Міністерство освіти і науки України
Криворізький національний університет
Кафедра моделювання та програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти бакалавра

зі спеціальності 121 – Інженерія програмного забезпечення

На тему: Автоматизація процесів розробки та впровадження програмного забезпечення в сучасних IT-проектах

Засвідчую, що в цій
кваліфікаційній роботі немає
запозичень із праць інших авторів
без відповідних посилань.
Студент гр. ІПЗ-21-1
_____ / Б. В. Корж/

Керівник
кваліфікаційної роботи _____ / А. В. Козиков /

Завідувач кафедри _____ / А. М. Стрюк /

Кривий Ріг
2025

Криворізький національний університет

Факультет: Інформаційних технологій

Кафедра: Моделювання та програмного забезпечення

Ступінь вищої освіти: бакалавр

Спеціальність: 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

Зав. Кафедри

_____ А. М. Стрюк

«___» _____ 20__ р.

ЗАВДАННЯ

на кваліфікаційну роботу

студенту групи ІПЗ-21-1 Коржу Богдану Вячеславовичу

1. Тема: Автоматизація процесів розробки та впровадження програмного забезпечення в сучасних ІТ-проектах
2. Термін подання студентом закінченої роботи: 20.06.2025
3. Вихідні дані по роботі: розроблено веб-додаток для обміну валют з використанням сучасних підходів для автоматизації процесів розробки та розгортання. Було використано такі інструменти як GitHub Actions для CI/CD, Docker для контейнеризації, Render як хостинг-платформи, Vite для фронтенд-збірки, Supabase як база даних.
4. Зміст пояснювальної записки: провести аналіз сучасних підходів до автоматизації процесів розробки ПЗ, спроєктувати мікросервісну архітектуру веб-застосунку, реалізувати веб-додаток для обміну валют, налаштувати автоматизоване розгортання за допомогою GitHub Actions та Docker.
5. Перелік ілюстративного матеріалу: схеми архітектури мікросервісної системи, структура бази даних, схеми взаємодії сервісів, інтерфейс веб-додатку.

КАЛЕНДАРНИЙ ПЛАН

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Аналіз сучасних підходів до автоматизації розробки ПЗ	11.04.2025 – 15.03.2025
2	Аналіз архітектурних рішень для створення масштабованої системи	16.04.2025 – 18.04.2025
3	Оцінка ефективності мікросервісної архітектури в умовах реального проєкту	19.04.2025 – 22.04.2025
4	Проєктування та реалізація архітектури системи обміну валют	23.04.2025 – 29.04.2025
5	Розробка функціональних модулів та інтерфейсу користувача	30.04.2025 – 05.05.2025
6	Налаштування CI/CD, контейнеризації та автоматичного розгортання	06.05.2025 – 12.05.2025
7	Формування теоретичних основ для першого розділу роботи	13.05.2025 – 15.05.2025
8	Розробка концептуальної архітектури для другого розділу: вибір мікросервісної моделі	16.05.2025 – 20.05.2025
9	Розробка бази даних для третього розділу: опис взаємодії та механізмів обробки даних	21.05.2025 – 25.05.2025
10	Оформлення пояснювальної записки	26.05.2025 – 30.05.2025

Дата видачі завдання: «10» квітня 2025 р.

Студент _____ / Б. В. Корж /

Керівник роботи _____ / А. В. Козиков /

РЕФЕРАТ

АВТОМАТИЗАЦІЯ, CI/CD, GITHUB ACTIONS, DOCKER, VITE, RENDER, ОБМІН ВАЛЮТ, МІКРОСЕРВІСИ, ВЕБ-ДОДАТОК

Пояснювальна записка: 91 с., 25 рис., 2 додатка, 11 джерел

Метою кваліфікаційної роботи є дослідження та провадження сучасних підходів до автоматизації процесів розробки та розгортання програмного забезпечення на прикладі створення веб-застосунку для обміну валют.

У роботі використано сучасні інструменти автоматизації, такі як GitHub Actions, Docker, Vite, а також хмарна платформа Render для хостингу. Розроблено мікросервісну архітектуру веб-додатку, що складається з кількох незалежних сервісів, кожен з яких виконує окрему бізнес-функцію.

Здійснено реалізацію автоматизованого розгортання за допомогою CI/CD-процесів. Описано структуру проєкту, конфігурації контейнерів, а також взаємодію між сервісами. Проведено тестування працездатності системи після автоматичного деплою.

Проєкт ілюструє практичну користь автоматизації при створенні та підтримці веб-сервісів, показуючи, як за допомогою популярних інструментів можна налагодити зручний і надійний процес розробки ПЗ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ПЗ	Програмне забезпечення
БД	База даних
CI	Continuous Integration
CD	Continuous Delivery
IDE	Integrated Development Environment
UI	User Interface
JSON	JavaScript Object Notation
HTTP	HyperText Transfer Protocol
Веб	Інтернет-простір
API	Application Programming Interface

ВСТУП

У сучасній розробці програмного забезпечення все частіше виникає потреба у швидких змінах, частих оновленнях і роботі великих команд одночасно. Через це традиційні підходи стають незручними. Зміни впроваджуються повільно, виникають конфлікти в коді, а деплой може займати години.

Саме тому автоматизація стала необхідністю. Вона з’явилась як відповідь на потребу в гнучкості, стабільності та простоті командної розробки. Такі підходи як CI/CD дозволяють запускати тести, збирати застосунок і деплоїти його автоматично, що значно спрощує роботу.

У цій роботі я вирішив показати, як за допомогою сучасних інструментів можна побудувати повний цикл автоматизованої розробки. Для цього було створено веб-сервіс для обміну валют, у якому використано GitHub Actions, Docker, Render та інші технології.

1 ТЕОРЕТИЧНІ ОСНОВИ АВТОМАТИЗАЦІЇ РОЗРОБКИ І ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Огляд методів автоматизації в IT-проектах

1.1.1 Історія виникнення автоматизації у розробці

Ще кілька десятиліть тому розробка програмного забезпечення була складним і майже повністю ручним процесом. Код писався у текстових редакторах без підсвітки синтаксису, тестування виконувалося вручну, а для запуску додатку іноді потрібно було пів дня налаштовувати середовище. Будь-яка дрібна помилка могла зупинити весь процес, а реліз нової версії програми був подією, до якої готувалися тижнями.

З часом стало зрозуміло, що чим більше проєктів, тим більше коду, і тим вища ймовірність помилок. У той момент почала виникати ідея автоматизації, як способу позбутися повторюваних завдань, зменшити людський фактор і пришвидшити розробку. Все починалося з простих скриптів для компіляції, згодом з’явилися більш складні рішення на кшталт CI/CD-процесів. Автоматизація пройшла великий шлях, але її головна мета залишилася незмінною: зробити життя розробника простішим, а продукт стабільнішим і надійнішим[1].

1.1.2 Автоматизація як ключовий компонент сучасної розробки програмного забезпечення

Автоматизація — це сукупність процесів, інструментів і практик, що дозволяють зменшити або повністю усунути ручну участь людини у повторюваних, стандартизованих завданнях.

У розробці програмного забезпечення автоматизація охоплює багато аспектів. Наприклад, автоматичне збирання проєкту після кожної зміни в коді. Або запуск тестів, які перевіряють, чи не зламався функціонал. Також можна

налаштувати автоматичне розгортання програми на сервер, щойно вона готова до публікації. Усе це дозволяє зменшити ризик помилки, пришвидшити роботу команди і зробити процес розробки більш передбачуваним.

Ключовими поняттями у сучасній автоматизації є CI (Continuous Integration — безперервна інтеграція) та CD (Continuous Delivery або Continuous Deployment — безперервне доставлення або розгортання).

Безперервна інтеграція передбачає, що кожен розробник регулярно додає свої зміни до загального репозиторію. Після кожного оновлення автоматично запускаються процеси перевірки: збирання програми, запуск юніт-тестів, статичний аналіз коду тощо. Мета цього підходу виявляти помилки якомога раніше, ще до того, як зміни потраплять у фінальну версію продукту.

Безперервне доставлення — це наступний крок після інтеграції. Якщо всі перевірки пройдені успішно, система може автоматично підготувати програму до релізу. Це включає створення збірок, налаштування середовища, генерацію артефактів (наприклад, архівів із додатком) та інше. У випадку безперервного розгортання ці збірки одразу потрапляють на сервер або в інше продуктивне середовище без ручного втручання. Це дозволяє оновлювати продукт кілька разів на день, що особливо важливо для великих команд та швидкого реагування на потреби користувачів.

CI/CD формує нову норму у процесі розробки: швидкість, стабільність і безперервне вдосконалення стають не винятками, а очікуваним стандартом. Завдяки таким підходам автоматизація перетворюється з додаткової опції на критично важливий елемент сучасного IT-виробництва. На рисунку 1.1 зображено типовий процес CI/CD, що включає етапи інтеграції, тестування, доставки й автоматичного розгортання додатку.

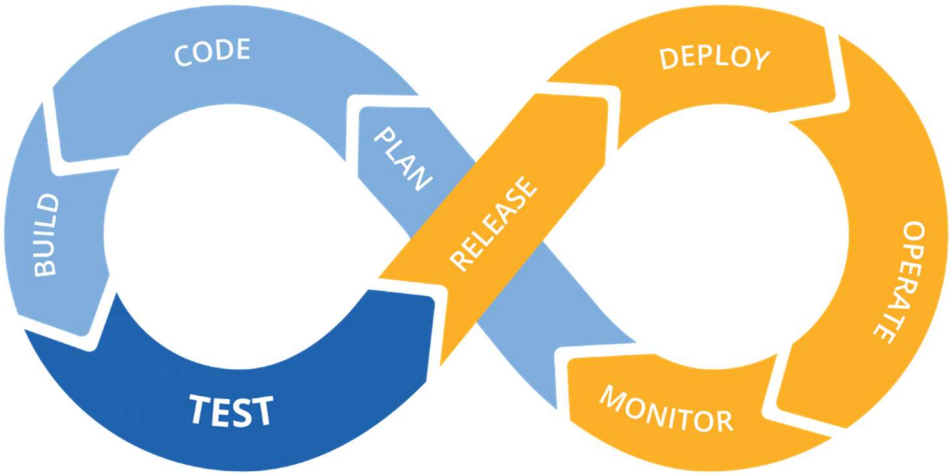

Рисунок 1.1 — Модель CI/CD процесів

1.1.3 Сучасний стан автоматизації та аналіз існуючих рішень

На сьогоднішній день автоматизація стала невід'ємною частиною процесу розробки програмного забезпечення. З постійним збільшенням складності програмних систем та швидким розвитком технологій, зростає потреба у швидкому і якісному тестуванні, інтеграції та доставці продуктів. В таких умовах автоматизація стає ключовим інструментом, який дозволяє розробникам уникати помилок, пов'язаних з людським фактором, а також значно прискорює процес створення програмного забезпечення. Автоматизація охоплює не лише процеси тестування, але й безперервну інтеграцію, розгортання і навіть моніторинг продуктивності, що робить розробку більш передбачуваною та ефективною. Сучасні інструменти автоматизації дозволяють команді зосередитись на розвитку продукту, зменшуючи рутинну роботу, що значно підвищує продуктивність і якість кінцевого результату.

Серед найбільш популярних рішень для автоматизації процесів розробки можна виділити такі інструменти, як GitHub Actions, GitLab CI/CD, Jenkins, Travis CI, Circle CI та. Кожен з них має свої особливості, що робить його підходящим для певних типів проектів, і допомагає команді ефективніше виконувати завдання.

GitHub Actions — інтегрована система CI/CD, яка вбудована прямо у GitHub. Вона дозволяє автоматизувати робочі процеси без необхідності використовувати сторонні інструменти. GitHub Actions підтримує налаштування робочих процесів на основі подій, що відбуваються в репозиторії. Наприклад, оновлення коду або створення релізу(рис. 1.2). Вона зручна і безкоштовна для відкритих проектів, що робить її ідеальним варіантом для невеликих команд або індивідуальних розробників[2].

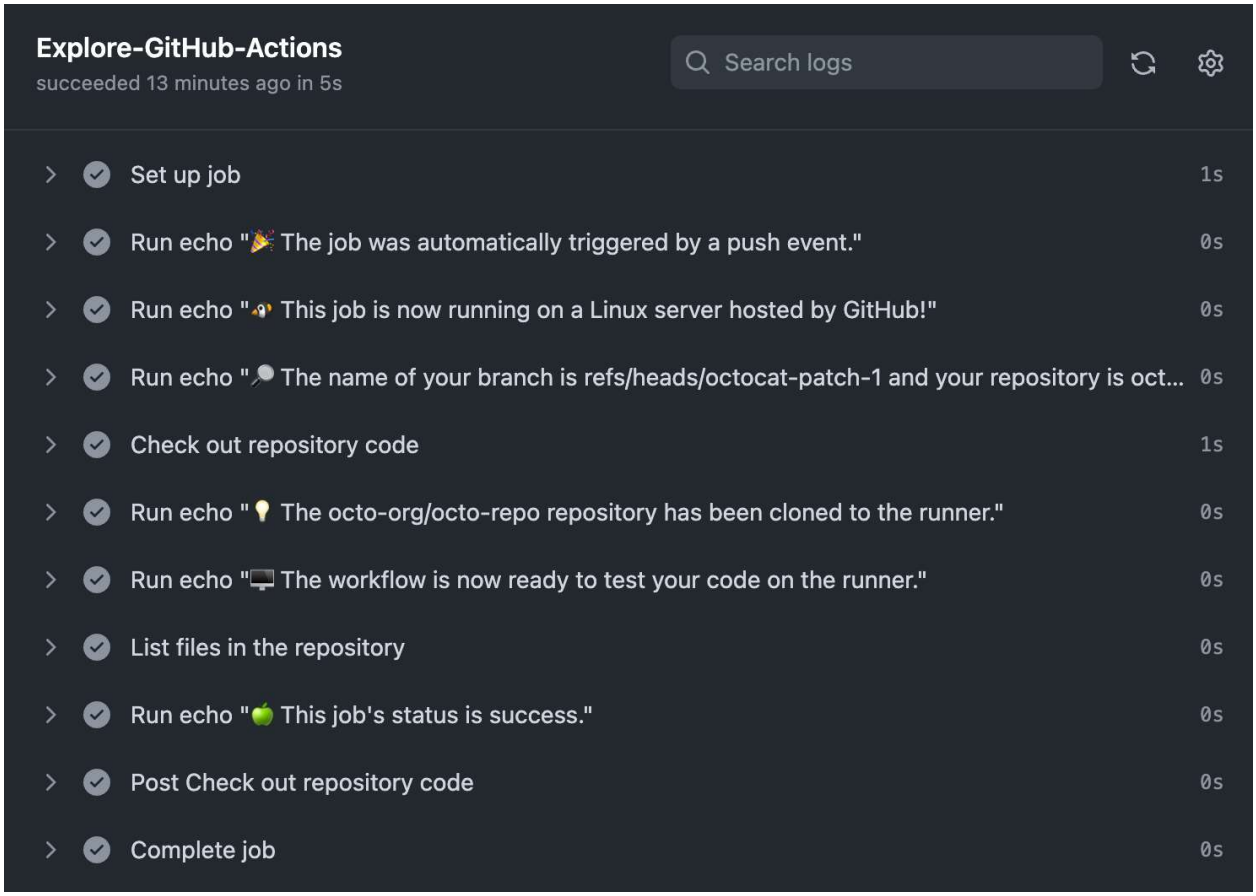

Рисунок 1.2 — Інтерфейс інструменту GitHub Actions

GitLab CI/CD — це система автоматизації(рис 1.3), вбудована прямо у GitLab. Вона дозволяє запускати збірку, тестування і розгортання одразу після змін у коді. Все налаштовується в одному файлі .gitlab-ci.yml, який зберігається в репозиторії. Це зручно, бо не потрібно використовувати сторонні сервіси. Підходить як для невеликих команд, так і для більших проєктів. А середовище GitLab дозволяє відстежувати помилки та є

можливість безкоштовного використання у приватних проєктах. Але для складніших процесів іноді потрібен досвід з написання пайплайнів.

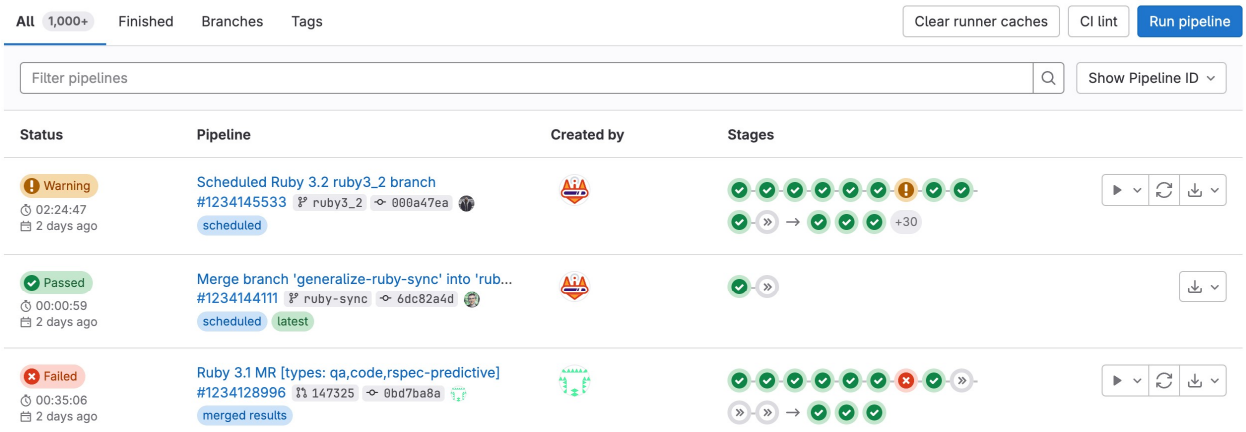

Рисунок 1.3 — Інтерфейс інструменту GitLab CI/CD

Jenkins — потужний інструмент з відкритим кодом, який дозволяє автоматизувати процеси CI/CD(рис 1.4). Jenkins підтримує велику кількість плагінів і забезпечує високу гнучкість у налаштуваннях. Підходить для великих проектів і команд, але може бути складним у налаштуванні для новачків. Головною перевагою є велика кількість доступних плагінів і гнучкість, що дозволяє налаштувати процес автоматизації під будь-які потреби. Однак, для ефективного використання потрібно багато часу на вивчення та налаштування системи.

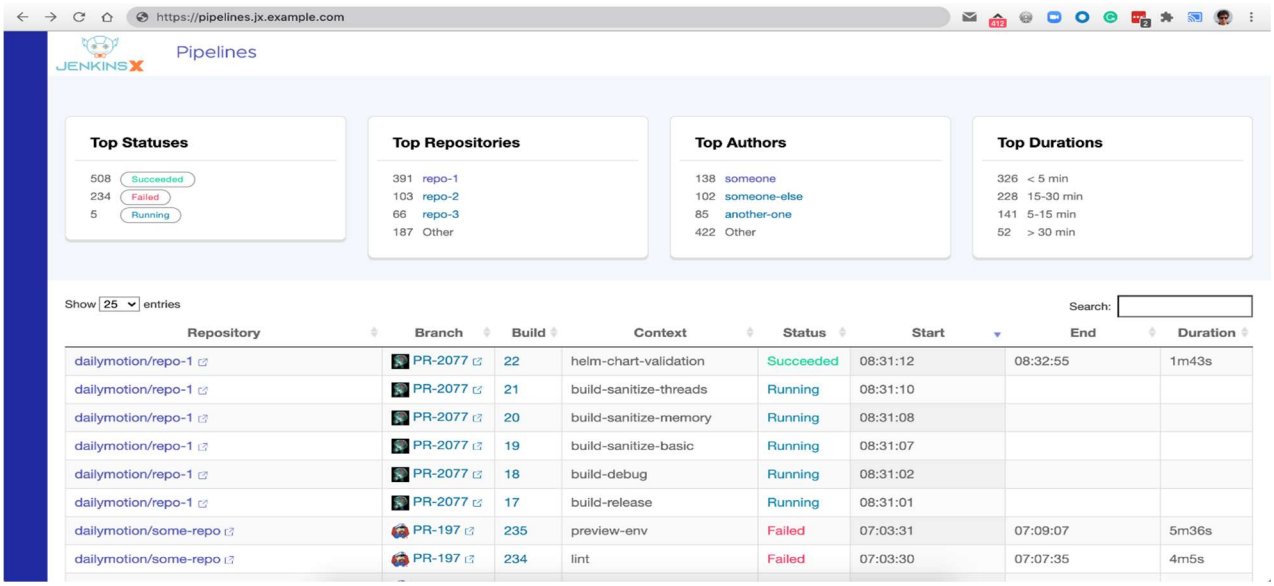

Рисунок 1.4 — Інтерфейс інструменту Jenkins

Travis CI — система для безперервної інтеграції та тестування(рис 1.5), інтегрована з GitHub. Travis CI пропонує просте налаштування для відкритих проектів і автоматично виконує тести при кожному коміті або пулл-реквесті. Він має безкоштовний план для відкритих репозиторіїв, що робить його популярним для open-source проектів. Однак для приватних репозиторіїв цей інструмент обмежений, оскільки для них необхідно підписатися на платний план. Головною перевагою є простота використання та інтеграція з GitHub, але для приватних проектів є деякі обмеження.

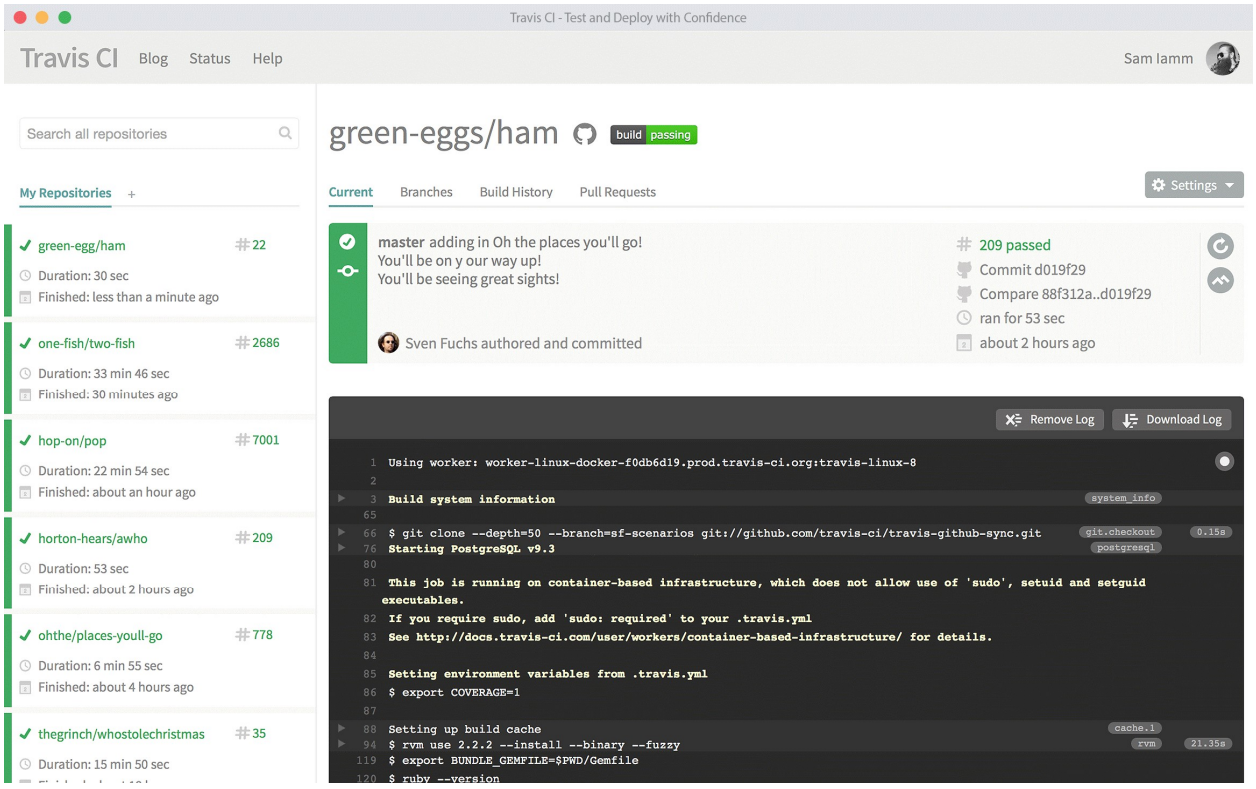

Рисунок 1.5 — Інтерфейс інструменту Travis CI

Circle CI — хмарний інструмент для безперервної інтеграції та тестування(рис 1.6), який дозволяє автоматизувати процеси виявлення помилок і тестування коду. Він популярний серед великих компаній, таких як Facebook, Spotify, Kickstarter, оскільки дозволяє швидко тестувати зміни в коді і робить процес розвитку більш ефективним. Однією з головних переваг є підтримка різноманітних тестів (unit, інтеграційних, функціональних), а також

швидкість і стабільність роботи. Проте він може бути складним для налаштування системи, особливо для новачків.

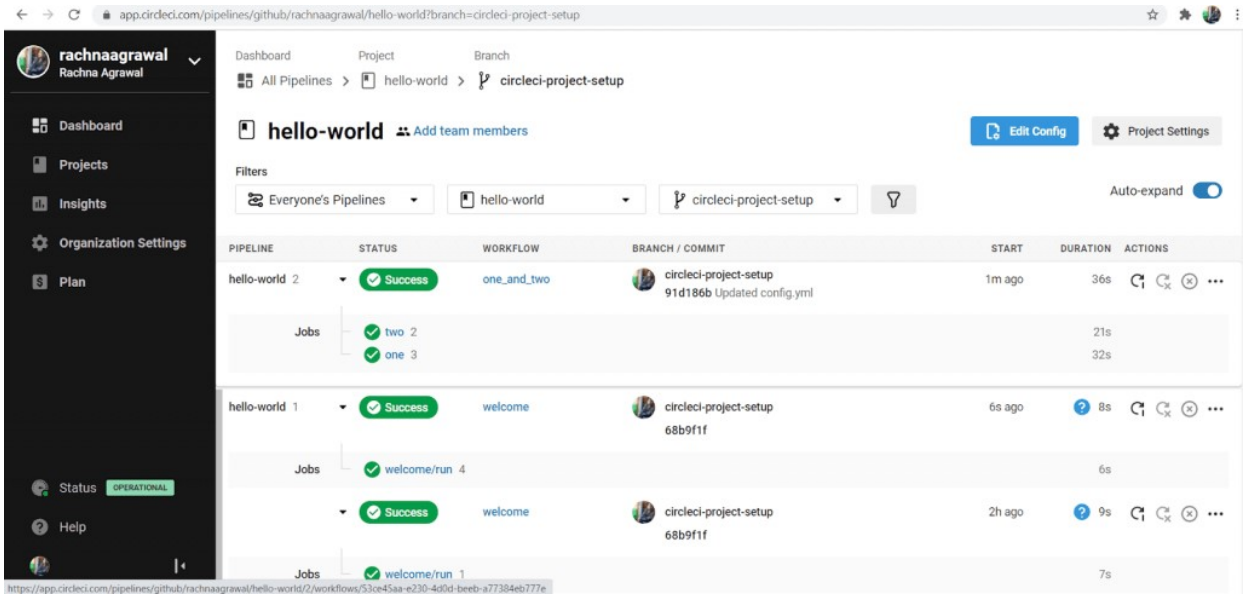

Рисунок 1.6 — Інтерфейс інструменту Circle CI

1.1.4 Важливість автоматизації у сучасній розробці

Поява DevOps-культури, безперервної інтеграції та розгортання, автоматичних тестів і моніторингу змінила очікування як з боку бізнесу, так і з боку користувача. Якщо раніше оновлення випускали раз на кілька місяців, то зараз мова йде про релізи кілька разів на день. У таких умовах людський фактор стає ризиком, який не можна ігнорувати.

Автоматизація дає змогу швидко перевіряти код, контролювати якість, тестувати на різних платформах, готувати збірки та миттєво реагувати на помилки. Це дозволяє команді залишатися конкурентоспроможною. Без автоматизації складно уявити розробку в умовах, коли кілька команд працюють паралельно над різними модулями одного продукту, коли потрібно зберігати узгодженість, синхронізацію і контроль якості.

Тому автоматизація стала не можливістю, а необхідністю. Вона вирішує не лише технічні завдання, а й організаційні, щоб забезпечити прозорість процесів, передбачуваність результатів, а саме головне стабільність в умовах постійних змін в IT середовищі.

1.2 Етапи життєвого циклу автоматизації розробки ПЗ

Життєвий цикл програмного забезпечення — це окремі послідовні етапи, які проходить продукт від моменту задуму до повноцінного використання та супроводу. У контексті автоматизації важливо не просто виконувати ці етапи вручну, а максимально автоматизувати всі повторювані дії. На рисунку 1.7 зображено етапи життєвого циклу яких слід дотримуватись при розробці програмного забезпечення.

Рисунок 1.7 — Каскадна модель життєвого циклу

Щоб краще зрозуміти суть кожного етапу, пройдемося по всіх кроках окремо. Це допоможе побачити, як усе працює в реальному проєкті, особливо коли процеси частково або повністю автоматизовані.

1.2.1 Аналіз вимог

На першому етапі потрібно чітко поставити мету програмного продукту, які цілі та завдання програмного забезпечення та його актуальність. Якщо це робота під замовлення (наприклад, фріланс або проект для компанії), то по-перше, потрібно домовитися із замовником. Треба поговорити, дізнатися, що саме він хоче бачити в результаті. По-друге, важливо розуміти, що замовник зазвичай не айтішник, і його уявлення про продукт можуть бути розмиті. Тому

необхідно знайти спільну мову, щоб і йому було зрозуміло, і нам було чітко, що треба робити.

Якщо проєкт розробляється як стартап або ініціатива з метою створення чогось нового або вдосконалення вже існуючого, то важливо звернути увагу на саму проблему та її вирішення. Потрібно розібратись, яка проблема існує, для кого цей проєкт може бути корисним, і чи є вже подібні рішення на ринку. Також важливо зрозуміти, чим наше рішення буде краще або у чому воно зможе запропонувати унікальність. Навіть без зовнішнього замовника, цей аналіз допомагає чітко визначити вимоги до проєкту і визначити, на чому ми будемо зосереджуватись у розробці.

1.2.2 Проєктування

Проєктування — це один із ключових етапів, на якому формується основа майбутнього програмного продукту. Тут ми визначаємо, з яких частин складатиметься система, як ці частини взаємодіятимуть між собою, і які технології будуть використані для реалізації задуму. Важливо одразу закласти таку структуру, яка буде зрозумілою, зручною в підтримці й здатною масштабуватись у майбутньому.

Все починається з аналізу вимог. Ми ще раз переглядаємо, що саме має робити система, і на основі цього створюємо загальну схему: модулі, база даних, API, користувацький інтерфейс. На цьому ж етапі обираються основні інструменти: мови програмування, фреймворки, бібліотеки, а також засоби автоматизації, які дозволяють пришвидшити розробку та зробити процес більш ефективним. Це можуть бути CI/CD-системи, генератори коду, інтегровані середовища розробки.

Окрему увагу варто приділити тому, як продукт виглядатиме для користувача. Інтерфейс має бути зрозумілим, логічним, зручним у повсякденному використанні. Це не менш важливо, ніж внутрішня архітектура, адже навіть якісна система без зручного доступу до функцій втрачає свою цінність.

Проєктування допомагає з самого початку уникнути багатьох помилок і труднощів. Якщо цей етап пройдений якісно, далі набагато легше реалізовувати функціонал, автоматизувати перевірки, тестування та впровадження, і швидше запускати продукт у роботу[3].

1.2.3 Розробка

На цьому етапі починається підготовка середовища. Встановлюються потрібні інструменти, налаштовуються системи контролю версій, підключаються CI/CD сервіси для автоматичного збирання, перевірки коду і розгортання. Також важливо визначити формат коду, структуру проєкту і базові правила взаємодії. Усе це є підготовкою, яка згодом економить десятки годин.

Далі розробка переходить до реалізації основної логіки. Створюються окремі модулі, компоненти інтерфейсу, обробники подій, API-запити, підключається база даних. Робота часто ведеться поетапно, тобто спочатку реалізується ключовий функціонал, а потім вводяться додаткові можливості. Паралельно формуються загальні підходи до написання коду, щоб усе залишалося зрозумілим і передбачуваним. Зміни регулярно фіксуються з коментарями, що дозволяє легко відслідковувати прогрес і швидко відкатитися у разі помилок. Завдяки CI/CD кожне оновлення автоматично перевіряється й може одразу потрапити в тестове середовище.

Після завершення основних задач настає етап тестування й доопрацювання. Фіксяться баги, покращується UX, додаються дрібні правки. Сама розробка не зупиняється і з’являються нові вимоги, щось оптимізується. Продукт продовжує розвиватися і адаптуватися до змін

1.2.4 Тестування

Тестування є найважливішим етапом, оскільки перевіряє чи відповідає продукт вимогам, чи працює кожен модуль так, як передбачено, та чи не виникають непередбачувані помилки. Тестування не повинно бути лише

фінальним кроком, а має стати частиною постійного процесу перевірки якості на всіх етапах розробки.

Перш за все, починають з написання юніт-тестів для основних функціональних блоків. Це дозволяє перевірити, чи кожна частина програми працює окремо. Автоматизація тестування на цьому етапі відіграє вирішальну роль. Тести виконуються автоматично щоразу, коли вносяться зміни в код. Це не тільки економить час, але й дає змогу оперативно виявляти помилки на ранніх етапах розробки, що знижує ризик їх появи в готовому продукті. Далі тестується інтеграція різних компонентів. Перевіряється, як модулі взаємодіють між собою, чи коректно передаються дані, чи правильно обробляються запити і відповіді. Для цього використовуються автоматизовані інтеграційні тести, що дають змогу швидко перевіряти, чи працює система в цілому, без необхідності запускати кожен компонент вручну.

Тестування не завершується після запуску продукту. В процесі його експлуатації з'являються нові помилки або проблеми, які могли залишитись непоміченими під час розробки. Саме на цьому етапі велике значення має фідбек від кінцевих користувачів. Вони можуть виявляти недоліки, які не були передбачені тестувальниками або розробниками,

1.2.5 Впровадження та підтримка

Останній етап включає в себе завершений продукт, який відповідає вимогам та готовий до впровадження. Після того як всі функціональні можливості реалізовані, протестовані та перевірені, наступає етап безпосереднього запуску. Це включає в себе налаштування необхідної інфраструктури, інтеграцію з іншими системами та надання доступу кінцевим користувачам.

Після запуску розпочинається етап підтримки, який є не менш важливим. Це не просто виправлення помилок, що можуть виникнути в процесі експлуатації, а й постійне вдосконалення продукту. На цьому етапі здійснюються оновлення, виправлення багів, додавання нових функцій, а

також адаптація продукту до змінюваних потреб користувачів та ринку. Завдяки постійній взаємодії з користувачами і збору фідбеку, продукт не лише підтримується на високому рівні, а й покращується з кожним оновленням.

Автоматизація на етапі підтримки відіграє важливу роль. Вона дозволяє за допомогою автоматизованих тестів та інструментів CI/CD швидко виявляти й усувати помилки, забезпечуючи стабільність і безперервну роботу продукту. Такий підхід гарантує, що всі компоненти працюють коректно і без конфліктів, що значно прискорює процес підтримки і оновлення продукту.

1.3 Цілі та завдання кваліфікаційної роботи

Метою кваліфікаційної роботи є продемонструвати важливість і необхідність автоматизації в сучасних проєктах. У сучасному світі, де проєкти стають дедалі складнішими, автоматизація процесів розробки, тестування, розгортання та підтримки є необхідністю. Особливо це стосується як великих, так і малих проєктів, де важливо забезпечити швидкий розвиток, стабільну роботу та високу якість кінцевого продукту.

Завданням роботи є створення демонстраційного прикладу використання технологій автоматизації на прикладі веб-сайту обміну валют. У рамках цього завдання будуть впроваджені автоматизовані процеси, що охоплюють розробку, тестування, збірку та деплой, а також їх інтеграцію з CI/CD інструментами. Зокрема, будуть налаштовані автоматизовані тести для перевірки основних функцій сайту, а також створено систему автоматичного розгортання для швидкого оновлення продукту без втрат у якості.

Отже, результатом кваліфікаційної роботи є демонстрація важливості автоматизації процесів розробки та впровадження програмного забезпечення в сучасних IT-проєктах. Використання інструментів CI/CD та інших технологій автоматизації дозволяє створювати якісні продукти в умовах постійних змін і вимог ринку. В результаті, правильне впровадження автоматизації є ключем до успіху і конкурентоспроможності проектів.

2 РОЗРОБКА СИСТЕМИ ОБМІНУ ВАЛЮТ І ІНТЕГРАЦІЯ З АВТОМАТИЗОВАНИМИ ПРОЦЕСАМИ

2.1 Огляд та вибір технологій для розробки системи обміну валют

Під час реалізації системи обміну валют я зіткнувся з необхідністю правильно підібрати технології, які б відповідали сучасним вимогам розробки. Оскільки проект передбачає взаємодію між клієнтською та серверною частинами, обробку запитів і обмін даними, а також подальше розгортання у продакшн-середовище, важливо було обрати такі рішення, які полегшать як процес створення системи, так і її підтримку надалі.

Особливу увагу я приділив можливості автоматизації процесів для уникнення людської помилки та пришвидшити розробку. З огляду на важливість правильного вибору інструментів для розробки, я зупинився на певному наборі сучасних технологій, які найкраще відповідають потребам проєкту. Для кращого розуміння їхньої ролі у створенні системи обміну валют я проведу короткий аналіз кожної технології, щоб показати їхній внесок у реалізацію проєкту:

Для серверної частини я обрав ASP.NET Core. Це сучасний фреймворк від Microsoft для розробки веб-додатків і API. Основними критеріями вибору стали його висока продуктивність, кросплатформенність, а також активна підтримка з боку спільноти. ASP.NET Core дозволяє легко створювати REST API для обробки запитів на обмін валют, працювати з базами даних через ORM-технології та реалізовувати безпеку через вбудовані механізми аутентифікації й авторизації. У процесі розробки я створив окремі ендпоінти для отримання актуальних курсів валют та проведення транзакцій, використовуючи зручну і зрозумілу архітектуру контролерів[4].

Щоб реалізувати клієнтську частину, я обрав React у поєднанні з Vite. React забезпечує компонентний підхід до побудови інтерфейсу, що дає можливість легко підтримувати та розширювати код у майбутньому. Використання Vite замість традиційних збирачів, таких як Webpack, значно

прискорило час старту проєкту і зробило розробку продуктивнішою. Vite забезпечує моментальну реакцію на зміни в коді завдяки "hot module replacement", що особливо важливо під час тестування інтерфейсних рішень. Інтерфейс системи обміну валют реалізований у вигляді SPA (Single Page Application), що забезпечує швидке реагування програми на дії користувача без потреби повного перезавантаження сторінки.

Однією з основних проблем, що часто виникають під час розробки, є різниця у конфігурації середовища на різних пристроях. Щоб цього уникнути, я вирішив використовувати Docker. Він дозволяє упаковувати застосунок разом із усіма залежностями в єдиний контейнер. Контейнеризація значно полегшує як локальну розробку, так і розгортання у хмарі. Під час роботи з Docker я створив окремі контейнери для бекенду та фронтенду, що дозволяє запускати і тестувати всю систему в ізольованому середовищі без складного локального налаштування. Крім того, завдяки використанню Docker стало можливим швидко масштабувати систему в майбутньому, якщо виникне потреба обслуговувати більшу кількість користувачів[5].

Для розгортання проєкту я обрав хмарну платформу Render. Основними перевагами Render стали:

- підтримка автоматичного розгортання після кожного оновлення в репозиторії Git;
- простий процес налаштування сервісів;
- безкоштовний план для невеликих проєктів, що дозволяє оптимізувати витрати на етапі розробки.

Під час роботи я створив окремі сервіси для фронтенду та бекенду, налаштував автоматичне оновлення контейнерів за змінами у вихідному коді та забезпечив безперервний доступ до застосунку через публічні URL.

Управління вихідним кодом у проекті здійснюється через Git, що є стандартом для колаборації в команді та забезпечує надійне збереження версій коду. Під час розробки я активно використовував можливості гілкування для розділення різних етапів роботи, наприклад, окремі гілки для розробки

фронтенду і бекенду, що дозволяло зручно тестувати нові функції без ризику пошкодити основну версію застосунку. Для автоматизації процесів тестування, збирання та розгортання застосунку я інтегрував GitHub Actions. Цей інструмент для автоматизації дозволяє запускати набори тестів, перевіряти код на наявність помилок і деплоїти нову версію на сервер після кожного оновлення репозиторію. GitHub Actions допомагає значно спростити процес розгортання та забезпечити стабільність і коректність роботи коду на всіх етапах розробки.

Таким чином, під час створення системи обміну валют я зробив свій вибір на користь ASP.NET Core, React app with Vite, Docker, Render та Git. Всі ці технології працюють у зв'язці, забезпечуючи швидку розробку, стабільну роботу застосунку, легкість масштабування та зручність підтримки. Кожен інструмент виконує свою важливу роль у проєкті, і завдяки їх поєднанню мені вдалося побудувати систему, яка відповідає сучасним вимогам до якості програмного забезпечення.

2.2 Проектування архітектури та реалізація основних функцій системи

Проектування розпочиналося з визначення базових принципів взаємодії між компонентами системи. Основою обраної архітектури стала клієнт-серверна модель, що передбачає чіткий розподіл логіки між фронтендом (користувацьким інтерфейсом) та бекендом (серверною обробкою даних). Такий підхід забезпечує зручність розробки, тестування та модернізації окремих частин системи. Під час проектування було враховано принципи мікросервісної архітектури, які дозволяють за потреби виділяти окремі компоненти у самостійні сервіси. Це забезпечує кращу масштабованість системи, підвищує її стійкість та спрощує інтеграцію нових функціональних можливостей. Процес проектування також включав вибір сучасних технологій для автоматизації розгортання та супроводу додатків[6].

У процесі розробки системи обміну валют була обрана клієнт-серверна архітектура, яка є класичним підходом для побудови веб-застосунків. межах цього підходу функціональність системи була розподілена на дві основні частини:

- клієнтська частина: реалізована у вигляді веб-застосунку, побудованого за допомогою React у поєднанні з Vite для оптимізації процесу розробки та збирання проєкту. Клієнтська сторона відповідає за відображення інформації для користувача, формування запитів до серверної частини та обробку отриманих результатів. Завдяки використанню React вдалося досягти високої швидкодії інтерфейсу та забезпечити гнучкість у розширенні функціоналу у майбутньому;
- серверна частина: реалізована на основі ASP.NET Core, що забезпечує обробку запитів клієнта, виконання бізнес-логіки та взаємодію з зовнішніми сервісами для отримання актуальних курсів валют. Використання цієї технології дозволяє створювати високопродуктивні веб-API, що легко масштабуються та інтегруються з іншими сервісами.

Також при проектуванні системи були враховані принципи мікросервісної архітектури. Її ідея полягає в тому, що окремі функціональні частини програми розділяються на незалежні сервіси, кожен з яких виконує свою чітко визначену задачу. Для забезпечення взаємодії між клієнтом та сервером використовується передача даних у форматі JSON через REST API, що забезпечує легкість у обробці інформації та високу сумісність. Такий підхід суттєво відрізняється від монолітної архітектури, де всі компоненти системи щільно інтегровані у єдине ціле[7].

У монолітних системах навіть невеликі зміни можуть потребувати повної перекомпіляції і повторного розгортання всього застосунку, що ускладнює масштабування і підтримку. Натомість мікросервісна архітектура дозволяє розвивати окремі компоненти незалежно один від одного, масштабувати лише ті частини системи, які зазнають підвищеного навантаження, та забезпечувати більшу стійкість, де збій одного сервісу не

призводить до повного виходу системи з ладу. На рисунку 2.1 зображено загальну архітектуру системи обміну валют, яка ілюструє взаємодію клієнтської частини, серверної логіки та допоміжних сервісів у рамках мікросервісної моделі[8].

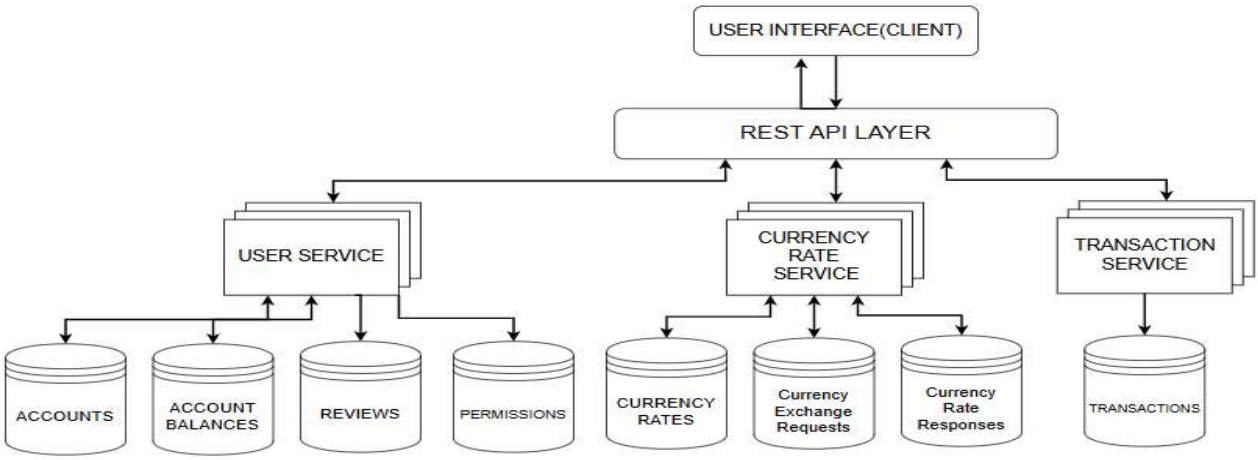

Рисунок 2.1 — Загальна архітектура системи обміну валют

Для реалізації бази даних було обрано систему управління базами даних PostgreSQL, яка завдяки своїй продуктивності та гнучкості ідеально підходить для потреб проєкту. На основі розробленої схеми (рис. 1.8) були визначені основні компоненти бази даних:

- accounts — таблиця для зберігання облікових записів користувачів. Кожен запис містить унікальний ідентифікатор користувача, логін та зашифрований пароль;
- account Balances — таблиця для зберігання поточних балансів користувачів у різних валютах. Вона пов'язана з обліковими записами за допомогою зовнішніх ключів;
- reviews — таблиця для зберігання відгуків користувачів про роботу системи. Ця функціональність дозволяє покращувати якість сервісу на основі реального досвіду користувачів;

- permissions — таблиця для зберігання інформації про права доступу користувачів, що дозволяє гнучко налаштовувати функціональні можливості облікових записів;
- currency Rates — таблиця для актуальних курсів валют. Вона є центральною для обробки запитів на обмін валют;
- currency Exchange Requests — таблиця, що відповідає за зберігання запитів користувачів на обмін валют;
- currency Rate Responses — таблиця, що відповідає за зберігання курсів валют до певної валюти. Завдяки цьому система може обробляти як миттєві операції;
- transactions — таблиця для реєстрації всіх транзакцій обміну валют. Вона містить інформацію про вихідну і цільову валюти, суми обміну та час виконання операції.

Побудована база даних дозволяє системі надійно працювати з обліковими записами, транзакціями та курсами валют.

Під час розробки системи обміну валют було реалізовано три окремих сервіси, кожен з яких відповідає за свій набір функціональностей. Такий підхід дозволяє не тільки краще організувати бізнес-логіку проєкту, а й спростити масштабування і обслуговування системи в майбутньому.

User Service відповідає за управління обліковими записами користувачів та авторизацію доступу до системи. Він забезпечує безпечну реєстрацію, автентифікацію та контроль прав доступу. Важливою функцією також є обробка відгуків користувачів для покращення якості сервісу. Сервіс інтегровано з механізмом генерації JWT-токенів для безпечної взаємодії в межах системи.

Основні функції сервісу:

- реєстрація нових користувачів: під час реєстрації сервіс зберігає у базі даних облікові дані нового користувача: логін та пароль у хешованому вигляді;

- аутентифікація користувачів: при вході у систему сервіс перевіряє надані облікові дані. У разі успіху генерується JWT-токен, який дозволяє користувачеві взаємодіяти із захищеними маршрутами без необхідності повторної авторизації;
- управління ролями: сервіс реалізує механізм ролей користувачів, що дозволяє контролювати рівень доступу до певних функцій у системі;
- збір відгуків: користувачі можуть залишати відгуки про якість роботи сервісу. Відгуки зберігаються у базі даних і можуть бути використані для подальшого аналізу й поліпшення системи.

Currency Rate Service реалізує функціональність, пов’язану з обробкою інформації про курси валют та обмінними операціями. Він є ключовим елементом, що забезпечує актуальність і точність фінансових даних, які використовуються в системі. Сервіс дозволяє користувачам швидко отримувати дані про поточні валютні курси та зручно здійснювати обмін. Його робота базується на інтеграції з зовнішніми фінансовими ресурсами для оновлення інформації в режимі реального часу. Таким чином, Currency Rate Service гарантує надійну фінансову аналітику для всіх операцій.

Основні функції сервісу:

- отримання актуальних курсів валют: сервіс надає можливість отримати останні курси валют, що є важливою частиною для правильного розрахунку суми обміну;
- пошук курсів окремої валюти: користувач може дізнатися актуальний курс конкретної валюти щодо іншої валюти, що робить обмін зручнішим і прозорішим;
- створення запитів на обмін валюти: користувачі можуть формувати власні запити на обмін, вказуючи бажану валюту і суму обміну. Сервіс обробляє ці запити і готує відповіді на основі поточних ринкових даних.

Transaction Service обробляє фінансові операції користувачів, пов'язані з обміном валют. Цей сервіс є центральною частиною логіки обробки транзакцій у системі. Він забезпечує правильність проведення обмінів,

перевіряє баланси та актуалізує залишки коштів на рахунках користувачів. Висока точність роботи Transaction Service критично важлива для запобігання помилкам у фінансових операціях і збереження цілісності даних. Завдяки йому забезпечується стабільна і безпечна обробка усіх обмінних запитів.

Основні функції сервісу:

- створення транзакцій обміну: після підтвердження запиту на обмін валюти створюється новий запис транзакції, де зберігається інформація про валюти, суми, курси та час операції;
- перевірка балансу користувача: сервіс перед проведенням транзакції перевіряє наявність достатньої кількості коштів на рахунку користувача, щоб уникнути ситуацій із перевищенням доступного балансу;
- оновлення залишків: після успішного завершення транзакції система автоматично змінює баланси відповідних рахунків користувача в базі даних.

У цьому розділі було розглянуто архітектуру системи, основні компоненти бази даних та функціональність кожного сервісу. Завдяки поділу на окремі сервіси вдалося чітко розмежувати обов’язки між частинами системи, що спрощує підтримку та розвиток. Проєктування бази даних дозволило структурувати інформацію та забезпечити надійне зберігання даних. Реалізовані сервіси взаємодіють між собою через чітко визначені запити, що забезпечує стабільну роботу всієї системи. Отже, на цьому етапі було створено надійну основу для подальшої розробки та розширення функціоналу.

2.3 Розробка інтерфейсу програми

Для розробки інтерфейсу програми було обрано сучасний інструментарій, а саме React у поєднанні з Vite. — високопродуктивним середовищем для розробки фронтенду. Vite є збирачем нового покоління, який відмовився від повної попередньої збірки на користь використання нативних ES-модулів у браузері, що дозволяє суттєво пришвидшити старт проєкту та

реакцію на зміну коду. Завдяки механізму hot module replacement (HMR) Vite забезпечує миттєве оновлення інтерфейсу під час редагування компонентів без перезавантаження всієї сторінки, що робить процес розробки не лише швидшим, а й комфортнішим для розробника. Крім того, Vite має мінімалістичну конфігурацію та добре інтегрується з екосистемою React, забезпечуючи швидку збірку у продакшн, оптимізацію залежностей та підтримку сучасних стандартів JavaScript. Усе це робить його зручним інструментом для побудови динамічного інтерфейсу користувача, який взаємодіє з бекендом системи через REST API.

Сам інтерфейс побудований за принципами компонентної архітектури, що дає змогу розділити функціональність програми на окремі логічні блоки. Це спрощує підтримку, дозволяє перевикористовувати код і поступово масштабувати застосунок. Кожен компонент відповідає за певну ділянку логіки, наприклад, відображення таблиці курсів або форму входу. Взаємодія між компонентами реалізована через передачу даних за допомогою пропсів і контекстів, що забезпечує гнучке реагування на зміну стану та синхронну роботу інтерфейсу.

На рисунку 2.2 представлено структуровану схему головної сторінки вебзастосунку, яка є основним компонентом "App" і забезпечує структуру та організацію всіх елементів інтерфейсу. У верхній частині інтерфейсу розташовано логотип, який водночас виконує функцію переходу на головну сторінку. Поруч знаходиться навігаційне меню, що дозволяє швидко переміщатися між ключовими розділами: курсами валют, особистим кабінетом, секцією з перевагими сервісу та формою для залишення власного відгуку. Доступ до особистого кабінету надається лише після реєстрації або авторизації, що здійснюється через відповідні розділи на головній сторінці, забезпечуючи безпечний доступ до персональних даних користувача. Компонент "HomePage" включає в себе центральну частину сторінки, яка займає таблиця з актуальними валютними курсами. Поряд реалізовано валютний калькулятор, що забезпечує можливість швидкого розрахунку

обмінних операцій. Також блок з коротким описом переваг сервісу, який формує у користувача первинне уявлення про функціональність і надійність платформи. Наприкінці сторінки секція з відгуками клієнтів, які вже скористалися сервісом, а також інтерактивна форма для подання нового відгуку.

Рисунок 2.2 — Схема головної сторінки вебзастосунку

Компонент "Login", показаний на рисунку 2.3, відповідає за процес автентифікації користувача, що є важливою частиною безпеки вебзастосунку.

Вікно компонента містить два основні поля: для введення логіну та пароля. У нижній частині форми розташована кнопка для входу, яка ініціює процес перевірки введених даних. Якщо введені дані не відповідають записам у базі даних, система виводить повідомлення про некоректні дані, що дозволяє користувачу усвідомити помилку. У разі успішної автентифікації користувача система автоматично перенаправляє на сторінку особистого кабінету.

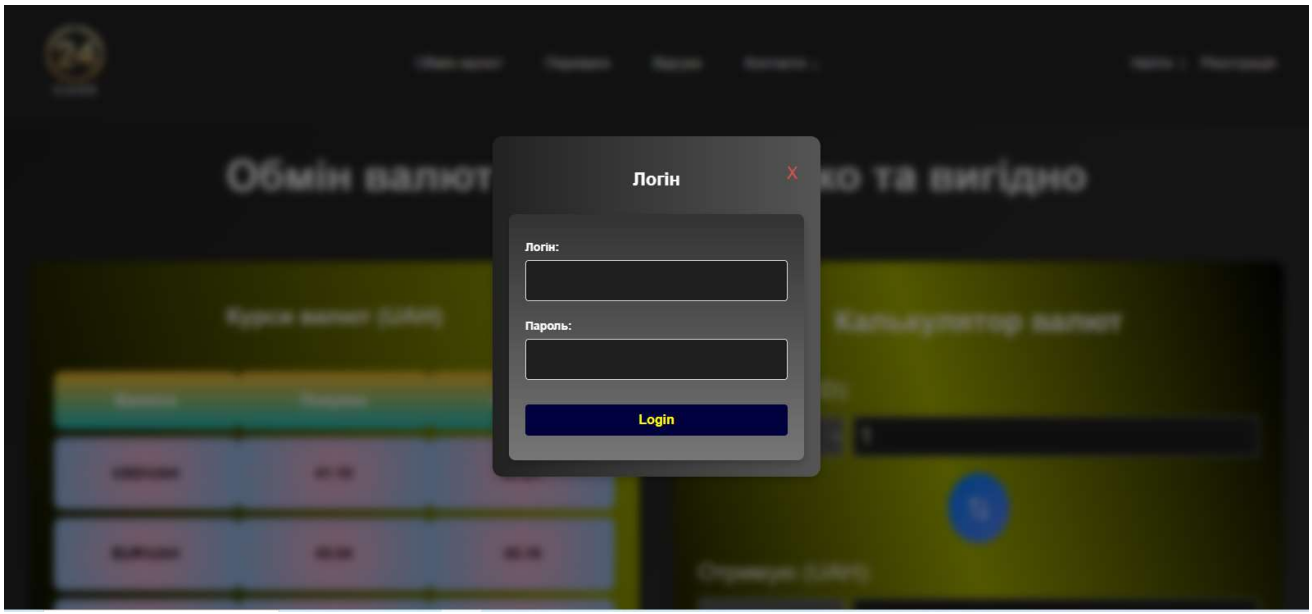

Рисунок 2.3 — Модульне вікно автентифікації користувача

Компонент "Register", показаний на рисунку 2.4, відповідає за процес реєстрації нового користувача в системі. Вікно компонента містить поля для введення логіну, пароля та підтвердження пароля. Користувач має ввести свій логін, пароль, а також повторно вказати пароль для перевірки на відповідність введених даних. У разі невідповідності паролів система виводить повідомлення про помилку, що дозволяє уникнути помилок при реєстрації. Після заповнення всіх полів користувач натискає кнопку реєстрації, яка ініціює перевірку введених даних. У разі успішної реєстрації система автоматично перенаправляє до особистого кабінету.

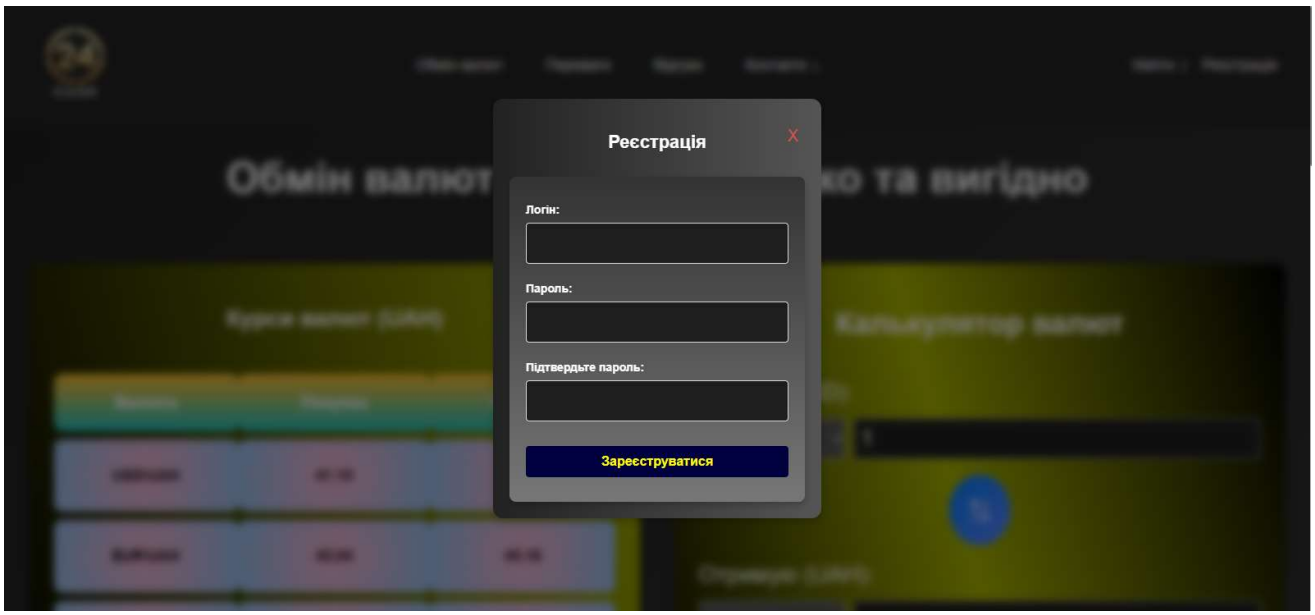

Рисунок 2.4 — Модульне вікно реєстрації користувача

Компонент "PersonalAccount", показаний на рисунку 2.13, відповідає за відображення особистої інформації користувача після успішної автентифікації. Після того, як користувач успішно входить у систему, вся ця інформація стає доступною в особистому кабінеті. Вікно компонента надає доступ до різних функцій, зокрема до інформації про активні запити на обмін, поточний стан рахунку та історію операцій. Користувач може переглядати всі свої попередні транзакції, статус запитів на обмін валют та баланс на рахунку. Крім того, в компоненті передбачено можливість редагувати пароль, що дає користувачу змогу оновлювати свої дані для покращення безпеки облікового запису. Також реалізовано функції поповнення рахунку та виводу коштів з гаманця, що дозволяє повноцінно керувати власними фінансами.

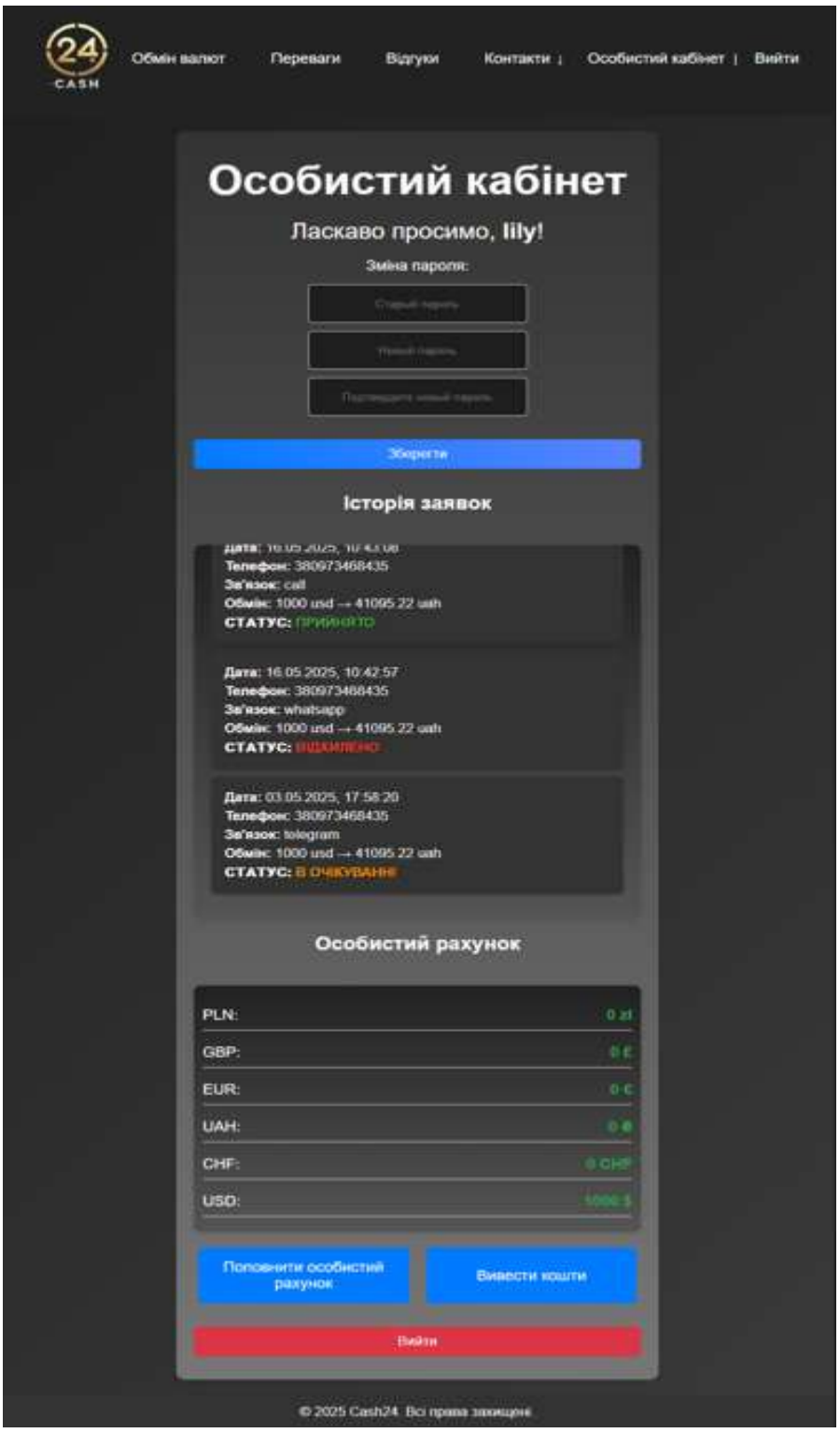

Рисунок 2.5 — Сторінка особистого кабінету

Компонент "CurrencyCalculator", зображений на рисунку 2.6, виконує роль інтерактивного калькулятора для розрахунку обмінного курсу між обраними валютами. Вікно компонента містить два випадаючих списки для вибору валюти, поле введення суми, яку користувач бажає обміняти, та поле з результатом перерахунку відповідно до актуального курсу. Усі обчислення

відбуваються в режимі реального часу, що забезпечує зручність та оперативність під час використання сервісу. Калькулятор взаємодіє з таблицею валют і динамічно підтягує актуальні значення курсів, дозволяючи користувачеві одразу бачити, скільки він отримає в результаті операції. Такий підхід значно спрощує процес планування обміну і знижує ймовірність помилки при розрахунках.

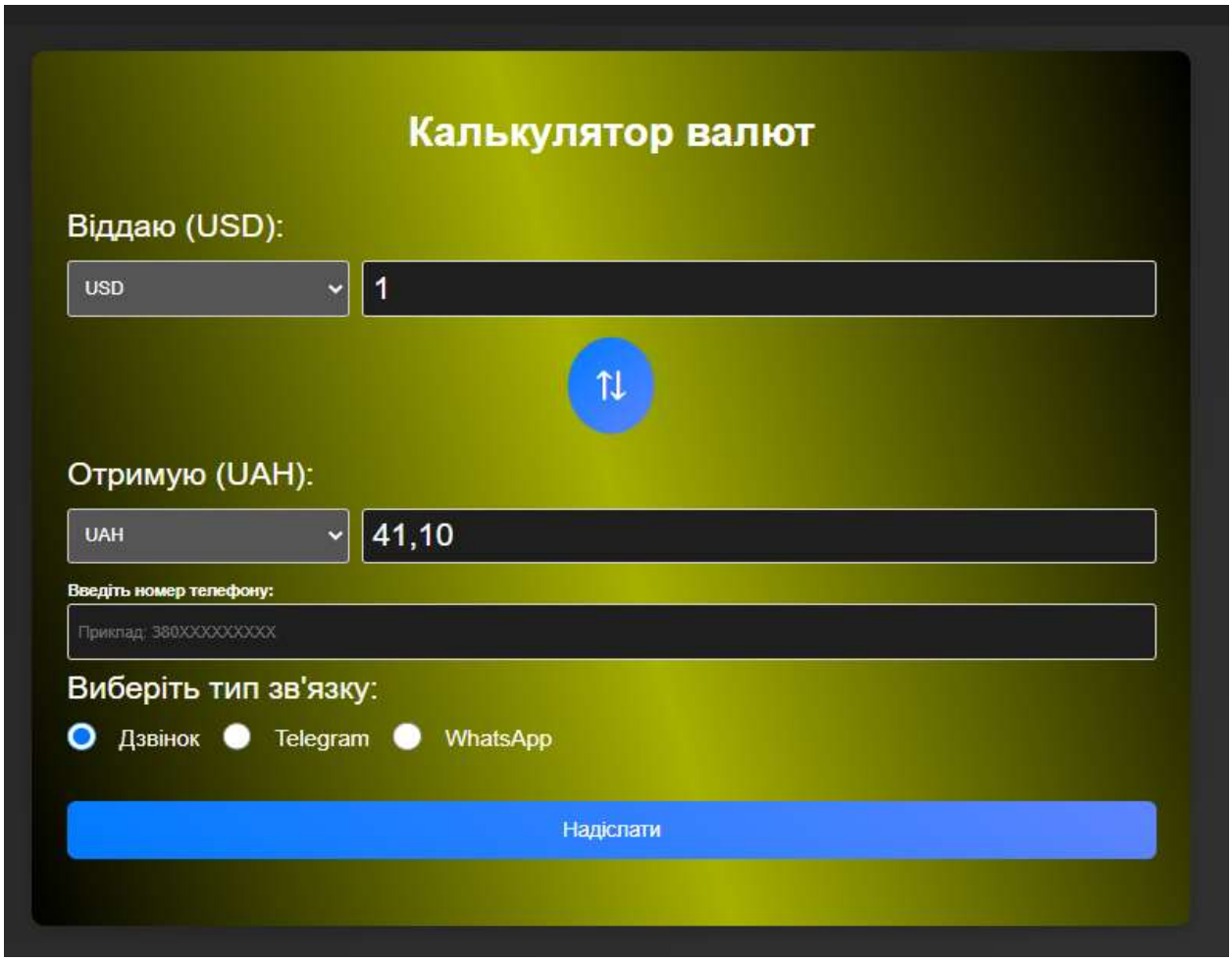

Рисунок 2.6 — Калькулятор валют

Компонент "CurrencyTable", який показано на рисунку 2.7, відповідає за відображення актуального курсу валют і дозволяє користувачам швидко орієнтуватися у співвідношенні гривні до інших валют. У таблиці подано співвідношення валют у форматі UAH/інша валюта, де зазначено курс купівлі та продажу для кожної пари. Дані таблиці автоматично оновлюються в реальному часі, що дозволяє користувачеві отримувати актуальну інформацію

без необхідності оновлювати сторінку. Цей компонент є основою для калькулятора обміну валют, де користувачі можуть здійснювати розрахунки на основі виведених курсів, що дає можливість швидко отримати точні результати обміну.

Рисунок 2.7 — Таблиця курсів валют

Компонент "PaymentModal", показаний на рисунку 2.8, відповідає за обробку платіжних операцій через модальне вікно. Він має два основні стани: поповнення та виведення коштів. Це вікно дозволяє користувачу вибрати тип картки (Visa або MasterCard), ввести номер картки та секретний код (CVC). Також передбачено поле для вибору валюти через випадаючий список та поле для введення суми, яку користувач планує вивести або поповнити.

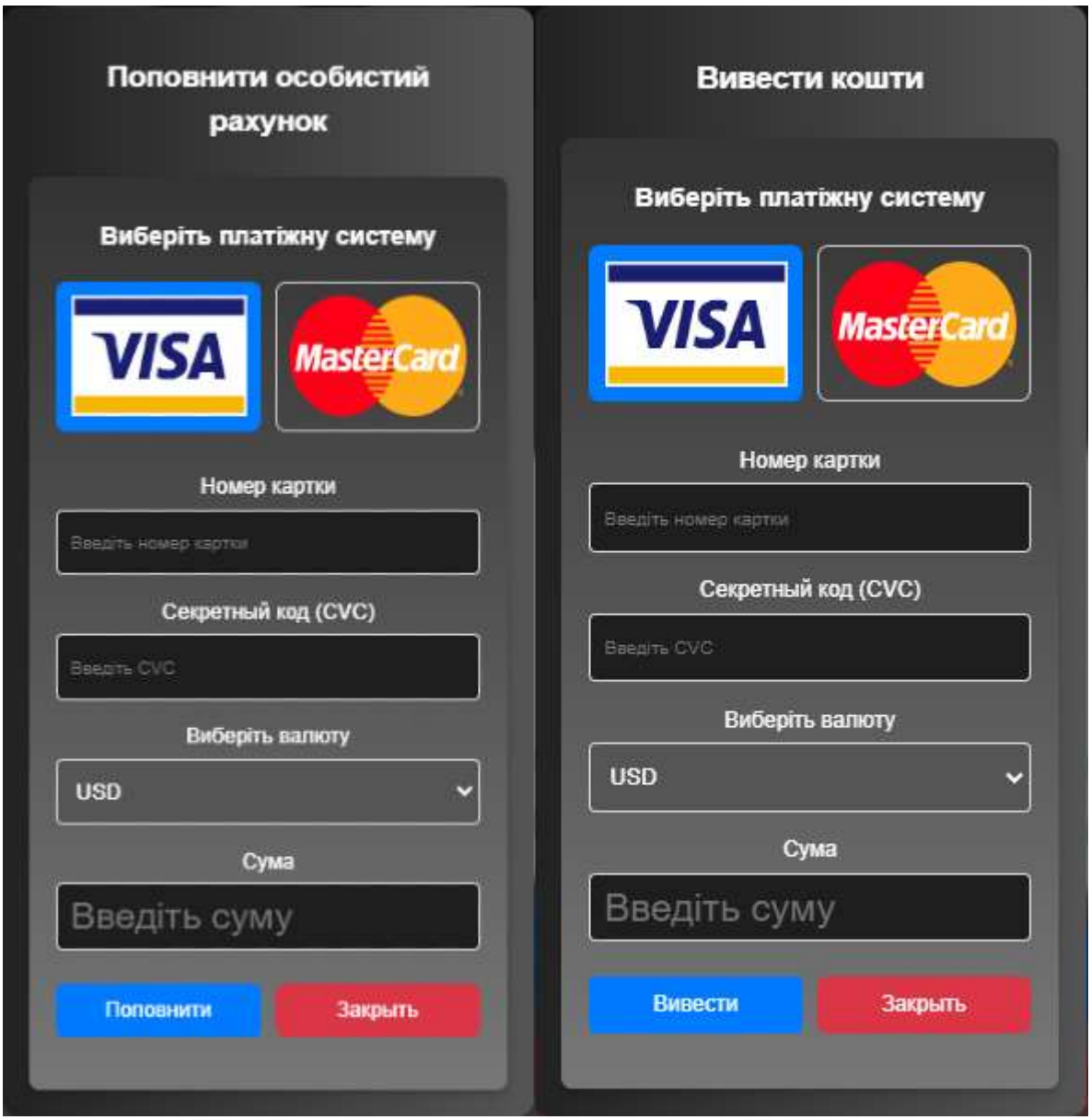

Рисунок 2.8 — Модульні вікна поповнення та виведення коштів

Компонент "ReviewSystem", показаний на рисунку 2.9, реалізує функціональність взаємодії користувачів із системою через відгуки. Головною його метою є надати можливість залишити власну думку про сервіс та ознайомитися з враженнями інших клієнтів. Компонент складається з двох основних частин: відображення вже опублікованих відгуків та форми для створення нового. У блоці відображення демонструються імена користувачів, короткий текст відгуку, рейтинг у вигляді зірок від 1 до 5, а також дата публікації, що створює враження активного та живого спілкування. Форма залишення відгуку містить текстове поле, селектор для вибору кількості зірок,

а також кнопку для підтвердження надсилання. Окрім додавання, користувач має можливість змінювати або видаляти власний відгук, що забезпечує гнучкість та контроль над особистим контентом. Усі нові повідомлення одразу з’являються в списку після успішного додавання. Такий підхід дозволяє підтримувати прозорість сервісу, а також сприяє покращенню довіри серед користувачів.

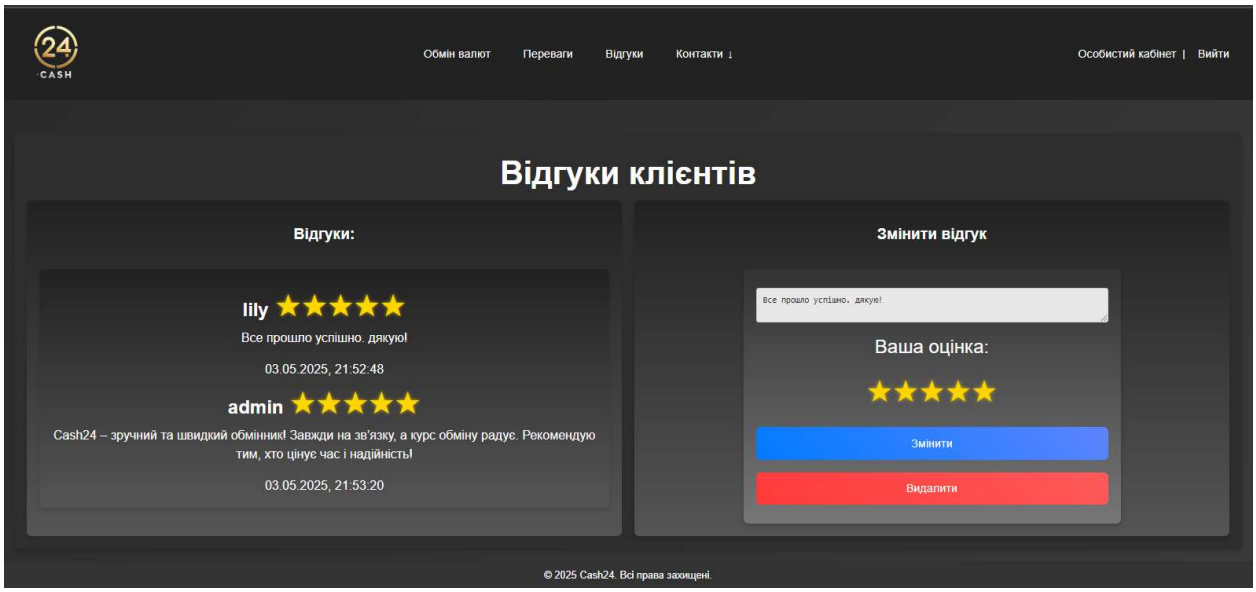

Рисунок 2.9 — Інтерфейс системи відгуків користувачів

3 АВТОМАТИЗАЦІЯ ПРОЦЕСІВ РОЗГОРТАННЯ І ВПРОВАДЖЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Налаштування середовища розробки та контейнеризація за допомогою Docker

У процесі розробки даного програмного забезпечення постала потреба у створенні зручного та керованого середовища, яке б дозволяло ізолювати сервіси, швидко запускати систему в різних умовах та уникати проблем із несумісністю залежностей. Оскільки обрана архітектура побудована на мікросервісному підході, кожен компонент системи виконує окрему функцію та потребує власного оточення для коректної роботи. Саме тому я вирішив застосувати технологію контейнеризації.

Контейнеризація — це процес упаковки програмного забезпечення разом із усіма необхідними залежностями в єдиний контейнер, який можна запустити на будь-якій машині з підтримкою відповідного середовища. Основним інструментом, який я використовував для цього, став Docker. Це сучасна платформа, що значно спрощує розгортання застосунків і дозволяє забезпечити однакову поведінку програмного продукту незалежно від платформи.

У якості основного середовища розробки я використовував Visual Studio 2022. Вибір саме цієї IDE був обумовлений тим, що вона має зручну інтеграцію з Docker, підтримує проєкти на .NET та дозволяє працювати з кількома сервісами в межах одного рішення. Крім того, Visual Studio 2022 значно зручніша у використанні ніж Visual Studio Code , оскільки більшість необхідних залежностей вже інтегровані, немає проблем з ручною конфігурацією, а менеджер пакетів NuGet забезпечує швидке та стабільне підключення зовнішніх бібліотек. Завдяки цьому я зміг організувати структуру рішення таким чином, щоб кожен мікросервіс мав окремий Dockerfile, налаштований відповідно до його потреб.

Під час розробки для кожного окремого мікросервісу було створено свій Dockerfile, який описує інструкції для побудови контейнера. Це дало змогу не лише ізолювати кожен сервіс, а й зробити їх незалежними від середовища, в якому вони запускаються. Описані інструкції включали копіювання файлів проєкту, встановлення залежностей, збірку та запуск, що дозволило отримати повноцінні контейнери, готові до розгортання.

Також під час налаштування окремих сервісів я використовував файл .env для зберігання змінних середовища. У ньому зберігалися ключові конфігураційні дані, зокрема рядки підключення до бази даних та API-ключі, які використовувалися сервісами. Такий підхід дозволив уникнути хардкодування чутливих даних у проєкті, що є критично важливим з міркувань безпеки, особливо в умовах продакшн-середовища.

Вибір обраних інструментів і підходів суттєво покращив процес розробки та розгортання проєкту. Завдяки ізольованому середовищу для кожного сервісу вдалося уникнути конфліктів між залежностями, що могло б ускладнити роботу з різними частинами проєкту. Це забезпечило стабільність і передбачуваність роботи кожного мікросервісу, що було критично важливо для подальшого масштабування та підтримки системи. Крім того, збереження чутливих даних в окремих конфігураційних файлах дозволило знизити ризики витоку інформації, забезпечивши безпеку і гнучкість у налаштуваннях для різних середовищ. Вибір середовища для розробки дозволив значно спростити роботу з різними частинами проєкту, забезпечивши зручний інтерфейс для керування мікросервісами, що сприяло швидшому виконанню завдань. В результаті, усе це дозволило зосередитись на основній логіці проєкту, мінімізуючи проблеми з налаштуванням середовища і забезпечуючи швидкість та безпеку роботи.

3.2 Моделювання та впровадження бази даних у прикладному середовищі

У прикладному середовищі системи використовується база даних PostgreSQL. Вона забезпечує надійне зберігання інформації, необхідної для роботи сервісів. Також PostgreSQL надає потужні можливості для автоматизації обробки даних. На рисунку 3.1 зображено структуру бази даних, яка містить таблиці, що зберігають дані про користувачів, транзакції, валютні курси та відгуки. Між таблицями реалізовано логічні зв’язки для підтримки цілісності даних і спрощення доступу до пов’язаної інформації[9].

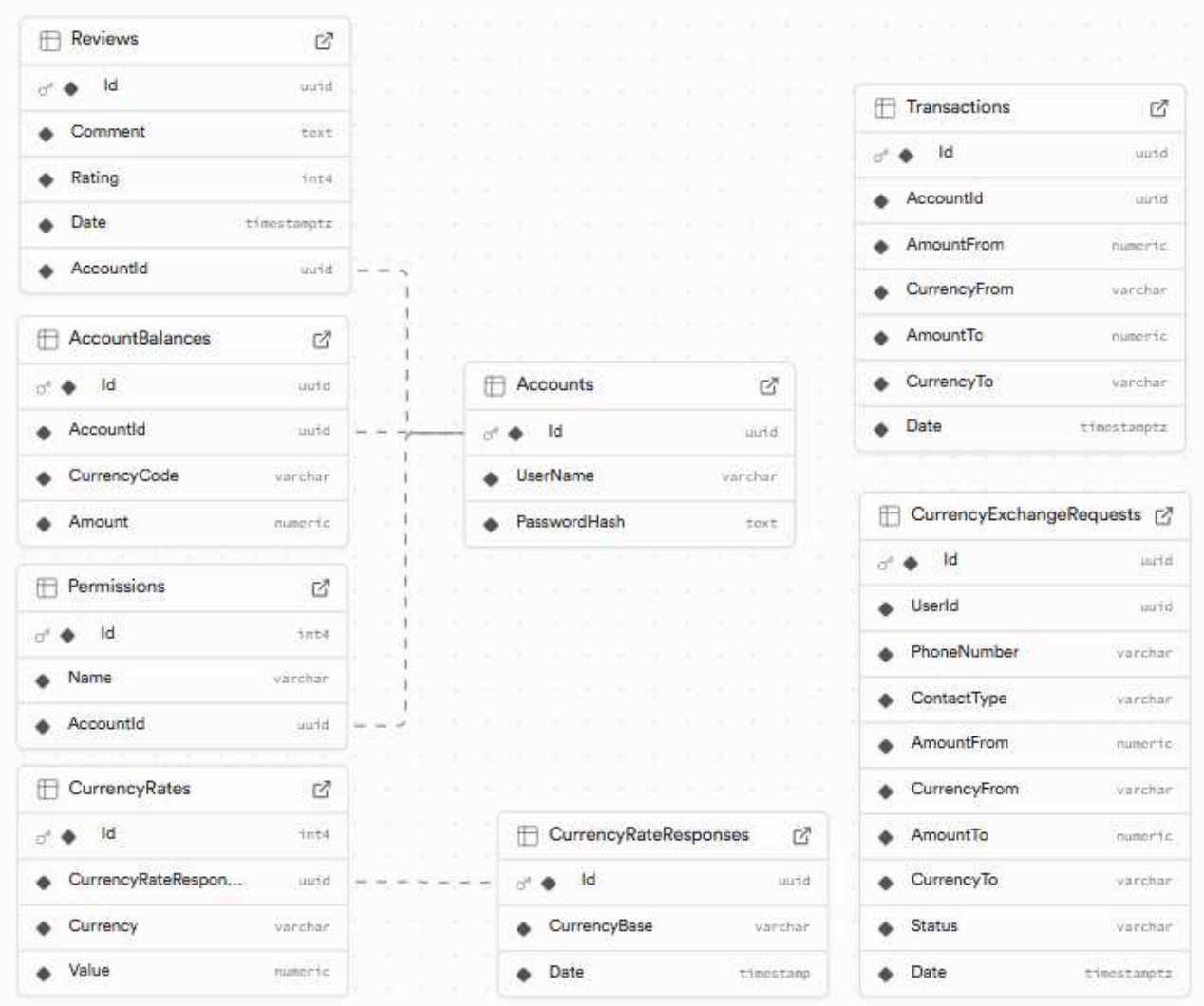

Рисунок 3.1 — Структура бази даних проєкту

Проєктування структури бази даних здійснювалося з урахуванням мікросервісної архітектури системи. Кожен сервіс взаємодіє лише з тими

таблицями, які відповідають його функціональності. Для підключення до бази даних використовувалися змінні середовища, що дозволило легко налаштовувати з'єднання для середовищ розгортання та розробки. У графічному середовищі адміністрування бази даних PGAdmin було застосовано вбудований функціонал календаря для автоматичного видалення старих записів курсу валют. Це допомагає підтримувати базу даних актуальною та знижує навантаження на систему.

Для взаємодії з базою даних було використано ORM (Object-Relational Mapping) — це технологія, що дозволяє працювати з базою даних на рівні об'єктів програмування, а не через SQL-запити. За допомогою ORM програма може використовувати об'єкти для маніпулювання даними, які зберігаються в реляційних таблицях. Це дозволяє абстрагуватися від прямого виконання SQL-запитів, що значно спрощує розробку та підвищує безпеку коду. ORM автоматично генерує SQL-запити для виконання операцій з базою даних, що знижує ймовірність помилок, спричинених неправильним написанням запитів. Однією з основних переваг ORM є зручність у роботі з даними, адже розробник працює з об'єктами, а не з таблицями, що дозволяє зменшити кількість коду та зробити його більш читабельним і зрозумілим.

Для реалізації було обрано Entity Framework. Це один із найпопулярніших ORM для .NET, який дозволяє розробникам працювати з базою даних, використовуючи об'єкти C#. Entity Framework автоматизує процес перетворення об'єктів на відповідні записи в таблицях бази даних, а також підтримує зв'язки між об'єктами та таблицями. Однією з головних переваг Entity Framework є його здатність працювати з різними типами баз даних, такими як SQL Server, PostgreSQL і інші, завдяки різноманітним провайдерам. Entity Framework надає зручний і потужний інтерфейс для виконання CRUD-операцій (створення, читання, оновлення, видалення) без необхідності написання SQL-запитів вручну.

Отже, база даних є ключовим елементом будь-якого додатку, оскільки без неї програма не може ефективно зберігати, обробляти та надавати

необхідну інформацію. Вибір правильної бази даних для проєкту впливає на його стабільність, масштабованість та ефективність. Тому важливо враховувати вимоги проєкту при виборі СУБД, оскільки кожна з них має свої переваги та можливості. Надійна база даних забезпечує безперебійну роботу сервісів і дозволяє легко впроваджувати новий функціонал. Її продумана структура спрощує подальшу підтримку та розвиток додатку.

3.3 Використання CI/CD для автоматичного деплою та тестування

GitHub — це зручний веб-сервіс для хостингу проектів, який дозволяє автоматизувати різні процеси прямо в репозиторії за допомогою GitHub Actions. Завдяки спеціальним файлам workflow, які зберігаються у папці .github/workflows, можна налаштувати запуск процесів при push, pull request або навіть за розкладом. Під час створення нового workflow GitHub одразу пропонує великий вибір шаблонів для різних мов і задач. Наприклад, для .NET уже є готовий шаблон, який автоматично виконує команди restore, build та test для проекту. Це дуже зручно, бо дозволяє швидко розпочати без складних налаштувань. У workflow можна використовувати змінні, секрети, умови виконання й інші функції, які допомагають будувати складні й гнучкі процеси. Також я працював з платформою Render, яка сама по собі підтримує автоматичний деплой при кожному push у репозиторій. Але Render не дає такого гнучкого контролю, як GitHub Actions. Наприклад, не можна додати етапи тестування або ручного затвердження. Такий підхід дозволяє краще контролювати якість і уникати помилок при розгортанні проєкту.

На рисунку 3.2 представлено набір стандартних шаблонів GitHub Actions, що використовуються для виконання типових етапів життєвого циклу розробки програмного забезпечення.

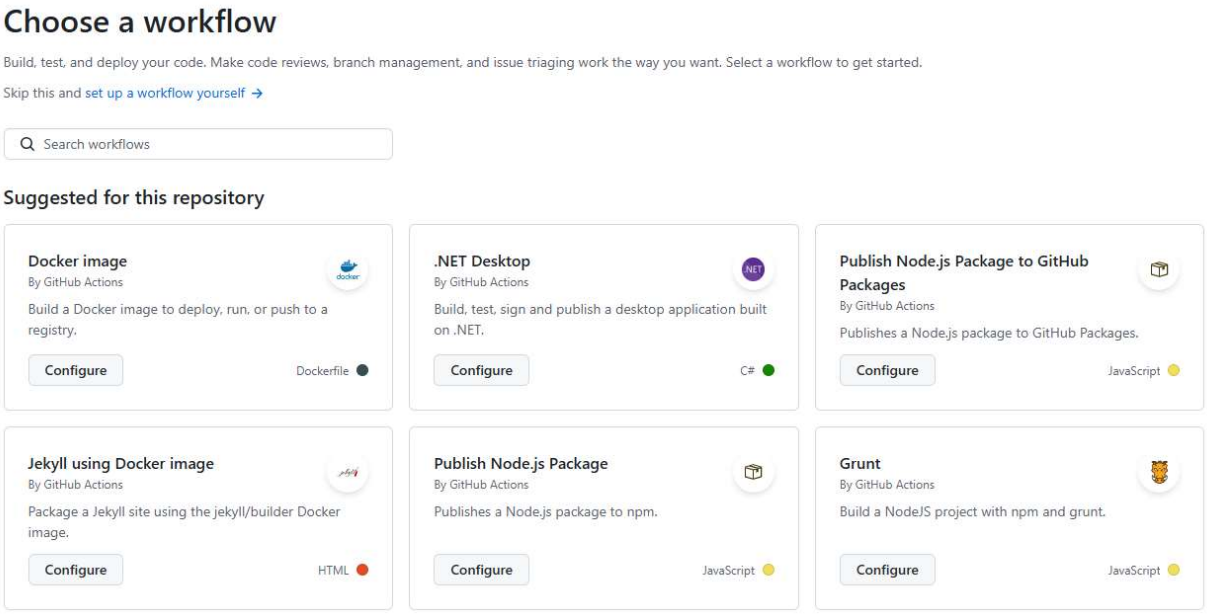

Рисунок 3.2 — Готові шаблони GitHub Actions для автоматизації задач

У проекті було реалізовано процес CI за допомогою GitHub Actions. Для цього я створив окремий файл deploy.yml(рис 3.3), який знаходиться в папці .github/workflows. У цьому файлі налаштував запуск автоматичних дій при кожному push та pull request у гілку main. Основні етапи процесу CI включають: відновлення залежностей, складання проєкту та запуск юніт-тестів. Спочатку запускається офіційний образ ubuntu-latest, після чого встановлюється .NET SDK відповідної версії. Далі команда dotnet restore відновлює всі необхідні пакети, а dotnet build перевіряє, чи немає помилок при компіляції. Після цього команда dotnet test запускає всі тести, і якщо хоча б один із них не проходить, процес переривається. Це дуже зручно, бо ще до об’єднання змін у головну гілку можна побачити, чи не зламався код. Усе це відбувається автоматично, без втручання користувача, що значно зменшує кількість людських помилок і забезпечує стабільність проєкту[10].

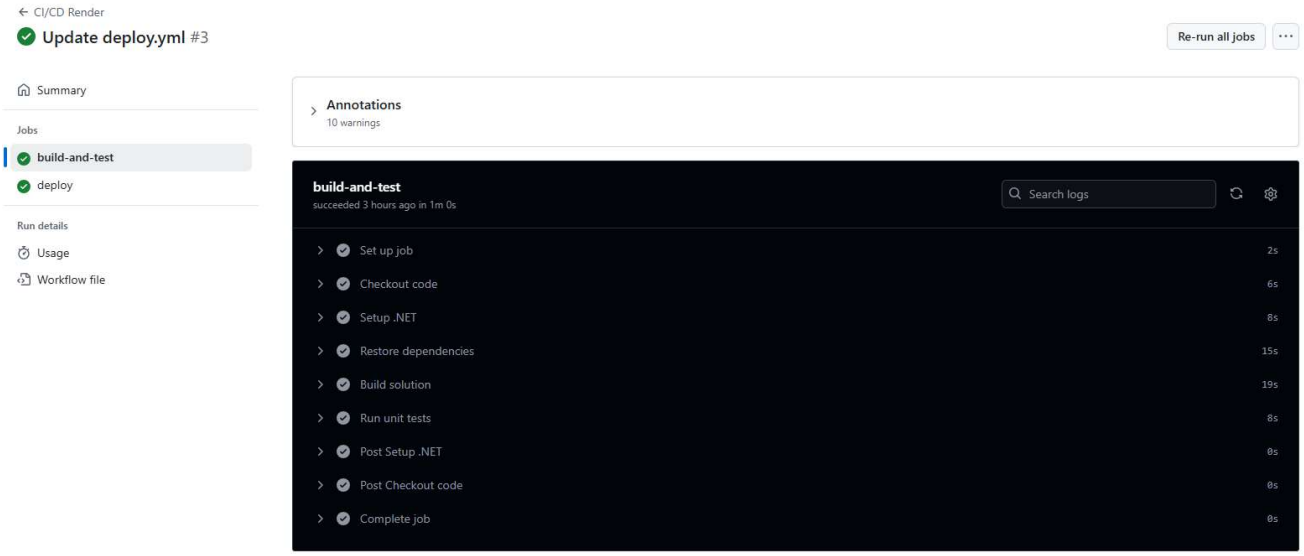

Рисунок 3.3 — Автоматизоване виконання етапів тестування через файл конфігурації deploy.yml

Процес CD було реалізовано за допомогою GitHub Actions і автоматичного деплою на платформу Render через вебхук. Цей вебхук, що називається Deploy Hook, — це унікальне посилання, яке надає Render для запуску процесу розгортання після отримання HTTP-запиту. Коли перевірки CI проходять успішно, усі зміни з головної гілки автоматично завантажуються у продакшн. Для цього на Render я прив’язав свій GitHub-репозиторій до конкретного вебсервісу. У налаштуваннях вказав, щоб сервіс автоматично оновлювався щоразу, коли змінюється гілка main. Завдяки цьому будь-яке оновлення коду одразу збирається і запускається на сервері без ручного втручання. Таким чином, саме GitHub контролює, коли слід виконати деплой, а Render відповідає лише за сам процес запуску. Це дозволяє краще керувати логікою деплою і при цьому зберігає простоту налаштування[11].

На рисунку 3.4 зображено деплой за допомогою Deploy Hook, який автоматично ініціюється після успішного проходження CI перевірок. Для забезпечення безпеки процесу деплою використовуються секрети GitHub, що дозволяє зберігати конфіденційну інформацію, необхідну для підключення до платформи Render.

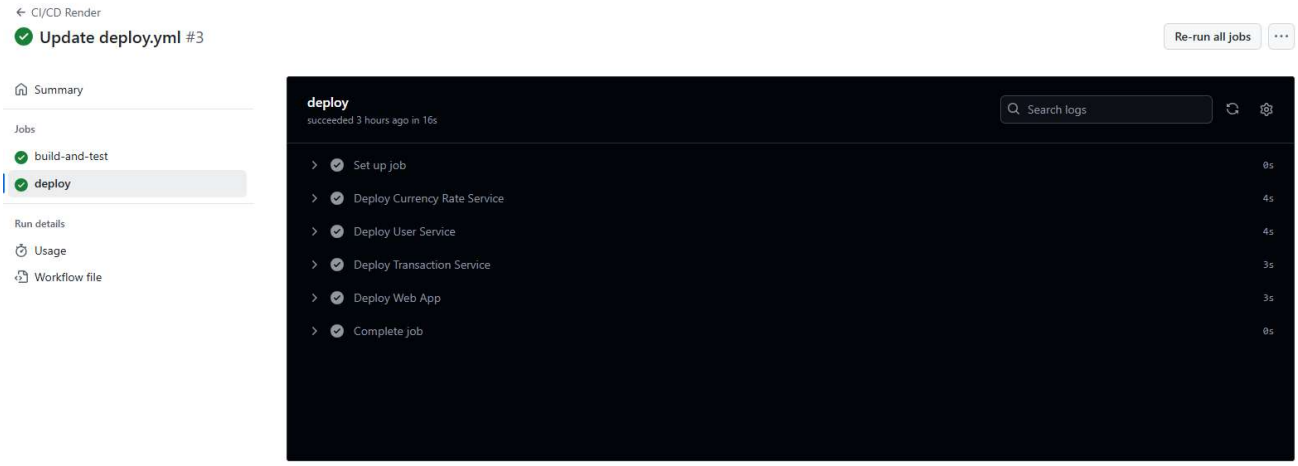

Рисунок 3.4 — Автоматизоване виконання етапів CD через файл конфігурації deploy.yml

Застосування процесів CI/CD значно спростило автоматизацію розгортання та тестування проєкту. Впровадження автоматичного деплою та тестування дозволило зменшити час, витрачений на ручні операції, та підвищити надійність проєкту. Окрім того, завдяки таким процесам знижено ймовірність виникнення помилок, а також забезпечено стабільність та якість коду на всіх етапах розробки. Впровадження CI/CD стало необхідним етапом у сучасному процесі розробки програмного забезпечення, забезпечуючи ефективність та зручність для розробників.

3.4 Розгортання та підтримка програмного проекту

Завершальним етапом роботи став вибір сервісів для розгортання проекту та зручності для його подальшої підтримки. У процесі роботи я зіткнувся з низкою проблем, пов’язаних із вибором платформи для хостингу та базою даних. Спочатку розглядав Railway та інші сервіси, однак обмеження безкоштовного тарифу, особливо щодо обсягу бази даних, ускладнювали реалізацію повноцінної системи. Власне розгортання PostgreSQL у Docker чи відкриття її в мережу також не дало бажаного результату. Тому було прийнято рішення скористатися платформою Supabase, яка надала сучасний інтерфейс, зручну авторизацію та достатньо ресурсів навіть на безкоштовному рівні. Для самої системи я обрав Render, адже ця платформа дозволяє легко розгортати

кілька проєктів одночасно, що ідеально підходить для мікросервісної архітектури мого проєкту. Ця платформа підтримує як backend-сервіси, так і frontend-додатки, а також пропонує корисні інструменти для їх подальшої підтримки.

Після вибору відповідних платформ для розгортання, я приступив до налаштування кожного компонента системи. Мій проєкт складається з трьох основних backend-сервісів: User Service, Currency Rate Service та Transaction Service. Кожен із них був розгорнутий окремо на Render як Web Service. Render дозволяє зв’язати репозиторій GitHub із сервісом, що забезпечує автоматичне оновлення після кожного пушу коміта. Для кожного сервісу я вказав необхідні build та start команди, а також налаштував змінні середовища через вбудований інтерфейс, що дозволило зберігати API-ключі та строки підключення до бази даних безпосередньо на платформі. Frontend частина, реалізована за допомогою React та Vite, була розгорнута як Static Site на Render. Це дозволило швидко та зручно опублікувати інтерфейс користувача, з автоматичним оновленням при кожному новому коміті до основної гілки. Render надає зручні інструменти для моніторингу стану сервісів, перегляду логів та налаштування автоматичних сповіщень про помилки. Це дозволяє оперативно реагувати на можливі проблеми та забезпечувати стабільну роботу застосунку в продакшн-середовищі.

Для зберігання та обробки даних я використав Supabase, який надає хмарну PostgreSQL базу даних із вбудованим REST API та системою авторизації. Supabase дозволив уникнути обмежень безкоштовних тарифів інших платформ і дозволив швидко підключити БД до проекту за API строки підключення. Supabase надає достатньо ресурсів навіть на безкоштовному рівні, що дозволило зосередитися на реалізації функціоналу без зайвих технічних труднощів. Після розгортання всіх компонентів системи важливою частиною стала їх підтримка.

Отже, розгортання та підтримка програмного проєкту є важливими етапами для забезпечення стабільної та безперебійної роботи системи після її

запуску. Підтримка включає в себе не лише моніторинг працездатності, а й регулярне оновлення компонентів, виправлення помилок та адаптацію до змін у середовищі. Платформи надають всі необхідні інструменти для автоматизації процесів та зручної підтримки системи, що дозволяє знижувати ризик помилок та спрощує управління проектом на всіх етапах його життєвого циклу. Завдяки таким платформам, підтримка і розвиток проєкту стає більш ефективним і менш ресурсоємним.

3.5 Керівництво користувача по роботі з системою обміну валют

Сайт обміну валют був створений для того, щоб користувачі могли швидко переглядати актуальні курси, порівнювати їх та виконувати прості розрахунки без зайвих складнощів. Щоб спростити навігацію та пояснити, як користуватись основними функціями, у цьому розділі наведено коротке керівництво. Інтерфейс максимально простий, тому навіть новий користувач без проблем розбереться з доступними можливостями.

Головна сторінка є центральним елементом сайту, на якій зібрано ключові функції для зручності користувача. Основний вміст сторінки складається з таблиці актуальних валютних курсів, інтерактивного калькулятора валют, секції з перевагами використання ресурсу та блоку з відгуками клієнтів. Уся інформація представлена у зрозумілому форматі, що дозволяє легко орієнтуватися навіть новим користувачам.

У верхній частині розташовано навігаційне меню, що прикріплене до екрану при прокручуванні. Зліва знаходиться логотип сайту, який дозволяє у будь-який момент повернутися на головну сторінку. По центру меню розміщені якірні посилання для швидкої навігації по розділах, а праворуч система автентифікації, що включає кнопки для реєстрації та входу в акаунт, а після входу можливість перейти або вийти з особистого кабінету.

На головній сторінці зліва розміщена таблиця актуальних валютних курсів, де користувач може побачити поточні значення купівлі та продажу

основних валют. Ця інформація оновлюється автоматично, щоб користувач завжди мав доступ до актуальних даних.

На головній сторінці зліва розміщена таблиця актуальних валютних курсів, де користувач може побачити поточні значення купівлі та продажу основних валют. Дані в таблиці оновлюються автоматично, що забезпечує користувачу доступ до актуальних курсів. Справа від таблиці зображено калькулятор валют, що дозволяє розрахувати суму обміну між обраними валютами. Калькулятор використовує дані з таблиці, тому користувач може легко орієнтуватися в розрахунках, не переходячи на інші сторінки або сервіси. На рисунку 3.5 зображено, як виглядають ці елементи на сторінці.

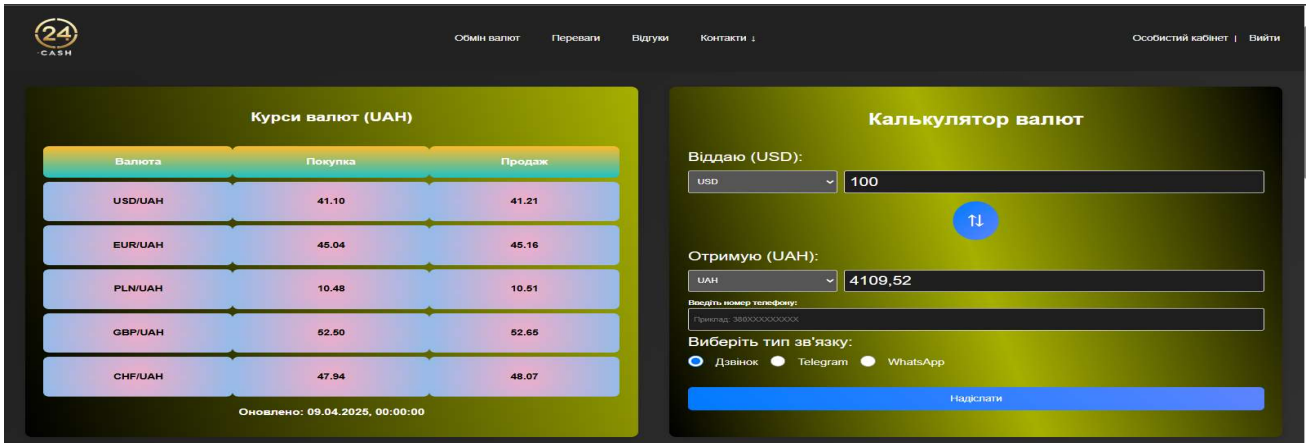

Рисунок 3.5 — Таблиця та калькултор валют

Далі посередині на головній сторінці розташований блок «Переваги роботи з нами», який надає користувачам короткий огляд основних переваг використання цієї платформи. У цьому блоці висвітлені ключові особливості, щоб викликати у користувача довіру і прозорість до платформи. Кожна перевага подана у вигляді короткого тексту з відповідною іконкою для візуальної ідентифікації, що робить інформацію доступною та зрозумілою. На рисунку 3.6 можна побачити приклад цього блоку.

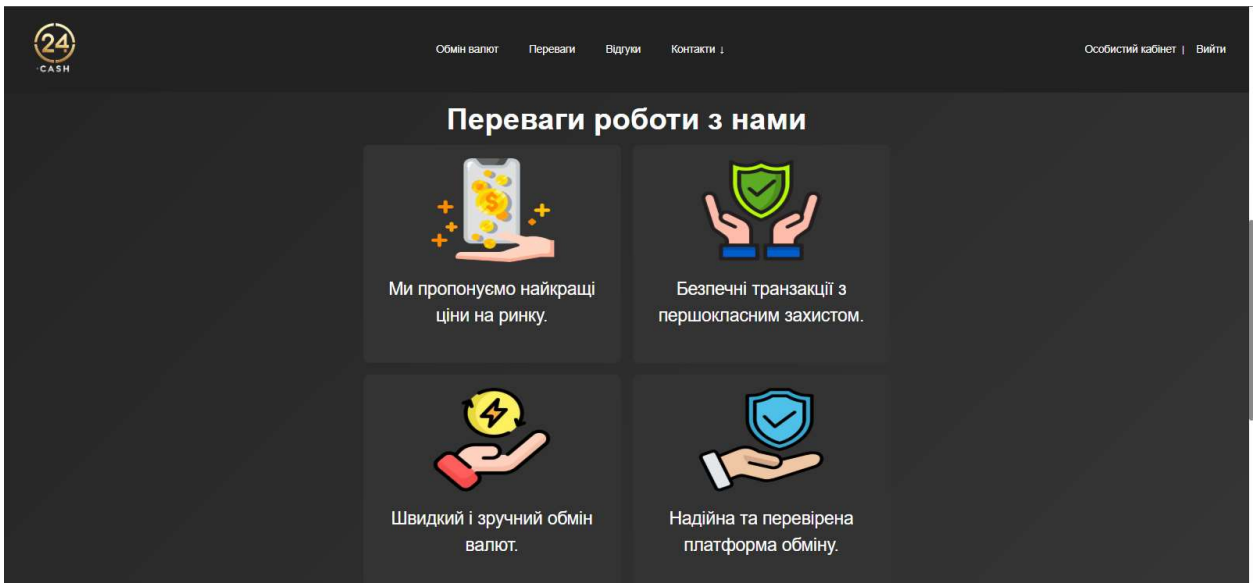

Рисунок 3.6 — Блок переваг платформи

У нижній частині головної сторінки розміщено розділ «Відгуки клієнтів», який слугує соціальним доказом якості та зручності сервісу. У цьому блоці відображаються короткі повідомлення від користувачів, які вже скористалися платформою для обміну валют. Відгуки подані у форматі карток, що містять ім’я користувача, короткий текст повідомлення та оцінку сервісу у вигляді зірок. Такий підхід допомагає новим відвідувачам швидко оцінити надійність системи на основі реального досвіду інших. Приклад відгуків зображено на рисунку 3.7.

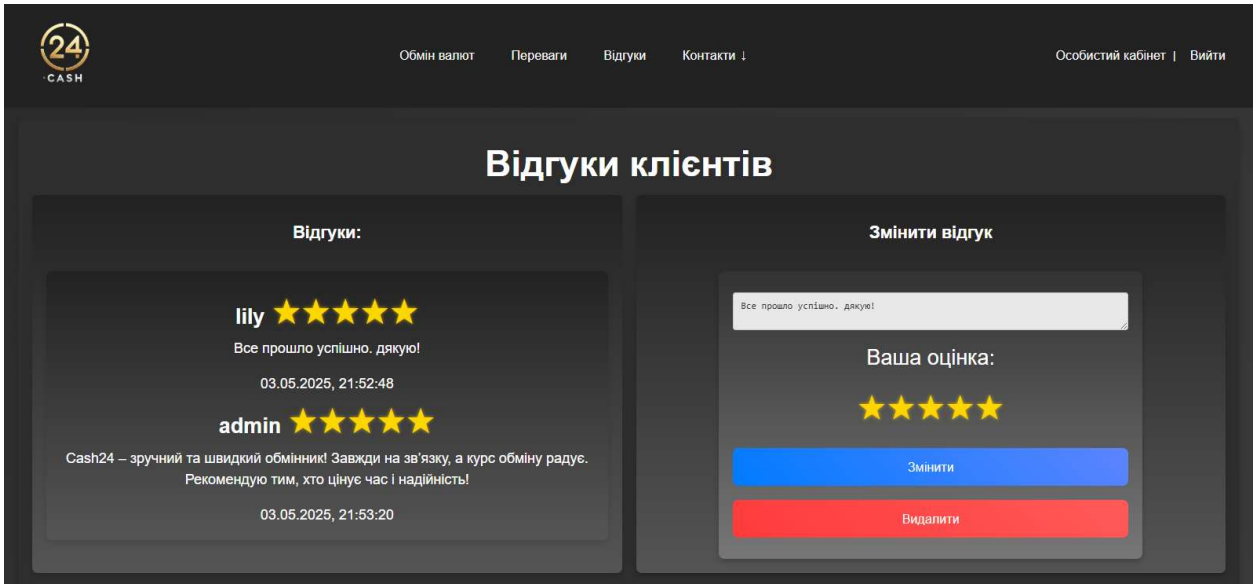

Рисунок 3.7 — Розділ відгуків клієнтів

Для забезпечення безпечного доступу до персоналізованих функцій системи реалізовано прості та зручні вікна реєстрації та входу(рис 3.8). Реєстрація передбачає заповнення мінімального набору полів, що дозволяє швидко створити обліковий запис. Після цього користувач може увійти до системи за допомогою форми авторизації. Обидва вікна реалізовано таким чином, щоб процес був інтуїтивно зрозумілим навіть для нових користувачів. У разі успішного входу з'являється доступ до особистого кабінету, де можна переглядати історію транзакцій, залишати заявки на обмін валют, писати відгуки та керувати налаштуваннями профілю.

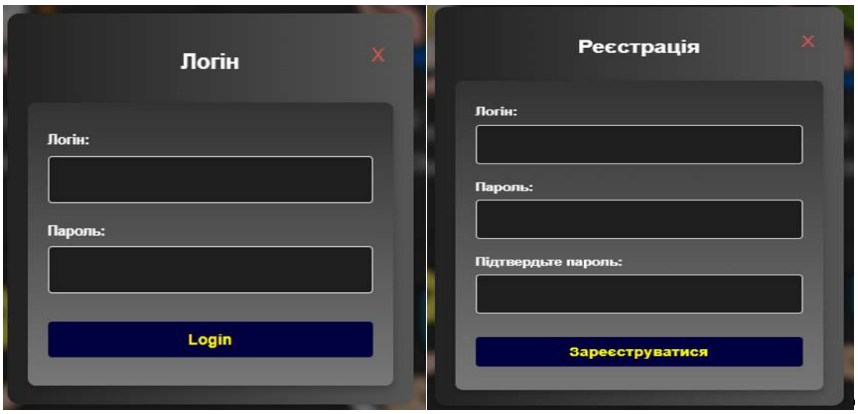

Рисунок 3.8 — Вікна автентифікації

Особистий кабінет користувача відкривається після успішної автентифікації через верхнє меню сайту. У цій сторінці користувач має доступ до персоналізованої інформації, зокрема до історії своїх транзакцій, даних профілю та можливості змінити особисті дані. Список операцій подано у вигляді таблиці з вказаними датами, сумами, валютами та типами операцій. Також передбачено можливість вийти з акаунту через відповідну кнопку, розташовану у верхній частині сторінки. Приклад вигляду особистого кабінету наведено на рисунку 3.9.

Рисунок 3.9 — Особистий кабінет користувача

Отже, користувацьке керівництво демонструє, що вебзастосунок розроблено з урахуванням зручності, інтуїтивності та доступності для звичайного користувача. Завдяки зрозумілій структурі сайту, продуманому навігаційному меню та чіткому поділу функціональностей. Навіть користувач без технічних знань зможе легко скористатися всіма можливостями системи. Таке рішення сприяє підвищенню якості взаємодії з програмним продуктом та задоволеності кінцевого користувача.

ВИСНОВКИ

У процесі розробки цієї кваліфікаційної роботи було створено автоматизовану систему для веб-сервісу обміну валют, що ілюструє важливість і практичну користь використання автоматизації в сучасній розробці програмного забезпечення. Завдяки інтеграції інструментів CI/CD, контейнеризації через Docker і використанню хмарних сервісів для розгортання вдалося значно спростити процеси створення, тестування та впровадження веб-додатків.

Основним результатом роботи стало автоматичне налаштування процесу розробки, що дозволяє швидко доставляти оновлення і гарантувати стабільну роботу програми, зберігаючи при цьому високий рівень гнучкості для подальших змін. Важливою складовою цього проекту є впровадження сучасних інструментів автоматизації, що дозволяють забезпечити безперервність процесів, знизити кількість помилок і підвищити загальну ефективність роботи команди. Використання Vite значно спростило розробку інтерфейсу користувача завдяки своїй швидкодії та простому налаштуванню, а хмарний сервіс Render дозволив розгорнути веб-застосунок, включно як з фронтендом, так і з бекенд-сервісами, забезпечивши їх доступність у будь-який час.

Розроблена система демонструє, як автоматизація може оптимізувати роботу команди, знижуючи ризики помилок і скорочуючи час на тестування та доставку нових функцій. Це дозволяє створювати більш стабільні і адаптивні продукти, що важливо для швидко змінюваного світу технологій. Такий підхід може бути застосований до різних проектів, забезпечуючи зручний і надійний процес розробки та підтримки програмного забезпечення.

ПЕРЕЛІК ПОСИЛАНЬ

1. Фаулер М. Архітектура корпоративних додатків. – К. : Вільямс, 2020.
2. GitHub Actions Documentation [Електронний ресурс]. – Режим доступу: https://docs.github.com/en/actions
3. Martin R.C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. – Prentice Hall, 2017.
4. Microsoft Docs. ASP.NET Core Guide [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core
5. Docker Docs. What is Docker? [Електронний ресурс]. – Режим доступу: https://docs.docker.com/get-started/overview/
6. Bass L., Clements P., Kazman R. Software Architecture in Practice. – Addison-Wesley, 2012.
7. OpenGroup. Microservice Architecture – Definition and Principles [Електронний ресурс]. – Режим доступу: https://www.opengroup.org/soa/source-book/msawp/p2.htm
8. Newman S. Building Microservices. – O'Reilly Media, 2021.
9. PostgreSQL Documentation [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/
10. Duvall P.M., Matyas S., Glover A. Continuous Integration. – Addison-Wesley, 2007.
11. Fowler M. Continuous Delivery and Integration [Електронний ресурс]. – Режим доступу: https://martinfowler.com/articles/continuousIntegration.html

Додаток А – Код програми

AccountService

```
using Auth.Persistence.Models;
using Auth.Persistence.Repositories;
using Microsoft.AspNetCore.Identity;

namespace Auth.BusinessLogic.Auth
{
    public class AccountService(IAccountRepository
accountRepository, JwtService jwtService) : IAccountService
    {
        public async Task<bool> RegisterAsync(string userName,
string password)
        {
            var IdentifyUserByName = await
accountRepository.GetByUserNameAsync(userName);
            if (string.Equals(IdentifyUserByName?.UserName,
userName, StringComparison.OrdinalIgnoreCase))
            {
                return false;
            }

            var newId = Guid.NewGuid();
            var account = new Account()
            {
                UserName = userName.ToLower(),
                Id = newId,
                Permissions = new List<Permission>
                {
                    new Permission
                    {
                        Name = Permissons.User,
                        AccountId = newId
                    }
                },

                Balances = Enum.GetValues<Currency>()
                    .Select(currency => new AccountBalance
                    {
                        AccountId = newId,
                        CurrencyCode = currency.ToString(),
                        Amount = 0m
                    })
                    .ToList()
            };

            var passHash = new
PasswordHasher<Account>().HashPassword(account, password);
            account.PasswordHash = passHash;
            await accountRepository.AddAccountAsync(account);
```

```
            return true;
        }

        public async Task<(string, bool)> LoginAsync(string
userName, string password)
        {
            var account = await
accountRepository.GetByUserNameAsync(userName);
            if(account == null) return ("", false);
            var result = new PasswordHasher<Account>()
                .VerifyHashedPassword(account,
account.PasswordHash, password);

            if (result == PasswordVerificationResult.Success)
            {
                return (jwtService.GenerateToken(account),
true);
            }
            else
            {
                return ("", false);
            }
        }

        public async Task<ICollection<Account>>
GetCollectionAccountsAsync()
        {
            return await
accountRepository.GetCollectionAccounts();
        }

        public async Task IncreaseBalanceAsync(Guid accountId,
decimal amount, string currencyCode)
        {
            var balances = await
accountRepository.GetAccountBalancesByUserIdAsync(accountId);

            var balance = balances.FirstOrDefault(b =>
b.CurrencyCode.Equals(currencyCode,
StringComparison.OrdinalIgnoreCase));
            balance.Amount += amount;

            await
accountRepository.UpdateAccountBalanceAsync(balance);
        }

        public async Task DecreaseBalanceAsync(Guid accountId,
decimal amount, string currencyCode)
        {
            var balances = await
accountRepository.GetAccountBalancesByUserIdAsync(accountId);
```

```
            var balance = balances.FirstOrDefault(b =>
b.CurrencyCode.Equals(currencyCode,
StringComparison.OrdinalIgnoreCase));
            if (balance == null) return;
            if (balance.Amount >= amount)
            {
                balance.Amount -= amount;
            }

            await
accountRepository.UpdateAccountBalanceAsync(balance);
        }

        public async Task<ICollection<Review>>
GetCollectionReviewsAsync()
        {
            return await
accountRepository.GetAllReviewsListAsync();
        }

        public async Task WriteReviewAsync(Guid accountId,
string commentary, int rating)
        {
            var review = new Review
            {
                Id = Guid.NewGuid(),
                Comment = commentary,
                Rating = rating,
                Date = DateTime.UtcNow,
                AccountId = accountId
            };

            await accountRepository.AddReviewAsync(review);
        }

        public async Task<string> GetUserNameByIdAsync(Guid
accountId)
        {
            var account = await
accountRepository.GetAccountByIdAsync(accountId);

            return account?.UserName ?? "";
        }

        public async Task UpdateReviewAsync(Guid id, string
comment, int rating)
        {
            var review = await
accountRepository.GetReviewByAccountIdAsync(id);

            review.Comment = comment;
            review.Rating = rating;
            review.Date = DateTime.UtcNow;
```

```
            await accountRepository.UpdateReviewAsync(review);
        }

        public async Task<Review> GetRiviewByAccountIdAsync(Guid
accountId)
        {
            return await
accountRepository.GetReviewByAccountIdAsync(accountId);
        }

        public async Task DeleteReviewAsync(Guid id)
        {
            var review = await
accountRepository.GetReviewByAccountIdAsync(id);

            await accountRepository.DeleteReviewAsync(review);
        }

        public async Task ChangePasswordAsync(Guid id, string
password)
        {
            var account = await
accountRepository.GetAccountByIdAsync(id);

            var passHash = new
PasswordHasher<Account>().HashPassword(account, password);

            account.PasswordHash = passHash;

            await accountRepository.UpdateAccountAsync(account);
        }

        public async Task<bool> VerifyOldPasswordAsync(Guid id,
string oldPassword)
        {
            var account = await
accountRepository.GetAccountByIdAsync(id);

            var result = new
PasswordHasher<Account>().VerifyHashedPassword(account,
account.PasswordHash, oldPassword);

            return result == PasswordVerificationResult.Success;
        }

        public async Task AssignUserPermissionsAsync(Guid
accountId, ICollection<Permission> permissions)
        {
            var account = await
accountRepository.GetAccountByIdAsync(accountId);
```

```
            var existingPermissionNames = new
HashSet<string>(account.Permissions.Select(p => p.Name));

            var newPermissions = permissions
                .Where(p =>
!existingPermissionNames.Contains(p.Name))
                .ToList();


            foreach (var permission in newPermissions)
            {
                account.Permissions.Add(permission);
            }

            await accountRepository.UpdateAccountAsync(account);
        }

        public async Task<ICollection<AccountBalance>>
GetAccountBalancesByUserIdAsync(Guid accountId)
        {
            return await
accountRepository.GetAccountBalancesByUserIdAsync(accountId);
        }

        public async Task<bool> CheckAccountBalanceAsync(Guid
accountId, decimal amountFrom, string currencyFrom)
        {
            var list = await
GetAccountBalancesByUserIdAsync(accountId);

            var balance = list.FirstOrDefault(b =>
b.CurrencyCode.Equals(currencyFrom,
StringComparison.CurrentCultureIgnoreCase));

            if (balance == null) return false;
            if(balance.Amount >= amountFrom)
            {
                return true;
            }

            return false;
        }
    }
}
```

IAccountService

```
using Auth.Persistence.Models;

namespace Auth.BusinessLogic.Auth
{
    public interface IAccountService
    {
        Task<bool> RegisterAsync(string userName, string
password);
```

```
        Task<(string, bool)> LoginAsync(string userName, string
password);
        Task<ICollection<Account>> GetCollectionAccountsAsync();
        Task IncreaseBalanceAsync(Guid accounId, decimal amount,
string currencyCode);
        Task DecreaseBalanceAsync(Guid accounId, decimal amount,
string currencyCode);
        Task<ICollection<Review>> GetCollectionReviewsAsync();
        Task WriteReviewAsync(Guid accountId, string commentary,
int rating);
        Task<string> GetUserNameByIdAsync(Guid accountId);
        Task UpdateReviewAsync(Guid id, string comment, int
rating);
        Task<Review> GetRiviewByAccountIdAsync(Guid accountId);
        Task DeleteReviewAsync(Guid id);
        Task ChangePasswordAsync(Guid id, string password);
        Task<bool> VerifyOldPasswordAsync(Guid id, string
oldPassword);
        Task AssignUserPermissionsAsync(Guid accountId,
ICollection<Permission> permissions);
        Task<ICollection<AccountBalance>>
GetAccountBalancesByUserIdAsync(Guid accountId);
        Task<bool> CheckAccountBalanceAsync(Guid accountId,
decimal amountFrom, string currencyFrom);
    }
}
```

AccountRepository

```
using Auth.Persistence.Data;
using Auth.Persistence.Models;
using Microsoft.EntityFrameworkCore;

namespace Auth.Persistence.Repositories
{
    public class AccountRepository(AuthDbContext context) :
IAccountRepository
    {
        public async Task AddAccountAsync(Account account)
        {
            await context.Accounts.AddAsync(account);
            context.SaveChanges();
        }

        public async Task<Account?> GetAccountByIdAsync(Guid
accountId)
        {
            return await context.Accounts
                .Include(a => a.Permissions)
                .Include(b => b.Balances)
                .Include(c => c.Review)
                .FirstOrDefaultAsync(a => a.Id == accountId);
        }
```

```
        public async Task<Account?> GetByUserNameAsync(string
userName)
        {
            return await context.Accounts
                .Include(a => a.Permissions)
                .Include(b => b.Balances)
                .Include(c => c.Review)
                .FirstOrDefaultAsync(x => x.UserName ==
userName.ToLower());
        }

        public async Task<ICollection<Account>>
GetCollectionAccounts()
        {
            return await context.Accounts
                .Include(a => a.Permissions)
                .Include(b => b.Balances)
                .Include(c => c.Review)
                .ToListAsync();
        }

        public async Task<ICollection<Permission>?>
GetPermissionByUserNameAsync(string userNameValue)
        {
            var account = await context.Accounts
                .Include(a => a.Permissions)
                .FirstOrDefaultAsync(a => a.UserName ==
userNameValue);

            return account?.Permissions ?? null;
        }

        public async Task
UpdateAccountBalanceAsync(AccountBalance balance)
        {
            context.AccountBalances.Update(balance);
            await context.SaveChangesAsync();
        }

        public async Task AddReviewAsync(Review review)
        {
            await context.Reviews.AddAsync(review);
            await context.SaveChangesAsync();
        }

        public async Task<ICollection<Review>>
GetAllReviewsListAsync()
        {
            return await context.Reviews.ToListAsync();
        }

        public async Task UpdateReviewAsync(Review review)
        {
```

```
            context.Reviews.Update(review);
            await context.SaveChangesAsync();
        }

        public async Task<Review> GetReviewByAccountIdAsync(Guid
accountId)
        {
            var account = await context.Accounts.Include(r =>
r.Review).FirstOrDefaultAsync(a => a.Id == accountId);
            return account.Review;
        }

        public async Task DeleteReviewAsync(Review review)
        {
            context.Reviews.Remove(review);
            await context.SaveChangesAsync();
        }

        public async Task UpdateAccountAsync(Account account)
        {
            context.Accounts.Update(account);
            await context.SaveChangesAsync();
        }

        public async Task<ICollection<AccountBalance>>
GetAccountBalancesByUserIdAsync(Guid accountId)
        {
            return await context.AccountBalances.Where(b =>
b.AccountId == accountId).ToListAsync();
        }
    }
}
```

IAccountRepository

```
using Auth.Persistence.Models;

namespace Auth.Persistence.Repositories
{
    public interface IAccountRepository
    {
        Task AddAccountAsync(Account account);
        Task<Account?> GetByUserNameAsync(string userName);
        Task<ICollection<Permission>>
GetPermissionByUserNameAsync(string userNameValue);
        Task<ICollection<Account>> GetCollectionAccounts();
        Task<Account?> GetAccountByIdAsync(Guid accountId);
        Task<ICollection<AccountBalance>>
GetAccountBalancesByUserIdAsync(Guid accountId);
        Task UpdateAccountBalanceAsync(AccountBalance balance);
        Task AddReviewAsync(Review review);
        Task<ICollection<Review>> GetAllReviewsListAsync();
        Task<Review> GetReviewByAccountIdAsync(Guid accountId);
        Task UpdateReviewAsync(Review review);
        Task DeleteReviewAsync(Review review);
```

```
        Task UpdateAccountAsync(Account account);
    }
}
```

AuthController

```
using Auth.BusinessLogic.Auth;
using Auth.Persistence.Models;
using AuthJWT.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace AuthJWT
{
    [ApiController]
    [Route("[controller]")]
    public class AuthController(IAccountService accountService)
: ControllerBase
    {
        [HttpPost("register")]
        public async Task<IActionResult>
Register([FromBody]RegisterUserRequest request)
        {
            bool success = await
accountService.RegisterAsync(request.UserName,
request.Password);

            if (!success)
            {
                return BadRequest(new { message = "Користувач
вже існує" });
            }

            return Ok(new { message = "Користувач успішно
зареєстрований" });
        }

        [HttpPost("login")]
        public async Task<IActionResult>
Login([FromBody]LoginRequest loginRequest)
        {
            var (token, success) = await
accountService.LoginAsync(loginRequest.UserName,
loginRequest.Password);

            if (!success)
            {
                return BadRequest(new { message = "Неправильний
логін або пароль" });
            }

            return Ok(token);
        }

        [HttpGet(nameof(GetAllAccounts))]
```

```
        public async Task<IActionResult> GetAllAccounts()
        {
            var result = await
accountService.GetCollectionAccountsAsync();

            return Ok(result);
        }

        [HttpGet(nameof(GetAllReviews))]
        public async Task<IActionResult> GetAllReviews()
        {
            var result = await
accountService.GetCollectionReviewsAsync();

            return Ok(result);
        }

        [HttpGet(nameof(GetUserNameById) + "/{accountId}")]
        public async Task<IActionResult> GetUserNameById(Guid
accountId)
        {
            var result = await
accountService.GetUserNameByIdAsync(accountId);

            return Ok(result);
        }

        [HttpGet(nameof(GetRiviewByAccountId) + "/{accountId}")]
        public async Task<IActionResult>
GetRiviewByAccountId(Guid accountId)
        {
            var result = await
accountService.GetRiviewByAccountIdAsync(accountId);

            return Ok(result);
        }

        [HttpPost(nameof(WriteReview))]
        public async Task<IActionResult> WriteReview([FromBody]
ReviewRequest request)
        {
            if (!(request.rating > 0 && request.rating <= 5))
                return BadRequest("Rating must be between 1 and
5");

            if(string.IsNullOrWhiteSpace(request.commentary))
                return BadRequest("Comment is empty");

            await accountService.WriteReviewAsync(request.id,
request.commentary, request.rating);

            return Ok("Review added");
        }
```

```
        [HttpPut(nameof(UpdateReview))]
        public async Task<IActionResult> UpdateReview([FromBody]
ReviewRequest request)
        {
            await accountService.UpdateReviewAsync(request.id,
request.commentary, request.rating);

            return Ok("Review updated");
        }

        [HttpDelete(nameof(DeleteReview) + "/{id}")]
        public async Task<IActionResult> DeleteReview(Guid id)
        {
            await accountService.DeleteReviewAsync(id);

            return Ok("Review deleted");
        }

        [HttpGet(nameof(VerifyOldPassword))]
        public async Task<IActionResult>
VerifyOldPassword([FromQuery] Guid accountId, [FromQuery] string
oldPassword)
        {
            var result = await
accountService.VerifyOldPasswordAsync(accountId, oldPassword);

            return Ok(result);
        }

        [HttpPut(nameof(ChangeUserPassword))]
        public async Task<IActionResult>
ChangeUserPassword([FromBody] ChangePasswordRequest request)
        {
            await
accountService.ChangePasswordAsync(request.accountId,
request.password);

            return Ok("User password updated");
        }

        [HttpPost(nameof(AssignAdminRole) + "/{accountId}")]
        public async Task<IActionResult> AssignAdminRole(Guid
accountId)
        {
            var permissionList = new List<Permission>
            {
                new Permission {AccountId = accountId, Name =
Permissons.User},
                new Permission {AccountId = accountId, Name =
Permissons.Admin},
            };
```

```
            await
accountService.AssignUserPermissionsAsync(accountId,
permissionList);

            return Ok("Assign roles completed");
        }
    }
}
                        BalanceController
using Auth.BusinessLogic.Auth;
using AuthJWT.Models;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace AuthJWT.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class BalanceController(IAccountService
accountService) : ControllerBase
    {
        [HttpPut(nameof(IncreaseBalance))]
        public async Task<IActionResult>
IncreaseBalance([FromBody] BalanceHandlerRequest request)
        {
            if (request.amount <= 0)
                return BadRequest("Amount must be greater than
zero.");

            await
accountService.IncreaseBalanceAsync(request.accountId,
request.amount, request.currencyCode);

            return Ok("Balance updated");
        }

        [HttpPut(nameof(DecreaseBalance))]
        public async Task<IActionResult>
DecreaseBalance([FromBody] BalanceHandlerRequest request)
        {
            if (request.amount <= 0)
                return BadRequest("Amount must be greater than
zero.");

            var success = await
accountService.CheckAccountBalanceAsync(request.accountId,
request.amount, request.currencyCode);

            if (!success)
            {
                return BadRequest("У користувача недостатньо
коштів на рахунку.");
            }
```

```
            await
accountService.DecreaseBalanceAsync(request.accountId,
request.amount, request.currencyCode);

            return Ok("Balance updated");
        }

        [HttpGet(nameof(GetAccountBalance) + "/{accountId}")]
        public async Task<IActionResult> GetAccountBalance(Guid
accountId)
        {
            var balances = await
accountService.GetAccountBalancesByUserIdAsync(accountId);

            return Ok(balances);
        }

        [HttpPost(nameof(CheckAccountBalance))]
        public async Task<IActionResult>
CheckAccountBalance([FromBody] BalanceHandlerRequest request)
        {
            var success = await
accountService.CheckAccountBalanceAsync(request.accountId,
request.amount, request.currencyCode);

            return Ok(success);
        }
    }
}
```

CurrencyRateService

```
using CurrencyRateService.Models;
using CurrencyRateService.Repositories;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;

namespace CurrencyRateService.Services
{
    public class CurrencyRateService(
        ICurrencyRateRepository currencyRateRepository,
        IServiceProvider serviceProvider,
        MarginService marginService
        ) : ICurrencyRateService
    {
        public async Task<CurrencyRateResponse>
GetCurrencyRateResponseAsync(JObject jsonObject)
        {
            var date = jsonObject["date"].ToString();

            DateTime parsedDate;
            if (!DateTime.TryParse(date, out parsedDate))
            {
                parsedDate = DateTime.MinValue;
```

```
            }

            parsedDate = DateTime.SpecifyKind(parsedDate,
DateTimeKind.Utc);

            string baseCurrency = jsonObject.Properties()
                .Where(p => p.Name != "date")
                .Select(p => p.Name)
                .FirstOrDefault();

            var newId = Guid.NewGuid();
            ICollection<CurrencyRate> currencies = new
List<CurrencyRate>();

            foreach (var property in jsonObject.Properties())
            {
                var currencyRates = property.Value as JObject;
                if (currencyRates != null)
                {
                    foreach (var rate in
currencyRates.Properties())
                    {
                        if
(decimal.TryParse(rate.Value.ToString(), out decimal rateValue)
                            &&
Enum.GetValues<Currency>().Select(c =>
c.ToString().ToLower()).Contains(rate.Name)
                            && rate.Name.ToLower() !=
baseCurrency?.ToLower())
                        {
                            currencies.Add(new CurrencyRate()
                            {
                                CurrencyRateResponseId = newId,
                                Currency = rate.Name,
                                Value = rateValue
                            });
                        }
                    }
                }
            }
            var currencyRateResponse = new
CurrencyRateResponse()
            {
                Id = newId,
                Date = parsedDate,
                CurrencyBase = baseCurrency,
                Rates = currencies
            };
            using (var scope = serviceProvider.CreateScope())
            {
                var currencyRateRepository =
scope.ServiceProvider.GetRequiredService<ICurrencyRateRepository
>();
```

```
                await
currencyRateRepository.AddCurrencyRateResponseAsync(currencyRate
Response);
            }
            return currencyRateResponse;
        }

        public virtual async
Task<ICollection<CurrencyRateResponse>>
GetCurrencyRateListAsync()
        {
            string url = "";

            var urls = Enum.GetValues<Currency>()
                .Select(c =>
                {
                    return url =
$"https://cdn.jsdelivr.net/npm/@fawazahmed0/currency-
api@latest/v1/currencies/{c.ToString().ToLower()}.json";
                })
                .ToList();

            var tasks = urls.Select(async url =>
            {
                var json = await
currencyRateRepository.GetJsonCurrencyAPIAsync(url);
                var currencyRateResponse = await
GetCurrencyRateResponseAsync(GetJObjestCurrencyRate(json));
                return currencyRateResponse;
            });

            var result = await Task.WhenAll(tasks);

            return result.ToList();
        }

        public JObject GetJObjestCurrencyRate(string json)
        {
            var jsonObject =
JsonConvert.DeserializeObject<JObject>(json);
            return jsonObject;
        }

        public async Task<CurrencyRateResponseDto>
GetExchangeRateFromAllRatesByNameAsync(ICollection<CurrencyRateR
esponseDto> exchangeRate, string currencyName)
        {
            var response = new CurrencyRateResponseDto
            {
                CurrencyBase = currencyName,
                Date = exchangeRate.FirstOrDefault(b =>
b.CurrencyBase == currencyName).Date,
                Rates = exchangeRate
```

```
                .SelectMany(currency => currency.Rates
                    .Where(rate => rate.Currency ==
currencyName.ToLower())
                    .Select(rate => new CurrencyRateDto
                    {
                        Currency = currency.CurrencyBase,
                        buyCurrency = rate.buyCurrency,
                        sellCurrency = rate.sellCurrency
                    }))
                .ToList()
            };
            return response;
        }

        public async Task<CurrencyRateResponseDto>
GetExchangeRateByNameAsync(ICollection<CurrencyRateResponse>
currencyList, string currencyName)
        {
            var response = new CurrencyRateResponseDto
            {
                CurrencyBase = currencyName,
                Date = currencyList.FirstOrDefault(b =>
b.CurrencyBase == currencyName).Date,
                Rates = currencyList
                .FirstOrDefault(n =>
n.CurrencyBase.Equals(currencyName,
StringComparison.OrdinalIgnoreCase)).Rates
                .Select(rate => new CurrencyRateDto
                {
                    Currency = rate.Currency,
                    buyCurrency = rate.Value * (1 -
marginService.GetMargin()),
                    sellCurrency = rate.Value * (1 +
marginService.GetMargin()),
                }).ToList()
            };

            return response;
        }

        public async Task<ICollection<CurrencyRateResponseDto>>
GetExchangeRateListAsync(ICollection<CurrencyRateResponse>
currencyList)
        {
            var tasks = Enum.GetValues<Currency>().Select(async
currency =>
            {
                return await
GetExchangeRateByNameAsync(currencyList,
currency.ToString().ToLower());
            }).ToList();

            var response = await Task.WhenAll(tasks);
```

```
            return response.ToList();
        }

        public async Task<ICollection<CurrencyRateResponse>>
GetActualCurrencyRateListAsync()
        {
            var actualCurrencyRateList = await
currencyRateRepository.GetCurrencyRateResponseListAsync();

            foreach (var currency in actualCurrencyRateList)
            {
                if(currency.Date != DateTime.UtcNow.Date)
                {
                    actualCurrencyRateList = await
GetCurrencyRateListAsync();
                    break;
                }
            }

            var latestRates = actualCurrencyRateList
                .GroupBy(d => d.CurrencyBase)
                .Select(g => g.OrderByDescending(d =>
d.Date).First())
                .ToList();

            return latestRates;
        }
    }
}
```

ICurrencyRateService

```
using CurrencyRateService.Models;
using Newtonsoft.Json.Linq;

namespace CurrencyRateService.Services
{
    public interface ICurrencyRateService
    {
        Task<ICollection<CurrencyRateResponse>>
GetCurrencyRateListAsync();
        Task<CurrencyRateResponse>
GetCurrencyRateResponseAsync(JObject jsonObject);
        JObject GetJObjestCurrencyRate(string json);
        Task<CurrencyRateResponseDto>
GetExchangeRateByNameAsync(ICollection<CurrencyRateResponse>
currencyList, string currencyName);
        Task<ICollection<CurrencyRateResponseDto>>
GetExchangeRateListAsync(ICollection<CurrencyRateResponse>
currencyList);
        Task<CurrencyRateResponseDto>
GetExchangeRateFromAllRatesByNameAsync(ICollection<CurrencyRateR
esponseDto> currencyList, string currencyName);
```

```
        Task<ICollection<CurrencyRateResponse>>
GetActualCurrencyRateListAsync();
    }
}
                    CurrencyRateRepository
using CurrencyRateService.Data;
using CurrencyRateService.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Caching.Memory;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;

namespace CurrencyRateService.Repositories
{
    public class CurrencyRateRepository
        (
        CurrencyRateDbContext context,
        HttpClient httpClient,
        IMemoryCache memoryCache
        ) : ICurrencyRateRepository
    {
        public async Task<string> GetJsonCurrencyAPIAsync(string
url)
        {
            var response = await httpClient.GetStringAsync(url);
            return response;
        }

        public async Task<CurrencyRateResponse?>
GetCurrencyRateResponseByIdAsync(Guid id)
        {
            return await
context.CurrencyRateResponses.FirstOrDefaultAsync(u => u.Id ==
id);
        }

        public async Task<ICollection<CurrencyRate>>
GetCurrencyRateAsync()
        {
            return await context.CurrencyRates.ToListAsync();
        }

        public async Task
UpdateCurrencyRateResponseAsync(CurrencyRateResponse
currencyRateResponse)
        {

context.CurrencyRateResponses.Update(currencyRateResponse);
            await context.SaveChangesAsync();
        }
```

```
        public async Task
AddCurrencyRateResponseAsync(CurrencyRateResponse
currencyRateResponse)
        {
            await
context.CurrencyRateResponses.AddAsync(currencyRateResponse);
            await context.SaveChangesAsync();
        }

        public async Task<ICollection<CurrencyRateResponse>>
GetCurrencyRateResponseListAsync()
        {
            return await context.CurrencyRateResponses
                .Include(c => c.Rates)
                .ToListAsync();
        }
    }
}
```

ICurrencyRateRepository

```
using CurrencyRateService.Models;
using Newtonsoft.Json.Linq;

namespace CurrencyRateService.Repositories
{
    public interface ICurrencyRateRepository
    {
        Task<string> GetJsonCurrencyAPIAsync(string url);
        Task<CurrencyRateResponse?>
GetCurrencyRateResponseByIdAsync(Guid id);
        Task<ICollection<CurrencyRate>> GetCurrencyRateAsync();
        Task
UpdateCurrencyRateResponseAsync(CurrencyRateResponse
currencyRateResponse);
        Task AddCurrencyRateResponseAsync(CurrencyRateResponse
currencyRateResponse);
        Task<ICollection<CurrencyRateResponse>>
GetCurrencyRateResponseListAsync();
    }
}
```

CurrencyRateController

```
using CurrencyRateService.Services;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace CurrencyRateService.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CurrencyRateController(ICurrencyRateService
currencyRateService) : ControllerBase
    {
        [HttpGet(nameof(GetCurrencyRateListRate))]
```

```
        public async Task<IActionResult>
GetCurrencyRateListRate()
        {
            var result = await
currencyRateService.GetCurrencyRateListAsync();

            return Ok(result);
        }

        [HttpGet(nameof(GetExchangeRateByName) +
"/{currencyName}")]
        public async Task<IActionResult>
GetExchangeRateByName(string currencyName)
        {
            var currencyList = await
currencyRateService.GetCurrencyRateListAsync();
            var result = await
currencyRateService.GetExchangeRateByNameAsync(currencyList,
currencyName);

            return Ok(result);
        }

        [HttpGet(nameof(GetExchangeCurrencyList))]
        public async Task<IActionResult>
GetExchangeCurrencyList()
        {
            var currencyList = await
currencyRateService.GetCurrencyRateListAsync();
            var result = await
currencyRateService.GetExchangeRateListAsync(currencyList);

            return Ok(result);
        }

        [HttpGet(nameof(GetExchangeRateFromAllRatesByName) +
"/{currencyName}")]
        public async Task<IActionResult>
GetExchangeRateFromAllRatesByName(string currencyName)
        {
            var currencyList = await
currencyRateService.GetActualCurrencyRateListAsync();
            var exchangeRate = await
currencyRateService.GetExchangeRateListAsync(currencyList);
            var result = await
currencyRateService.GetExchangeRateFromAllRatesByNameAsync(excha
ngeRate, currencyName);

            return Ok(result);
        }
    }
}
```

TransactionService

```
using System.Net.Http;
using System.Text.Json;
using System.Text;
using TransactionService.Models;
using TransactionService.Repositories;

namespace TransactionService.Services
{
    public class TransactionService
        (
            ITransactionRepository transactionRepository,
            HttpClient httpClient
        ) : ITransactionService
    {
        public async Task ChangeRequestStatusAsync(Guid
accountId, DateTime date, string status)
        {
            if (status.ToLower() != "rejected" &&
status.ToLower() != "pending" && status.ToLower() != "approved")
                return;

            var requestList = await
transactionRepository.GetCurrencyExchangeRequestListByUserIdAsyn
c(accountId);

            var request = requestList.FirstOrDefault(r => r.Date
== date);

            if(request == null) return;

            request.Status = status.ToLower();
            await
transactionRepository.UpdateRequestAsync(request);
        }

        public bool
ChechMaxRequests(ICollection<CurrencyExchangeRequest>
currencyExchangeRequests)
        {
            int pendingRequests = currencyExchangeRequests
                .Where(r => r.Status.ToLower() == "pending")
                .Count();
            return pendingRequests < 3;
        }

        public async Task<bool> CheckAccountBalanceAsync(Guid
accountId, decimal amountFrom, string currencyFrom)
        {
            var json = JsonSerializer.Serialize(new
            {
                accountId,
                amount = amountFrom,
                currencyCode = currencyFrom
```

```
            });
            var content = new StringContent(json, Encoding.UTF8,
"application/json");
            var response = await
httpClient.PostAsync($"http://localhost:5010/api/Balance/CheckAc
countBalance", content);
            if (!response.IsSuccessStatusCode)
            {
                throw new Exception($"API error:
{response.StatusCode}");
            }
            string result = await
response.Content.ReadAsStringAsync();
            bool isSuccess = bool.Parse(result);
            return isSuccess;
        }

        public async Task
CreateCurrencyExchangeRequestAsync(Guid userId, string
phoneNumber, string contactType, decimal amountFrom, string
currencyFrom, decimal amountTo, string currencyTo)
        {
            var request = new CurrencyExchangeRequest
            {
                Id = Guid.NewGuid(),
                UserId = userId,
                PhoneNumber = phoneNumber,
                ContactType = contactType,
                AmountFrom = amountFrom,
                CurrencyFrom = currencyFrom,
                AmountTo = amountTo,
                CurrencyTo = currencyTo,
                Date = DateTime.UtcNow,
                Status = "pending"
            };

            await
transactionRepository.AddCurrencyExchangeRequestAsync(request);
        }

        public async Task CreateTransactionAsync(Guid accountId,
decimal amountFrom, string currencyFrom, decimal amountTo,
string currencyTo)
        {
            var transaction = new Transaction()
            {
                AccountId = accountId,
                AmountFrom = amountFrom,
                CurrencyFrom = currencyFrom,
                AmountTo = amountTo,
                CurrencyTo = currencyTo,
                Date = DateTime.UtcNow
            };
```

```
            var json = JsonSerializer.Serialize(new
            {
                accountId,
                amount = amountFrom,
                currencyCode = currencyFrom
            });
            var content = new StringContent(json, Encoding.UTF8,
"application/json");
            var response = await
httpClient.PutAsync($"http://localhost:5010/api/Balance/Decrease
Balance", content);

            json = JsonSerializer.Serialize(new
            {
                accountId,
                amount = amountTo,
                currencyCode = currencyTo
            });
            content = new StringContent(json, Encoding.UTF8,
"application/json");
            response = await
httpClient.PutAsync($"http://localhost:5010/api/Balance/Increase
Balance", content);
            if(!response.IsSuccessStatusCode)
            {
                throw new Exception($"API error:
{response.StatusCode}");
            }
            await transactionRepository.AddAsync(transaction);
        }

        public async Task<ICollection<CurrencyExchangeRequest>>
GetCurrencyExchangeRequestListByUserIdAsync(Guid userId)
        {
            return await
transactionRepository.GetCurrencyExchangeRequestListByUserIdAsyn
c(userId);
        }

        public async Task<ICollection<CurrencyExchangeRequest>>
GetUsersRequestsListAsync()
        {
            var list = await
transactionRepository.GetCurrencyExchangeRequestListAsync();

            return list.Where(r => r.Status.ToLower() ==
"pending").ToList();
        }

        public async Task<IEnumerable<Transaction>>
GetUserTransactions(int userId)
        {
```

```
            return await
transactionRepository.GetTransactionByUserIdAsync(userId);
        }
    }
}
```

ITransactionService

```
using TransactionService.Models;

namespace TransactionService.Services
{
    public interface ITransactionService
    {
        Task CreateTransactionAsync(Guid accountId, decimal
amountFrom, string currencyFrom, decimal amountTo, string
currencyTo);
        Task<IEnumerable<Transaction>> GetUserTransactions(int
userId);
        Task CreateCurrencyExchangeRequestAsync(Guid userId,
string phoneNumber, string contactType, decimal amountFrom,
string currencyFrom, decimal amountTo, string currencyTo);
        Task<ICollection<CurrencyExchangeRequest>>
GetCurrencyExchangeRequestListByUserIdAsync(Guid userId);
        bool
ChechMaxRequests(ICollection<CurrencyExchangeRequest>
currencyExchangeRequests);
        Task<ICollection<CurrencyExchangeRequest>>
GetUsersRequestsListAsync();
        Task ChangeRequestStatusAsync(Guid accountId, DateTime
date, string status);
        Task<bool> CheckAccountBalanceAsync(Guid accountId,
decimal amountFrom, string currencyFrom);
    }
}
```

TransactionRepository

```
using Microsoft.EntityFrameworkCore;
using TransactionService.Data;
using TransactionService.Models;

namespace TransactionService.Repositories
{
    public class TransactionRepository(TransactionDbContext
context) : ITransactionRepository
    {
        public async Task AddAsync(Transaction transaction)
        {
            await context.Transactions.AddAsync(transaction);
            await context.SaveChangesAsync();
        }

        public async Task
AddCurrencyExchangeRequestAsync(CurrencyExchangeRequest
currencyExchangeRequest)
        {
```

```
            await
context.CurrencyExchangeRequests.AddAsync(currencyExchangeReques
t);
            await context.SaveChangesAsync();
        }

        public async Task<ICollection<CurrencyExchangeRequest>>
GetCurrencyExchangeRequestListAsync()
        {
            return await
context.CurrencyExchangeRequests.ToListAsync();
        }

        public async Task<ICollection<CurrencyExchangeRequest>>
GetCurrencyExchangeRequestListByUserIdAsync(Guid userId)
        {
            return await
context.CurrencyExchangeRequests.Where(id => id.UserId ==
userId).ToListAsync();
        }

        public async Task<IEnumerable<Transaction>>
GetTransactionByUserIdAsync(int userId)
        {
            return await context.Transactions.ToListAsync();
        }

        public async Task
UpdateRequestAsync(CurrencyExchangeRequest request)
        {
            context.CurrencyExchangeRequests.Update(request);
            await context.SaveChangesAsync();
        }
    }
}
                    ITransactionRepository
using TransactionService.Models;

namespace TransactionService.Repositories
{
    public interface ITransactionRepository
    {
        Task AddAsync(Transaction transaction);
        Task<IEnumerable<Transaction>>
GetTransactionByUserIdAsync(int userId);
        Task
AddCurrencyExchangeRequestAsync(CurrencyExchangeRequest
currencyExchangeRequest);
        Task<ICollection<CurrencyExchangeRequest>>
GetCurrencyExchangeRequestListByUserIdAsync(Guid userId);
        Task<ICollection<CurrencyExchangeRequest>>
GetCurrencyExchangeRequestListAsync();
```

```
        Task UpdateRequestAsync(CurrencyExchangeRequest
request);
    }
}
                        TransactionController
using Microsoft.AspNetCore.Mvc;
using TransactionService.Models;
using TransactionService.Services;

namespace ExchangeService.Controllers
{
    [ApiController]
    [Route("api/transaction")]
    public class TransactionController(ITransactionService
transactionService) : ControllerBase
    {
        [HttpPost(nameof(CreateTransaction))]
        public async Task<IActionResult>
CreateTransaction([FromBody] TransactionRequest request)
        {
            bool success = await
transactionService.CheckAccountBalanceAsync(request.accountId,
request.amountFrom, request.currencyFrom);
            if (!success)
            {
                return BadRequest("У користувача недостатньо
коштів на рахунку.");
            }
            await
transactionService.CreateTransactionAsync(request.accountId,
request.amountFrom, request.currencyFrom, request.amountTo,
request.currencyTo);

            return Ok("Balance updated");
        }

        [HttpPost(nameof(CreateCurrencyExchangeRequest))]
        public async Task<IActionResult>
CreateCurrencyExchangeRequest([FromBody]
CurrencyExchangeRequestDto request)
        {
            var list = await
transactionService.GetCurrencyExchangeRequestListByUserIdAsync(r
equest.UserId);

            var success =
transactionService.ChechMaxRequests(list);

            if (!success)
            {
                return BadRequest("Ви можете подати лише 3 заяви
в очікуванні.");
            }
```

```
            await
transactionService.CreateCurrencyExchangeRequestAsync(request.Us
erId, request.PhoneNumber, request.ContactType,
request.AmountFrom, request.CurrencyFrom, request.AmountTo,
request.CurrencyTo);

            return Ok("Request created");
        }

        [HttpGet(nameof(GetCurrencyExchangeRequestListByUserId)
+ "/{userId}")]
        public async Task<IActionResult>
GetCurrencyExchangeRequestListByUserId(Guid userId)
        {
            var list = await
transactionService.GetCurrencyExchangeRequestListByUserIdAsync(u
serId);

            return Ok(list);
        }

        [HttpGet(nameof(GetUsersRequestsList))]
        public async Task<IActionResult> GetUsersRequestsList()
        {
            var list = await
transactionService.GetUsersRequestsListAsync();

            return Ok(list);
        }

        [HttpPut(nameof(ChangeRequestStatus))]
        public async Task<IActionResult>
ChangeRequestStatus([FromBody] ChangeStatusRequest request)
        {
            await
transactionService.ChangeRequestStatusAsync(request.accountId,
request.date, request.status);

            return Ok("Request status changed");
        }
    }
}
```

CurrencyRateServiceTests

```
using CurrencyRateService.Services;
using CurrencyRateService.Repositories;
using CurrencyRateService.Models;
using Moq;

namespace CurrencyRateService.Tests
{
    public class CurrencyRateServiceTests
    {
```

```
        [Fact]
        public async Task GetActualCurrencyRateList()
        {
            var fakeRates = new List<CurrencyRateResponse>
            {
                new CurrencyRateResponse { CurrencyBase = "USD",
Date = new DateTime(2025, 1, 1) },
                new CurrencyRateResponse { CurrencyBase = "USD",
Date = new DateTime(2025, 5, 1) },
                new CurrencyRateResponse { CurrencyBase = "EUR",
Date = new DateTime(2025, 5, 1) },
                new CurrencyRateResponse { CurrencyBase = "EUR",
Date = new DateTime(2025, 1, 1) },
            };

            var mockRepo = new Mock<ICurrencyRateRepository>();
            mockRepo.Setup(r =>
r.GetCurrencyRateResponseListAsync())
                    .ReturnsAsync(fakeRates);

            var mockProvider = new Mock<IServiceProvider>();
            var mockMarginService = new Mock<MarginService>();

            var mockService = new
Mock<CurrencyRateService.Services.CurrencyRateService>
                (
                mockRepo.Object,
                mockProvider.Object,
                mockMarginService.Object
                ) { CallBase = true };

            mockService.Setup(s => s.GetCurrencyRateListAsync())
                       .ReturnsAsync(fakeRates);

            var result = await
mockService.Object.GetActualCurrencyRateListAsync();

            Assert.Contains(result, r => r.CurrencyBase == "USD"
&& r.Date == new DateTime(2025, 5, 1));
            Assert.Contains(result, r => r.CurrencyBase == "EUR"
&& r.Date == new DateTime(2025, 5, 1));
        }
    }
}
```

TransactionServiceTests

```
using Moq;
using TransactionService.Models;
using TransactionService.Repositories;
using TransactionService.Services;

namespace TransactionService.Tests
{
    public class TransactionServiceTests
```

```
    {
        [Fact]
        public async Task CreateCurrencyExchangeRequest()
        {
            var mockRepo = new Mock<ITransactionRepository>();
            var mockHttp = new Mock<HttpMessageHandler>();
            var httpClient = new HttpClient(mockHttp.Object);
            var service = new
TransactionService.Services.TransactionService(mockRepo.Object,
httpClient);

            var userId = Guid.NewGuid();
            var phoneNumber = "1234567890";
            var contactType = "telegram";
            var amountFrom = 100m;
            var currencyFrom = "USD";
            var amountTo = 90m;
            var currencyTo = "EUR";

            CurrencyExchangeRequest? capturedRequest = null;

            mockRepo
                .Setup(r =>
r.AddCurrencyExchangeRequestAsync(It.IsAny<CurrencyExchangeReque
st>()))
                .Callback<CurrencyExchangeRequest>(r =>
capturedRequest = r)
                .Returns(Task.CompletedTask);

            await
service.CreateCurrencyExchangeRequestAsync(userId, phoneNumber,
contactType, amountFrom, currencyFrom, amountTo, currencyTo);

            mockRepo.Verify(r =>
r.AddCurrencyExchangeRequestAsync(It.IsAny<CurrencyExchangeReque
st>()), Times.Once);

            Assert.NotNull(capturedRequest);
            Assert.Equal(userId, capturedRequest!.UserId);
            Assert.Equal(phoneNumber,
capturedRequest.PhoneNumber);
            Assert.Equal(contactType,
capturedRequest.ContactType);
            Assert.Equal(amountFrom,
capturedRequest.AmountFrom);
            Assert.Equal(currencyFrom,
capturedRequest.CurrencyFrom);
            Assert.Equal(amountTo, capturedRequest.AmountTo);
            Assert.Equal(currencyTo,
capturedRequest.CurrencyTo);
            Assert.Equal("pending", capturedRequest.Status);
        }
    }
```

```
}
                        UserServiceTests
using Auth.BusinessLogic.Auth;
using Auth.Persistence.Models;
using Auth.Persistence.Repositories;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Options;
using Moq;

namespace UserService.Tests
{
    public class UserServiceTests
    {
        [Fact]
        public async Task Register_User()
        {
            var mockRepo = new Mock<IAccountRepository>();

            mockRepo.Setup(r =>
r.GetByUserNameAsync("testuser"))
                    .ReturnsAsync((Account?)null);

            mockRepo.Setup(r =>
r.AddAccountAsync(It.IsAny<Account>()))
                    .Returns(Task.CompletedTask);

            var mockAuthSettings = new
Mock<IOptions<AuthSettings>>();
            mockAuthSettings.Setup(x => x.Value).Returns(new
AuthSettings
            {
                SecretKey =
"mysecretKeywfwgwefwewefnlwefnwoenvoiwnevoiwnevoinwoeivnowinevio
nwv",
                Expires = TimeSpan.FromHours(1)
            });

            var jwtService = new
JwtService(mockAuthSettings.Object);

            var service = new AccountService(mockRepo.Object,
jwtService);

            var result = await service.RegisterAsync("testuser",
"password123");

            Assert.True(result);
        }

        [Fact]
        public async Task Login_User()
        {
            var newId = Guid.NewGuid();
```

```
            var account = new Account()
            {
                UserName = "testuser",
                Id = newId,
                Permissions = new List<Permission>
                {
                    new Permission
                    {
                        Name = Permissons.User,
                        AccountId = newId
                    }
                },

                Balances = Enum.GetValues<Currency>()
                    .Select(currency => new AccountBalance
                    {
                        AccountId = newId,
                        CurrencyCode = currency.ToString(),
                        Amount = 0m
                    })
                    .ToList()
            };

            var passHash = new
PasswordHasher<Account>().HashPassword(account, "password123");
            account.PasswordHash = passHash;

            var mockRepo = new Mock<IAccountRepository>();
            mockRepo.Setup(r =>
r.GetByUserNameAsync("testuser"))
                    .ReturnsAsync(account);

            var mockAuthSettings = new
Mock<IOptions<AuthSettings>>();
            mockAuthSettings.Setup(x => x.Value).Returns(new
AuthSettings
            {
                SecretKey =
"mysecretKeywfwgwefwewefnlwefnwoenvoiwnevoiwnevoinwoeivnowinevio
nwv",
                Expires = TimeSpan.FromHours(1)
            });

            var jwtService = new
JwtService(mockAuthSettings.Object);

            var service = new AccountService(mockRepo.Object,
jwtService);

            var (token, success) = await
service.LoginAsync("testuser", "password123");

            Assert.True(success);
```

```
            Assert.True(!string.IsNullOrWhiteSpace(token));
        }

    }
}
```

JwtService

```
using Auth.Persistence.Models;
using Microsoft.Extensions.Options;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using Microsoft.IdentityModel.Tokens;
using System.Text;

namespace Auth.BusinessLogic.Auth
{
    public class JwtService(IOptions<AuthSettings> options)
    {
        public virtual string GenerateToken(Account account)
        {
            var claims = new List<Claim>
            {
                new Claim(ClaimTypes.NameIdentifier,
account.UserName),
                new Claim("id", account.Id.ToString()),
            };

            foreach (var permission in account.Permissions)
            {
                claims.Add(new Claim(ClaimTypes.Role,
permission.Name));
            }

            var jwtToken = new JwtSecurityToken(
                expires:
DateTime.UtcNow.Add(options.Value.Expires),
                claims: claims,
                signingCredentials:
                new SigningCredentials(
                    new SymmetricSecurityKey(

Encoding.UTF8.GetBytes(options.Value.SecretKey)),
                                SecurityAlgorithms.HmacSha256));

            return new
JwtSecurityTokenHandler().WriteToken(jwtToken);
        }
    }
}
```

PermissonRequirements

```
using Microsoft.AspNetCore.Authorization;

namespace Auth.BusinessLogic.Auth
{
```

```
    public class PermissonRequirements :
IAuthorizationRequirement
    {
        public string Permisson { get; }

        public PermissonRequirements(string permisson)
        {
            Permisson = permisson;
        }
    }
}
```

PermissonRequirementsHandler

```
using Auth.Persistence.Repositories;
using Microsoft.AspNetCore.Authorization;
using Microsoft.Extensions.DependencyInjection;
using System.Security.Claims;

namespace Auth.BusinessLogic.Auth
{
    public class
PermissonRequirementsHandler(IServiceScopeFactory
serviceScopeFactory) :
AuthorizationHandler<PermissonRequirements>
    {
        protected override async Task HandleRequirementAsync(
            AuthorizationHandlerContext context,
            PermissonRequirements requirement)
        {
            var userName = context.User.Claims
                .FirstOrDefault(x => x.Type ==
ClaimTypes.NameIdentifier);
            var scope = serviceScopeFactory.CreateScope();
            var accountRepository =
scope.ServiceProvider.GetService<IAccountRepository>();
            var permissions = await
accountRepository.GetPermissionByUserNameAsync(userName.Value);

            if (permissions.Any(x => x.Name ==
requirement.Permisson))
            {
                context.Succeed(requirement);
            }
        }
    }
}
```

AuthDbContext

```
using Auth.Persistence.Models;
using Microsoft.EntityFrameworkCore;

namespace Auth.Persistence.Data
{
    public class AuthDbContext : DbContext
    {
```

```
        public AuthDbContext(DbContextOptions options) :
base(options)
        {

        }

        public DbSet<Account> Accounts { get; set; }
        public DbSet<Permission> Permissions { get; set; }
        public DbSet<AccountBalance> AccountBalances { get; set;
}
        public DbSet<Review> Reviews { get; set; }

        protected override void OnModelCreating(ModelBuilder
modelBuilder)
        {
            modelBuilder.Entity<Permission>()
                .HasOne(a => a.Account)
                .WithMany(e => e.Permissions)
                .HasForeignKey(p => p.AccountId)
                .OnDelete(DeleteBehavior.Cascade);

            base.OnModelCreating(modelBuilder);
        }
    }
}
```

CurrencyRateDbContext

```
using CurrencyRateService.Models;
using Microsoft.EntityFrameworkCore;

namespace CurrencyRateService.Data
{
    public class CurrencyRateDbContext(DbContextOptions options)
: DbContext(options)
    {
        public DbSet<CurrencyRateResponse> CurrencyRateResponses
{ get; set; }
        public DbSet<CurrencyRate> CurrencyRates { get; set; }
    }
}
```

TransactionDbContext

```
using Microsoft.EntityFrameworkCore;
using TransactionService.Models;

namespace TransactionService.Data
{
    public class TransactionDbContext(DbContextOptions options)
: DbContext(options)
    {
        public DbSet<Transaction> Transactions { get; set; }
        public DbSet<CurrencyExchangeRequest>
CurrencyExchangeRequests { get; set; }
    }
}
```

Account

```
using System.ComponentModel.DataAnnotations;

namespace Auth.Persistence.Models
{
    public class Account
    {
        [Key]
        public Guid Id { get; set; }
        [Required]
        [MaxLength(16)]
        public string UserName { get; set; }
        [Required]
        public string PasswordHash { get; set; }

        [Required]
        public ICollection<Permission> Permissions { get; set; }
        [Required]
        public ICollection<AccountBalance> Balances { get; set;
}
        [Required]
        public Review Review { get; set; }
    }
}
```

AccountBalance

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace Auth.Persistence.Models
{
    public class AccountBalance
    {
        [Key]
        public Guid Id { get; set; } = Guid.NewGuid();

        [Required]
        [ForeignKey(nameof(Account))]
        public Guid AccountId { get; set; }
        [JsonIgnore]
        public virtual Account Account { get; set; }

        [Required]
        public string CurrencyCode { get; set; }

        [Required]
        [Column(TypeName = "decimal(18,4)")]
        public decimal Amount { get; set; }
    }
}
```

Permission

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
```

```
using System.Text.Json.Serialization;

namespace Auth.Persistence.Models
{
    public class Permission
    {
        [Key]
        public int Id { get; set; }

        [Required]
        [MaxLength(16)]
        public string Name { get; set; }

        [Required]
        [ForeignKey(nameof(Account))]
        public Guid AccountId { get; set; }
        [JsonIgnore]
        public virtual Account Account { get; set; }
    }
}
```

Review

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Text.Json.Serialization;

namespace Auth.Persistence.Models
{
    public class Review
    {
        [Key]
        public Guid Id { get; set; }
        [Required]
        public string Comment {  get; set; }
        [Range(1, 5, ErrorMessage = "Значение должно быть от 1
до 5.")]
        public int Rating { get; set; }
        [Required]
        public DateTime Date { get; set; }
        [Required]
        [ForeignKey(nameof(Account))]
        public Guid AccountId { get; set; }
        [JsonIgnore]
        public virtual Account Account { get; set; }
    }
}
```

Currency

```
namespace Auth.Persistence.Models
{
    public enum Currency
    {
        UAH,
        USD,
        EUR,
```

```
        PLN,
        GBP,
        CHF
    }
}
```

CurrencyRate

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Text.Json.Serialization;

namespace CurrencyRateService.Models
{
    public class CurrencyRate
    {
        [Key]
        public int Id { get; set; }

        [ForeignKey(nameof(CurrencyRateResponse))]
        public Guid CurrencyRateResponseId { get; set; }
        [JsonIgnore]
        public virtual CurrencyRateResponse CurrencyRateResponse
{ get; set; }

        [Required]
        public string Currency { get; set; }
        [Required]
        public decimal Value { get; set; }
    }
}
```

Додаток Б – Посилання на веб-застосунок

https://reactwebapp-z1ek.onrender.com/