НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри

Комп'ютерних наук

_____Голуб Б. Л., к.т.н., доцент

_____

«___»_____2025р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

На тему: « Програмне забезпечення для введення домашньої бухгалтерії з підтримкою синхронізації між пристроями »

Спеціальність 121 «Інженерія програмного забезпечення»

Гарант освітньої програми:_____Вайганг Г.О_____

Керівник дипломного проекту:_____ / Ніколаєнко Д.В. /
підпис ПІБ

Виконав: _____ / Прохоренко А.О. /
підпис ПІБ

КИЇВ-2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

«ЗАТВЕРДЖУЮ»
Завідувач кафедри
Комп'ютерних наук
Голуб Б. Л., к. тех.н., доцент

«__» _____ 20__ р.

З А В Д А Н Н Я

на виконання бакалаврської кваліфікаційної роботи студенту

Прохоренко Артем Олександрович
(прізвище, ім'я, по батькові)

Спеціальність 121 «Інженерія програмного забезпечення»

Тема кваліфікаційної бакалаврської роботи: «Програмне забезпечення для ведення домашньої бухгалтерії з підтримкою синхронізації між пристроями»

затверджена наказом ректора НУБіП України від "_16_" грудня 2025 р. № 2248С

Термін подання завершеної роботи на кафедру _____
(рік, місяць, число)

Вихідні дані до кваліфікаційної бакалаврської роботи

_____

Перелік питань, що підлягають розробці:
1. Аналіз предметної області
2. Проектування системи
3. Реалізація системи
4. Рекомендаціїї щодо впровадження та експлуатації системи

Дата видачі завдання "_____"_____2024 р.

Керівник бакалаврської роботи _____ Ніколаєнко Д.В.
(підпис) (прізвище та ініціали)

Завдання прийняв до виконання _____ Прохоренко А.О.
(підпис) (прізвище та ініціали студента)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

WPF – Windows Presentation Foundation

XAML – Extensible Application Markup

CLR – Common Language Runtime

BCL – Базова бібліотека класів у .NET

ASP.NET – Фреймворк для створення веб-додатків на .NET

API – Інтерфейс програмування додатків, набір функцій для взаємодії між програмами

SQL) – Мова структурованих запитів

СУБД – Система управління базами даних

HTTP – Протокол передачі гіпертексту

TCP – Протокол керування передачею

IP – Інтернет-протокол, основа для роботи мережі

UML – Уніфікована мова моделювання

ID – Ідентифікатор, унікальний код для запису

UX – Досвід користувача ПЗ

ВСТУП

Фінансова грамотність і вміння слідкувати за власними витратами набувають дедалі більшої ваги у сучасному світі. Економічна нестабільність, зростання цін, інфляція та широке поширення безготівкових розрахунків набирають все більшої популярності, тому введення домашньої бухгалтерії стає справжньою необхідністю. Це допомагає не тільки планувати свої витрати наперед, а й досягати поставлених фінансових цілей, уникати заборгованостей і накопичувати гроші. На сьогоднішній день існує велика кількість додатків для введення обліку витрат, але багато з них мають свої проблеми – недостатня функціональність, неможливість синхронізації даних на різних пристроях, тощо. Такі обмеження створюють незручність для тих, хто прагне вести облік транзакцій і отримувати аналітику транзакцій з різних пристроїв, ділитися даними з членами сім’ї .

Мета дипломної роботи є створення програмного забезпечення для ведення домашньої бухгалтерії з підтримкою синхронізації між пристроями. Система має відповідати сучасним вимогам клієнтів і мати ключові можливості: зручний та інтуїтивно зрозумілий інтерфейс для швидкого внесення доходів та витрат, категоризація транзакцій, детальну історію операцій, синхронізацію даних між кількома пристроями в реальному часі, надійне зберігання персональної інформації користувача, легке створення звітності в різних форматах. Проект передбачає створення клієнтського додатку для комп’ютера на основі технології WPF з використанням мови програмування C#. Для управління базою даних використовується PostgreSQL, розгорнутий у хмарному середовищі, що забезпечує високу доступність, гарну масштабованість і безпечне зберігання даних. Використання хмарної інфраструктури дозволяє реалізувати просту та надійну синхронізацію між пристроями, а також забезпечує резервне копіювання для захисту від втрати будь-якої інформації.

Дана система є актуальною, оскільки вона вирішує основні проблеми ведення домашньої бухгалтерій в даний та відкриває можливості для користувача покращити своє фінансове положення.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Актуальність теми

Ведення своїх фінансів – це важливий інструмент для управління доходами і витратами, що допомагає покращити своє фінансове становище. У сучасному світі, де економічна ситуація часто змінюється, а купівельна спроможність споживачів знижується через інфляцію, ведення домашньої бухгалтерії стає необхідністю для більшості людей.

Домашня бухгалтерія допомагає:

1. Контролювати витрати: Регулярне відстеження своїх витрат допомагає звернути увагу на непотрібні покупки та на основі цього оптимізувати свій бюджет.

2. Планувати майбутнє: Облік фінансів сприяє створенню фінансових планів для накопичення грошей на великі покупки, відпуску, пенсію тощо. Особи, які слідкують за своїм бюджетом, значно частіше досягають успіхів у досягненні своїх фінансових планів. Дані свідчать про те, що поведінка, пов’язана з плануванням, тісно пов’язана з досягненням цілей, незалежно від рівня доходу чи освіти.

3. Запобігати боргам: Своєчасне відстеження доходів і витрат допомагає уникати ситуацій, коли витрати перевищують доходи, що є основною причиною заборгованостей. Важливою частиною управління боргами є знання того, як уникнути боргів. Створення домашнього бюджету і його дотримання допоможе контролювати виплати боргів і систематично заощаджувати на інші цілі.

4. Підвищувати фінансову грамотність: Ведення обліку допомагає краще розуміти фінансові процеси, такі як вплив відсоткових ставок чи інфляції, що підвищує загальну фінансову обізнаність людини.

Сучасне суспільство стикається з різними економічними викликами, що роблять ведення домашньої бухгалтерії ще більш актуальним у наш час:

1. Інфляція: Зростання цін на товари та послуги знижує купівельну спроможність. У 2023 році середній рівень інфляції в Україні склав 12,9%.

2. Поширення безготівкових розрахунків: Зростання популярності таких платежів ускладнює слідкування за витратами. За даними НБУ, у 2024 році частка цифрових операцій в Україні досягла 65% від загального обсягу усіх транзакцій.

3. Економічна нестабільність: Постійна зміна курсів валют та нестабільність доходів створюють додаткові перешкоди при плануванні капіталу. У таких умовах ведення обліку за грошима стає потрібним інструментом для миттєвої адаптації до змін.

1.2 Визначення вимог до програмного забезпечення

1.2.1 Функціональні вимоги

При вивченні предметної області одним із ключових етапів є визначення основних вимог до системи. Завдяки цьому можна визначити функціональність, рівень захисту, сферу використання, що надає можливість покращити кінцеву якість програмного продукту.

Функціональні вимоги – це вимоги до програмного забезпечення, які описують основні можливості внутрішньої роботи системи, її поведінку.

Для системи введення домашньої бухгалтерії можна виділити декілька основних вимог:

1. Реєстрація – це механізм який надає можливість створювати новим користувачам облікові записи у системі для подальшої роботи з нею.

2. Авторизація – забезпечує вхід для вже зареєстрованого користувача в систему після перевірки вірності облікових даних

3. Облік доходів та витрат – програма повинна надавати користувачам доступ до основних функцій для роботи з транзакціями: додавання, оновлення, видалення.

4. Планування бюджету – це функція, яка дозволяє створювати бюджет з певним лімітом на певні часові періоди для окремих категорій або загальних витрат. Вона дозволяє користувачу краще відстежувати свої витрати з метою накопичення грошей на майбутні витрати.

5. Звітність – можливість генерувати звіти у вигляді таблиць, графіків або діаграм для подальшого експорту користувачем. Допомагає структурувати дані за певними параметрами для аналізу даних.

6. Синхронізація між різними пристроями – забезпечення технології синхронізації, через розгорнуту в хмарному середовищі базу даних. Дозволяє отримувати потрібну інформацію з різних пристроїв використовуючи обліковий запис користувача.

7. Автоматичне резервне копіювання та відновлення – це механізми, які відповідають за регулярне створення резервних копій у хмарному середовищі з можливістю швидкого відновлення інформації.

1.2.2 Нефункціональні вимоги

Нефункціональні або технічні вимоги – це вимоги до програмного забезпечення, які стосуються продуктивності, масштабованості, продуктивності та інших критеріїв, що визначають систему. Ці вимоги поділяються на види: апаратні , операційні та інтерфейсні.

До основних нефункціональних вимог входять:

1. Продуктивність – додаток повинен швидко обробляти запити користувачів. Хмарна база даних повинна забезпечувати мінімальний час затримки під час звичайних операцій, таких як читання, запис або оновлення даних. Клієнтський додаток має запускатися без перебоїв.

2. Масштабованість – архітектура має підтримувати можливість додавання нових серверів без необхідності зміни самої логіки програми. Необхідно гарантувати стабільну роботу програмного забезпечення за умови високого навантаження з боку користувачів.

3. Надійність – хмарна інфраструктура має включати резервне копіювання даних та автоматичне перемикання на резервний сервер, якщо виникнуть проблеми з працездатністю.

4. Безпека – усі дані користувачів, включно з транзакціями і особистою інформацією, мають бути завжди зашифровані алгоритмом.

5. Інтерфейс – має бути зрозумілим для користувача і простим у використовувані. Система повинна підтримувати українську мову та надавати допоміжні підказки для полегшення роботи з нею.

6. Керування помилками – автоматичне виявлення помилок на всіх рівнях та повідомлення користувача о про них.

Дотримання функціональних і нефункціональних вимог зазначених вище дозволяє забезпечити роботу програми згідно очікуванням користувачів і сприяє оперативному впровадженню програмного забезпечення.

2 ПРОЕКТУВАННЯ СИСТЕМИ

2.1 Вибір платформ та інструментів розробки

Розробка будь-якої системи вимагає ретельного вибору платформ та інструментів, які допомагають забезпечити усі вимоги визначені на етапі планування. Зручне середовище розробки, бібліотеки та інші інструменти скорочують час створення системи й зменшують кількість помилок на етапі розробки. Отже, виникає необхідність у ретельному виборі відповідних платформ та інструментів для розробки.

2.1.1 WPF додаток

Система введення домашньої бухгалтерії з підтримкою синхронізації між пристроями складається з двох головних частин: клієнтської, яка відповідає за взаємодію з користувачем та серверної, що займається зберіганням та обробкою даних. Клієнтська частина реалізована за допомогою WPF – графічного підсистеми Microsoft для розробки застосунків на платформі Windows. Ця платформа використовується для створення комп’ютерних додатків, які мають складні інтерфейси, наприклад, фінансові системи, текстові редактори чи інструменти для аналізу даних.

Одним із плюсів використання WPF є створення зрозумілих та привабливих інтерфейсів, які відповідатимуть встановленим вимогам дизайну. Завдяки адаптивним макетам, можна легко налаштовувати систему для різних екранів, незважаючи на розміри екрану, що робить її зручною для різних користувачів при використанні на будь-якому настільному пристрої.

Також обрана технологія використовує апаратне прискорення DirectX для швидкого і плавного показу складних графічних елементів. Апаратне прискорення – це використання графічного процесу замість центрального при обробці графіки, анімацій тощо. Це важливо враховувати при створені фінансових систем, оскільки потрібно стрімко відображати велику кількість

даних без збоїв. Важливо, що механізм рендерингу оптимізований для мінімізації затримок при роботі на комп’ютерах із середніми технічними характеристиками.

Ця платформа дозволяє створювати інтерактивні інтерфейси, які реагують на різні події, такі як кліки миші, введення з клавіатури або перетягування. Користувач може легко додати транзакцію через натискання кнопки, фільтрувати дані в таблиці за категоріями або отримувати підказки при наведенні на певні елементи при роботі з системою. В результаті користувачі можуть виконувати звичні операції швидше та з меншими зусиллями.

WPF дає змогу створювати стилі та шаблони для елементів інтерфейсу, що в свою чергу дозволяє змінювати вигляд елементів: кнопок, таблиць, графіків згідно з потребами користувачів. Ця гнучкість допомагає створити єдиний та привабливий вигляд системи.

Підтримка української локалізації в обраній підсистемі забезпечує простоту адаптації інтерфейсу для українських користувачів. Усі елементи: меню, підказки чи повідомлення про помилки, відображаються українською мовою, а формати дати, години та валюти відповідають місцевим стандартам.

WPF має модульну структуру, що робить додавання нових функцій простим. Модульна архітектура – це підхід, який дозволяє розподіляти велику програму на окремі незалежні частини, або модулі. Кожен модуль відповідає за певну функцію. Це забезпечує гнучкість у розвитку системи, зокрема можливість впроваджувати нові види аналітики, формати звітів та функціонал управління бюджетом. Платформа підтримує швидке оновлення даних, що дає змогу автоматично оновлювати баланс рахунку або графік витрат, коли додається нова транзакція тощо. Це дуже важливо для фінансових систем, адже споживачі хочуть миттєво мати доступ до щойно внесеної інформації.

2.1.2 Фреймворк .NET

WPF є частиною екосистеми .NET що в свою чергу є платформою від Microsoft, яка допомагає створювати додатки для різних пристроїв та операційних систем. Вона має бібліотеки, інструменти та мови програмування, які дозволяють розробляти різноманітні програми. .NET підтримує об'єктно-орієнтоване програмування, асинхронні операції, безпеку даних і працює на різних платформах. У розробці WPF-застосунку використовується .NET 8, що забезпечує сучасну основу для створення функціональних настільних програм.

Одним з основних компонентів фреймворку є CLR – основа, яка відповідає за виконання коду, управління пам’яттю, обробку помилок і безпеку даних. Код спочатку компілюється у проміжне представлення і перетворює його у машинний код за допомогою JIT-компіляції. Для WPF CLR забезпечує стабільну роботу клієнтського додатку, обробку подій інтерфейсу, належне управління ресурсами. Він займається керуванням пам’яті, автоматично видаляючи зайві об’єкти, щоб уникнути витоків пам’яті. До цього ж, цей компонент забезпечує обробку помилок, що допомагає системі і користувачу вчасно реагувати на збій.

Також в .NET входить BCL – це набір бібліотек, які допомагають працювати з файлами, мережею, базами даних, шифруванням і обробкою даних. У WPF BCL використовують для обробки даних, наприклад, для категоризації транзакцій чи створення звітів. Коли користувач додає транзакцію, бібліотека перевіряє дані на правильність і готує їх для збереження в базі. Для створення звітів BCL пропонує інструменти для обробки даних, такі як підрахунок витрат по категоріях або створення статистики для діаграм. Завдяки підтримці мережевих функцій, можлива реалізація асинхронної взаємодії з хмарним сервером для обміну та синхронізації даних.

Останньою ключовою особливістю .NET є підтримка асинхронного програмування. Це дозволяє виконувати такі операції, як запити до бази даних або API, не заважаючи користувачеві під час роботи з програмним забезпеченням. Коли споживач синхронізує дані з хмарним сервером, .NET дає

можливість робити це у фоновому режимі, тож користувач може продовжувати вводити транзакції або переглядати звіти без затримок.

Мова програмування C# використовується в WPF для опису логіки програми. C# – це об’єктно-орієнтована мова програмування з простим синтаксисом для платформи .NET, яку створила Microsoft. Вона схожа на C і Java, але з кількома додатковими можливостями для полегшення роботи над складними проектами. Завдяки зрозумілому синтаксису можна легко писати логіку для обробки ключових подій.

C# підтримує об’єктно-орієнтоване програмування, що важливо для створення модульних систем. Це дозволяє створювати класи для представлення сутностей з різними властивостями і методами. Такий підхід дозволяє зберігати гнучкість архітектури, що спрощує додавання нових функцій із мінімальними змінами в коді.

Мова розмітки XAML – декларативна мова, що базується на XML. Це важливий інструмент, який відповідає за створення графічного інтерфейсу при роботі з WPF. Вона дозволяє описати, як виглядатимуть різні елементи інтерфейсу, такі як кнопки, текстові поля, таблиці чи діаграми.

XAML допоможе визначити структуру інтерфейсу. Завдяки цьому можна описати, як будуть розташовані елементи, такі як поле для введення даних, випадаючий список або кнопки, у вигляді ієрархічного коду. Такий підхід дозволяє швидко змінювати розташування або вигляд елементів без змін у логіці програми, що робить дизайн гнучким і простим для адаптації до нових вимог

Для створення веб-додатку серверної частини системи домашньої бухгалтерії буде використано технологію ASP.NET. Вона призначена для створення веб-сервісів і веб-застосунків, яка дозволяє створювати API для обміну даними між клієнтом і сервером.

Однією з ключових переваг обраного фреймворка над аналогами є можливість миттєвої обробки запитів. Для створення REST API, яке взаємодіє з

WPF-клієнтом, було обрано технологію ASP.NET. API-ендпоінти, будуть відповідати за частину функцій в системі. Такий підхід дозволить частково розділити клієнтську та серверну логіку, роблячи систему більш модульною.

Серверну частину, побудовану на ASP.NET, легко розгорнути в хмарному середовищі. Хмарне розгортання забезпечить високу зручність з різних пристроїв, автоматичне масштабування під час зростання кількості користувачів, що відповідає зазначеним раніше вимогам до надійності та доступності. ASP.NET підтримує контейнеризацію через Docker, що спрощує розгортання та оновлення серверної частини.

2.1.3 СУБД PostgreSQL

Для управління даними обрана СУБД PostgreSQL – відкрита реляційна система управління базами даних. PostgreSQL дає змогу виконувати складні запити, підтримує цілісність транзакцій, зберігає структуровані дані. Завдяки тому що є підтримка роботи в хмарі можна забезпечити синхронізацію і резервне копіювання.

Обрана СУБД підтримує індекси, як B-дерево чи GIN, що допомагає прискорити запити, зменшивши час виконання навіть для великих обсягів даних. PostgreSQL також може обробляти запити одночасно, що в свою чергу дозволяє швидко виконувати агрегацію даних. Обробка запитів повинна займати мінімальний час тому дуже важливо, щоб користувачі швидко отримували свої дані під час роботи з програмою.

Для ефективного управління базами даних та взаємодії з реляційною структурою SQL важливо використовувати зручні інструменти адміністрування, які поєднують візуалізацію і функціональність. pgAdmin це інструмент для адміністрування та розробки баз даних PostgreSQL, що надає графічний інтерфейс для зручного управління реляційними базами даних на локальних і віддалених серверах.

Функціональність pgAdmin включає графічний редактор SQL з підсвічуванням синтаксису, інструменти для перегляду та редагування даних. Інтерфейс дозволяє зручно переглядати об’єкти бази даних в деревоподібному вигляді, де можна побачити схеми, таблиці та їхні властивості, а також виконувати адміністративні завдання. Додаткові можливості включають відображення графічних планів запитів і планувальник завдань. Для моніторингу системи доступні інформаційні панелі, які відображають стан сервера та бази даних у реальному часі.

2.2 Архітектура системи

2.2.1 Опис клієнто-серверної архітектури

Для системи введення домашньої бухгалтерії важливо обрати правильну архітектуру яка передбачає синхронізацію між пристроями. Клієнт-серверна модель (рис. 2.1) є однією з ключових архітектур у розробці програмного забезпечення, яка визначає спосіб взаємодії між двома основними компонентами: клієнтом і сервером. У цій моделі клієнт, зазвичай додаток або пристрій користувача, відправляє запити до сервера. Сервер обробляє ці запити та повертає дані або виконує певні дії. Це дає можливість розподіляти ресурси, централізовано керувати даними та забезпечувати взаємодію між багатьма користувачами. Із-за своєї ефективності архітектура клієнт-сервер лежить в основі багатьох сучасних систем, включно з програмними додатками, корпоративними системами, базами даних і мобільними програмами.

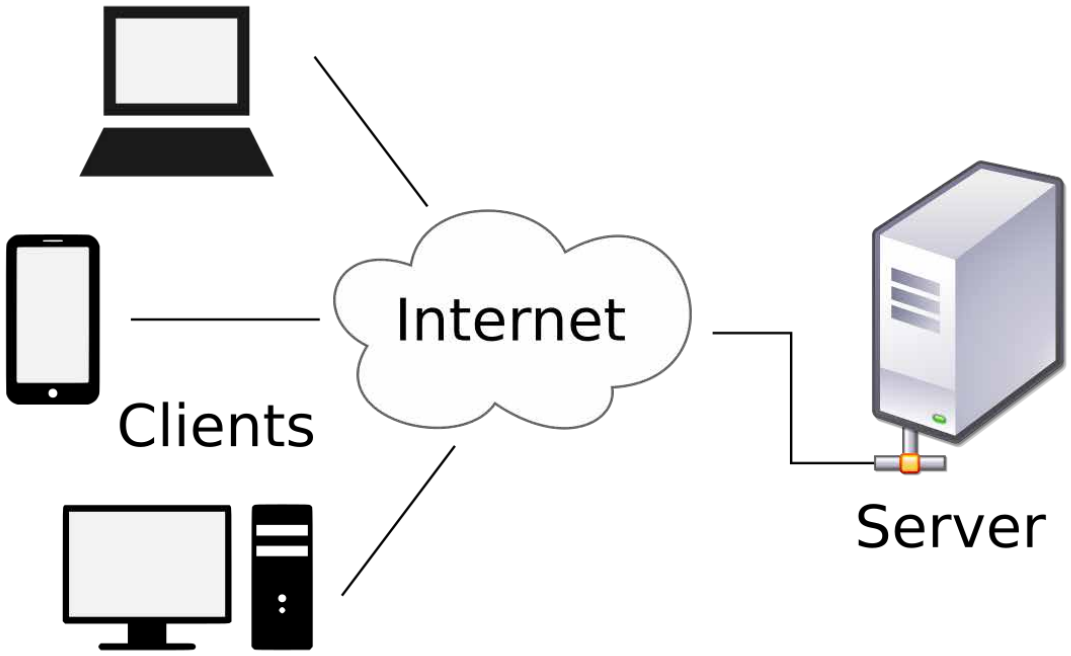


Рис. 2.1 Клієнто-серверна архітектура

В порівнянні з іншими типами архітектур таких, як однорангова або автономна, клієнт-серверна модель більше підходить для обраної системи. Хоча однорангова модель може синхронізувати пристрої, її реалізація складніша, і вона менш надійна для фінансових даних через відсутність централізованого контролю. Автономна модель, де пристрої працюють незалежно, не підтримує синхронізацію, що робить її непридатною для використання між різними пристроями.

В описаній моделі клієнт виконує роль інтерфейсу для користувача. Це дає змогу користувачеві взаємодіяти із системою, надсилати запити і отримувати відповіді. Сервер обробляє ці запити, зберігає дані і забезпечує безпеку. Комунікація між клієнтом і сервером відбувається завдяки використанню мережевих протоколів, таких як HTTP, HTTPS, TCP/IP та інших, залежно від потреб системи. Ця модель допомагає розділити функції на окремі частини, що спрощує розробку, підтримку і масштабування програмного забезпечення. Наприклад, сервер може займатися обчисленнями або зберігати великі обсяги даних, а клієнт пропонувати зручний інтерфейс.

Головна перевага обраної моделі – централізоване управління даними. Воно дає змогу забезпечити великий рівень безпеки, що важливо для програмного забезпечення, яке обробляє фінансові дані. Сервер використовує різні методи шифрування, щоб захистити дані під час передачі та зберігання. Для додаткового забезпечення безпеки впроваджуються різні способи перевірки особи, щоб уникнути несанкціонованого доступу третіми особами. Сервер перевіряє, чи вірні облікові дані користувача, перш ніж надати йому доступ до фінансової інформації. Завдяки цьому користувачі можуть бути впевненими у безпеці своїх даних.

2.2.2 Порівняння товстого і тонкого клієнту

Для реалізації системи було обрано двошарову клієнт-серверну архітектуру завдяки її простоті в розробці та відповідності вимогам проекту. Така модель забезпечує ефективну взаємодію між клієнтською та серверною частинами, спрощує підтримку та масштабування системи, а також дозволяє швидко впроваджувати необхідний функціонал.

Двошарова архітектура складається з двох основних рівнів: представлення і даних. У клієнт-серверній моделі ці рівні розподілені між клієнтом і сервером по-різному, в залежності від того, чи є клієнт товстим чи тонким. Товстий клієнт і тонкий клієнт – це два основних способи втілення клієнтської частини, кожен з яких має свої особливості, переваги та недоліки. Вони можуть впливати на вибір архітектури для конкретного програмного забезпечення.

Щоб визначитись із конкретним видом двошарової клієнтно-серверної моделі треба спочатку розглянути плюси і мінуси доступних варіантів:

Товстий клієнт характеризується тим, що більшість роботи з обробкою даних і бізнес-логікою проходить на стороні користувача. Це означає, що додаток, має всі необхідні функції, здатен виконувати складні обчислення. Частиною товстого клієнта є зручний графічний інтерфейс, що покращує взаємодію з користувачем.

Тонкий клієнт більше покладається на сервер для виконання різних обчислень і логіки. У такій моделі клієнтська частина зазвичай має простий інтерфейс, який показує дані від сервера та передає запити від користувачів. Тонкий клієнт часто реалізується у вигляді веб-додатку, що працює у браузері, або терміналі, який залежить від сервера для виконання всіх операцій. Це допомагає зменшити навантаження на клієнтський бік тому, що основна обчислювальна потужність зосереджена на сервері.

Таблиця 2.1

Критерій	Товстий клієнт	Тонкий клієнт
Обробка даних	Переважно на стороні клієнта	Переважно на стороні сервера
Бізнес-логіка	Реалізована локально в додатку	Реалізована на сервері
Інтерфейс користувача	Повноцінний інтерфейс	Спрощений, часто веб-інтерфейс
Обчислювальне навантаження	На клієнтському пристрої	На сервері
Оновлення програмного забезпечення	Потрібно оновлювати клієнт вручну або через систему оновлень	Централізоване оновлення на сервері
Ресурси клієнта	Потребує більше ресурсів від комп’ютера	Працює навіть на слабких пристроях

На основі цих порівнянь було обрано клієнт-серверну модель з товстим клієнтом. Вона пропонує зручний інтерфейс, забезпечує надійну синхронізацію

між пристроями і високий рівень безпеки. Товстий клієнт дозволяє користувачам виконувати обчислення локально і швидко отримувати відповіді.

2.3 Архітектурний шаблон у клієнтській частині

2.3.1 Шаблон MVVM

Архітектурний шаблон який використовується у WPF додатках називається Model-View-ViewModel. MVVM – спосіб організації коду, який допомагає створювати модульні й прості в обслуговуванні клієнтські додатки. Використовується для поділу моделі та її уявлення, що необхідно для їх зміни окремо друг від друга. Використання цієї архітектури є ефективним для товстих клієнтів, адже вона дозволяє чітко розділити обов'язки між інтерфейсом, обробкою даних і бізнес-логікою. Це робить код гнучким для масштабування й тестування.

MVVM поділяється на три головні частини: Model, View і ViewModel. Model працює з даними і бізнес-логікою, яка не пов’язана з інтерфейсом користувача. Він містить дані, такі записи про транзакції, витратні категорії і бюджети, а також логіку для обробки.

View є шаром представлення, який відповідає за відображення даних користувачу і обробляє його дій в інтерфейсі. Він включає такі елементи: таблиці транзакцій, діаграми витрат, форми для введення даних і кнопки для виконання дій. View не містить бізнес-логіки, просто показує дані, які надходять від ViewModel, і передає дії користувача, такі як натискання кнопок чи введення тексту, до ViewModel для подальшої обробки. Це допомагає тримати інтерфейс і логіку окремо.

ViewModel слугує посередником між Model і View, забезпечуючи зв’язок між даними та їх відображенням. Він отримує дані від Model, обробляє їх, щоб вони були зрозумілі для View, і передає їх далі. Також обробляє дії користувача з View та викликає відповідні функції в Model.

Головними перевагами MVVM є те, що цей шаблон підтримує двостороннє зв’язування даних. Крім того, MVVM дає змогу легко налаштувати інтерфейс для різних платформ, бо View можна змінювати незалежно від ViewModel і моделі. Оскільки ViewModel не залежить від View і працює тільки з даними та логікою, її можна тестувати окремо, використовуючи модульні тести. Model також можна тестувати окремо, перевіряючи, чи правильно зберігаються дані чи як проходить взаємодія з сервером.

Порівняно з іншими архітектурними шаблонами, а саме MVC або MVP, MVVM більше підходить для реалізації у товстому клієнті, адже підтримує двостороннє зв’язування даних і дає змогу чітко розділити логіку відображення. MVC, часто має близький зв'язок між View і Controller, що може ускладнити тестування і зміну інтерфейсу. MVP не ефективно підтримує двостороннє зв’язування, тому може знадобитися більше коду для синхронізації даних.

Таким чином, MVVM була обрана для клієнтської частини програми ведення домашньої бухгалтерії через її здатність забезпечувати чітке розділення відповідальності, підтримку двостороннього зв’язування даних, легкість тестування та сумісність із товстим клієнтом у клієнт-серверній моделі. MVVM пропонує оптимальний баланс між функціональністю і простотою розробки, що робить її найкращим вибором для реалізації програмного забезпечення.

2.4 Моделювання системи

2.4.1 Діаграма UseCase

Для забезпечення чіткого розуміння структури та поведінки програмної системи застосовується моделювання з допомогою UML. Це уніфікована мова моделювання, яка допомагає візуалізувати, проектувати і документувати програмне забезпечення. Головна ідея UML – показати архітектуру системи так, щоб кожен міг її зрозуміти. Моделювання

UseCase – це діаграма UML, що відображає взаємодію між акторами та випадками використання, якітпредставляють конкретні дії або функції. Мета

такої діаграми показати як користувачі взаємодіють із системою і які функції вона має.

Діаграма складається з кількох ключових елементів: акторів та варіантів використання. Актори зображуються в вигляді фігурок людини для користувачів або зовнішніх систем, і вони представляють тих, хто взаємодіє із системою. Варіанти використання відображаються овалами всередині яких описуються конкретні задачі. Лінії між акторами та варіантами використання показують дії доступні кожному актору. Також є спеціальні зв’язки, як включення, розширення чи узагальнення, для деталізації взаємодій.

UseCase діаграма не вдається в деталі, а зосереджується на тому, що система має робити з погляду користувача.

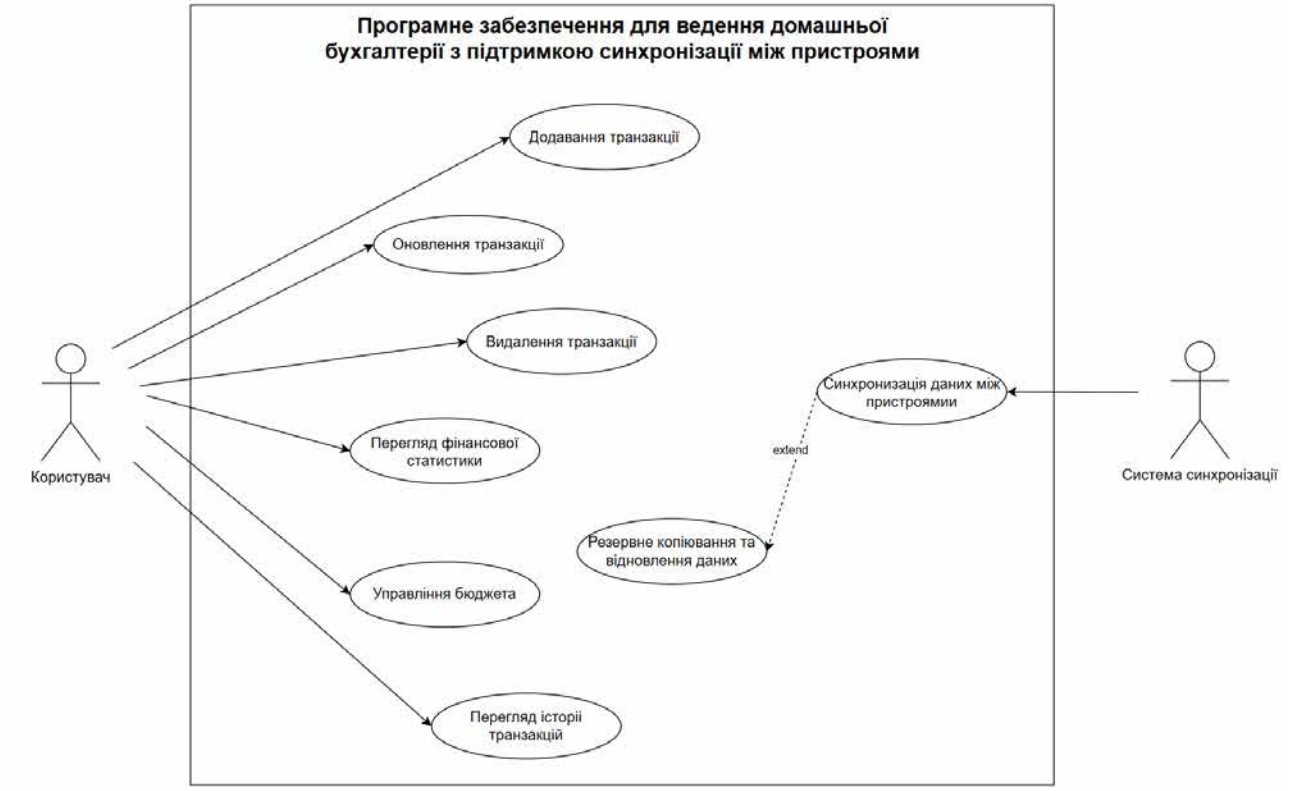


Рис. 2.2 Діаграма UseCase

На діаграмі (рис 2.2) представлені два актори: користувач зліва і система синхронізації справа. Користувач – основний актор, який працює з програмою, а система синхронізації виступає як зовнішній актор, що забезпечує обмін даними між пристроями. Між цими акторами розташовані варіанти використання, які показують основні функції програми.

Користувач має доступ до варіантів використання, таких як додавання транзакцій, оновлення транзакцій, видалення транзакцій, перегляд фінансової статистики, управління бюджетом і перегляд історії транзакцій. Ці функції дають змогу користувачу взаємодіяти з транзакціями, бюджетом та фінансовими звітами. Система синхронізації забезпечує синхронізацію даних між пристроями. Цей варіант використання включає в себе додаткову дію - резервне копіювання та відновлення даних.

2.4.2 Діаграма послідовності

Діаграма послідовності – це діаграма UML, яка показує, як об'єкти взаємодіють один з одним у певному сценарії чи процесі з плином часу. Вона відображає послідовність повідомлень або викликів між об'єктами для виконання конкретної функції чи задачі. Головна мета діаграми послідовності показати як передається інформація чи контроль між частинами системи, підкреслюючи часові відношення між діями.

Діаграма має кілька основних елементів. Об'єкти або актори у графіку показані у вигляді прямокутників або фігурки людини з вертикальними лініями, які показують їхній стан протягом усього процесу. Повідомлення між об'єктами відображаються горизонтальними стрілками, що показують передачу даних. Час рухається зверху вниз, що допомагає побачити порядок дій.

Створення діаграми послідовності допомагає зрозуміти, як працює взаємодія, виявити проблеми з часом виконання чи синхронізацією та поліпшити процеси. Діаграма стає важливим інструментом на етапі проектування та тестування, даючи чітке уявлення, як система працює.

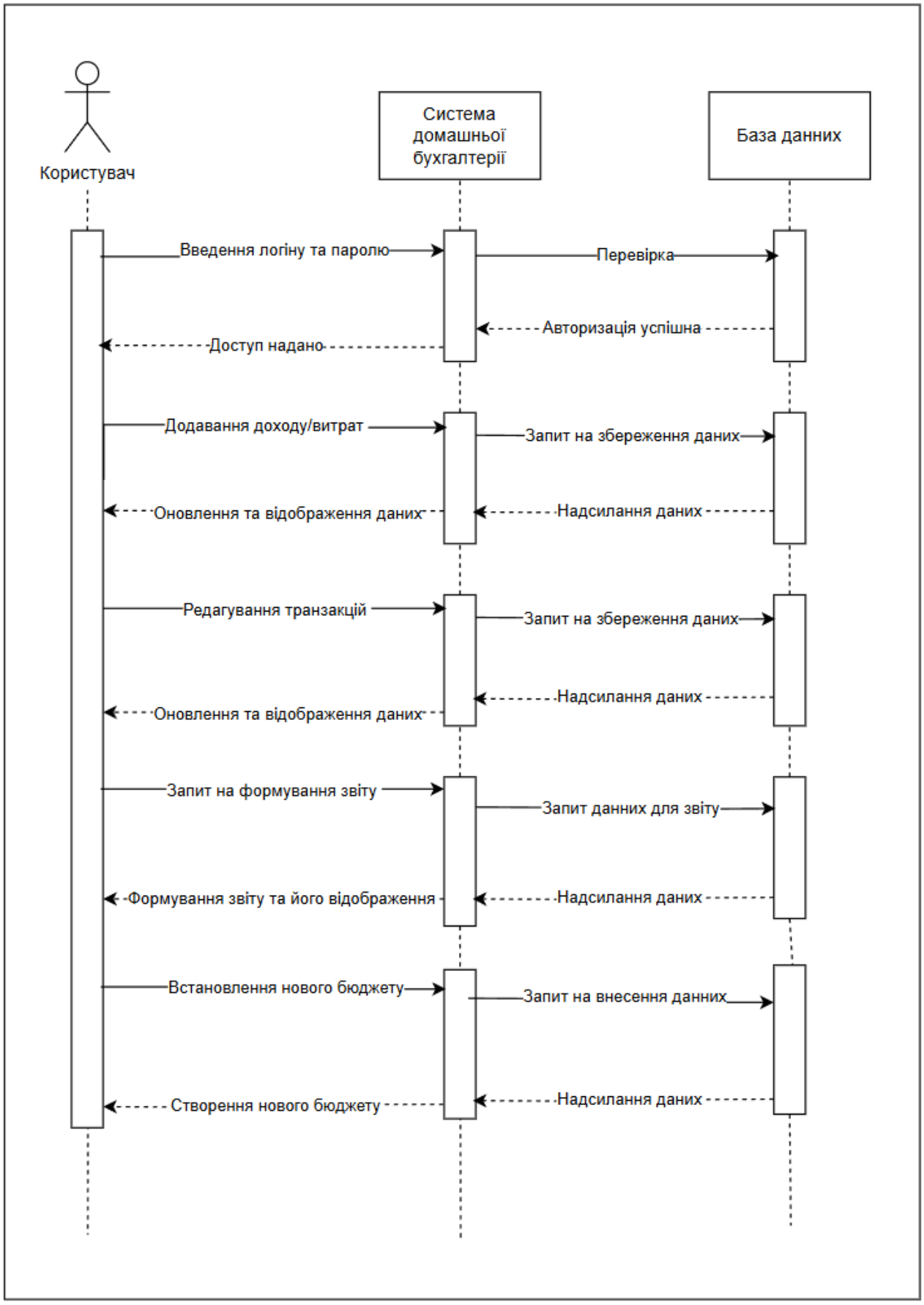


Рис. 2.3 Діаграма послідовності

Створена діаграма послідовності (рис. 2.3) зображує, як користувач взаємодіє із системою для ведення домашньої бухгалтерії, включаючи роботу з базою даних. На діаграмі є три ключові елементи: користувач, система управління фінансами і база даних . Користувач починає дії, які йдуть до системи, а система потім працює з базою даних для їх виконання. Події відбуваються зверху вниз, що показує порядок дій.

Все починається з того, що користувач надсилає запит на ведення логіну та паролю. Система отримує цей запит і передає його в базу даних на перевірку. Після перевірки при коректності даних користувач отримує до неї доступ. Далі користувач може додавати та редагувати транзакції, надсилаючи зміни у базу даних, запитувати формування звіту, та створювати новий бюджет.

Діаграма ілюструє, що система виступає як посередник, передаючи запити і отримуючи дані, завдяки чому користувач може безперервно взаємодіяти з базою даних. Це відображає, як функціонує клієнт-серверна модель з товстим клієнтом, де дані обробляються локально поєднуючись з синхронізацією через сервер.

2.4.3 Діаграма компонентів

Діаграма компонентів – це діаграма UML, яка показує структуру системи. Вона відображає фізичні компоненти та їх взаємодію. Головна мета цієї діаграми – надати огляд архітектури системи на рівні реалізації і як усі частини взаємодіють.

Вона має кілька основних частин: компоненти та зв’язки між ними. Компоненти показані у вигляді прямокутників з тегом «component» та їхніми назвами. Зв’язки між ними показані лініями зі стрілками. Це можуть бути залежності, інтерфейси або порти. Компоненти бувають як програмні, так і

апаратні. Лінії асоціації показують, як компоненти обмінюються даними чи викликають один одного.

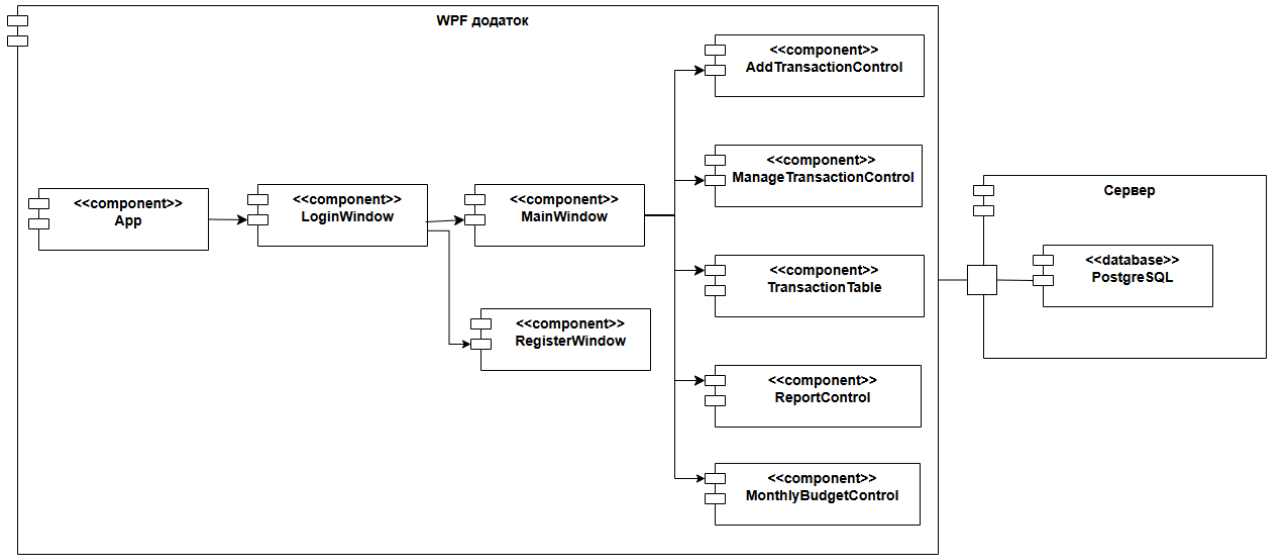


Рис. 2.4 Діаграма компонентів

У цій діаграмі (рис 2.4) представлена архітектура програмного забезпечення, реалізованого з використанням WPF і хмарного серверу. Центральним елементом є WPF Додаток, який складається з кількох компонентів. Початковий компонент, запускає додаток і потім переходить до вікна входу. В ньому користувач може увійти в свій обліковий запас, або перейти до вікна реєстрації, де можна створити новий обліковий запис, якщо його ще нема.

Функції головного меню доповнюються кількома контролерами: для додавання транзакцій, для оновлення та видалення транзакцій, для створення звітів, для роботи з бюджетом та для перегляду історії транзакцій.

WPF додаток під’єднаний до Сервера, що взаємодіє з базою даних. Дані зберігаються та обробляються на сервері, а додаток надає користувачеві інтерфейс і логіку для роботи з системою.

2.4.4 Діаграма розгортання

Діаграма розгортання – це тип діаграми UML, який показує фізичне розміщення програмних компонентів на апаратному забезпечені. Вона фокусується на фізичних елементах, таких як сервери, комп'ютери та мобільні пристрої, і на тому, як ці елементи взаємодіють один з одним. Головна мета діаграми – дати ясне уявлення про інфраструктуру, яка потрібна для роботи системи, і як її можна масштабувати або управляти.

Основні елементи: вузли, які виглядають як куби або прямокутники і представляють фізичні або віртуальні пристрої. Усередині цих вузлів компоненти, такі як виконувані файли або бібліотеки, і вони позначаються як артефакт або компонент. Зв'язки між вузлами відображені у вигляді ліній, з вказівкою типу зв'язку.

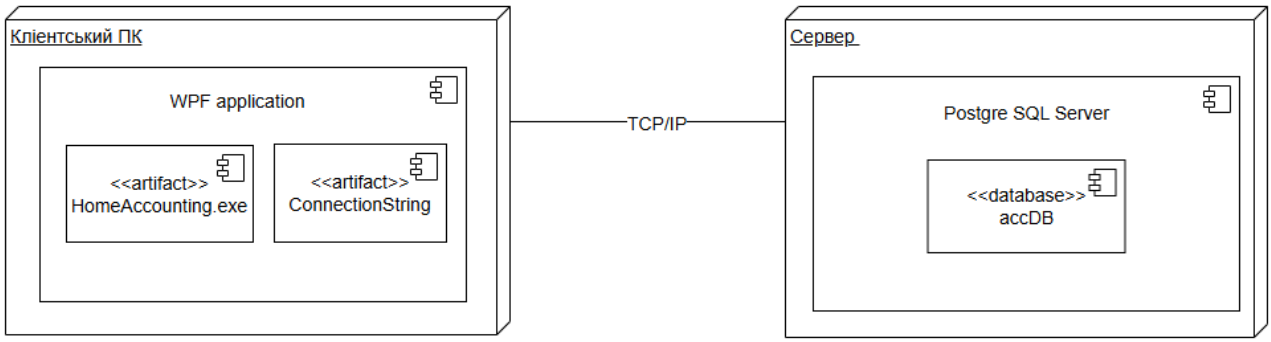


Рис. 2.5 Діаграма розгортання

На діаграмі (рис 2.5) представлено фізичну архітектуру, яка показує, як програмне забезпечення розподілене між апаратними вузлами та їх взаємодію. Два основні вузли: клієнтський ПК і сервер. На клієнтському ПК встановлено WPF програма, яка складається з файлу програми і рядка підключення. Клієнтська програма працює прямо на комп'ютері користувача і використовує рядок підключення для роботи з віддаленим сервером.

На сервері розташована база даних, що зберігає та обробляє усю інформацію, яка використовується у додатку. Клієнтський комп’ютер підключається до сервера через TCP/IP протокол, що дозволяє йому надсилати запити до бази даних і отримувати відповіді. Це відображає товсту клієнт-серверну модель: клієнт обробляє інтерфейс і логіку, а сервер займається управлінням даними.

2.5 Моделювання даних

2.5.1 Реляційність баз даних

У процесі розробки інформаційних систем важливу роль відіграє правильне структурування та організація даних. Щоб дані зберігались, оброблялись і аналізувались ефективно, потрібно добре розуміти їхню структуру, зв'язки та обмеження. Для цього використовують моделювання даних. Воно передбачає створення різних концептуальних, логічних чи фізичних моделей, щоб представити дані у зрозумілому вигляді. Моделювання даних використовують при розробці баз даних, проектуванні систем і управлінні інформацією. Це полегшує розуміння вимог до даних. Також зменшує ризик помилок під час реалізації, дозволяючи виявити проблеми з даними ще на етапі проектування. Завдяки добре спроектованій моделі можна легко додавати нові функції, без значних змін у структурі бази даних.

Під час проєктування бази даних необхідно дотримуватись принципів реляційної моделі та забезпечити нормалізацію до третьої нормальної форми

Реляційна модель – це основний підхід у теорії баз даних, який спирається на математику і реляційну алгебру. У цій моделі дані представлені у вигляді таблиць де кожна з них має свої рядки та стовпці, що утворюють структуроване сховище інформації. Для забезпечення унікальності рядків використовується первинний ключ. Цей ключ може посилатися на зовнішні ключі в інших таблицях, завдяки чому створюються зв’язки між таблицями.

У реляційній моделі дані організовані так, щоб зменшити повторення та зберегти цілісність. Зв'язки між таблицями визначаються за допомогою операцій реляційної алгебри, такі як проекція, селекція, об’єднання та перетин.

Нормалізація в реляційних базах даних – це спосіб організації даних для усунення повторення та забезпечення їх цілісності. Це досягається через приведення таблиць до певних нормальних форм.

Перша нормальна форма говорить про те, що кожне значення в таблиці не може бути розбитим на менші частини без втрати сенсу, а також у реляції не повинно бути повторюваних груп атрибутів. Головна ідея першої нормальної форми позбутися повторюваних груп і переконатися, що в таблиці є первинний ключ, який унікально ідентифікує кожен запис.

Друга нормальна форма застосовується до таблиць, які вже знаходяться в першій нормальній формі. Вона вимагає, щоб усі неключові атрибути були залежні від усього первинного ключа, а не від його частини. Це допомагає позбутися часткових залежностей у таблицях з комбінованими ключами. Основна мета другої нормальної форми - уникнути зайвих даних, які виникають через ці часткові залежності.

Третя нормальна форма будується на другій нормальній формі і вимагає, щоб усі неключові атрибути залежали безпосередньо від первинного ключа, а не від інших неключових атрибутів. Головна мета третьої нормальної форми уникнути транзитивних залежностей, зменшуючи надлишковість і полегшуючи оновлення даних без ризику помилок.

Мета реляційної моделі – це створення логічної структури даних, яка дозволяє зручно зберігати, оновлювати та отримувати інформацію за допомогою SQL. Вона підтримує цілісність даних через різні обмеження, такі як унікальність і зв'язки між таблицями.

2.5.2 ER-діаграма

Перші рівні моделювання даних концептуальній та логічний. На концептуальному рівні моделювання визначаються основні сутності та їх зв'язки. На логічному рівні додається більше деталей про структуру даних, вказуючи атрибути. Для зображення цих рівнів використовується ER-діаграма.

У цій діаграмі сутності виглядають як прямокутники, які позначають основні об’єкти системи. Атрибути, що описують ці об’єкти, знаходяться в відповідних прямокутниках. Два ключові атрибути у ER-діаграмі : первинний та вторинний ключ.

Первинний ключ - це атрибут або група атрибутів, які допомагають точно ідентифікувати кожен запис в сутності. В сутності первинним ключем може бути атрибут, який гарантує, що жодні два записи не мають однакового значення. На ER-діаграмі, первинні ключі як позначають як PK.

Зовнішній ключ - це атрибут або група атрибутів, які з’єднують дві сутності, посилаючись на первинний ключ іншої сутності. У сутності може бути атрибут , який слугує зовнішнім ключем та вказує на інший атрибут, який здійснив дію. Таким чином, цей атрибут переймає значення з таблиці на яку посилається, забезпечуючи зв'язок та цілісність даних. На ER-діаграмі, зовнішні ключі позначають як FK.

Між сутностями зв’язки показуються у вигляді ліній з позначками (рис 2.6), що визначають тип відносин. Основні типи зв'язків між сутностями: один до одного, один до багатьох, багато до багатьох. Вони показують, скільки об'єктів однієї сутності можуть бути пов'язані з іншими.

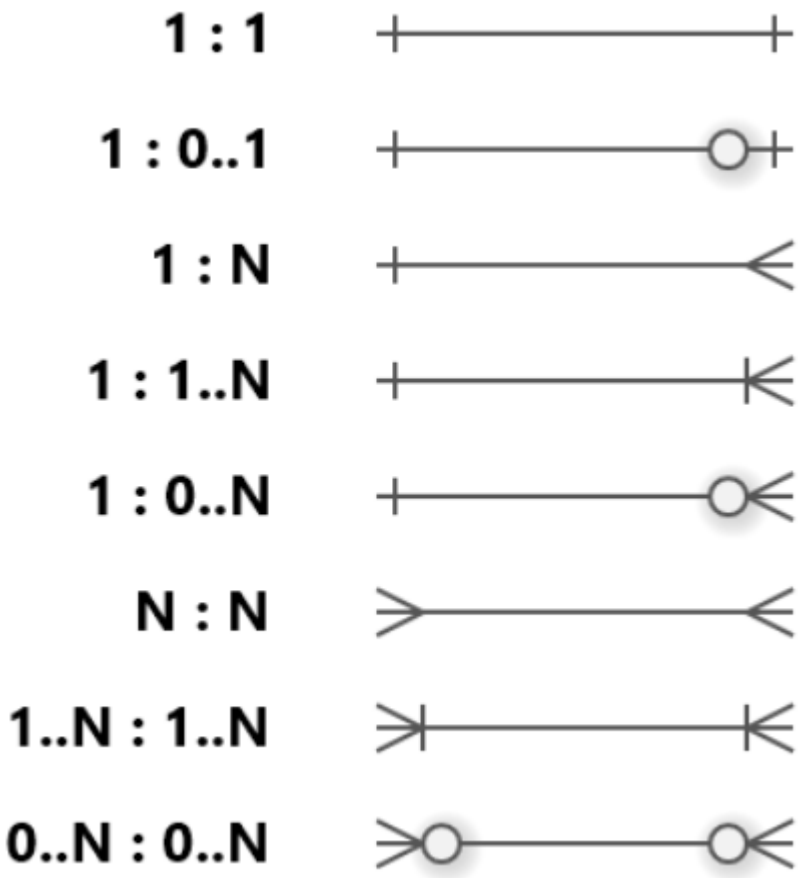


Рис. 2.6 Типи зв’язків

Зв'язок один до одного демонструє, що кожен екземпляр першої сутності може бути пов'язаний тільки з одним екземпляром другої. Використовується коли потрібно інформацію логічно та доцільно розділити на дві сутності.

Зв'язок один до багатьох є найпоширенішим. Він вказує на те, що один екземпляр першої сутності може мати багато зв’язків з другою сутністю, але кожен екземпляр другої сутності може бути пов’язаний лише з одним екземпляром першої.

Зв'язок багато до багатьох , відображає що один екземпляр першої сутності може бути пов’язаний з багатьма екземплярами другої сутності і навпаки.

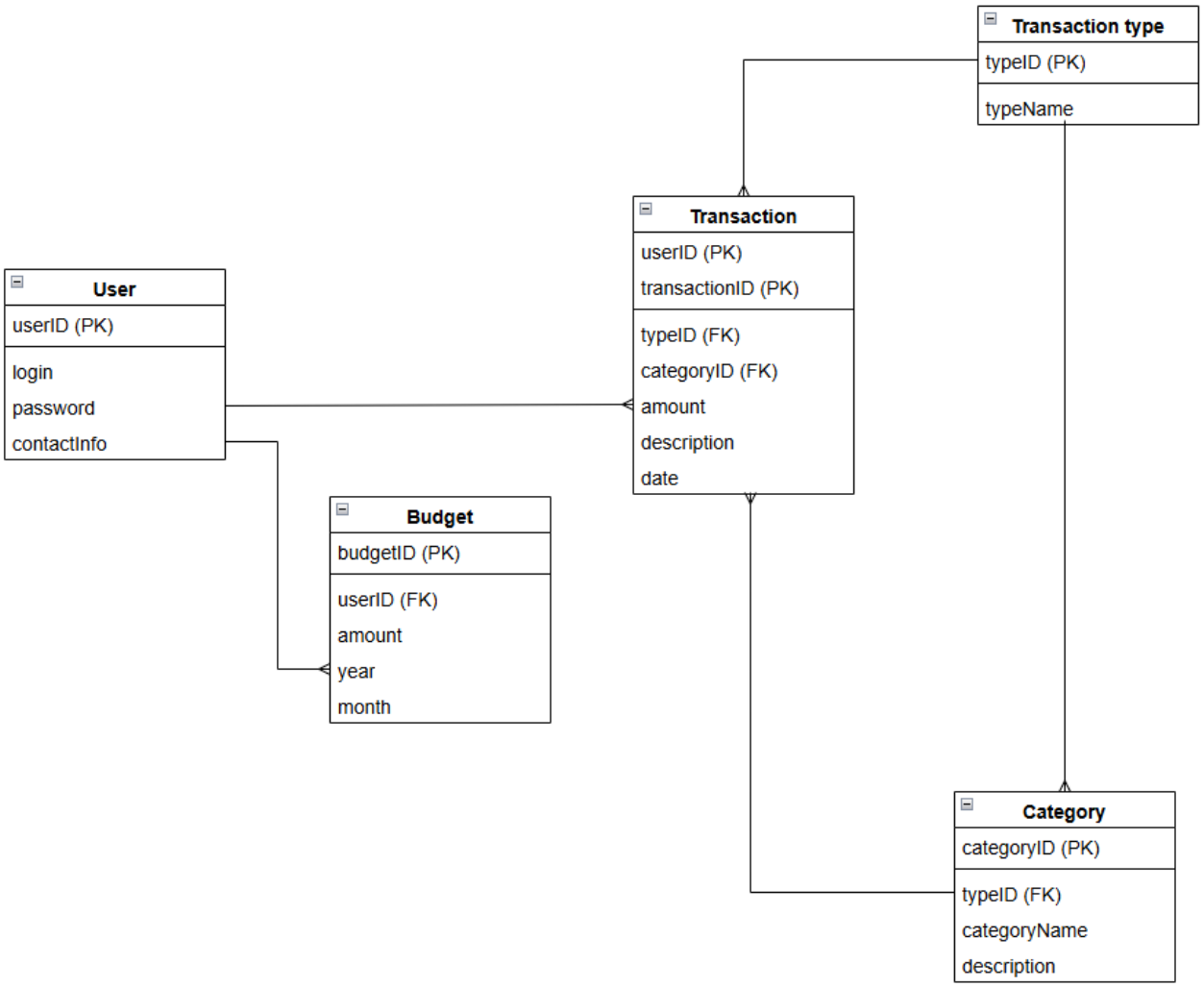


Рис. 2.7 ER-діаграма

На зображенні представлена ER-діаграма, яка моделює структуру даних. Діаграма має п’ять основних сутностей: користувач, бюджет, транзакція, тип транзакції та категорія, а також їхні зв’язки та властивості.

Сутність User має первинний ключ userID, а також атрибути login, password і contactInfo. Вона представляє собою користувачів системи. Завдяки зв’язкам один до багатьох між таблицями кожен користувач може мати кілька бюджетів та транзакцій.

У сутності Budget первинний ключ budgetID, а userID використовується як зовнішній ключ. Також дана сутність має атрибути amount, year і month, що описують бюджет на певний період.

Transaction характеризується первинними ключами transactionID та userID і атрибутами, categoryID, typeID, amount, description і date, які описують фінансові операції користувача. Зв’язок з сутностями Transaction type та category забезпечується зовнішніми ключами typeID та categoryID відповідно.

Сутність Category пов’язана з типами транзакцій через зовнішній ключ typeID і має первинний ключ categoryID та атрибути categoryName і description . Зв’язок один до багатьох встановлено між Category і Transaction, де categoryID виступає зовнішнім ключем у Transaction. Це дозволяє класифікувати транзакції за категоріями, пов’язаними з їх типами.

Остання сутність типи транзакцій має первинний ключ typeID і атрибут typeName, які містять інформацію про тип транзакції.

Отже, представлена модель є реляційною, бо всі елементи представлені у вигляді таблиць з чіткими первинними та зовнішніми ключами. Вона відповідає третій нормальній формі: кожен атрибут залежить лише від первинного ключа, транзитивні залежності відсутні, і повторювані групи виключені. Ніякі частини моделі не порушують принципи реляційності, і сама структура не є ні ієрархічною, ні об'єктною, ні документною.

3 РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Розробка інтерфейсу

3.1.1 Приципи зручності користування інтерфейсом

Створення графічного інтерфейсу для програмного забезпечення важливий крок у розробці програми. Інтерфейс визначає зручність та інтуїтивність взаємодії користувачів із додатком, що безпосередньо впливає на ефективність. Особлива увага приділяється врахуванню потреб користувачів. Інтерфейс інтегрується з програмною логікою, щоб він міг виконувати необхідні дії, такі як додавання транзакцій, перегляд бюджетів тощо.

Дотримання принципів зручності користування інтерфейсом - це один з важливих факторів успіху програмного продукту. Вони описані в стандарті ISO 9241 і є загально визначеними рекомендаціями для забезпечення комфортної і безпечної взаємодії між користувачем і системою. У контексті розробки інтерфейсу для програмного забезпечення, ці принципи поділяються на:

1. Ефективність інтерфейсу – досягається шляхом забезпечення швидкого та точного виконання завдань із мінімальними зусиллями. Інтерфейс повинен бути простим, з зрозумілими полями, щоб користувач міг швидко завершити роботу. Це відповідає стандарту, який вимагає економити час і зусилля користувача для досягнення своїх цілей.

2. Зручність системи – визначає ступінь, до якого система може бути використана без зайвих труднощів. Важливо зробити усі елементи дизайну однаковими на всіх екранах. Це зменщує навантаження на користувача та сприяє швидкій адаптації до інтерфейсу. Інтерфейс має бути простим у використанні та з передбачуваними діями, що відповідає рекомендаціям по простому управлінню.

3. Задоволеність користувача – залежить від швидкого зворотного зв’язку. Система повинна давати миттєву інформацію про результати дій. Допомагає підвищити довіру до системи.

4. Доступність – допомагає налаштувати інтерфейс під потреби різних користувачів, включаючи тих, хто має обмежені можливості. Полягає в підтримки контрастних кольорів, великих шрифтів тощо.

3.1.2 Діаграма навігації

Для забезпечення логічної структури інтерфейсу використовують діаграма навігації - інструмент моделювання, який показує перехід між різними екранами або сторінками в інтерфейсі. Вона фокусується на логіці переходів, показуючи переміщення між функціональними елементами програми, і допомагає спроєктувати UX.

Основні частини діаграми - це вузли, які показують екрани, вікна або сторінки. Стрілки показують, як можна переходити між цими елементами. Діаграма може також містити розгалуження, цикли і точки входу/виходу, щоб показати всі можливі варіанти взаємодії.

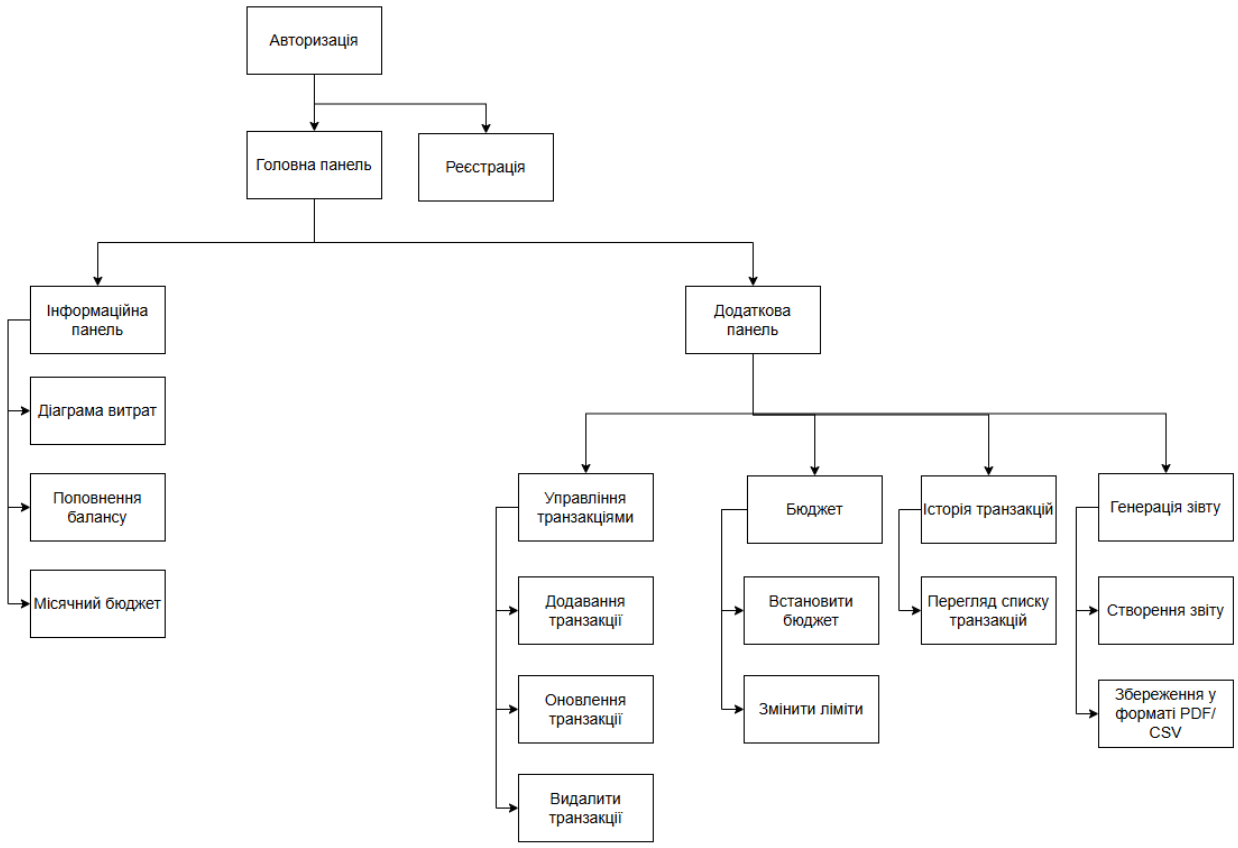


Рис. 3.8 Діаграма навігації

На діаграмі навігації зображеній на (рис. 3.8) , відображені усі переходи між екранами та їх наповнення. Початковою точкою є екран авторизація, який розділяється на два варіанти: головна панель для тих, хто вже зареєстрований і ввійшов в обліковий запис, і реєстрація для нових користувачів.

З головної панелі користувач може отримати доступ до різних функцій: інформаційна панель та додаткова панель. Інформаційна панель відповідає за відображення графіків та панелей: витрат, поповнення балансу, місячний бюджет. Додаткова панель потрібна для переходу до інших ключових функцій програми: управління транзакціями , бюджет , історія транзакцій, генерація звіту.

3.1.3 Програмування інтерфейсів

Після визначення основних елементів, структури ключових вікон, а також підходу для реалізації навігації між функціональними модулями програми йде їх реалізація. Нижче наведено детальний опис створених інтерфейсів для основних вікон та контролерів.

1. LoginWindow.xaml. Вікно входу (рис. 3.9) в систему перший екран при запуску програми створений для авторизації користувачів. Файл містить розмітку інтерфейсу, де є поля для вводу логіна і пароля, а також кнопка для входу і кнопка для переходу до вікна реєстрації.

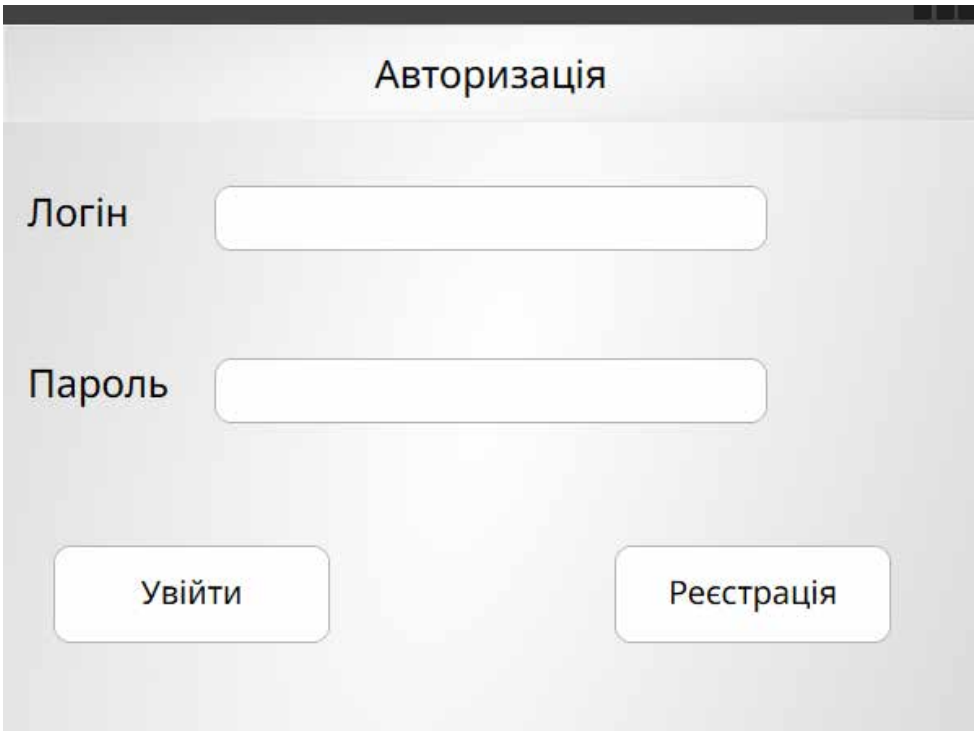


Рис. 3.9 Вікно входу

2. RegisterWindow.xaml. Вікно реєстрації (рис. 3.10) призначене для реєстрації користувачі в яких немає облікового запису. Файл містить розмітку для введення логіну, паролю і контактних даних і кнопка для підтвердження реєстрації.

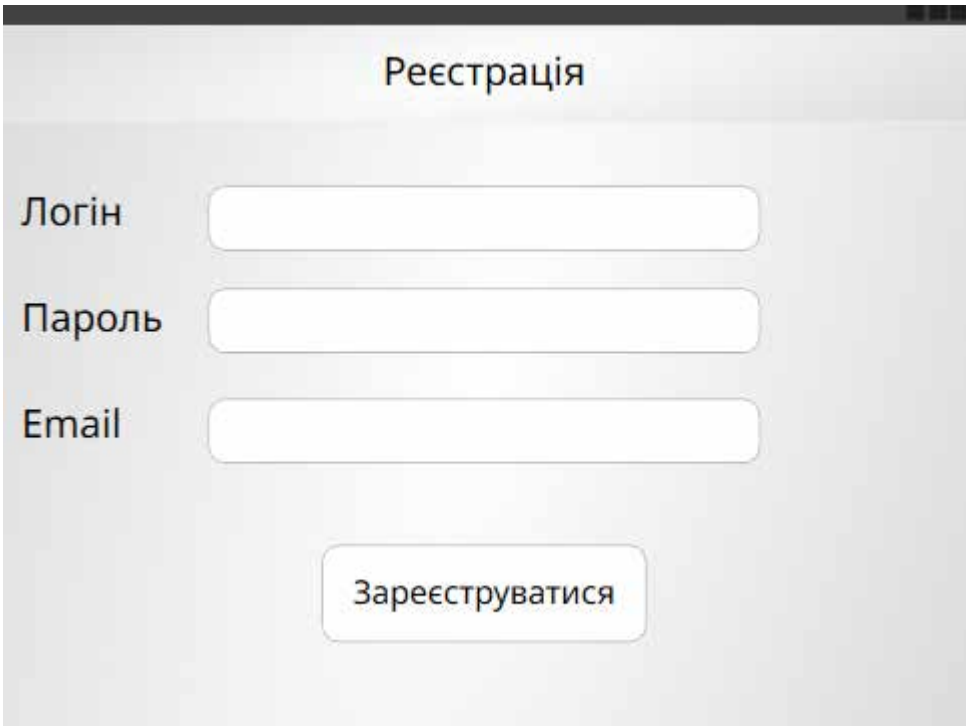


Рис. 3.10 Форма реєстрації

3. MainWindow.xaml. Головне вікно (рис. 3.11) основний інтерфейс програми, який надає доступ до ключових функцій. MainWindow.xaml містить панель для подальшої навігації та відображення графіків і панелей.

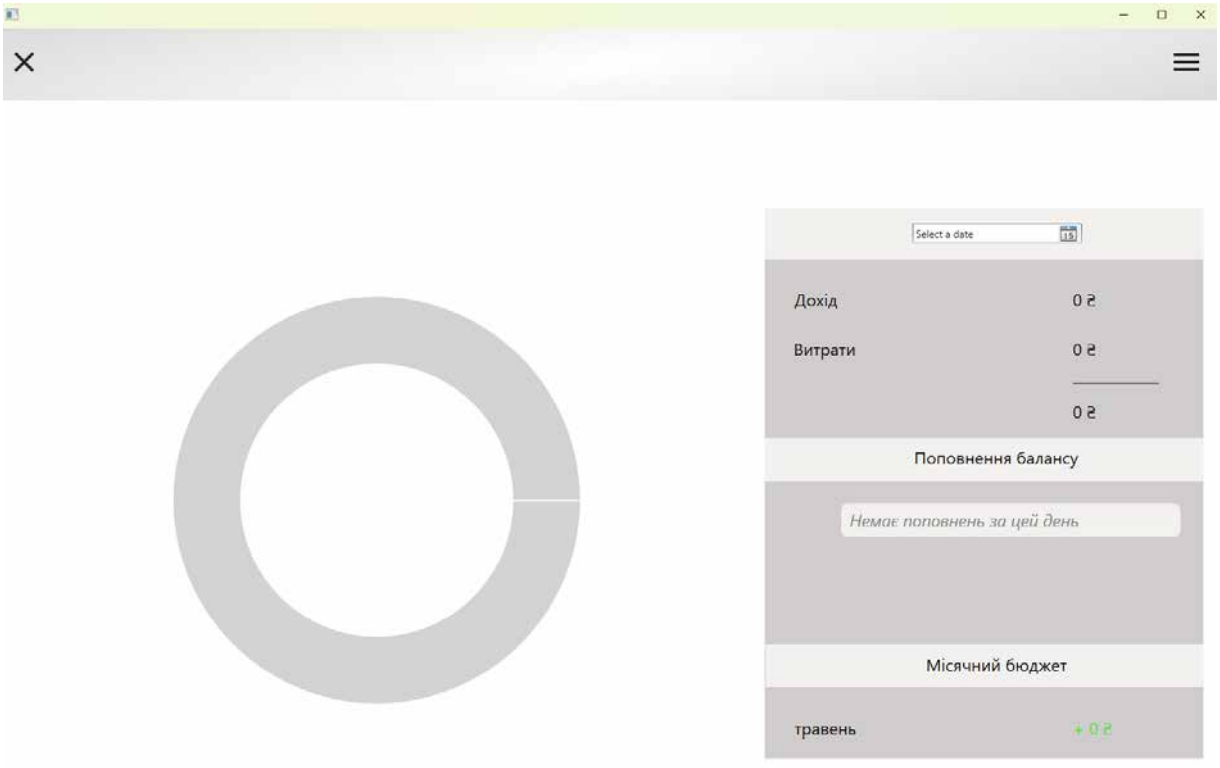


Рис. 3.11 Головне вікно

4. AddTransactionControl.xaml. Контролер додавання транзакцій (рис. 3.12) інтегрований у головне вікно для введення нових фінансових операцій. У файлі розміщені поля введення для суми, дати, категорії, опису та вибору типу транзакції.

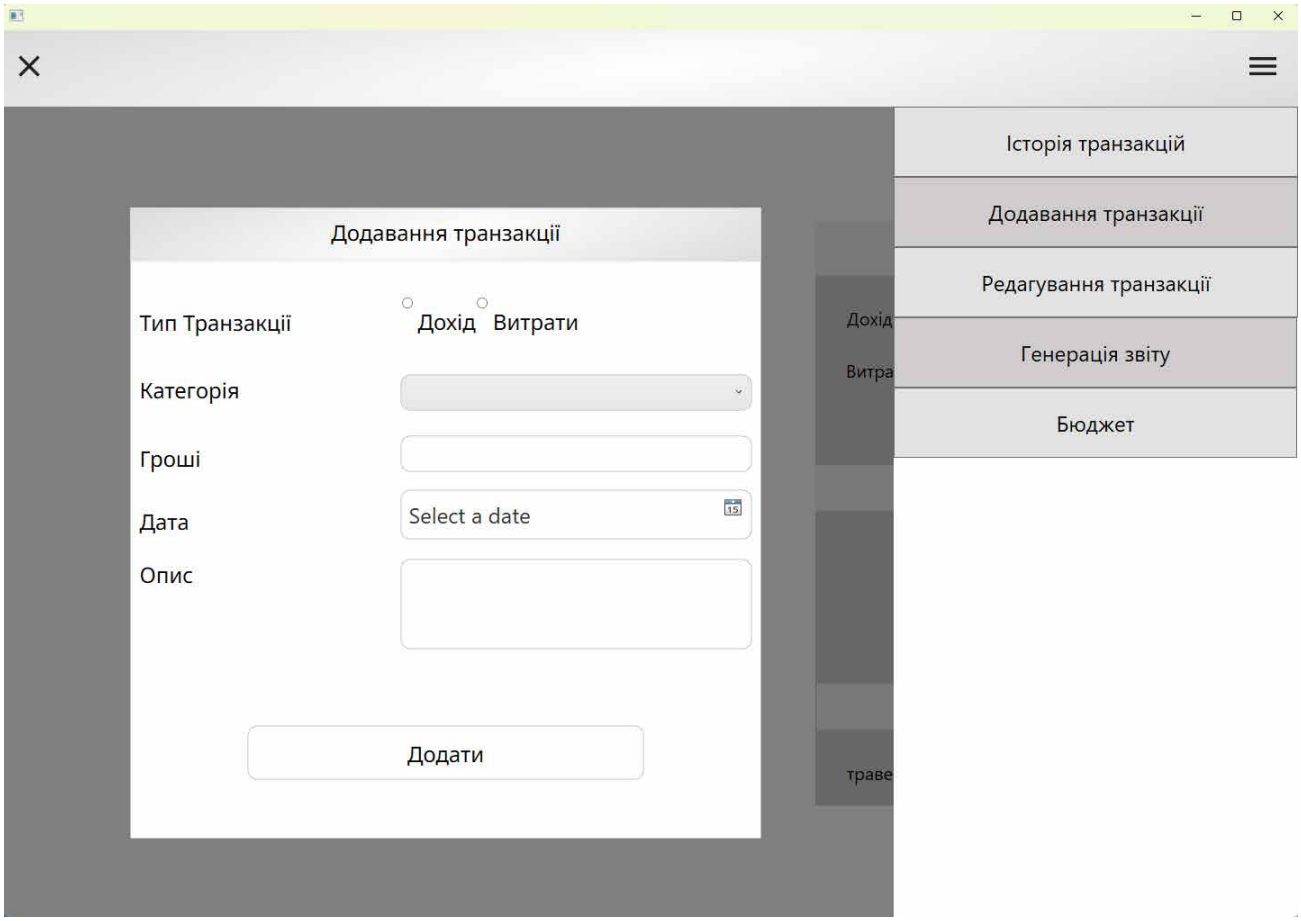


Рис. 3.12 Контролер додавання транзакцій

5. ManageTransactionControl.xaml. Контролер (рис. 3.13) дозволяє редагувати і видаляти вже існуючі транзакції. Розміщенні поля для введення ID транзакції, категорії, суми, дати , опису та вибору типу транзакції.

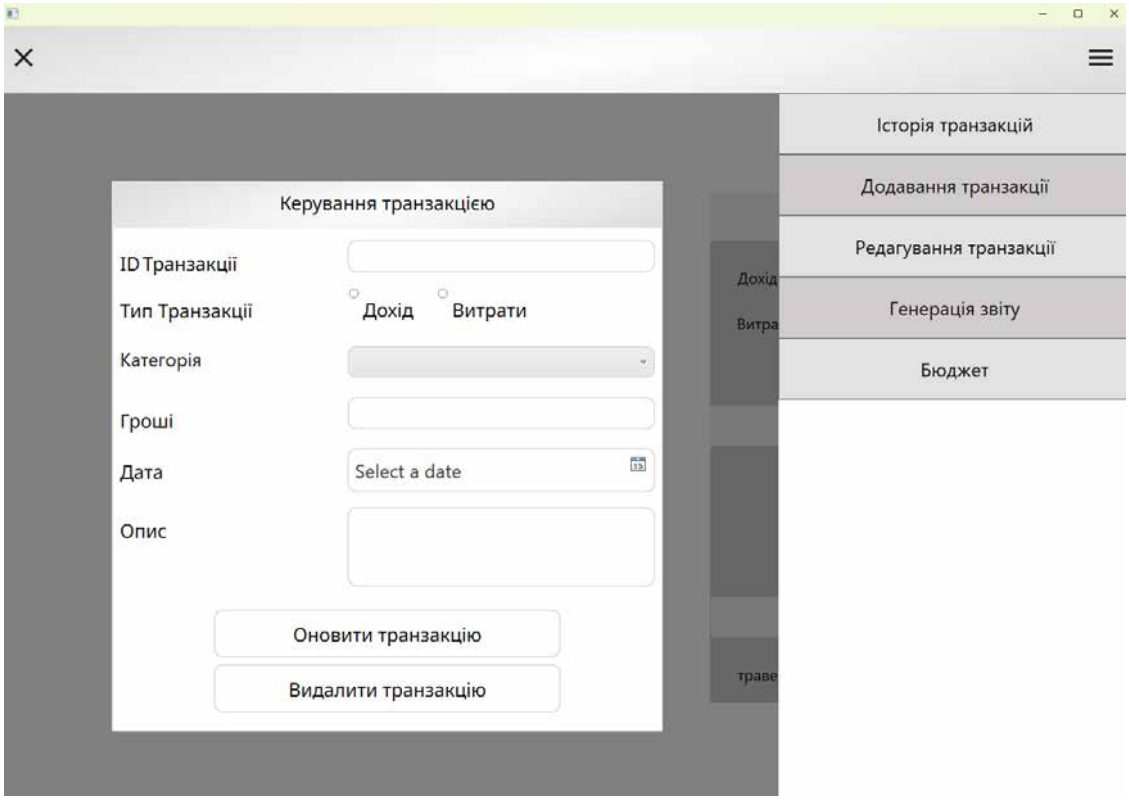


Рис. 3.13 Контролер редагування транзакцій

6. TransactionTable.xaml. Контролер (рис. 3.14) має таблиці транзакцій, які мають інформацію про всі транзакції.

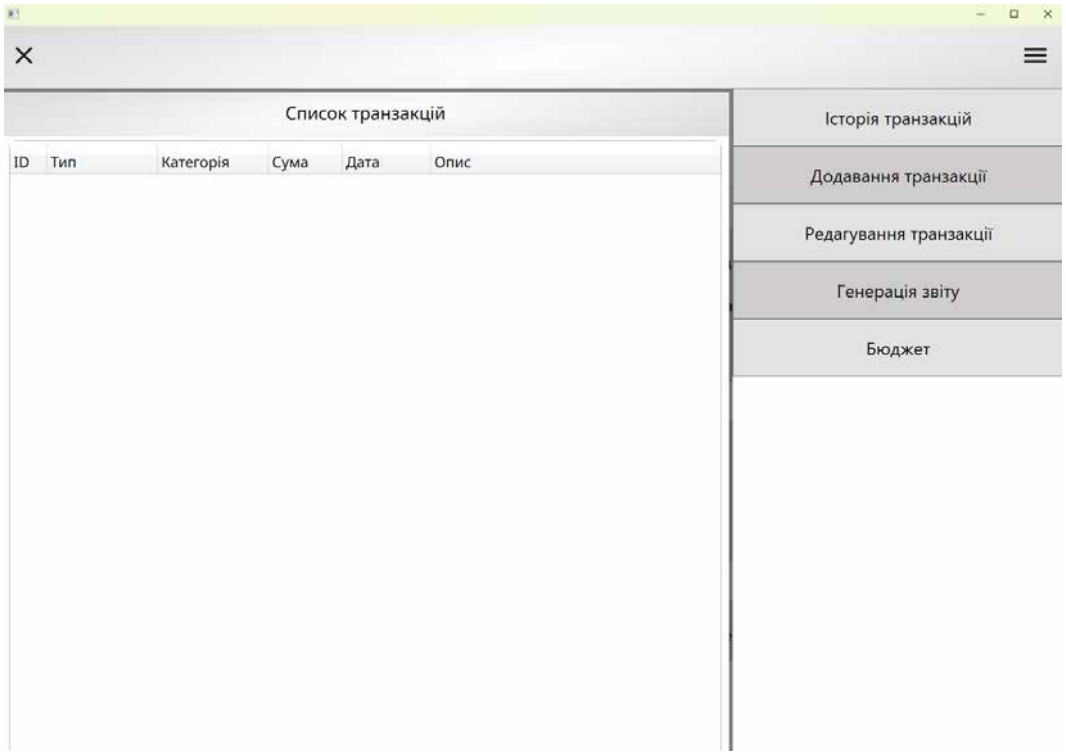


Рис. 3.14 Контролер створення звітів

7. ReportControl.xaml. Контролер звітів (рис. 3.15) відповідає за створення фінансових звітів та їх збереження у різних форматах. Інтерфейс програми має панель для вибору часових періодів та кнопки для генерації звітів у форматі pdf або csv на вибір користувача.

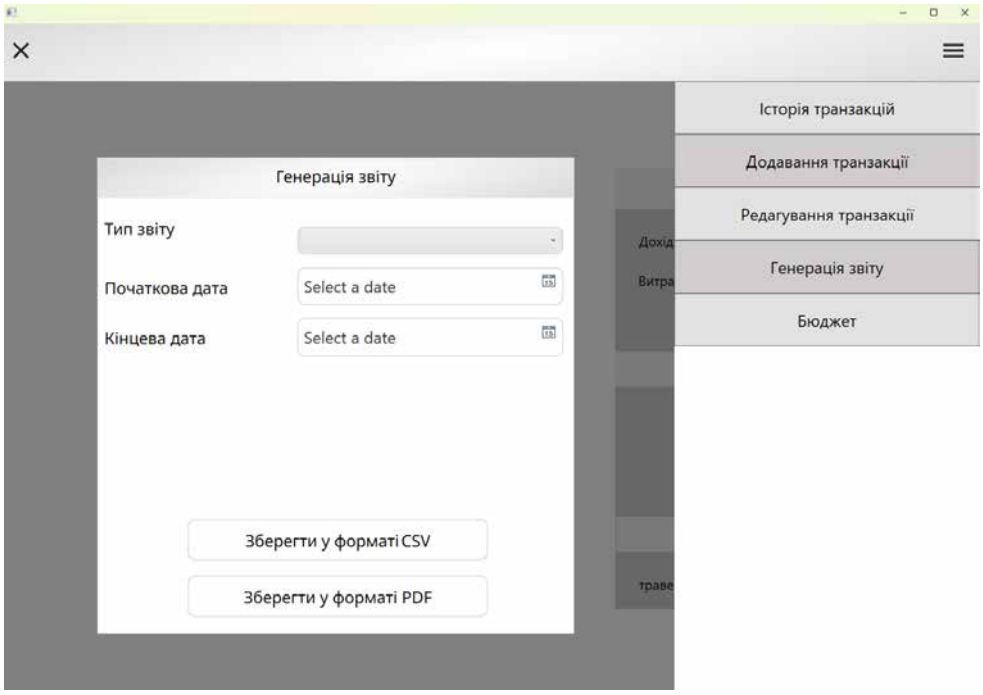


Рис. 3.15 Контролер генерації звітів

8. MonthlyBudgetControl.xaml. Контролер місячного бюджету (рис. 3.16) показує і допомагає управляти бюджетами. Дозволяє обирати місяць, рік та ліміт для встановлення або зміни бюджету за цей період.

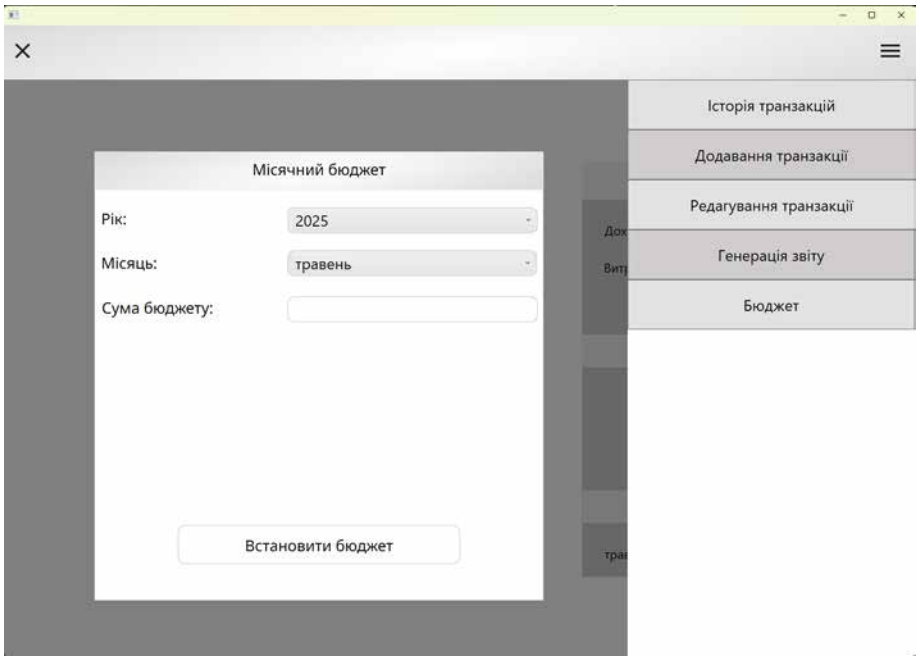


Рис. 3.16 Контролер місячного бюджету

3.2 Алгоритмізація та програмування модулів.

Після розробки інтерфейсу користувача за допомогою мови розмітки XAML, наступним етапом стає реалізація функціональної логіки програмного забезпечення. Цей крок полягає у програмуванні модулів, які забезпечують обробку даних, керування поведінкою додатку та взаємодію між користувачем і системою. Впровадження логіки є ключовим для забезпечення коректної роботи програмного продукту. Нижче наведено опис основних алгоритмів системи:

1. Модуль AddTransactionControl.xaml.cs відповідає за додавання транзакцій. Повний код наведений у Додатку А.

1.1 Алгоритм додавання транзакції спочатку перевіряє заповненість обов’язкових полів: категорії, суми та дати. У разі їх відсутності відображається повідомлення про необхідність заповнення всіх полів. Далі перевіряється коректність формату суми шляхом перетворення текстового введення в десяткове число. Якщо формат суми некоректний, відображається повідомлення про помилку. Після успішної перевірки даних встановлюється з’єднання з базою даних, формується SQL-запит для вставки нової транзакції в таблицю Transaction. Запит включає ідентифікатор користувача, тип транзакції, суму,

ідентифікатор категорії, дату та опис. Після виконання запиту відображається повідомлення про успішне додавання транзакції.

2. Модуль MainWindow.xaml.cs відображає статистику по транзакціями і діаграму витрат. Повний код наведений у Додатку А.

2.1 Алгоритм оновлення і створення діаграми реалізовано в методі UpdateChartUI. Створюється модель діаграми PlotModel з використанням бібліотеки OxyPlot, до якої додається кругова діаграма PieSeries. Якщо список категорій витрат порожній, створюється єдиний сірий круг без позначень. В іншому випадку для кожної категорії витрат із отриманих даних додається частина на діаграму із назвою категорії, сумою та вказується процент від усього графіку. Модель присвоюється елементу myPlot, розміщеному в частині XAML, для відображення графіку. Також відповідає за оновлення текстових блоків відображення загального доходу, витрат, чистого балансу та залишку місячного бюджету.

Алгоритм оновлення інтерфейсу списку доходів реалізовано в методі UpdateTopIncomesUI. Панель TopIncomePanel очищається. Якщо список доходів порожній, відображається повідомлення про відсутність поповнень. В іншому випадку для кожної категорії доходу створюється горизонтальна панель StackPanel, що містить два текстові блоки: один для назви категорії, інший для суми.

3. Модуль MonthlyBudgetControl.xaml.cs забезпечує функціонал управління місячним бюджетом, дозволяючи завантажувати, переглядати та зберігати бюджет для обраного року та місяця. Повний код наведений у Додатку А.

3.1 Алгоритм завантаження місячного бюджету реалізовано в методі LoadMonthlyBudget. Виконується SQL запит для вибірки суми бюджету з таблиці MonthlyBudget для заданого користувача, року та місяця. Якщо результат запиту

містить вже значення, сума бюджету відображається в елементі AmountTextBox. Якщо бюджет не знайдено, поле AmountTextBox очищається.

3.2 Алгоритм збереження місячного бюджету реалізовано в методі SaveButton_Click. Спочатку перевіряється наявність обраних року та місяця. Якщо вони відсутні, відображається повідомлення про пусті поля. Після перевірки даних встановлюється з’єднання з базою даних. Виконується SQL запит для перевірки наявності запису про бюджет для заданого користувача, року та місяця до базих даних. Якщо запис відсутній, виконується запит на вставку нового запису в таблицю MonthlyBudget із зазначенням ідентифікатора користувача, року, місяця та суми. Якщо запис вже існує, виконується SQL запит на оновлення суми бюджету.

4. Модуль ReportControl.xaml.cs забезпечує функціонал створення звітів про транзакції та бюджет у двох форматах PDF і CSV на основі даних, отриманих з інтерфейсу. Повний код наведений у Додатку А.

4.1 Алгоритм створення PDF звітів реалізовано в методі SavePdfButton_Click. Перевіряється наявність обраного типу Бюджет в елементі ReportTypeComboBox. Формується ім’я звіту, що включає поточний час і дату створення. При створені звіту типу Бюджет викликається метод LoadBudgetReportRows для отримання даних. Після цього створюється документ із таблицею, що містить стовпці для місяця та року, бюджету, витрат і різниці завдяки використанню бібліотеки QuestPDF. Якщо дані відсутні, відображається повідомлення про відсутність даних за обраний період. Для звіту типу Транзакції викликається метод LoadTransactions для отримання даних транзакцій. Якщо транзакції відсутні, відображається повідомлення про їх відсутність. Генерується PDF-документ із таблицею, що містить стовпці для дати, суми, категорії та опису транзакцій. Після генерації створені звіти відкривається за допомогою переглядача встановленого в системі за замовчуванням.

4.2 Алгоритм створення CSV звіту реалізовано в методі SaveCsvButton_Click. Він працює так само як і створення звіту SavePdfButton_Click, але має інше розширення файлу.

5. Модуль ManageTranctionControl.xaml.cs оновлює та видаляє транзакції завдяки відповідним алгоритмам. Повний код наведений у Додатку А.

5.1 Алгоритм оновлення транзакції реалізовано в методі UpdateTransactionButton_Click. Спершу перевіряється заповненість обов’язкових полів: ідентифікатора транзакції, категорії, суми та дати. У разі їх відсутності відображається повідомлення про необхідність заповнення всіх полів. Потім перевіряється коректність вказаного ідентифікатора транзакції та суми шляхом перетворення текстового введення в десяткове число. При неправильному форматі даних, відображається повідомлення про помилку. Після успішної перевірки даних встановлюється з’єднання з базою даних, формується SQL-запит для оновлення запису в таблиці Transaction. Запит включає ідентифікатор транзакції, ідентифікатор користувача, ідентифікатор категорії, суму, дату та опис. Після виконання запиту перевіряється кількість оновлених рядків, і відображається повідомлення про успішне оновлення або відсутність транзакції.

5.2 Алгоритм видалення транзакції реалізовано в методі DeleteTransactionButton_Click. Перевіряється наявність та коректність ідентифікатора транзакції. У разі некоректного формату відображається відповідне повідомлення. Після підтвердження правильності даних встановлюється з’єднання з базою даних, формується SQL-запит для видалення запису з таблиці Transaction за ідентифікатором транзакції та користувача. Після виконання запиту перевіряється кількість видалених рядків, і відображається повідомлення про успішне видалення або відсутність транзакції.

6. Модуль TransactionTable.xaml.cs забезпечує зручний перегляд транзакцій та їхню подальшу обробку. Повний код наведений у Додатку А.

6.1 Алгоритм завантаження транзакцій реалізовано в методі LoadTransactions. Створюється список для зберігання об’єктів, що представляють собою транзакції. Виконується SQL запит, який об’єднує три таблиці: Transaction, TransactionType та ExpenseCategory для отримання потрібних атрибутів. Запит фільтрує транзакції за ідентифікатором користувача та сортує їх за датою. Дані зчитуються і формуються в об’єкти. У разі відсутності даних рядок залишається порожнім. Список транзакцій присвоюється властивості ItemsSource елемента TransactionDataGrid для відображення в таблиці у ньому.

7. Модуль UserService.cs забезпечує функціонал управління користувачами, зокрема пошук користувача за логіном, створення нового користувача та налаштування ролей бази даних для нього. Повний код наведений у Додатку Б.

7.1 Алгоритм пошуку користувача за логіном реалізовано в методі GetUserByLogin. Встановлюється з’єднання з базою даних через об’єкт NpgsqlConnection. Виконується SQL запит до таблиці Users для вибірки ідентифікатора користувача, логіна, пароля та контактної інформації за заданим логіном. Дані зчитуються за допомогою NpgsqlDataReader. Якщо запис знайдено, створюється об’єкт типу User із заповненими полями userID, login, password та contactInfo.

7.2 Алгоритм створення нового користувача реалізовано в методі CreateUser. Встановлюється з’єднання з базою даних. Виконується SQL-запит для вставки нового запису в таблицю Users із полями login, password та contactInfo. Після виконання запиту повертається ідентифікатор нового користувача

7.3 Алгоритм створення ролі бази даних для користувача реалізовано в методі CreateDbRoleForUser. Формується SQL запит, який перевіряє існування ролі з ім’ям, що відповідає логіну користувача, у системному каталозі pg_roles.

Якщо роль не існує, створюється нова роль із логіном і паролем користувача. Виконується запит на створення ролі. Далі виконується серія команд для надання прав доступу

8. Модуль AuthController.cs забезпечує обробку HTTP-запитів для автентифікації та реєстрації користувачів через REST API. Повний код наведений у Додатку Б.

8.1 Алгоритм автентифікації користувача реалізовано в методі Login. Метод приймає об’єкт типу User із даними логіна та пароля, отриманими з HTTP запиту. Викликається метод GetUserByLogin сервісу UserService для пошуку користувача за логіном. Якщо користувача не знайдено або пароль не збігається, повертається HTTP статус 401 із повідомленням про неправильний логін або пароль. У разі успішної автентифікації повертається HTTP статус 200.

8.2 Алгоритм реєстрації користувача реалізовано в методі Register. Метод приймає об’єкт типу User із даними логіна, пароля та, контактної інформації. Спочатку перевіряється правильність пароля: він не повинен бути порожнім і має містити від 6 до 16 символів. У разі неправильно введених даних повертається HTTP статус 400 із повідомленням про помилку. Далі викликається метод GetUserByLogin для перевірки існування користувача з таким логіном. Якщо користувач уже існує, знову повертається HTTP статус 400 із повідомленням про помилку. У разі відсутності користувача викликається метод CreateUser для створення нового запису в базі даних. Після успішного створення користувача викликається метод CreateDbRoleForUser для налаштування ролі бази даних. Якщо створення ролі не вдалося, повертається HTTP статус 500 із відповідним повідомленням. У разі успіху повертається HTTP статус 200 із об’єктом, що містить повідомлення про успішну реєстрацію та ідентифікатор нового користувача.

Реалізовані модулі являють собою основні функції для програмного забезпечення для ведення домашньої бухгалтерії з підтримкою синхронізації між

пристроями. Система підтримує усі важливі функції визначені раніше від автентифікації до створення звітів.

3.3 Розробка бази даних

3.3.1 Опис мови SQL та принципів роботи з базою даних

Мова SQL, або структурована мова запитів – це стандартна мова програмування для роботи з реляційними базами даних. Вона дозволяє користувачам легко управляти, запитувати, оновлювати та аналізувати дані.

SQL поділяється на кілька категорій команд, кожна з яких виконує специфічну функцію. Команди визначення даних: CREATE, ALTER, DROP, використовуються для створення, модифікації та видалення структур бази даних, наприклад, таблиць чи індексі. Команди маніпуляції даними: SELECT, INSERT, UPDATE і DELETE, призначені для додавання, зміни, видалення та вибірки даних. Команди контролю даних: GRANT та REVOKE, регулюють права доступу до даних, забезпечуючи безпеку.

Однією з головних особливостей SQL декларативність: користувач визначає, які дані потрібно отримати чи змінити, а не як саме це має бути зроблено, що дозволяє системі управління базами даних оптимізувати виконання запитів за допомогою індексів. SQL дозволяє об'єднувати таблиці завдяки команди JOIN , що допомагає зв'язувати дані з різних джерел. Агрегатні функції, такі як SUM, AVG, COUNT, полегшують аналіз даних при використані в SQL запиті.

3.3.2 Створення таблиць

Створення таблиці в реляційній базі даних за допомогою мови SQL є основним процесом, який дозволяє визначити структуру для зберігання даних, забезпечивши їхню організацію, цілісність і доступність для подальшої роботи. Цей процес здійснюється вручну за допомогою SQL команди CREATE TABLE. У запиті потрібно вказати унікальну назву таблиці і у дужках перерахувати стовпці таблиці. Кожен стовпець має мати своє ім’я і тип даних.

Таблиця Users створена для зберігання даних користувачів у системі ведення домашньої бухгалтерії.

userID - унікальний ідентифікатор користувача. Тип serial автоматично генерує послідовні значення для нових записів. Використовується як первинний ключ для забезпечення швидкого доступу та унікальності.

login - логін користувача. Використовується для зберігання текстових рядків змінної довжини. Є обов’язковим і має обмеження унікальності, щоб уникнути дублювання логінів. Тип varchar.

password - пароль користувача. Призначений для зберігання пароля. Обов’язкове поле для забезпечення автентифікації. Тип varchar.

contactInfo - контактна інформація користувача. Тип text, що дозволяє зберігати email довільної довжини. Поле необов’язкове, може залишатися порожнім.

SQL запит для створення таблиці Users:

```
CREATE TABLE "User" (
   userID SERIAL PRIMARY KEY,
   login VARCHAR(100) NOT NULL UNIQUE,
   password VARCHAR(100) NOT NULL,
   contactInfo TEXT
);
```

Таблиця Transaction_type призначена для класифікації типів транзакцій у системі ведення домашньої бухгалтерії, що дозволяє організувати фінансові операції за категоріями, такими як Доходи чи Витрати.

transactionTypeID - унікальний ідентифікатор типу транзакції. Використовується як первинний ключ для забезпечення швидкого доступу та унікальності. Тип serial.

typeName - назва типу транзакції. Є обов’язковим полем і має обмеження унікальності, щоб уникнути дублювання назв, що забезпечує чітку ідентифікацію категорій. Тип varchar.

SQL запит для створення таблиці Transaction_type:

```
CREATE TABLE "Transaction_type" (
    transactionTypeID SERIAL PRIMARY KEY,
    typeName VARCHAR(50) NOT NULL UNIQUE
);
```

Таблиця Expense_category призначена для класифікації категорій витрат у системі ведення домашньої бухгалтерії, що дозволяє групувати транзакції за певними категоріями, та пов’язувати їх із типами транзакцій.

categoryID - унікальний ідентифікатор категорії витрат. Автоматично генерує послідовні значення для нових записів. Використовується як первинний ключ для забезпечення унікальності та швидкого доступу до записів. Тип: serial.

categoryName - назва категорії витрат. Є обов’язковим, що гарантує наявність назви для кожної категорії. Тип varchar.

description - опис категорії витрат. Дозволяє зберігати довільний текстовий опис без обмеження довжини. Поле є необов’язковим, що дає можливість залишити його порожнім, якщо опис не потрібен. Тип: TEXT.

transactionTypeID – ідентифікатор типу транзакції. Є зовнішнім ключем, що посилається на поле transactionTypeID таблиці Transaction_type. Вказує, до якого типу транзакції належить категорія. Є обов’язковим, що забезпечує зв’язок кожної категорії з відповідним типом транзакції. Тип: INT.

SQL запит для створення таблиці Expense_category:

```
CREATE TABLE "Expense_category" (
    categoryID SERIAL PRIMARY KEY,
    categoryName VARCHAR(100) NOT NULL,
    description TEXT,
    transactionTypeID INT NOT NULL REFERENCES
"Transaction_type"(transactionTypeID)
);
```

Таблиця Transaction призначена для зберігання інформації про транзакції в системі, що дозволяє відстежувати доходи, витрати та їх деталі для кожного користувача.

userID - ідентифікатор користувача, який здійснив транзакцію. Є зовнішнім ключем, що посилається на поле userID таблиці Users. Налаштування ON DELETE CASCADE забезпечує автоматичне видалення всіх транзакцій користувача при видаленні його запису з таблиці Users. Тип int.

transactionID - унікальний ідентифікатор транзакції. Автоматично генерує послідовні значення для нових записів. Разом із userID формує складений первинний ключ, що забезпечує унікальність транзакцій для кожного користувача. Тип serial.

transactionTypeID - ідентифікатор типу транзакції. Є зовнішнім ключем, що посилається на поле typeID таблиці Transaction_type. Вказує, до якого типу належить транзакція. Тип int.

amount – сума транзакції. Дозволяє зберігати числове значення до 12 цифр із двома знаками після коми. Є обов’язковим, що гарантує наявність суми для кожної транзакції. Тип numeric.

categoryID – ідентифікатор категорії витрат. Є зовнішнім ключем, що посилається на поле categoryID таблиці Expense_category. Вказує категорію транзакції. Поле є необов’язковим, що дозволяє створювати транзакції без прив’язки до категорії. Тип int.

date - дата здійснення транзакції. Є обов’язковим, що забезпечує наявність дати для кожної транзакції. Тип date.

description - опис транзакції. Дозволяє зберігати довільний текстовий опис без обмеження довжини. Поле є необов’язковим, що дає можливість залишити його порожнім, якщо опис не потрібен. Тип text.

SQL запит для створення таблиці Transaction:

```
CREATE TABLE "Transaction" (
    userID INT REFERENCES "User"(userID) ON DELETE CASCADE,
    transactionID SERIAL,
    transactionTypeID INT REFERENCES "Transaction_type"(typeID),
    amount NUMERIC(12,2) NOT NULL,
    categoryID INT REFERENCES "Expense_category"(categoryID),
    date DATE NOT NULL,
    description TEXT,
    PRIMARY KEY (userID, transactionID)
);
```

Таблиця MonthlyBudget призначена для зберігання інформації про місячні бюджети користувачів у системі ведення домашньої бухгалтерії, що дозволяє планувати та відстежувати фінансові ліміти для кожного користувача на певний місяць.

monthlyBudgetID - унікальний ідентифікатор місячного бюджету. Автоматично генерує послідовні значення для нових записів. Використовується як первинний ключ для забезпечення унікальності та швидкого доступу до записів. Тип serial.

userID - ідентифікатор користувача, якому належить бюджет. Є зовнішнім ключем, що посилається на поле userID таблиці "User". Налаштування ON DELETE CASCADE забезпечує автоматичне видалення всіх бюджетів користувача при видаленні його запису з таблиці User. Тип int.

year - рік, для якого встановлено бюджет. Є обов’язковим, що гарантує наявність року для кожного запису. Тип int.

month - місяць, для якого встановлено бюджет. Є обов’язковим, що забезпечує наявність місяця для кожного запису. Тип int.

amount - сума місячного бюджету. Дозволяє зберігати числове значення до 10 цифр із двома знаками після коми. Є обов’язковим, що гарантує наявність суми для кожного бюджету. Тип numeric.

SQL запит для створення таблиці MonthlyBudget:

```
CREATE TABLE "MonthlyBudget" (
    "monthlyBudgetID" SERIAL PRIMARY KEY,
    "userID" INTEGER NOT NULL REFERENCES "User"("userID") ON DELETE CASCADE,
    "year" INTEGER NOT NULL,
    "month" INTEGER NOT NULL,
    "amount" NUMERIC(10,2) NOT NULL,
);
```

3.3.2 Функції, тригери , залежності та обмеження в PostgreSQL

Для забезпечення цілісності даних, автоматизації повторюваних операцій та реалізації бізнес-логіки на рівні бази даних використовуються елементи PostgreSQL: функції, тригери, послідовності та обмеження.

Ці інструменти дозволяють суттєво знизити навантаження на клієнтську частину застосунку, підвищити надійність виконання операцій і забезпечити обробку даних незалежно від джерела доступу.

У мові SQL функція позначає спеціалізований програмний об’єкт, призначений для виконання певних операцій над даними з поверненням результату. Функції є важливою частиною реляційних баз даних, адже вони допомагають виконувати обчислення, обробляти дані або змінювати значення в зручний і повторно використовуваний спосіб. Вони полегшують складні запити та забезпеченню модульність при роботі з базами даних. Функції - інструмент, який приймає вхідні дані, обробляє їх за заданою логікою та повертає результат у вигляді одного значення або набору даних.

Функції в SQL поділяються на три основні типи залежно від їхнього призначення. Скалярні функції приймають один або кілька вхідних параметрів і повертають єдине значення. До скалярних функцій належать функції для роботи з рядками, функції для роботи з датами або математичні функції.

Агрегатні функції працюють з набором рядків і повертають одне значення, що підсумовує дані. Вони часто використовуються разом з оператором групування для аналітичних розрахунків

Функції, які повертають результат у вигляді таблиці, що може бути використана в SQL-запросах називаються табличними. Табличні функції бувають двох підвидів: вбудовані і багатовиразні. Вбудовані табличні функції повертають результат як єдиний запит, тоді як багатовиразні функції містять складнішу логіку, яка реалізується через кілька SQL-виразів. Такі функції

корисні для створення складних наборів даних, які можна повторно використовувати в різних запитах.

Використання функцій спрощує написання складних запитів, дозволяючи замінити великі вирази меншими викликами функцій. Також вони сприяють повторному використанню в коді, оскільки одна й та сама функція може застосовуватися в різних запитах.

Для системи було створено функцію add_or_create_monthly_budget для створення запису про місячний бюджет користувача в таблиці MonthlyBudget. Вона приймає чотири вхідні параметри: userID, year, month, amount, після чого виконує умовну операцію вставки даних.

Функція спочатку перевіряє, чи існує запис у таблиці MonthlyBudget, який відповідає комбінації вхідних параметрів: userID, year і month. Ця перевірка здійснюється за допомогою запиту з умовою SELECT 1, який повертає рядок, якщо відповідний запис існує. Якщо такий запис відсутній, функція виконує операцію вставки нового рядка в таблицю MonthlyBudget, використовуючи значення вхідних параметрів для заповнення стовпців. Якщо запис із заданими параметрами вже існує, функція не виконує жодних дій, завершуючи виконання.

SQL запит для створення функції add_or_create_monthly_budget:

```
CREATE FUNCTION add_or_create_monthly_budget(p_userid integer, p_year integer, p_month integer, p_amount numeric) RETURNS void
  AS $$
BEGIN
  IF NOT EXISTS (
    SELECT 1 FROM "MonthlyBudget"
    WHERE "userID" = p_userID AND "year" = p_year AND "month" = p_month
  ) THEN
```

```
    INSERT INTO "MonthlyBudget" ("userID", "year", "month", "amount")
    VALUES (p_userID, p_year, p_month, p_amount);
  END IF;
END;
$$;
```

Інша створена тригер-функція називається assign_transaction_id і використовується для автоматично призначення transactionID транзакції для нового запису в таблиці Transaction. Вона активується під час вставки даних і змінює значення стовпця transactionID у новому рядку перед його збереженням у таблиці. Функція генераціє ідентифікатор на основі максимального значення transactionID для конкретного користувача, забезпечуючи унікальність ідентифікаторів у межах кожного користувача.

SQL запит для створення функції assign_transaction_id:

```
CREATE FUNCTION assign_transaction_id() RETURNS trigger
  AS $$
DECLARE
  next_id INT;
BEGIN
  SELECT COALESCE(MAX("transactionID"), 0) + 1
  INTO next_id
  FROM "Transaction"
  WHERE "userID" = NEW."userID";
  NEW."transactionID" := next_id;
  RETURN NEW;
```

```
END;
$$;
```

Тригер - це програмний об’єкт бази даних, який автоматично виконується або активується у відповідь на події, пов’язані з операціями над даними в таблиц. Вони є важливою частиною систем управління базами даних. Тригери виконуються без явного виклику користувачем, що відрізняє їх від функцій чи процедур, які викликаються явно.

Тригери які реагують на операції маніпуляції даними називаються DML тригерами. Вони активуються при виконанні операцій вставки, оновлення або видалення даних у таблиці. Такі тригери дозволяють налаштувати поведінку операції, виконуючи замість неї інші інструкції.

Тригери, що реагують на зміни структури бази даних, називаються DDL тригерами, оскільки вони пов’язані з мовою визначення даних. Вони активуються при виконанні операцій, таких як створення, зміна або видалення об’єктів бази даних, наприклад, таблиць, індексів чи схем. Такі тригери зазвичай використовуються для аудиту змін у структурі бази даних або для обмеження несанкціонованих модифікацій. Вони можуть застосовуватися на рівні окремої бази даних або всієї системи, залежно від реалізації в конкретній системі керування базами даних.

Інший вид тригеру DDL реагує на зміни в структурі бази даних.Вони спрацьовують, коли ви створюєте, змінюєте або видаляєте об'єкти, як-от таблиці, індекси чи схеми. Вони активуються при виконанні операцій, таких як створення, зміна або видалення об’єктів бази даних, наприклад, таблиць, індексів чи схем. Їх можна застосовувати на рівні окремої бази даних або всієї системи.

Шоб доповнити функцію public.assign_transaction_id створено тригер trg_assign_transaction_id, що автоматично викликає її перед виконанням операції вставки у таблицю Transaction. Він належить до категорії DML тригерів. Виконується для кожного нового рядка, що вставляється в таблицю. Тригер

забезпечує автоматичне призначення унікального ідентифікатора транзакції шляхом виклику функції.

SQL запит для створення тригеру trg_assign_transaction_id:

CREATE TRIGGER trg_assign_transaction_id BEFORE INSERT ON "Transaction" FOR EACH ROW EXECUTE FUNCTION assign_transaction_id();

Послідовність останній елемент який використано для автоматизації. У мові SQL є спеціальним об’єктом реляційної бази даних, призначеним для генерації унікальних числових значень у заданому порядку. Послідовності використовуються для створення унікальних ідентифікаторів забезпечуючи автоматизоване та контрольоване створення послідовних чисел. Послідовності не прив’язані до конкретної таблиці, і їх можуть використовувати різні таблиці чи запити. Цей об’єкт забезпечує гнучкість, масштабованість і надійність у генерації унікальних значень.

Однією з ключових особливостей послідовностей є їхня незалежність від транзакцій. Завдяки цьому значення завжди унікальні, але це може призводити до пропусків у послідовності, якщо транзакція, що отримала значення, не завершується успішно.

При використанні типу поля serial у базі даних автоматично створюються послідовності які збільшують значення поля на 1 якщо число не вказано коли додається новий запис.

Початкове значення встановлено на 1 за допомогою START WITH 1. INCREMENT BY 1, вказує, що кожне наступне значення збільшується на 1. Виклик функції NEXTVAL звертається до бази даних для отримання нового значення, що забезпечує точність, але може знижувати продуктивність у високонавантажених системах.

Приклад SQL запит для створення послідовності Budget_budgetID_seq:

CREATE SEQUENCE "Budget_budgetID_seq"

```
AS integer
START WITH 1
INCREMENT BY 1
NO MINVALUE
NO MAXVALUE
CACHE 1;
```

Обмеження – це елемент, що допомагає підтримувати правильність і цілісність даних у таблицях.. Обмеження є частиною структури реляційної бази даних і застосовуються до стовпців або таблиць для контролю значень. Вони гарантують дотримання заданих умов запобігаючи введенню некоректних даних у поля. Обмеження є важливим механізмом для забезпечення надійності даних і уникненню помилок. Обмеження в базі даних працюють на рівні схеми й перевіряються автоматично під час виконання операцій. Обмеження можуть стосуватися окремих стовпців або таблиць.

Обмеження MonthlyBudget_month_check створене вручну для створення унікальних ідентифікаторів для стовпця monthlyBudgetID у таблиці MonthlyBudget. Обмеження гарантує коректність значень стовпця month, обмежуючи їх діапазоном чисел від 1 до 12 що відповідають місяцям.

Приклад SQL запит для створення обмеження MonthlyBudget_month_check

```
CONSTRAINT "MonthlyBudget_month_check" CHECK (((month >= 1) AND
(month <= 12)))
```

3.4 Розгортання системи

Після завершення розробки важливо правильно розгорнути систему для забезпечення зручного доступу, стабільної роботи . При використанні товстої клієнто-серверної моделі найкраще підходить гібридне розгортання. Цей підхід

дозволяє комбінувати локальну обробку даних і хмарні сервіси для налаштування інформаційних систем.

Локальна частина знаходиться на комп’ютері користувача, у вигляді програмного додатку розробленого з використанням WPF. Через нього користувач взаємодіє з серверною частиною при відправки запитів до бази даних. База даних і серверна логіка працюють через хмарний сервіс. Сервер зберігає дані централізовано та синхронізує їх між різними пристроями. Процес розгортання бази даних PostgreSQL починається з вибору сервісу.

Серед популярних хмарних сервісів для баз даних можна виділити Amazon Web Services RDS, Microsoft Azure SQL Database та Render. Кожен з них має свої особливості.

Amazon Web Services RDS сервіс для роботи з реляційними базами даних. Він пропонує хорошу надійність завдяки тому, що бази даних розміщені в кількох зонах доступності в одному регіоні. Якщо станеться проблема з сервером або з електрикою, система зможе працювати без збоїв. Важливо для великих проєктів, де відсутність доступу до даних може призвести до великих втрат. RDS підтримує різні системи управління базами даних, особливо PostgreSQL і працює разом з іншими сервісами компанії Amazon які, надають можливість моніторити продуктивність, відстежувати оброблення запитів, зберігати резервні копії. Інтеграція цього сервісу потребує значних фінансових витрат на його підтримку, що робить його використання економічно неефективним для малих проектів.

Microsoft Azure SQL Database інший варіант для розгортання реляційних баз даних, який надає можливості, щоб настроїти ресурси під конкретні потреби. Модель гнучкого сервера дозволяє динамічно налаштовувати віртуальні ядра і обсяг пам'яті. Така модель створює баланс між продуктивністю та витратами, залежно від навантаження, яке виникає під час роботи з запитами. Має погану підтримку PostgreSQL, як в інших аналогах, що може обмежувати його функціональність у контексті реалізації бази дани, яка потребує специфічних

можливостей цієї СУБД, таких як розширені можливості роботи з транзакціями чи підтримка складних індексів. Azure SQL Database легко інтегрується з іншими сервісами компанії Microsoft для аналізу продуктивності та для серверної логіки, що полегшує створення API для синхронізації даних. Використання даного сервісу також потребує значних фінансових витрат на його підтримку, що робить його менш вигідним у порівнянні з іншими варіантами для невеликих систем.

Render зосереджується на спрощенні процесу розгортання та оптимізації ресурсів для малих і середніх проєктів, що робить його гарним вибором для бази даних. Платформа підтримує PostgreSQL в управлінні, даючи можливість користувачам створювати базу даних через простий веб-інтерфейс. Render пропонує кращу цінову модель, ніж його аналоги. Серед основних функцій платформи - автоматичне резервне копіювання, можливість відновлення, шифрування даних і вбудований моніторинг продуктивності. Render має деякі обмеження в масштабуванні для великих проектів.

Таблиця 4.2

Критерій	Amazon Web Services RDS	Microsoft Azure SQL Database	Render
Тип сервісу	Керований сервіс для реляційних баз даних	Керований сервіс для реляційних баз даних	Платформа для розгортання та управління БД
Підтримка PostgreSQL	Підтримує	Погана підтримка	Підтримує
Надійність	Мультизональна доступність, безперебійна робота при збоях	Гнучке масштабування, але обмеження у функціональності PostgreSQL	Автоматичне резервне копіювання, відновлення, шифрування
Масштабованість	Розміщення в кількох зонах доступності	Динамічне налаштування ресурсів	Обмежена масштабованість для великих проектів
Інтеграція	Інтеграція з іншими сервісами AWS для моніторингу, резервного копіювання	Легка інтеграція з Microsoft сервісами, API для синхронізації	Простий веб-інтерфейс для управління
Цінова модель	Високі фінансові витрати, неефективний для малих проектів	Значні витрати, менш вигідний для невеликих систем	Краща цінова модель для малих і середніх проектів

Платформа Render має вигідну цінову модель, що зменшує витрати на налаштування і обслуговування, підтримує PostgreSQL, має авто-резервне копіювання, шифрування даних і вбудована перевірка продуктивності роблять зберігання інформації безпечним і надійним що робить його найкращим вибором для впровадження в систему програмного забезпечення для ведення домашньої бухгалтерії з підтримкою синхронізації між пристроями.

Розгортання частини системи на платформі Render включає кілька етапів. Спочатку потрібно зареєструватися на офіційному сайті Render, де користувач заповнює форму з даними. Після цього можна працювати з панеллю управління.

У панелі керування створюється проект в якому буде знаходитись база даних. Для створення нової бази даних PostgreSQL на платформі Render заповнюється вся потрібна інформація і обирається тип підписки. Після чого в розділі General (рис 3.17) вказуються загальні налаштування обрані раніше, а саме унікальна назва бази даних, дата створення, статус роботи, версія PostgreSQL і розмір сховища даних. Через кнопку Add Read Replica можна створити додаткові копії для покращення швидкості читання. Завдяки Datadog API Key є можливість додати API ключ для логування.

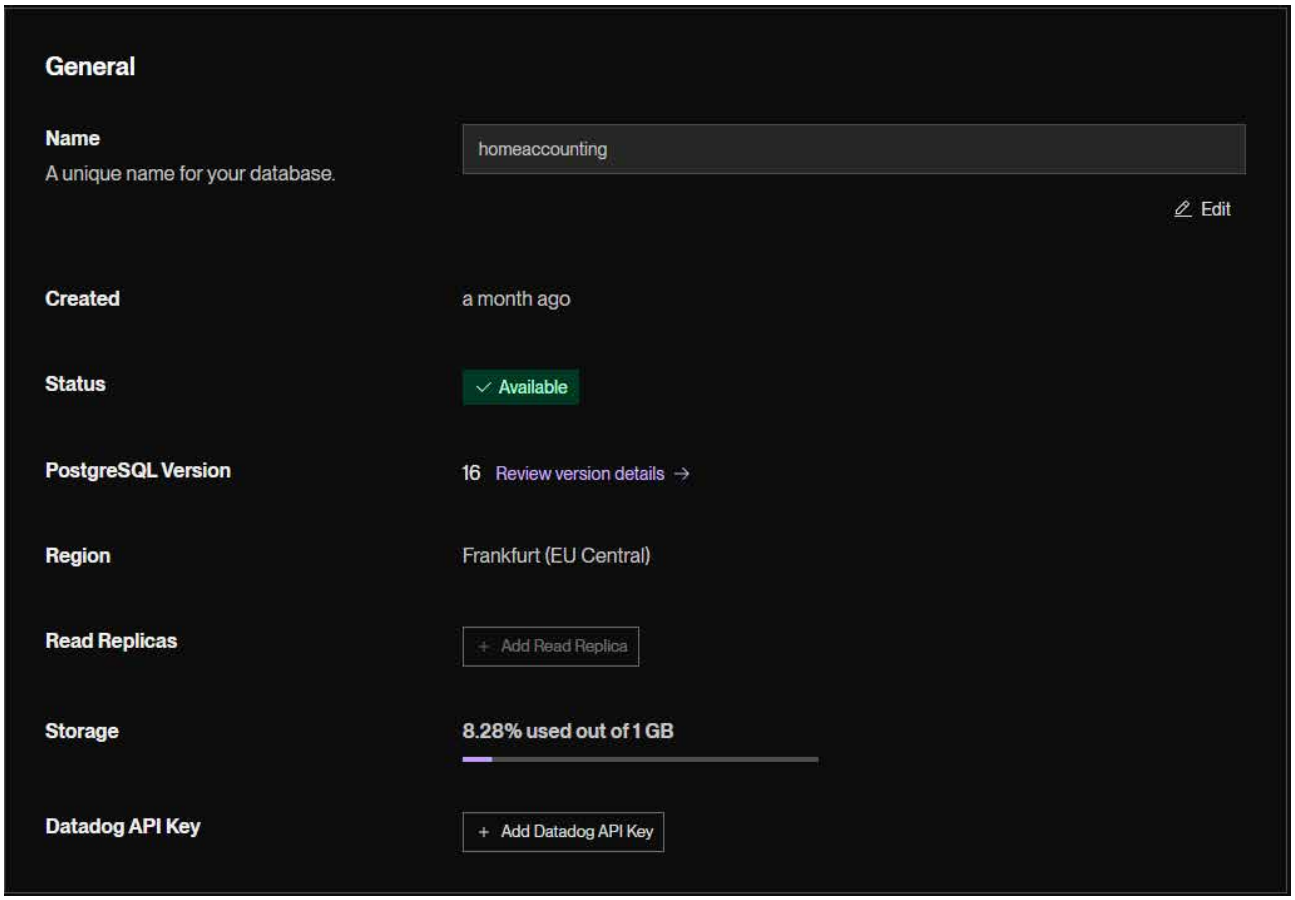


Рис. 3.17 Загальна інформація про базу даних

На наступному етапі потрібно підготувати веб-сервіс для роботою з API до запуску. Це включає перевірку працездатності коду створеного на ASP.NET Core.

Проект повинен мати два файли конфігурації для розміщення проекту ASP.NET на Render. Файл конфігурації задає параметри і налаштування для роботи програми. Він дозволяє легко управляти різними моментами: підключення до бази даних, налаштування середовища чи логування. Для веб-додатків, зокрема для програм домашньої бухгалтерії, цей файл є місцем, де зберігаються всі налаштування. Це дає змогу змінювати додаток під різні умови не змінюючи код. Формат файлу конфігурацій JSON з назвою appsettings.json. Цей файл служить основним джерелом налаштувань в ASP.NET Core, де він працює з системою конфігурації. Завдяки цьому легко читати параметри за допомогою вбудованого методу IConfiguration.Файл конфігурації має такі секцію ConnectionStrings, що необхідна для додатків, що працюють з базами

даних. В ньому знаходиться рядок DefaultConnection без вказаного строки підключення для забезпечення безпеки. Ці дані будуть зберігатися як змінні середовища в Render.

Docker-файл починається з визначення базового образу для процесу створення образу. Цей образ містить .NET SDK 8.0, потрібний для компіляції та публікації додатка. Наступна команда встановлює робочу директорію у контейнері, де відбуватимуться всі подальші дії. Інструкція copy копіює файл рішення з репозиторію в контейнер, і переносить файл проекту у вказану директорію. Це підготовує структуру для відновлення залежностей. Команда run виконує відновлення всіх NuGet-пакетів, зазначених у файлі, завантажуючи їх із репозиторіїв. Потім інструкція copy копіює решту файлів проекту, включаючи код і конфігурації, у контейнер. Наступна команда змінює робочу директорію, де виконується команда run, компілюючи додаток та публікуючи зібрані файли в папку.

Другий етап починається з іншого образу для виконання додатка. Цей містить лише .NET Runtime без компіляторів, що допомагає зменшити розмір кінцевого образу. Команда workdir створює робочу директорію в новому контейнері, а copy переносить зібрані файли з попереднього етапу у поточну директорію. EntryPoint задає команду для запуску контейнера, вказуючи, що при запуску виконається команда dotnet, яка запускає ASP.NET Core додаток.

Код конфігураційного файлу Docker:

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build

WORKDIR /app

COPY *.sln ./

COPY AuthAPI/*.csproj ./AuthAPI/

RUN dotnet restore

COPY AuthAPI/. ./AuthAPI/
```

```
WORKDIR /app/AuthAPI

RUN dotnet publish -c Release -o out

FROM mcr.microsoft.com/dotnet/aspnet:8.0

WORKDIR /app

COPY --from=build /app/AuthAPI/out ./

ENTRYPOINT ["dotnet", "AuthAPI.dll"]
```

Після завершення етапу тестування працездатності програмного забезпечення в локальному, перевірки повноти структури проекту та наявності всіх необхідних файлів, програмний продукт завантажується до віддаленого репозиторію на платформі GitHub. Завдяки цьому код зберігається в одному місці та готовий до розгортання на хмарному середовищі Render.

Щоб завантажити проект до Github потрібно виконати команди всередині проекту. Перша команда echo додає текст до файлу, який використовується для опису проєкту в репозиторії. Після команда git init ініціалізує локальний репозиторій у поточній директорії, створюючи приховану папку .git, яка міститиме всю історію змін. Це перетворює робочу папку на простір, де система відстежує файли та записує їхні оновлення для подальшої роботи з версіями коду. Git add . додає всі файли та папки з поточної директорії до індексу репозиторію. Це включає вихідний код, конфігураційні файли, документацію та інші ресурси проєкту, які мають бути в історії змін, після чого система готує весь вміст для наступного етапу. Git commit –m створює початковий коміт із повідомленням "Initial commit", фіксуючи поточний стан проєкту в історії репозиторію. В ньому зберігаються усі додані файли та їхній вміст, дозволяючи відслідковуати зміни та повертатися до цього стану коли потрібно. Кожне повідомлення коротко описує суть зробленої роботи. Наступна команда git remote add origin додає віддалений репозиторій із назвою origin, прив’язуючи локальний репозиторій до віддаленого сховища на GitHub за вказаною адресою.

Команда встановлює зв’язок між локальним і віддаленим сховищами для подальшої синхронізації. Остання команда відправляє вміст репозиторію до віддаленого сховища на GitHub.

Приклад команди для завантаження проекту до Github

```
echo "# НазваПроєкту" >> README.md
git init
git add .
git commit -m "Initial commit"
git remote add origin https://github.com/user/projectname.git
git push -u origin main
```

Після завершення процесу завантаження проєкту, програмний код стає доступним у репозиторії (рис. 3.18) на платформі GitHub. Надалі він використовується як джерело для автоматизованого розгортання на платформі Render. Під час створення веб-додатку потрібно вказати посилання до репозиторію, після чого система отримує необхідні файли для компіляції.

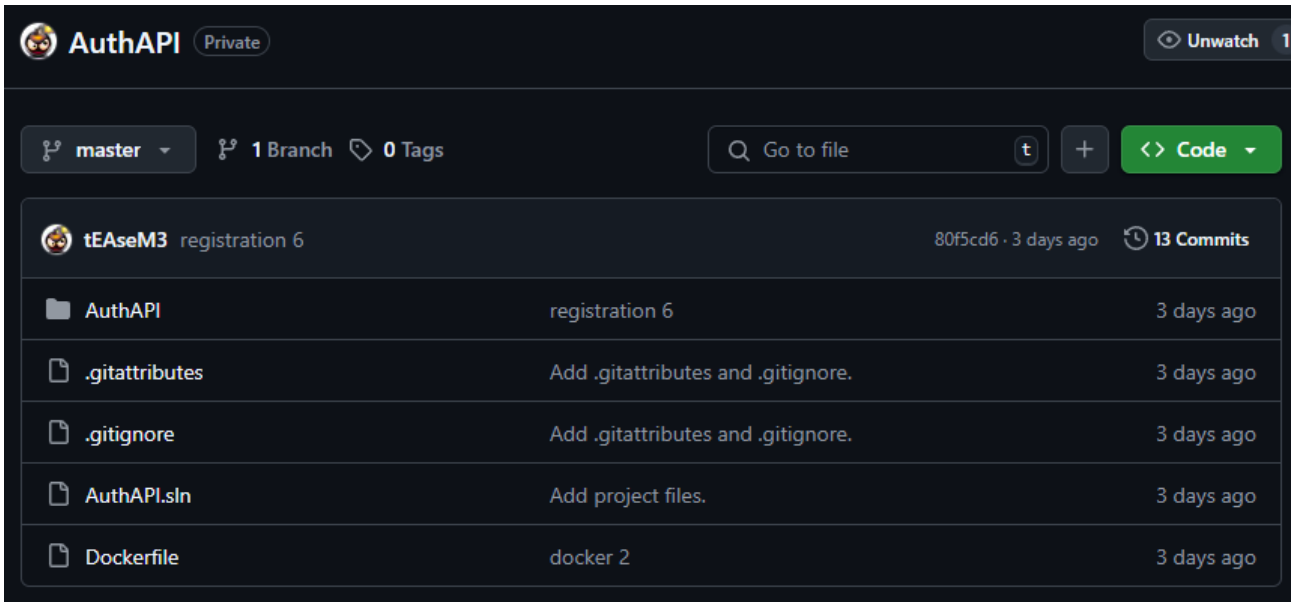


Рис. 3.18 Створений репозиторій у GitHub

4 РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ

4.1 Вимоги до апаратного та програмного забезпечення для користувача

Для забезпечення стабільної роботи системи необхідно чітко визначити вимоги до апаратного та програмного забезпечення. Визначення мінімальних та рекомендованих характеристик дозволяє забезпечити безперервне функціонування програмного додатку в умовах використування користувачем.

Апаратне забезпечення для роботи з WPF додатоком потребує:

Процесор: Багатоядерний процесор 2 ГГц.

Оперативна пам’ять: 8 ГБ.

Вільне місце на диску: 2 ГБ.

Відеокарта: Сумісний із DirectX 11 або новішим для підтримки рендерингу у WPF-інтерфейсі.

Монітор: Роздільна здатність екрана не нижче 1024x1440 пікселів для відображення графічного інтерфейсу.

Периферійні пристрої: Клавіатура та миша для взаємодії з інтерфейсом.

Програмне забезпечення для роботи з WPF додатоком потребує:

Операційна система: Microsoft Windows 10.

Фреймворк: .NET 8.0 для підтримки WPF-додатка.

Бібліотеки:

OxyPlot для відображення кругових діаграм.

QuestPDF для генерації PDF-звітів.

CsvHelper для створення CSV-файлів.

Npgsql для роботи з базою даних

4.2 Тестування системи

Після завершення етапів розробки інтерфейсу, реалізації логіки та інтеграції всіх компонентів програмного забезпечення, потрібно виконати останній етап розробки програмного забезпечення, а саме тестування. Тестування допомагає знайти помилки, перевірити, чи все працює як потрібно, і чи відповідає система вимогам. Проведення тестування дає змогу впевнитися, що програма стабільна, надійна та готова до використання користувачами.

Один із методів перевірки системи є тестові сценарії - плани, які описують кроки, умови і результати, щоб перевірити, як працює програмне забезпечення. Вони допомагають знайти помилки, перевірити, чи програмне забезпечення відповідає вимогам і як воно веде себе в різних ситуаціях. Тестові сценарії створюються на основі вимог, технічних документів і того, як система буде використовуватися.

1. Тестування модулю реєстрації

Мета тестування: Перевірити обробку введених користувачем даних під час створення нового облікового запису.

Таблиця 4.3

№	Назва тесту	Вхідні дані	Очікуваний результат	Фактичний результат	Статус
1	Успішна реєстрація	Логін: userTest Пароль: passwordTest Email: usertest@example.com	Користувача успішно зареєстровано, перенаправлення до головного вікна	Збігається	Пройдено
2	Порожні всі поля	Логін: Пароль: Email:	Кнопка Зареєструватися неактивна або повідомлення Заповніть всі поля	Збігається	Пройдено
3					

	Відсутнє підключення до сервера				
4	Пароль занадто великий	Логін: userTest Пароль: passwordTest123123 Email: usertest@example.com	Повідомлення про перевищення кількості символів	Збігається	Пройдено

Тестування показало, що форма реєстрації справно працює з усіма важливими ситуаціями, які можуть виникнути у користувача. Це стосується перевірки введених даних, а також обробки мережевих помилок і вимог до паролів.

2. Тестування модулю авторизації

Мета тестування: Перевірити правильність обробки даних при вході користувача до системи.

Таблиця 4.4

№	Назва тесту	Вхідні дані	Очікуваний результат	Фактичний результат	Статус
1	Вхід з коректними обліковими даними	Логін: userTest Пароль: passwordTest Email: usertest@example.com	Користувача перенаправлено до головного вікна додатку	Збігається	Пройдено
2	Вхід з некоректним паролем	Логін: userTest Пароль: passwordTest Email: usertest@example.com	Виведено повідомлення Невірний логін або пароль	Збігається	Пройдено
3	Порожні поля форми	Логін: userTest Пароль: passwordTest Email: usertest@example.com	Кнопка входу неактивна або виведено повідомлення Заповніть всі поля	Збігається	Пройдено

За результатами тестів видно, що форма авторизації працює, як треба. Вона справляється зі звичайними і крайніми ситуаціями, дає зворотний зв’язок користувачу.

3. Тестування модулю додавання транзакцій

Мета тестування: Перевірити коректність введення та збереження фінансових операцій у систему.

Таблиця 4.5

№	Назва тесту	Вхідні дані	Очікуваний результат	Фактичний результат	Статус
1	Успішне додавання транзакції	Тип: Витрата Категорія: Продукти Сума: 500 Дата: 22.05 Опис : груші	Транзакція збережена, відображається у списку операцій	Збігається	Пройдено
2	Відсутність вибору категорії	Тип: Доход Категорія: не вибрана Сума: 1000 Дата: поточна дата Опис : груші	Повідомлення: Оберіть категорію, транзакція не додається	Збігається	Пройдено
3	Додавання транзакції з сумою 0	Тип: Витрата Категорія: Продукти Сума: 0 Дата: поточна дата. Опис : груші	Відображається повідомлення Сума має бути більше нуля, транзакція не додається	Збігається	Пройдено
4	Введення негативної суми	Тип: Витрата Категорія: Розваги Сума: -200 Дата:	Відображається повідомлення транзакція не додається	Збігається	Пройдено

Тестування показало, що форма для додавання транзакцій добре обробляє різні варіанти введення, навіть якщо там є помилки. Крім того, вона взаємодіє з користувачем, надаючи правильні повідомлення про помилки.

4. Тестування модулю редагування транзакцій

Мета тестування: перевірка функцій оновлення та видалення у модулі редагування.

Таблиця 4.6

№	Назва тесту	Вхідні дані	Очікуваний результат	Фактичний результат	Статус
1	Успішне редагування транзакції	ID: 1 Тип: Витрата Категорія: Продукти Сума: 900 Дата: 22.05 Опис : сливи	Транзакція оновлена, зміни відображаються у списку операцій	Збігається	Пройдено

2	Спроба редагування з некоректною сумою	ID: 1 Тип: Витрата Категорія: Продукти Сума: -900 Дата: 22.05 Опис : сливи	Відображається повідомлення Сума має бути більше нуля, зміни не збережені	Збігається	Пройдено
3	Спроба оновлення без заповнення обов’язкових полів	ID: -1 Тип: Витрата Категорія: Продукти Сума: -900 Дата: 22.05 Опис : сливи	Повідомлення: зміни не збережені	Збігається	Пройдено
4	Успішне видалення транзакції	ID: -1 Тип: Витрата Категорія: Продукти Сума: -900 Дата: 22.05 Опис : сливи	Транзакція видалена зі списку операцій	Збігається	Пройдено

Проведене тестування модуля редагування транзакцій на відповідність функціоналу встановленим вимогам. Система коректно обробляє зміни даних, не допускає збереження некоректних значень та забезпечує видалення транзакцій.

5. Тестування модулю місячного бюджету

Мета тестування : перевірка відображення, додавання та оновлення даних про місячний бюджет користувача.

Таблиця 4.7

№	Назва тесту	Вхідні дані	Очікуваний результат	Фактичний результат	Статус
1	Завантаження існуючого бюджету	Рік: 2025 Місяць травень Сума 200	У полі відображається сума бюджету	Збігається	Пройдено
2	Завантаження бюджету, якого немає	Рік: 2025 Місяць червень Сума 200	Повідомлення: бюджет не встановлено	Збігається	Пройдено
3	Оновлення існуючого бюджету	Рік: 2025 Місяць червень Сума 900	Повідомлення: оновлення бюджету	Збігається	Пройдено
4	Спроба зберегти бюджет із некоректною	Рік: 2025 Місяць червень Сума -900	Відображається повідомлення: Введіть коректну суму бюджету,	Збігається	Пройдено

	сумою		збереження не відбувається		

Тестування показало, що модуль працює справно: він може завантажувати, додавати і оновлювати місячний бюджет для вибраного року і місяця.

Повне тестування системи показало, що вона працює так, як і було заплановано. Були перевірили основні модулі програми. Результати підтвердили, що система правильно обробляє дані. Створені сценарії функціонального тестування допомогли перевірити основні шляхи виконання програми.

4.3 Використання програмного продукту

Для початку роботи необхідно запустити програмний додаток та увійти до облікового запису (рис. 3.19). Щоб пройти перевірку та отримати доступ до програми, користувач повинен ввести правильні облікові дані – логін і пароль. У разі успішної автентифікації відбувається встановлення з’єднання з сервером авторизації, який перевіряє правильність введених даних користувач заходить у програму.

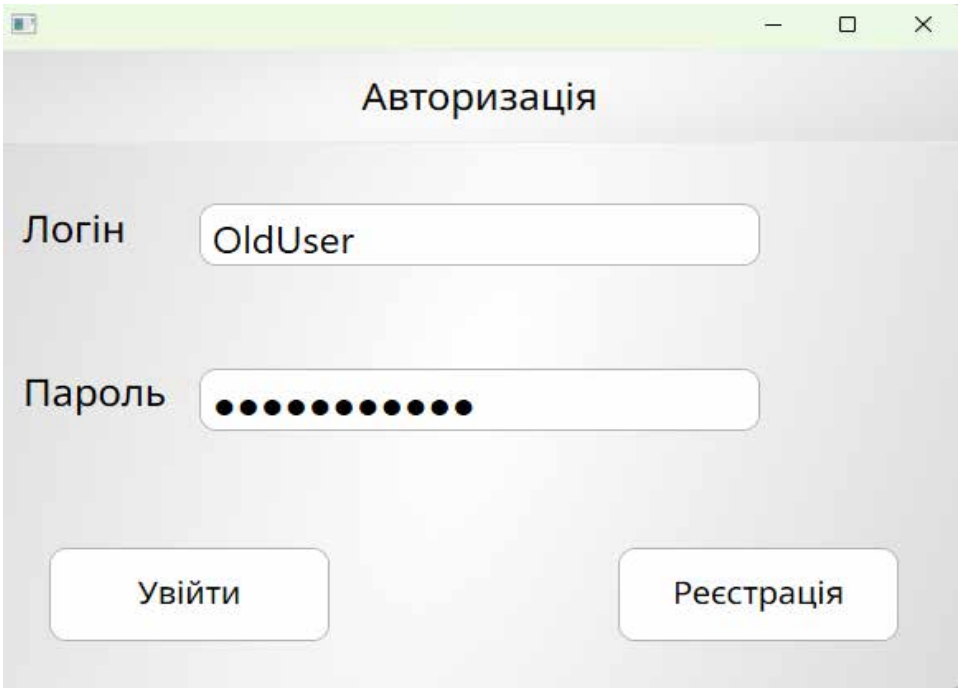


Рис. 4.19 Заповнене вікно авторизації

Якщо користувач новий то потрібно спочатку зареєструвати (рис. 3.19) обліковий запис заповнивши поля логіну, паролю та пошти. Важливо, щоб пароль був від 6 до 16 символів.

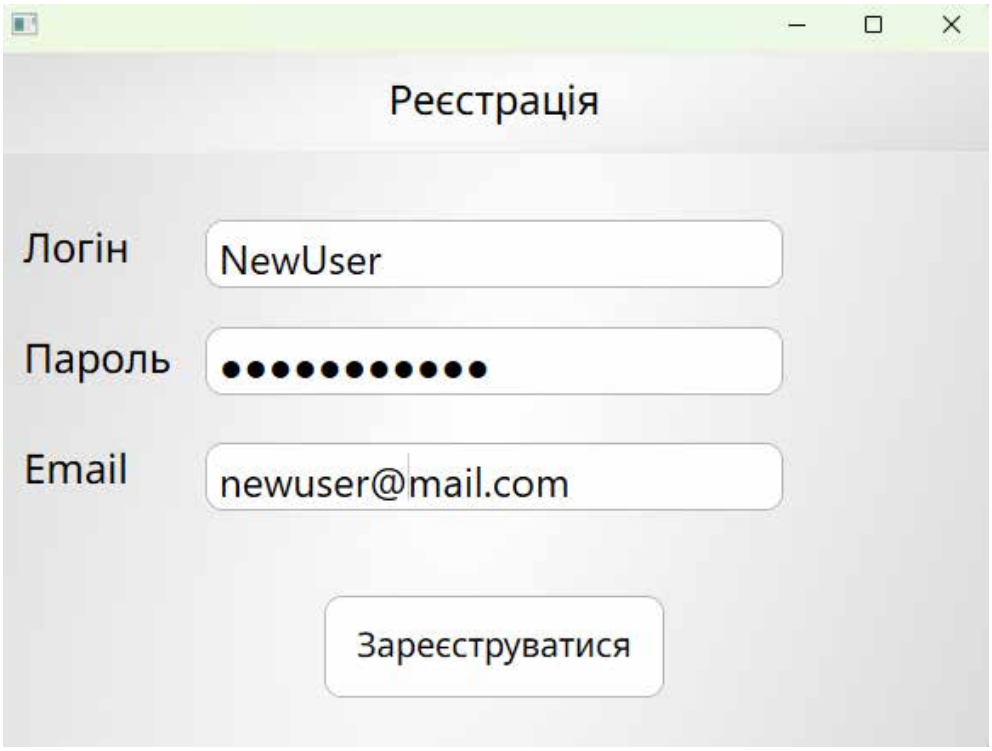


Рис. 4.19 Заповнене вікно реєстрації

Після входу до облікового запису користувача зустрічає головне меню (рис 3.11) . У ньому можна переглянути звітність останніх транзакцій або перейти далі натиснув на додаткове меню праворуч зверху. Якщо в базі даних є записи про витрати, вони відображаються на графіку в головному меню. Також за наявності записів у місячному бюджеті та транзакціях пов’язаних з поповненням відображається статистика на панелі праворуч.

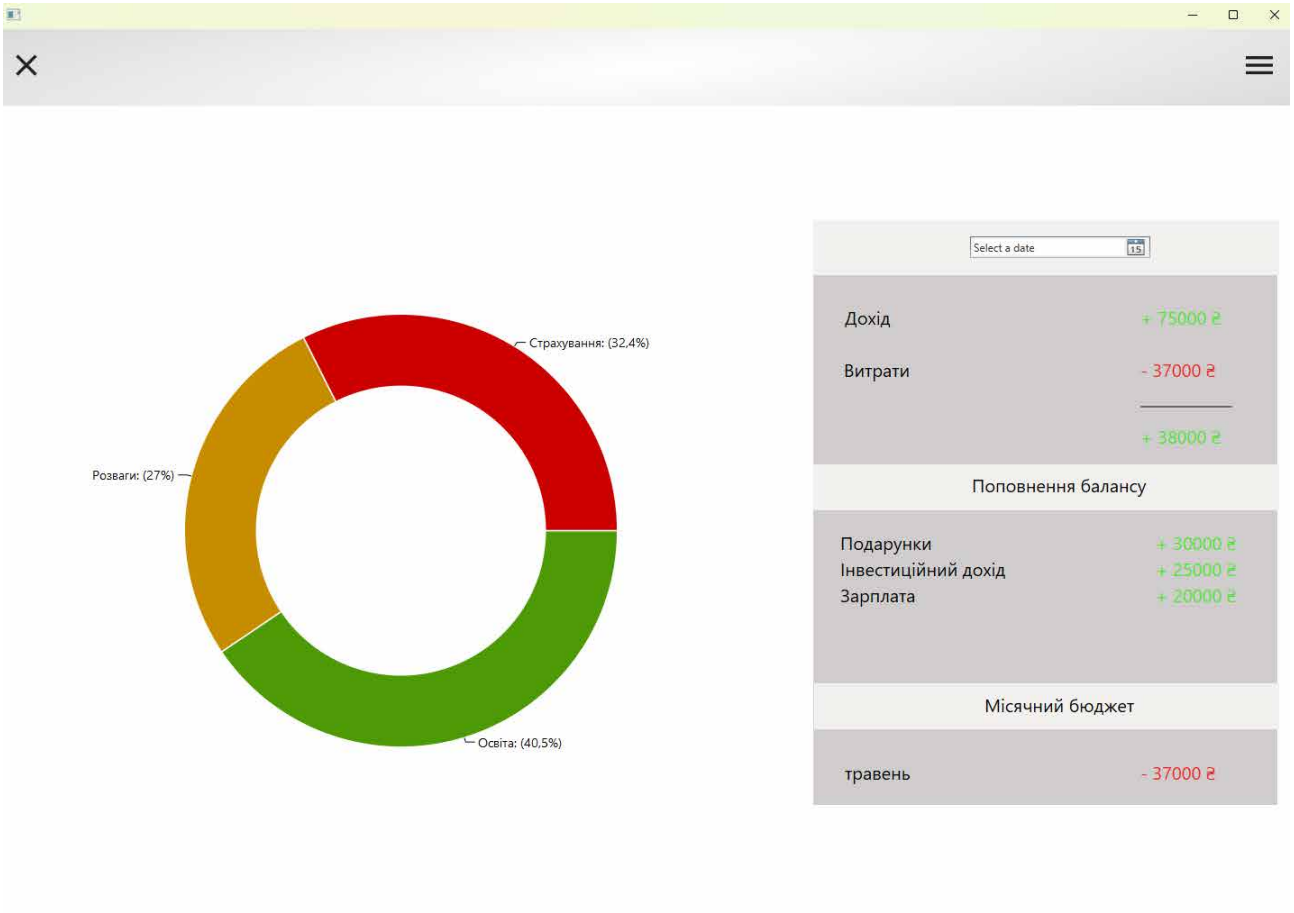


Рис. 4.20 Заповнене головне меню

При натисканні на додаткове меню спливають основні функції (рис. 4.21) системи де користувач може обрати, що робити: переглядати транзакції, додати або редагувати транзакцію, переглянути звіти, керувати бюджетом. Інтерфейс зроблено так, щоб користувач мав швидкий доступ до всіх ключовичможливостей, що робить користування зручнішим і допомагає легше взаємодіяти з системою.

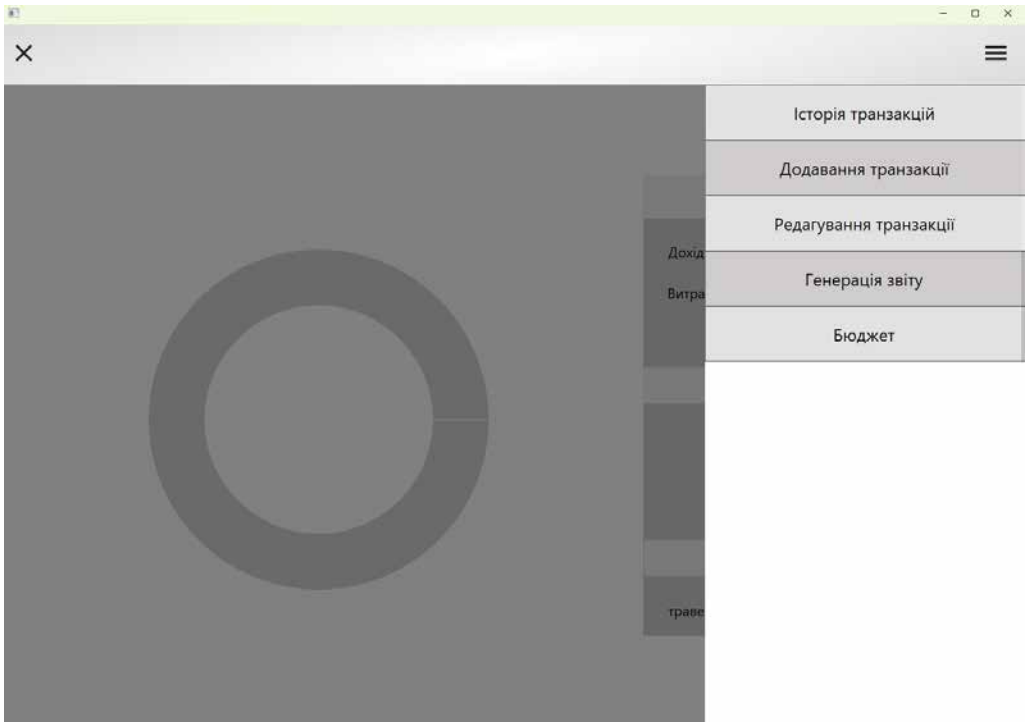


Рис. 4.21 Розгорнуте додаткове меню

При виклику контролера додавання транзакцій (рис 4.22) користувач може обрати тип транзакції, категорію та заповнити полія грошей, дати, опис після чого додати новий запис до бази даних.

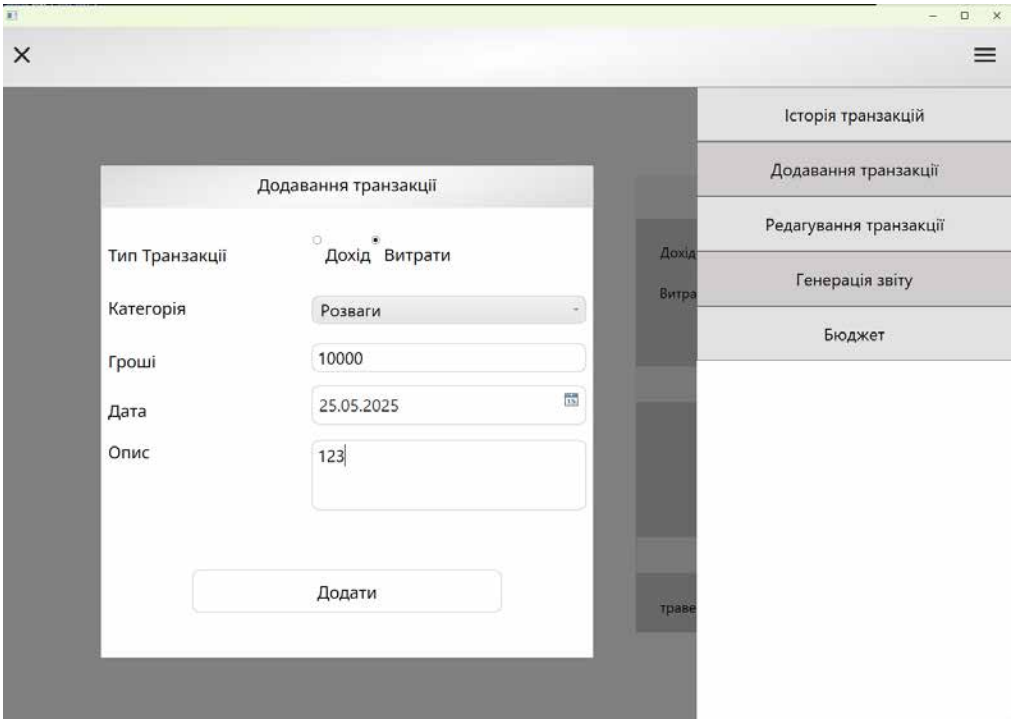


Рис. 4.22 Заповнений контролер додавання транзакцій

При переході до контролеру редагування транзакцій (рис. 4.23) користувач може обрати тип транзакції, категорію та заповнити поля ID, грошей, дати, опис після чого оновити або видалити новий запис до бази даних.

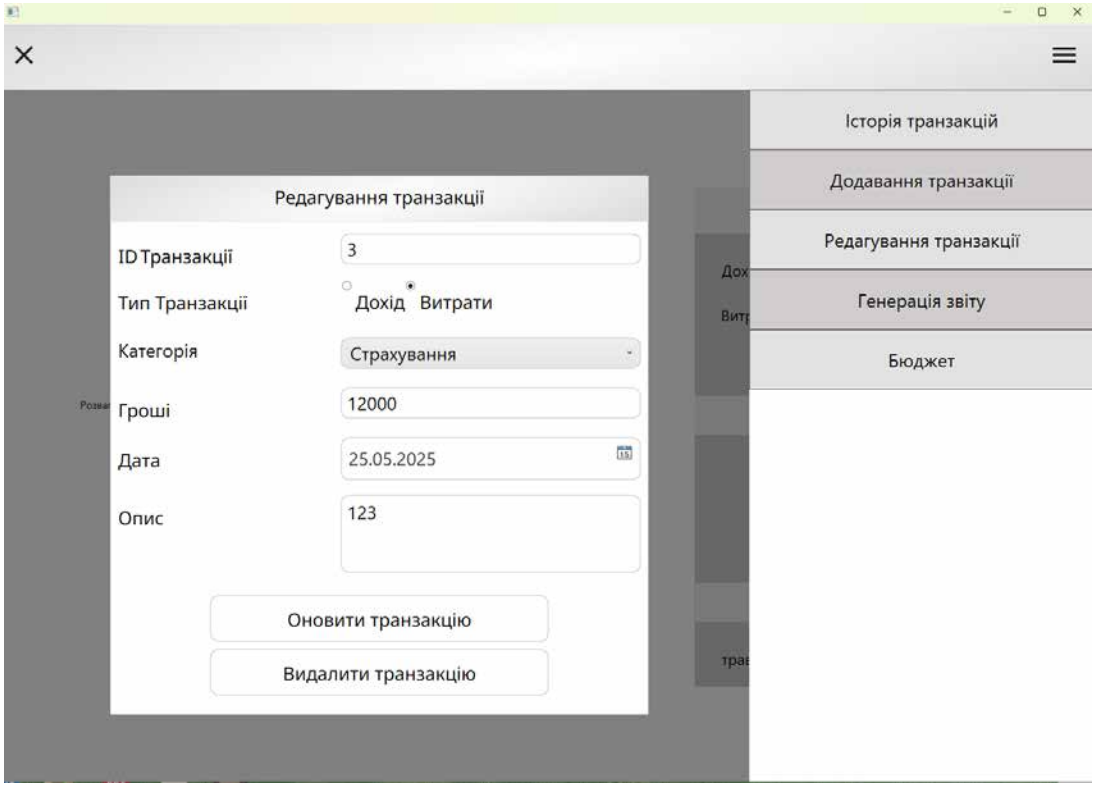


Рис. 4.23 Заповнений контролер додавання транзакцій

Коли користувач звертається до контролера історії (рис. 4.24) транзакцій, він може бачити всі записи транзакцій, пов’язані з його обліковим записом. Це допомагає швидко знаходити потрібну інформацію, аналізувати витрати і доходи, а також тримати під контролем фінансову активність. При подвійному натисканні на поле вся інформація про транзакцію перейде в контролер редагування

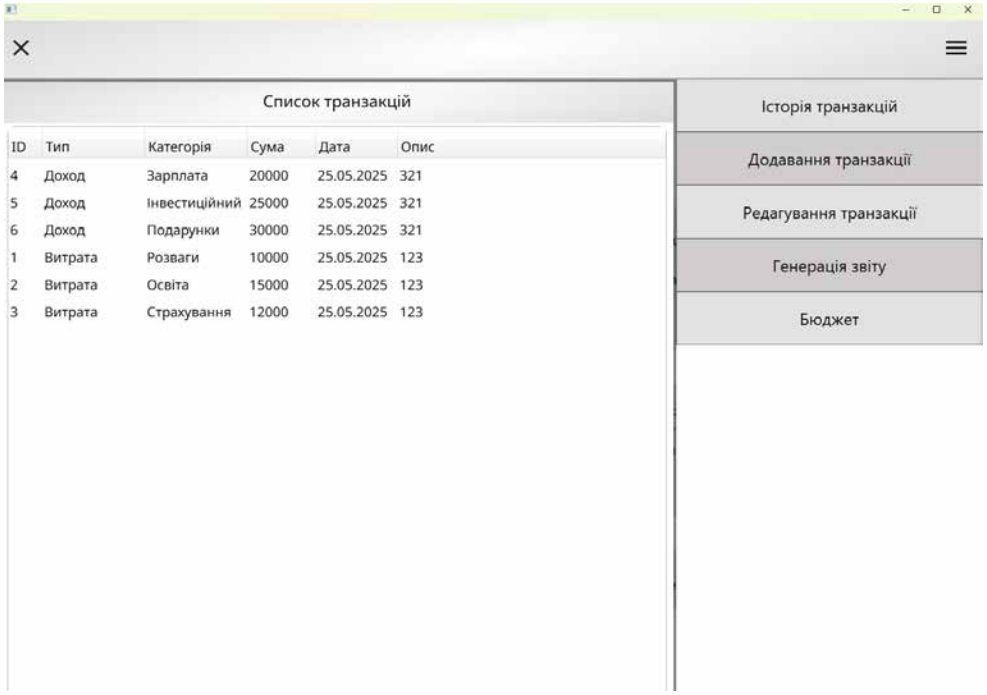


Рис. 4.24 Заповнений контролер додавання транзакцій

При виклику контролеру генерації звітів (рис 4.25) користувач може обрати тип звіту і вказати діапазон дат, після чого згенерувати звіт у форматі PDF або CSV на вибір. Звіт можна зберегти на комп'ютері для подальшого використання чи аналізу.

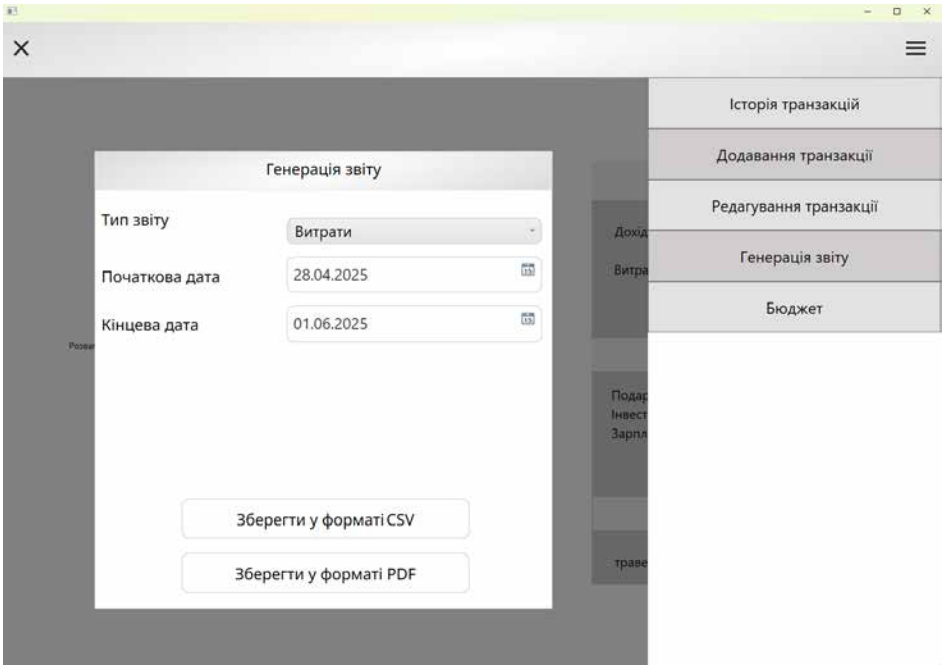
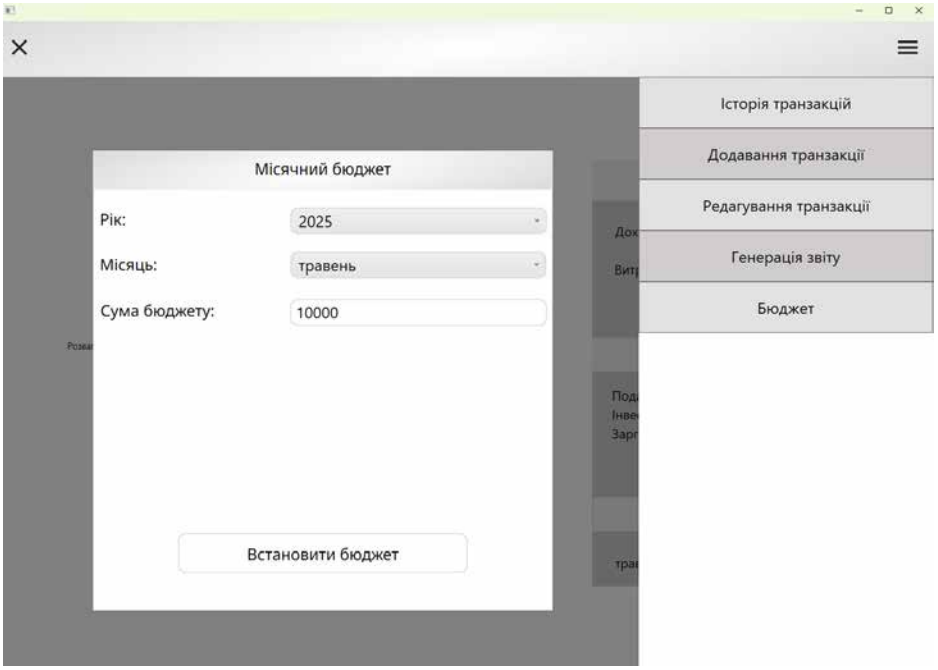


Рис. 4.25 Заповнений контролер генерації звіту

При виклику контролеру місячних бюджетів користувач може встановити певні ліміти на обраний місяць, вказавши допустимі суми витрат за категоріями або загальний бюджет. транзакцій та відображає залишок бюджету в інтерфейсі користувача.



ВИСНОВКИ

У рамках дипломної роботи було успішно розроблено програмне забезпечення для ведення домашньої бухгалтерії, яке відповідає сучасним вимогам користувачів. Створена система забезпечує зручний та інтуїтивно зрозумілий інтерфейс для швидкого внесення транзакцій . Особливу увагу було приділено реалізації синхронізації даних між кількома пристроями в реальному часі, що стало можливим завдяки використанню хмарного середовища Render і бази даних PostgreSQL. Це забезпечило високу доступність, масштабованість і безпечне зберігання даних, а також захист від їх втрати через резервне копіювання.

Розроблений клієнтський додаток на основі технології WPF з використанням мови програмування C#. Система вирішує основні проблеми, пов’язані з веденням домашньої бухгалтерії, такі як недостатня

функціональність і відсутність синхронізації, що робить її актуальним і корисним інструментом для управління особистими фінансами. У перспективі можливе розширення функціональності.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

Бойко В. В. Програмування на мові C# для створення додатків на платформі .NET.

Гавриленко О. П. Бази даних та інформаційні системи: основи проектування та використання PostgreSQL.

Коваленко С. М. Хмарні технології в розробці програмного забезпечення:

Садбері Д. PostgreSQL 14: адміністрування та оптимізація баз даних.

Шилдт Г. Повний довідник по C#.

Чакон С., Страуб Б. Pro Git: все про систему контролю версій Git.

Іванов І. І. Розробка графічних інтерфейсів із використанням WPF: практичний посібник.

Лисенко О. В. Основи роботи з хмарними сервісами для розробників.

Петренко А. М. Сучасні підходи до створення клієнт-серверних додатків.

ISO 9241-11:2018. Ergonomics of human-system interaction – Part 11: Usability: Definitions and concepts. – Geneva: International Organization for Standardization, 2018. – 28 p.

ДОДАТКИ

Додаток А

Розгорнутий код реалізації основних модулей через C#

LoginWindow.xaml.cs

```
using System;
```

```
using System.Windows;
using Microsoft.Extensions.Configuration;
using DiplomTest.Models;
using DiplomTest.Services;

namespace DiplomTest
{
    public partial class LoginWindow : Window
    {
        private readonly string connectionStringTemplate;
        private readonly AuthService _authService;

        public int LoggedInUserID { get; private set; }
        public string LoginText { get; private set; }
        public string PasswordText { get; private set; }
        public string ConnectionString { get; private set; }

        private const string ApiUrl = "https://authapi-62qs.onrender.com/api/auth/login";

        public LoginWindow()
        {
            InitializeComponent();
            this.WindowStartupLocation = WindowStartupLocation.CenterScreen;
            this.Closed += (_, _) =>
            {
                if (this.DialogResult != true)
                    Application.Current.Shutdown();
            };

            var builder = new ConfigurationBuilder()
                .SetBasePath(AppDomain.CurrentDomain.BaseDirectory)
```

```
            .AddJsonFile("appsettings.json", optional: false, reloadOnChange: false);
        IConfiguration config = builder.Build();

        connectionStringTemplate = config.GetConnectionString("DefaultConnection");

        _authService = new AuthService(ApiUrl);
    }

    private async void LoginButton_Click(object sender, RoutedEventArgs e)
    {
        string login = LoginTextBox.Text;
        string password = PasswordBox.Password;

        if (string.IsNullOrWhiteSpace(login) || string.IsNullOrWhiteSpace(password))
        {
            MessageBox.Show("Введіть логін та пароль.");
            return;
        }

        MessageBox.Show("Перевірка вірності даних...");

        var (success, user, error) = await _authService.LoginAsync(login, password);

        if (success && user != null)
        {
            ConnectionString =
$"{connectionStringTemplate}Username={login};Password={password};";
            LoggedInUserID = user.userID;
            LoginText = login;
            PasswordText = password;

            this.DialogResult = true;
```

```
            this.Close();
        }
        else
        {
            MessageBox.Show(error ?? "Невідома помилка.");
        }
    }

    private void RegisterButton_Click(object sender, RoutedEventArgs e)
    {
        var registerWindow = new RegisterWindow();
        bool? result = registerWindow.ShowDialog();
        if (result == true)
        {
            MessageBox.Show("Тепер ви можете увійти.");
        }
    }
  }
}
```

MainWindow.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Threading;
using DiplomTest.Services;
using OxyPlot;
```

```
using OxyPlot.Series;

namespace DiplomTest
{
    public partial class MainWindow : Window
    {
        private readonly DispatcherTimer refreshTimer;
        private readonly string connectionString;
        private readonly DashboardService dashboardService;

        public int CurrentUserID { get; private set; }
        public string CurrentLogin { get; private set; }
        public string CurrentPassword { get; private set; }

        public MainWindow()
        {
            InitializeComponent();

            var loginWindow = new LoginWindow();
            if (loginWindow.ShowDialog() == true)
            {
                CurrentUserID = loginWindow.LoggedInUserID;
                connectionString = loginWindow.ConnectionString;
            }
            else
            {
                MessageBox.Show("Не вдалося підключитися");
                Application.Current?.Shutdown();
                return;
            }
```

```
        // Инициализируем сервис только с connectionString (по твоему первому коду DashboardService)
        dashboardService = new DashboardService(connectionString);

        LoadChartAndTopIncomes();

        refreshTimer = new DispatcherTimer
        {
            Interval = TimeSpan.FromSeconds(5)
        };
        refreshTimer.Tick += (s, e) => LoadChartAndTopIncomes();
        refreshTimer.Start();
    }

    private void LoadChartAndTopIncomes()
    {
        var selectedDate = ChartDatePicker.SelectedDate ?? DateTime.Today;

        try
        {
            // По твоему первому сервису:
            ChartDataResult chartData = dashboardService.GetChartData(CurrentUserID, selectedDate);
            UpdateChartUI(chartData);

            List<IncomeCategoryData> topIncomes = dashboardService.GetTopIncomes(CurrentUserID, selectedDate);
            UpdateTopIncomesUI(topIncomes);
        }
        catch (Exception ex)
        {
            MessageBox.Show("Помилка при завантаженні даних: " + ex.Message,
                            "Error", MessageBoxButton.OK, MessageBoxImage.Error);
```

```
    }
}

private void UpdateChartUI(ChartDataResult data)
{
    var model = new PlotModel();

    var pie = new PieSeries
    {
        StrokeThickness = 1.5,
        AngleSpan = 360,
        StartAngle = 0,
        InnerDiameter = 0.4,
        Diameter = 0.6,
        OutsideLabelFormat = "{1}: ({2:0.#}%)",
        InsideLabelFormat = null,
        TickHorizontalLength = 10,
        TickRadialLength = 5,
        FontSize = 14
    };

    if (data.ExpenseCategories.Count == 0)
    {
        pie.OutsideLabelFormat = null;
        pie.InsideLabelFormat = null;
        pie.Slices.Add(new PieSlice("", 1.0)
        {
            Fill = OxyColor.Parse("#D3D3D3")
        });
    }
    else
```

```
        {
            // Добавляем срезы пирога из ExpenseCategories
            foreach (var category in data.ExpenseCategories)
            {
                pie.Slices.Add(new PieSlice(category.CategoryName, category.TotalAmount));
            }
        }

        model.Series.Add(pie);
        myPlot.Model = model;

        IncomeTextBlock.Text = data.TotalIncome > 0 ? $"+ {data.TotalIncome} ₴" : "0 ₴";
        IncomeTextBlock.Foreground = new
SolidColorBrush((Color)ColorConverter.ConvertFromString(
            data.TotalIncome > 0 ? "#50E233" : "Black"));

        ExpenseTextBlock.Text = data.TotalExpense > 0 ? $"- {data.TotalExpense} ₴" : "0 ₴";
        ExpenseTextBlock.Foreground = new
SolidColorBrush((Color)ColorConverter.ConvertFromString(
            data.TotalExpense > 0 ? "#ED2222" : "Black"));

        double net = data.TotalIncome - data.TotalExpense;
        if (data.TotalIncome == 0 && data.TotalExpense == 0)
        {
            NetBalanceTextBlock.Text = "0 ₴";
            NetBalanceTextBlock.Foreground = new
SolidColorBrush((Color)ColorConverter.ConvertFromString("Black"));
        }
        else
        {
            string sign = net >= 0 ? "+" : "-";
            string color = net >= 0 ? "#50E233" : "#ED2222";
```

```
            NetBalanceTextBlock.Text = $"{sign} {Math.Abs(net)} ₴";
            NetBalanceTextBlock.Foreground = new
SolidColorBrush((Color)ColorConverter.ConvertFromString(color));
        }

        MonthLabelTextBlock.Text = data.MonthName; // Украинское название месяца

        MonthlyBudgetTextBlock.Text = $"{(data.MonthlyBudget - data.MonthlyExpense >= 0 ?
"+" : "-")} {Math.Abs(data.MonthlyBudget - data.MonthlyExpense)} ₴";
        MonthlyBudgetTextBlock.Foreground = new
SolidColorBrush((Color)ColorConverter.ConvertFromString(
            (data.MonthlyBudget - data.MonthlyExpense) >= 0 ? "#50E233" : "#ED2222"));
    }

    private void UpdateTopIncomesUI(List<IncomeCategoryData> incomes)
    {
        TopIncomePanel.Children.Clear();

        Brush greenBrush = new
SolidColorBrush((Color)ColorConverter.ConvertFromString("#50E233"));

        if (incomes.Count == 0)
        {
            var placeholder = new Border
            {
                Height = 40,
                Width = 400,
                Background = new SolidColorBrush(Color.FromRgb(243, 241, 239)),
                CornerRadius = new CornerRadius(8),
                Margin = new Thickness(0, 5, 0, 0),
                Child = new TextBlock
                {
                    Text = "Немає поповнень за цей день",
```

```
            Foreground = Brushes.Gray,
            FontStyle = FontStyles.Italic,
            FontFamily = new FontFamily("Poppins"),
            FontSize = 20,
            Padding = new Thickness(10)
        }
    };
    TopIncomePanel.Children.Add(placeholder);
    return;
}

foreach (var income in incomes)
{
    var row = new StackPanel
    {
        Orientation = Orientation.Horizontal,
        Margin = new Thickness(0, 5, 0, 0),
    };

    var categoryText = new TextBlock
    {
        Text = income.CategoryName,
        FontSize = 20,
        FontFamily = new FontFamily("Poppins"),
        Foreground = Brushes.Black,
        Width = 350
    };

    var amountText = new TextBlock
    {
        Text = $"+ {income.TotalAmount} ₴",
```

```
            FontSize = 20,
            FontFamily = new FontFamily("Poppins"),
            Foreground = greenBrush,
        };

        row.Children.Add(categoryText);
        row.Children.Add(amountText);

        TopIncomePanel.Children.Add(row);
    }
}

private void ChartDatePicker_SelectedDateChanged(object sender,
SelectionChangedEventArgs e)
{
    LoadChartAndTopIncomes();
}

private void MenuIcon_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    bool isOpen = SlidingPanel.Width > 0;
    var panelAnim = new DoubleAnimation(isOpen ? SlidingPanel.Width : 0,
                                        isOpen ? 0 : 450,
                                        TimeSpan.FromMilliseconds(300));
    var shadowAnim = new DoubleAnimation(isOpen ? ShadowOverlay.Opacity : 0.5,
                                         isOpen ? 0 : 0.5,
                                         TimeSpan.FromMilliseconds(300));

    if (!isOpen) ShadowOverlay.Visibility = Visibility.Visible;
    else shadowAnim.Completed += (_, __) => ShadowOverlay.Visibility =
Visibility.Collapsed;
```

```
    SlidingPanel.BeginAnimation(WidthProperty, panelAnim);
    ShadowOverlay.BeginAnimation(OpacityProperty, shadowAnim);
}

private void Button1_Click(object sender, RoutedEventArgs e)
{
    if (MainControl.Content is TransactionTable)
    {
        MainControl.Content = null;
    }
    else
    {
        MainControl.Content = new TransactionTable(CurrentUserID, connectionString);
    }
}

private void AddTransactionButton_Click(object sender, RoutedEventArgs e)
{
    if (MainControl.Content is AddTransactionControl)
    {
        MainControl.Content = null;
    }
    else
    {
        MainControl.Content = new AddTransactionControl(CurrentUserID, connectionString);
    }
}

private void ManageTransactionButton_Click(object sender, RoutedEventArgs e)
{
    if (MainControl.Content is ManageTransactionControl)
```

```
        {
            MainControl.Content = null;
        }
        else
        {
            MainControl.Content = new ManageTransactionControl(CurrentUserID,
connectionString);
        }
    }

    private void ReportControlButton_Click(object sender, RoutedEventArgs e)
    {
        if (MainControl.Content is ReportControl)
        {
            MainControl.Content = null;
        }
        else
        {
            MainControl.Content = new ReportControl(CurrentUserID);
        }
    }

    private void MonthlyBudgetButton_Click(object sender, RoutedEventArgs e)
    {
        if (MainControl.Content is MonthlyBudgetControl)
        {
            MainControl.Content = null;
        }
        else
        {
            MainControl.Content = new MonthlyBudgetControl(CurrentUserID, connectionString);
        }
```

```
        }

        private void CloseIcon_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
        {
            Application.Current.Shutdown();
        }
    }
}
```

RegisterWindow.xaml.cs

```
using System;
using System.Windows;
using DiplomTest.Services;

namespace DiplomTest
{
    public partial class RegisterWindow : Window
    {
        private readonly AuthService _authService;

        private const string ApiLoginUrl = "https://authapi-
62qs.onrender.com/api/auth/login";
        private const string ApiRegisterUrl = "https://authapi-
62qs.onrender.com/api/auth/register";

        public RegisterWindow()
        {
            InitializeComponent();
            this.WindowStartupLocation = WindowStartupLocation.CenterScreen;
```

```
        _authService = new AuthService(ApiLoginUrl, ApiRegisterUrl);
    }

    private async void RegisterButton_Click(object sender, RoutedEventArgs e)
    {
        ErrorMessage.Text = "";

        string login = LoginTextBox.Text.Trim();
        string email = EmailTextBox.Text.Trim();
        string password = PasswordBox.Password;

        if (string.IsNullOrEmpty(login) || string.IsNullOrEmpty(email) ||
string.IsNullOrEmpty(password))
        {
            ErrorMessage.Text = "Будь ласка, заповніть усі поля.";
            return;
        }

        var (success, error) = await _authService.RegisterAsync(login, email,
password);

        if (success)
        {
            MessageBox.Show("Реєстрація успішна! Тепер ви можете увійти.",
"Успіх",
                            MessageBoxButton.OK, MessageBoxImage.Information);
            this.DialogResult = true;
            this.Close();
```

```
            }
            else
            {
                ErrorMessage.Text = error ?? "Невідома помилка при реєстрації.";
            }
        }
    }
}
```

AddTransanctionControl.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls;
using Npgsql;

namespace DiplomTest
{
    public partial class AddTransactionControl : UserControl
    {
        private int userId;
        private string connectionString;

        public AddTransactionControl(int currentUserId, string connString)
        {
            InitializeComponent();
            userId = currentUserId;
            connectionString = connString;
            LoadCategories(1);
        }
```

```
private class CategoryItem
{
    public int Id { get; set; }
    public string Name { get; set; }

    public override string ToString()
    {
        return Name;
    }
}

private void LoadCategories(int transactionTypeID)
{
    CategoryComboBox.Items.Clear();

    try
    {
        using (var conn = new NpgsqlConnection(connectionString))
        {
            conn.Open();

            var cmd = new NpgsqlCommand(
                "SELECT \"categoryID\", \"categoryName\" FROM \"ExpenseCategory\" WHERE \"transactionTypeID\" = @transactionTypeID", conn);
            cmd.Parameters.AddWithValue("transactionTypeID", transactionTypeID);

            using (var reader = cmd.ExecuteReader())
            {
                while (reader.Read())
                {
                    CategoryComboBox.Items.Add(new CategoryItem
```

```
                {
                    Id = reader.GetInt32(0),
                    Name = reader.GetString(1)
                });
            }
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show("Помилка завантаження категорій: " + ex.Message);
    }
}

private void TransactionTypeRadioButton_Checked(object sender, RoutedEventArgs e)
{
    if (ExpenseRadioButton.IsChecked == true)
    {
        LoadCategories(1);
    }
    else if (IncomeRadioButton.IsChecked == true)
    {
        LoadCategories(2);
    }
}

private void AddTransactionButton_Click(object sender, RoutedEventArgs e)
{
    if (CategoryComboBox.SelectedItem == null ||
string.IsNullOrWhiteSpace(AmountTextBox.Text) || DatePicker.SelectedDate == null)
    {
        MessageBox.Show("Будь ласка, заповніть усі поля.");
```

```
    return;
}

var selectedCategory = CategoryComboBox.SelectedItem as CategoryItem;
if (selectedCategory == null)
{
    MessageBox.Show("Вибрано недійсну категорію.");
    return;
}

int categoryId = selectedCategory.Id;
decimal amount;
if (!decimal.TryParse(AmountTextBox.Text, out amount))
{
    MessageBox.Show("Невірний формат суми.");
    return;
}

if (amount <= 0)  // <-- Проверка на отрицательные и нулевые значения
{
    MessageBox.Show("Сума має бути більшою за нуль.");
    return;
}

DateTime date = DatePicker.SelectedDate.Value;
string description = DescriptionTextBox.Text ?? "";

try
{
    using (var conn = new NpgsqlConnection(connectionString))
    {
```

```
                conn.Open();


                var cmd = new NpgsqlCommand(
                    "INSERT INTO \"Transaction\" (\"userID\", \"transactionTypeID\", \"amount\", \"categoryID\", \"date\", \"description\") " +
                    "VALUES (@userID, @transactionTypeID, @amount, @categoryID, @date, @description)", conn);


                cmd.Parameters.AddWithValue("userID", userId);
                cmd.Parameters.AddWithValue("transactionTypeID", ExpenseRadioButton.IsChecked == true ? 1 : 2);
                cmd.Parameters.AddWithValue("amount", amount);
                cmd.Parameters.AddWithValue("categoryID", categoryId);
                cmd.Parameters.AddWithValue("date", date);
                cmd.Parameters.AddWithValue("description", description);


                cmd.ExecuteNonQuery();


                MessageBox.Show("Транзакцію додано успішно.");
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show("Помилка додавання транзакції: " + ex.Message);
        }
    }


}
}
```

ManageTransactionControl.xaml.cs

```
using System;
using System.Windows;
```

```
using System.Windows.Controls;
using Npgsql;

namespace DiplomTest
{
    public partial class ManageTransactionControl : UserControl
    {
        private int userId;
        private string connectionString;

        public ManageTransactionControl(int currentUserId, string connString)
        {
            InitializeComponent();
            userId = currentUserId;
            connectionString = connString;
            LoadCategories(1);
        }

        private class CategoryItem
        {
            public int Id { get; set; }
            public string Name { get; set; }

            public override string ToString()
            {
                return Name;
            }
        }

        private void LoadCategories(int transactionTypeID)
        {
            CategoryComboBox.Items.Clear();

            try
            {
                using (var conn = new NpgsqlConnection(connectionString))
                {
                    conn.Open();

                    var cmd = new NpgsqlCommand(
                        "SELECT \"categoryID\", \"categoryName\" FROM \"ExpenseCategory\" WHERE
\"transactionTypeID\" = @transactionTypeID", conn);
                    cmd.Parameters.AddWithValue("transactionTypeID", transactionTypeID);

                    using (var reader = cmd.ExecuteReader())
                    {
                        while (reader.Read())
                        {
                            CategoryComboBox.Items.Add(new CategoryItem
                            {
                                Id = reader.GetInt32(0),
                                Name = reader.GetString(1)
```

```
                });
            }
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show("Помилка під час завантаження категорій: " + ex.Message);
    }
}

private void TransactionTypeRadioButton_Checked(object sender, RoutedEventArgs e)
{
    if (ExpenseRadioButton.IsChecked == true)
    {
        LoadCategories(1);
    }
    else if (IncomeRadioButton.IsChecked == true)
    {
        LoadCategories(2);
    }
}

private void UpdateTransactionButton_Click(object sender, RoutedEventArgs e)
{
    if (string.IsNullOrWhiteSpace(TransactionIdTextBox.Text) ||
        CategoryComboBox.SelectedItem == null ||
        string.IsNullOrWhiteSpace(AmountTextBox.Text) ||
        DatePicker.SelectedDate == null)
    {
        MessageBox.Show("Заповніть усі поля.");
        return;
    }

    if (!(CategoryComboBox.SelectedItem is CategoryItem selectedCategory))
    {
        MessageBox.Show("Некоректно вибрано категорію.");
        return;
    }

    if (!int.TryParse(TransactionIdTextBox.Text, out int transactionId))
    {
        MessageBox.Show("Некоректний ID транзакції.");
        return;
    }

    if (!decimal.TryParse(AmountTextBox.Text, out decimal amount))
    {
        MessageBox.Show("Некоректний формат суми.");
        return;
    }
```

```
        if (amount <= 0)  // <-- Добавленная проверка на отрицательные и нулевые значения
        {
            MessageBox.Show("Сума має бути більшою за нуль.");
            return;
        }

        DateTime date = DatePicker.SelectedDate.Value;
        string description = DescriptionTextBox.Text ?? "";

        try
        {
            using (var conn = new NpgsqlConnection(connectionString))
            {
                conn.Open();

                var cmd = new NpgsqlCommand(
                    "UPDATE \"Transaction\" SET \"categoryID\" = @categoryID, \"amount\" =
@amount, \"date\" = @date, \"description\" = @description " +
                    "WHERE \"transactionID\" = @transactionID AND \"userID\" = @userID", conn);

                cmd.Parameters.AddWithValue("transactionID", transactionId);
                cmd.Parameters.AddWithValue("userID", userId);
                cmd.Parameters.AddWithValue("categoryID", selectedCategory.Id);
                cmd.Parameters.AddWithValue("amount", amount);
                cmd.Parameters.AddWithValue("date", date);
                cmd.Parameters.AddWithValue("description", description);

                int rows = cmd.ExecuteNonQuery();
                MessageBox.Show(rows > 0 ? "Транзакцію оновлено." : "Транзакцію не
знайдено.");
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show("Помилка при оновленні: " + ex.Message);
        }
    }


    private void DeleteTransactionButton_Click(object sender, RoutedEventArgs e)
    {
        if (string.IsNullOrWhiteSpace(TransactionIdTextBox.Text))
        {
            MessageBox.Show("Введіть ID транзакції для видалення.");
            return;
        }

        if (!int.TryParse(TransactionIdTextBox.Text, out int transactionId))
        {
            MessageBox.Show("Некоректний ID транзакції.");
            return;
        }
```

```
        try
        {
            using (var conn = new NpgsqlConnection(connectionString))
            {
                conn.Open();

                var cmd = new NpgsqlCommand(
                    "DELETE FROM \"Transaction\" WHERE \"transactionID\" = @transactionID
AND \"userID\" = @userID", conn);
                cmd.Parameters.AddWithValue("transactionID", transactionId);
                cmd.Parameters.AddWithValue("userID", userId);

                int rows = cmd.ExecuteNonQuery();

                MessageBox.Show(rows > 0 ? "Транзакцію видалено." : "Транзакцію не знайдено
або не видалено.");
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show("Помилка при видаленні: " + ex.Message);
        }
    }

    public void LoadTransactionData(int transactionId)
    {
        try
        {
            using (var conn = new NpgsqlConnection(connectionString))
            {
                conn.Open();

                var cmd = new NpgsqlCommand(
                    "SELECT * FROM \"Transaction\" WHERE \"transactionID\" = @id AND
\"userID\" = @uid", conn);
                cmd.Parameters.AddWithValue("id", transactionId);
                cmd.Parameters.AddWithValue("uid", userId);

                using (var reader = cmd.ExecuteReader())
                {
                    if (reader.Read())
                    {
                        TransactionIdTextBox.Text = transactionId.ToString();
                        AmountTextBox.Text = reader["amount"].ToString();
                        DescriptionTextBox.Text = reader["description"]?.ToString() ?? "";
                        DatePicker.SelectedDate = Convert.ToDateTime(reader["date"]);

                        int typeId = Convert.ToInt32(reader["transactionTypeID"]);
                        if (typeId == 1) ExpenseRadioButton.IsChecked = true;
                        else IncomeRadioButton.IsChecked = true;
```

```
                    LoadCategories(typeId);

                    int categoryId = Convert.ToInt32(reader["categoryID"]);

                    foreach (var item in CategoryComboBox.Items)
                    {
                        if (item is CategoryItem cat && cat.Id == categoryId)
                        {
                            CategoryComboBox.SelectedItem = item;
                            break;
                        }
                    }
                }
                else
                {
                    MessageBox.Show("Транзакцію не знайдено.");
                }
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show("Помилка завантаження транзакції: " + ex.Message);
        }
    }
}
```

MonthlyBudgetControl.xaml.cs

```
using System;
using System.Linq;
using System.Windows;
using System.Windows.Controls;
using Npgsql;

namespace DiplomTest
{
    public partial class MonthlyBudgetControl : UserControl
    {
        private int userId;
        private string connectionString;

        public MonthlyBudgetControl(int currentUserId, string connString)
        {
            InitializeComponent();

            userId = currentUserId;
            connectionString = connString;

            LoadYearMonthSelectors();
```

```
        YearComboBox.SelectionChanged += YearMonth_SelectionChanged;
        MonthComboBox.SelectionChanged += YearMonth_SelectionChanged;

        YearComboBox.SelectedItem = DateTime.Today.Year;
        MonthComboBox.SelectedIndex = DateTime.Today.Month - 1;
    }

    private void LoadYearMonthSelectors()
    {
        int currentYear = DateTime.Today.Year;
        for (int y = currentYear - 5; y <= currentYear + 5; y++)
            YearComboBox.Items.Add(y);

        var monthNames = System.Globalization.CultureInfo.GetCultureInfo("uk-
UA").DateTimeFormat.MonthNames
            .Where(m => !string.IsNullOrEmpty(m))
            .ToList();

        MonthComboBox.ItemsSource = monthNames;
    }

    private void YearMonth_SelectionChanged(object sender, SelectionChangedEventArgs e)
    {
        if (YearComboBox.SelectedItem == null || MonthComboBox.SelectedIndex == -1)
            return;

        int year = (int)YearComboBox.SelectedItem;
        int month = MonthComboBox.SelectedIndex + 1;

        LoadMonthlyBudget(year, month);
    }

    private void LoadMonthlyBudget(int year, int month)
    {
        try
        {
            using var conn = new NpgsqlConnection(connectionString);
            conn.Open();

            var cmd = new NpgsqlCommand(
                "SELECT amount FROM \"MonthlyBudget\" WHERE \"userID\" = @uid AND year =
@year AND month = @month", conn);

            cmd.Parameters.AddWithValue("uid", userId);
            cmd.Parameters.AddWithValue("year", year);
            cmd.Parameters.AddWithValue("month", month);

            var result = cmd.ExecuteScalar();

            if (result != null && result != DBNull.Value)
            {
```

```
                AmountTextBox.Text = Convert.ToDecimal(result).ToString("F2");
                MessageBox.Show($"Бюджет на {MonthComboBox.SelectedItem} {year} завантажено.");
            }
            else
            {
                AmountTextBox.Text = "";
                MessageBox.Show($"Бюджет на {MonthComboBox.SelectedItem} {year} не встановлено.");
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show("Помилка завантаження бюджету: " + ex.Message);
        }
    }

    private void SaveButton_Click(object sender, RoutedEventArgs e)
    {
        if (YearComboBox.SelectedItem == null || MonthComboBox.SelectedIndex == -1)
        {
            MessageBox.Show("Оберіть рік і місяць.");
            return;
        }

        if (!decimal.TryParse(AmountTextBox.Text, out decimal amount) || amount < 0)
        {
            MessageBox.Show("Введіть коректну суму бюджету.");
            return;
        }

        int year = (int)YearComboBox.SelectedItem;
        int month = MonthComboBox.SelectedIndex + 1;

        try
        {
            using var conn = new NpgsqlConnection(connectionString);
            conn.Open();

            var checkCmd = new NpgsqlCommand(
                "SELECT COUNT(*) FROM \"MonthlyBudget\" WHERE \"userID\" = @uid AND year = @year AND month = @month", conn);

            checkCmd.Parameters.AddWithValue("uid", userId);
            checkCmd.Parameters.AddWithValue("year", year);
            checkCmd.Parameters.AddWithValue("month", month);

            long count = (long)checkCmd.ExecuteScalar();

            if (count == 0)
            {
                var insertCmd = new NpgsqlCommand(
```

```
                "INSERT INTO \"MonthlyBudget\" (\"userID\", year, month, amount) VALUES
(@uid, @year, @month, @amount)", conn);

                insertCmd.Parameters.AddWithValue("uid", userId);
                insertCmd.Parameters.AddWithValue("year", year);
                insertCmd.Parameters.AddWithValue("month", month);
                insertCmd.Parameters.AddWithValue("amount", amount);

                insertCmd.ExecuteNonQuery();
                MessageBox.Show($"Бюджет на {MonthComboBox.SelectedItem} {year} додано.");
            }
            else
            {
                var updateCmd = new NpgsqlCommand(
                    "UPDATE \"MonthlyBudget\" SET amount = @amount WHERE \"userID\" = @uid
AND year = @year AND month = @month", conn);

                updateCmd.Parameters.AddWithValue("amount", amount);
                updateCmd.Parameters.AddWithValue("uid", userId);
                updateCmd.Parameters.AddWithValue("year", year);
                updateCmd.Parameters.AddWithValue("month", month);

                updateCmd.ExecuteNonQuery();
                MessageBox.Show($"Бюджет на {MonthComboBox.SelectedItem} {year}
оновлено.");
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show("Помилка збереження бюджету: " + ex.Message);
        }
    }
  }
}
```

ReportControl.xaml.cs

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using System.Windows;
using System.Windows.Controls;
using Npgsql;
using QuestPDF.Fluent;
using QuestPDF.Infrastructure;
using CsvHelper;
using System.Diagnostics;
using QuestPDF.Helpers;
using System.Text;
```

```
namespace DiplomTest
{
    public partial class ReportControl : UserControl
    {
        private int userId;


        public ReportControl(int currentUserId)
        {
            InitializeComponent();
            userId = currentUserId;
            QuestPDF.Settings.License = LicenseType.Community;
        }

        private void SavePdfButton_Click(object sender, RoutedEventArgs e)
        {
            if (ReportTypeComboBox.SelectedItem is not ComboBoxItem selectedItem)
            {
                MessageBox.Show("Будь ласка, оберіть тип звіту.", "Помилка",
MessageBoxButton.OK, MessageBoxImage.Warning);
                return;
            }

            string reportType = selectedItem.Content.ToString();
            string datePart = DateTime.Now.ToString("yyyyMMdd_HHmmss");
            string filePath = $"{reportType}_Звіт_{datePart}.pdf";

            if (reportType == "Бюджет")
            {
                var rows = LoadBudgetReportRows();
                if (rows.Count == 0)
                {
                    MessageBox.Show("Немає даних бюджету для обраного періоду.", "Увага",
MessageBoxButton.OK, MessageBoxImage.Warning);
                    return;
                }

                Document.Create(container =>
                {
                    container.Page(page =>
                    {
                        page.Margin(30);
                        page.Header().Text("Бюджетний звіт")
                            .FontSize(20).Bold().AlignCenter();

                        page.Content().PaddingTop(20).Table(table =>
                        {
                            table.ColumnsDefinition(columns =>
                            {
                                columns.RelativeColumn(2);
                                columns.RelativeColumn(2);
                                columns.RelativeColumn(2);
```

```
                    columns.RelativeColumn(2);
                });

                table.Header(header =>
                {

header.Cell().Border(1).BorderColor(Colors.Black).Padding(5).Text("Місяць").Bold();

header.Cell().Border(1).BorderColor(Colors.Black).Padding(5).Text("Бюджет").Bold();

header.Cell().Border(1).BorderColor(Colors.Black).Padding(5).Text("Витрати").Bold();

header.Cell().Border(1).BorderColor(Colors.Black).Padding(5).Text("Різниця").Bold();
                });

                foreach (var row in rows)
                {
                    table.Cell().Border(1).BorderColor(Colors.Black).Padding(5)

.Text($"{CultureInfo.CurrentCulture.DateTimeFormat.GetMonthName(row.Month)}
{row.Year}");

                    table.Cell().Border(1).BorderColor(Colors.Black).Padding(5)
                        .Text($"{row.Budget} грн");

                    table.Cell().Border(1).BorderColor(Colors.Black).Padding(5)
                        .Text($"{row.TotalExpenses} грн");

                    table.Cell().Border(1).BorderColor(Colors.Black).Padding(5)
                        .Text($"{row.Difference} грн")
                        .FontColor(row.Difference >= 0 ? Colors.Green.Medium :
Colors.Red.Medium);
                }
            });

            page.Footer().AlignCenter().Text($"Сформовано {DateTime.Now:g}");
        });
    })
    .GeneratePdf(filePath);

    Process.Start(new ProcessStartInfo(filePath) { UseShellExecute = true });
    MessageBox.Show("PDF-звіт успішно збережено.", "Успіх", MessageBoxButton.OK,
MessageBoxImage.Information);
    return;
}

// --- обычный транзакционный отчет ---
var transactions = LoadTransactions();
if (transactions.Count == 0)
{
    MessageBox.Show("Немає транзакцій в обраний період.", "Увага",
MessageBoxButton.OK, MessageBoxImage.Warning);
```

```
            return;
        }

        Document.Create(container =>
        {
            container.Page(page =>
            {
                page.Margin(30);
                page.Header().Text("Звіт по транзакціях")
                    .FontSize(20).Bold().AlignCenter();

                page.Content().PaddingTop(20).Table(table =>
                {
                    table.ColumnsDefinition(columns =>
                    {
                        columns.RelativeColumn(1);
                        columns.RelativeColumn(2);
                        columns.RelativeColumn(2);
                        columns.RelativeColumn(3);
                    });

                    table.Header(header =>
                    {

header.Cell().Border(1).BorderColor(Colors.Black).Padding(5).Text("Дата").Bold();

header.Cell().Border(1).BorderColor(Colors.Black).Padding(5).Text("Сума").Bold();

header.Cell().Border(1).BorderColor(Colors.Black).Padding(5).Text("Категорія").Bold();

header.Cell().Border(1).BorderColor(Colors.Black).Padding(5).Text("Опис").Bold();
                    });

                    foreach (var tx in transactions)
                    {
                        table.Cell().Border(1).BorderColor(Colors.Black).Padding(5).Text(tx.Дата);
                        table.Cell().Border(1).BorderColor(Colors.Black).Padding(5).Text(tx.Сума);
                        table.Cell().Border(1).BorderColor(Colors.Black).Padding(5).Text(tx.Категорія);
                        table.Cell().Border(1).BorderColor(Colors.Black).Padding(5).Text(tx.Опис);
                    }
                });

                page.Footer().AlignCenter().Text($"Сформовано {DateTime.Now:g}");
            });
        })
        .GeneratePdf(filePath);

        Process.Start(new ProcessStartInfo(filePath) { UseShellExecute = true });
        MessageBox.Show("PDF-звіт успішно збережено.", "Успіх", MessageBoxButton.OK,
MessageBoxImage.Information);
    }
```

```
        private void SaveCsvButton_Click(object sender, RoutedEventArgs e)
        {
            if (ReportTypeComboBox.SelectedItem is not ComboBoxItem selectedItem)
            {
                MessageBox.Show("Будь ласка, оберіть тип звіту.", "Помилка",
MessageBoxButton.OK, MessageBoxImage.Warning);
                return;
            }

            string reportType = selectedItem.Content.ToString();
            string datePart = DateTime.Now.ToString("yyyyMMdd_HHmmss");
            string filePath = $"{reportType}_Звіт_{datePart}.csv";

            if (reportType == "Бюджет")
            {
                var rows = LoadBudgetReportRows();
                if (rows.Count == 0)
                {
                    MessageBox.Show("Немає даних бюджету для обраного періоду.", "Увага",
MessageBoxButton.OK, MessageBoxImage.Warning);
                    return;
                }

                using (var writer = new StreamWriter(filePath, false, new UTF8Encoding(true)))
                {
                    writer.WriteLine("Місяць;Бюджет;Витрати;Різниця");

                    foreach (var row in rows)
                    {
                        string monthName =
CultureInfo.CurrentCulture.DateTimeFormat.GetMonthName(row.Month);
                        string dateStr = $"{monthName} {row.Year}";
                        string budget = row.Budget.ToString("0.00", CultureInfo.InvariantCulture);
                        string expenses = row.TotalExpenses.ToString("0.00",
CultureInfo.InvariantCulture);
                        string difference = row.Difference.ToString("0.00", CultureInfo.InvariantCulture);

                        writer.WriteLine($"{dateStr};{budget};{expenses};{difference}");
                    }
                }

                MessageBox.Show("CSV-звіт бюджету успішно збережено.", "Успіх",
MessageBoxButton.OK, MessageBoxImage.Information);
                Process.Start(new ProcessStartInfo(filePath) { UseShellExecute = true });
                return;
            }

            // --- звіт по транзакціях ---
            var transactions = LoadTransactions();
            if (transactions.Count == 0)
            {
```

```
            MessageBox.Show("Немає транзакцій в обраний період.", "Увага",
MessageBoxButton.OK, MessageBoxImage.Warning);
            return;
        }

        using (var writer = new StreamWriter(filePath, false, new UTF8Encoding(true)))
        {
            writer.WriteLine("Дата;Сума;Категорія;Опис");

            foreach (var tx in transactions)
            {
                string amount = decimal.Parse(tx.Сума.Replace(" грн.", ""),
CultureInfo.CurrentCulture)
                    .ToString("0.00", CultureInfo.InvariantCulture);
                writer.WriteLine($"{tx.Дата};{amount};{tx.Категорія};{tx.Опис}");
            }
        }

        MessageBox.Show("CSV-звіт транзакцій успішно збережено.", "Успіх",
MessageBoxButton.OK, MessageBoxImage.Information);
        Process.Start(new ProcessStartInfo(filePath) { UseShellExecute = true });
    }



    private List<TransactionRow> LoadTransactions()
    {
        var list = new List<TransactionRow>();

        if (!StartDatePicker.SelectedDate.HasValue || !EndDatePicker.SelectedDate.HasValue)
        {
            MessageBox.Show("Будь ласка, виберіть обидві дати.", "Помилка",
MessageBoxButton.OK, MessageBoxImage.Warning);
            return list;
        }

        var startDate = StartDatePicker.SelectedDate.Value;
        var endDate = EndDatePicker.SelectedDate.Value;

        if (startDate > endDate)
        {
            MessageBox.Show("Початкова дата не може бути більшою за кінцеву.", "Помилка",
MessageBoxButton.OK, MessageBoxImage.Warning);
            return list;
        }

        if (ReportTypeComboBox.SelectedItem is not ComboBoxItem selectedItem)
        {
            MessageBox.Show("Будь ласка, оберіть тип звіту.", "Помилка",
MessageBoxButton.OK, MessageBoxImage.Warning);
            return list;
        }
```

```
            int transactionTypeId = selectedItem.Content.ToString() == "Доходи" ? 1 : 2;

            string connString = "Host=dpg-d03csr49c44c73b251g0-a.frankfurt-postgres.render.com;" +

"Username=admin;Password=FMVLOQHm8DG10NZUbRXtFx9W3MKwZYaV;" +
                        "Database=accdb_95ld";

            using (var conn = new NpgsqlConnection(connString))
            {
                conn.Open();
                var cmd = new NpgsqlCommand(
                    "SELECT t.\"date\", t.\"amount\", t.\"description\", ec.\"categoryName\" " +
                    "FROM \"Transaction\" t " +
                    "JOIN \"ExpenseCategory\" ec ON t.\"categoryID\" = ec.\"categoryID\" " +
                    "WHERE t.\"userID\" = @userID " +
                    "AND t.\"transactionTypeID\" = @transactionTypeID " +
                    "AND t.\"date\" BETWEEN @start AND @end " +
                    "ORDER BY t.\"date\"", conn);

                cmd.Parameters.AddWithValue("userID", userId);
                cmd.Parameters.AddWithValue("transactionTypeID", transactionTypeId);
                cmd.Parameters.AddWithValue("start", startDate);
                cmd.Parameters.AddWithValue("end", endDate);

                using (var reader = cmd.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        var date = reader.GetDateTime(0);
                        var amount = reader.GetDecimal(1);
                        var description = reader.IsDBNull(2) ? "" : reader.GetString(2);
                        var category = reader.GetString(3);

                        list.Add(new TransactionRow
                        {
                            Дата = date.ToShortDateString(),
                            Сума = $"{amount} грн.",
                            Категорія = category,
                            Опис = description
                        });
                    }
                }
            }

            return list;
        }

        private List<MonthlyBudgetReportRow> LoadBudgetReportRows()
        {
            var list = new List<MonthlyBudgetReportRow>();
```

```
        if (!StartDatePicker.SelectedDate.HasValue || !EndDatePicker.SelectedDate.HasValue)
        {
            MessageBox.Show("Будь ласка, виберіть обидві дати.", "Помилка",
MessageBoxButton.OK, MessageBoxImage.Warning);
            return list;
        }

        var startDate = StartDatePicker.SelectedDate.Value;
        var endDate = EndDatePicker.SelectedDate.Value;

        string connString = "Host=dpg-d03csr49c44c73b251g0-a.frankfurt-postgres.render.com;" +

"Username=admin;Password=FMVLOQHm8DG10NZUbRXtFx9W3MKwZYaV;" +
                    "Database=accdb_95ld";

        using (var conn = new NpgsqlConnection(connString))
        {
            conn.Open();

            var cmd = new NpgsqlCommand(@"
            SELECT
                mb.year, mb.month, mb.amount AS budget,
                COALESCE(SUM(t.amount), 0) AS total_expenses
            FROM ""MonthlyBudget"" mb
            LEFT JOIN ""Transaction"" t
                ON EXTRACT(YEAR FROM t.date) = mb.year
                AND EXTRACT(MONTH FROM t.date) = mb.month
                AND t.""userID"" = mb.""userID""
                AND t.""transactionTypeID"" = 2
            WHERE mb.""userID"" = @userId
                AND (mb.year > @startYear OR (mb.year = @startYear AND mb.month >=
@startMonth))
                AND (mb.year < @endYear OR (mb.year = @endYear AND mb.month <=
@endMonth))
            GROUP BY mb.year, mb.month, mb.amount
            ORDER BY mb.year, mb.month;
        ", conn);

            cmd.Parameters.AddWithValue("userId", userId);
            cmd.Parameters.AddWithValue("startYear", startDate.Year);
            cmd.Parameters.AddWithValue("startMonth", startDate.Month);
            cmd.Parameters.AddWithValue("endYear", endDate.Year);
            cmd.Parameters.AddWithValue("endMonth", endDate.Month);

            using (var reader = cmd.ExecuteReader())
            {
                while (reader.Read())
                {
                    list.Add(new MonthlyBudgetReportRow
                    {
                        Year = reader.GetInt32(0),
                        Month = reader.GetInt32(1),
```

```
                    Budget = reader.GetDecimal(2),
                    TotalExpenses = reader.GetDecimal(3)
                });
            }
        }
    }

    return list;
}

public class MonthlyBudgetReportRow
{
    public int Year { get; set; }
    public int Month { get; set; }
    public decimal Budget { get; set; }
    public decimal TotalExpenses { get; set; }
    public decimal Difference => Budget - TotalExpenses;
}
public class TransactionRow
{
    public string Дата { get; set; }
    public string Сума { get; set; }
    public string Категорія { get; set; }
    public string Опис { get; set; }
}
}
}
```

TransactionTable.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls;
using Npgsql;

namespace DiplomTest
{
    public partial class TransactionTable : UserControl
    {
        private readonly int userId;
        private readonly string connectionString;
```

```
public TransactionTable(int currentUserId, string connString)
{
    InitializeComponent();
    userId = currentUserId;
    connectionString = connString;
    LoadTransactions();
}

private void LoadTransactions()
{
    var transactions = new List<dynamic>();

    try
    {
        using (var conn = new NpgsqlConnection(connectionString))
        {
            conn.Open();

            var cmd = new NpgsqlCommand(@"
            SELECT t.""transactionID"",
                tt.""typeName"",
                ec.""categoryName"",
                t.""amount"",
                t.""date"",
                t.""description""
            FROM ""Transaction"" t
            LEFT JOIN ""TransactionType"" tt ON t.""transactionTypeID"" =
tt.""transactionTypeID""
            LEFT JOIN ""ExpenseCategory"" ec ON t.""categoryID"" = ec.""categoryID""
            WHERE t.""userID"" = @uid
            ORDER BY t.""date"" DESC", conn);
```

```
            cmd.Parameters.AddWithValue("uid", userId);

            using (var reader = cmd.ExecuteReader())
            {
                while (reader.Read())
                {
                    transactions.Add(new
                    {
                        transactionID = reader.GetInt32(0),
                        transactionType = reader.IsDBNull(1) ? "—" : reader.GetString(1),
                        category = reader.IsDBNull(2) ? "—" : reader.GetString(2),
                        amount = reader.GetDecimal(3),
                        date = reader.GetDateTime(4).ToShortDateString(),
                        description = reader.IsDBNull(5) ? "" : reader.GetString(5)
                    });
                }
            }
        }

        TransactionDataGrid.ItemsSource = transactions;
    }
    catch (Exception ex)
    {
        MessageBox.Show("Помилка при завантаженні транзакцій:\n" + ex.Message,
                        "Помилка", MessageBoxButton.OK,
                        MessageBoxImage.Error);
    }
}

private void TransactionDataGrid_MouseDoubleClick(object sender,
System.Windows.Input.MouseButtonEventArgs e)
{
```

```
            var selectedTransaction = TransactionDataGrid.SelectedItem;
            if (selectedTransaction != null)
            {
                dynamic transaction = selectedTransaction;

                var manageControl = new ManageTransactionControl(userId, connectionString);
                manageControl.LoadTransactionData(transaction.transactionID);

                var mainWindow = Application.Current.MainWindow as MainWindow;
                if (mainWindow != null)
                {
                    mainWindow.MainControl.Content = manageControl;
                }
            }
        }
    }
}
```

Додаток Б

Розгорнутий код реалізації основних модулей через ASP.NET

AuthAPI

```
using AuthAPI.Models;
using AuthAPI.Services;
using Microsoft.AspNetCore.Mvc;

namespace AuthAPI.Controllers
```

```
{
    [ApiController]
    [Route("api/[controller]")]
    public class AuthController : ControllerBase
    {
        private readonly UserService _userService;

        public AuthController(UserService userService)
        {
            _userService = userService;
        }

        [HttpPost("login")]
        public IActionResult Login([FromBody] User loginRequest)
        {
            var user = _userService.GetUserByLogin(loginRequest.login);
            if (user == null || user.password != loginRequest.password)
                return Unauthorized("Неправильний логін або пароль");

            return Ok(new { user.userID, user.login, user.contactInfo });
        }

        [HttpPost("register")]
        public IActionResult Register([FromBody] User user)
        {
            if (string.IsNullOrWhiteSpace(user.password) || user.password.Length < 6 ||
user.password.Length > 16)
                return BadRequest("Пароль повинен містити від 6 до 16 символів");

            var existing = _userService.GetUserByLogin(user.login);
            if (existing != null)
                return BadRequest("Користувач з таким логіном вже існує");
```

```
            var newUserId = _userService.CreateUser(user);
            if (newUserId == null)
                return StatusCode(500, "Помилка при створенні користувача");

            var roleCreated = _userService.CreateDbRoleForUser(user.login, user.password);

            if (!roleCreated)
                return StatusCode(500, "Помилка при створенні ролі БД");

            return Ok(new { message = "Реєстрація успішна", userID = newUserId });
        }
    }
}
```

UserService.cs

```
using AuthAPI.Models;
using Microsoft.Extensions.Configuration;
using Npgsql;
using System;

namespace AuthAPI.Services
{
    public class UserService
    {
        private readonly string _connectionString;

        public UserService(IConfiguration configuration)
        {
            _connectionString = configuration.GetConnectionString("DefaultConnection")
                ?? throw new InvalidOperationException("Строка підключення 'DefaultConnection' не знайдена.");
```

```
    }

    public User? GetUserByLogin(string login)
    {
        using var conn = new NpgsqlConnection(_connectionString);
        conn.Open();

        var cmd = new NpgsqlCommand(
            "SELECT \"userID\", login, password, \"contactInfo\" FROM \"Users\" WHERE login =
@login", conn);
        cmd.Parameters.AddWithValue("login", login);

        using var reader = cmd.ExecuteReader();
        if (reader.Read())
        {
            return new User
            {
                userID = reader.GetInt32(0),
                login = reader.GetString(1),
                password = reader.GetString(2),
                contactInfo = reader.IsDBNull(3) ? null : reader.GetString(3)
            };
        }

        return null;
    }

    public int? CreateUser(User user)
    {
        using var conn = new NpgsqlConnection(_connectionString);
        conn.Open();
```

```
        var cmd = new NpgsqlCommand(
            "INSERT INTO \"Users\" (login, password, \"contactInfo\") " +
            "VALUES (@login, @password, @info) RETURNING \"userID\";", conn);

        cmd.Parameters.AddWithValue("login", user.login);
        cmd.Parameters.AddWithValue("password", user.password);
        cmd.Parameters.AddWithValue("info", (object?)user.contactInfo ?? DBNull.Value);

        var result = cmd.ExecuteScalar();
        return result != null ? Convert.ToInt32(result) : null;
    }

    public bool CreateDbRoleForUser(string login, string password)
    {
        try
        {
            using var conn = new NpgsqlConnection(_connectionString);
            conn.Open();


            var safeLogin = "\"" + login.Replace("\"", "\"\"") + "\"";
            var safePassword = password.Replace("'", "''");
            var sql = $@"
                DO
                $$
                BEGIN
                    IF NOT EXISTS (SELECT FROM pg_catalog.pg_roles WHERE rolname =
'{login}') THEN
                        EXECUTE 'CREATE ROLE {safeLogin} LOGIN PASSWORD
"{safePassword}'";
                    END IF;
                END
```

```
            $$;
        ";

        var createRoleCmd = new NpgsqlCommand(sql, conn);
        createRoleCmd.ExecuteNonQuery();

        var grantCmd = new NpgsqlCommand($@"
            GRANT CONNECT ON DATABASE {conn.Database} TO {safeLogin};
            GRANT USAGE ON SCHEMA public TO {safeLogin};
            GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO {safeLogin};
            ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO {safeLogin};
        ", conn);

        grantCmd.ExecuteNonQuery();

        return true;
      }
      catch (Exception ex)
      {
        Console.WriteLine($"ERROR: {ex.Message}");
        return false;
      }
    }


  }
}
```