Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

Мобільний застосунок для автоматизації розподілу обов'язків у сім'ї
Назва теми

Рівень вищої освіти Перший (бакалаврський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр КвРІПЗс.230129.01.02.ПЗ

Виконав студент III курсу, група ІПЗс-23-1 _____ Родіон ВАНДОЛЯК
Підпис Ім'я, ПРІЗВИЩЕ

Керівник старший викладач _____ Ганна БЕДРАТЮК
Науковий ступінь, вчене звання Підпис Ім'я, ПРІЗВИЩЕ

Нормоконтролер кандидат технічних наук,доцент _____ Юрій ФОРКУН
Посада Підпис Ім'я, ПРІЗВИЩЕ

До захисту допускаю:
Завідувач кафедри інженерії
програмного забезпечення _____ Леонід БЕДРАТЮК
Підпис Ім'я, ПРІЗВИЩЕ

1 червня 2026 р.

Хмельницький 2026

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Інженерії програмного забезпечення

Рівень вищої освіти Перший (бакалаврський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІПЗ
Л. П. Бедратюк
20 01 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Вандоляку Родіону Анатолійовичу
Прізвище, ім'я, по батькові студента

1. Тема роботи Мобільний застосунок для автоматизації розподілу обов'язків в сім`ї

Керівник роботи Бедратюк Ганна Іванівна, старший викладач
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 20.01.2026 р. № 7

2. Строк подання студентом роботи на кафедру 01.06.2026 р.

3. Вихідні дані до роботи Матеріали переддипломної практики

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити) Дослідження предметної області та постановка задачі, проєктування програмного забезпечення, програмна реалізація та тестування

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

1. Діаграма варіантів використання
2. Діаграма зв'язків модулів
3. Діаграма класів

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Форкун Ю.В., доцент	05.05.2026	05.25.2026
Антиплагіат	Форкун Ю. В., доцент	05.05.2026	05.25.2026

7. Дата видачі завдання « 20 » січня 2026 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1 Ознайомлення з тематикою дипломного проєктування, визначення та узгодження індивідуальної теми кваліфікаційної роботи (КвР)	01.12– 31.12.2026	
2 Збір матеріалу за темою КвР; дослідження предметної області, в якій планується використання програмного забезпечення (ПЗ), визначення задач та вимог.	01.01 – 20.02.2026	
3 Проєктування програмного забезпечення	21.02 – 20.03 2026	
4 Програмна реалізація з використанням відповідних засобів розроблення. Тестування ПЗ	21.03 – 30.04.2026	
5 Написання вступу, загальних висновків, оформлення переліку джерел посилання та додатків. Оформлення пояснювальної записки КвР згідно вимог	01.05 – 25.05.2026	
6 Попередній захист КвР	Травень	Згідно графіка
7 Перевірка КвР на плагіат, нормоконтроль, отримання відгуків, рецензій та інших супровідних документів. Брошурування (зшиття) пояснювальної записки.	26.05 – 30.05.2026	
8 Здача КвР на кафедру; підготовка КвР для розміщення у репозитарії ХНУ; підготовка до захисту та захист КвР	з 01.06.2026	

Студент _____ Родіон ВАНДОЛЯК
Підпис Ім'я, ПРІЗВИЩЕ

Керівник роботи _____ Ганна БЕДРАТЮК
Підпис Ім'я, ПРІЗВИЩЕ

АНОТАЦІЯ

Тема кваліфікаційної роботи «Мобільний застосунок для автоматизації розподілу домашніх обов'язків в сім'ї».

Автор роботи: Вандоляк Родіон Анатолійович.

Керівник роботи: Бедратюк Ганна Іванівна.

Пояснювальна записка: 78 с., 14 рисунків, 4 табл., 4 дод., 40 джерел.

Графічна частина: 3 креслення ф. А3.

МОБІЛЬНИЙ ЗАСТОСУНОК, JS, SQL, Kotlin.

Мета кваліфікаційної роботи: розроблення мобільного застосунку для автоматизації розподілу обов'язків у сім'ї та організації спільного побуту.

У кваліфікаційній роботі проведено аналіз предметної області та наявного програмного забезпечення, визначено функціональні та нефункціональні вимоги до застосунку, розроблено загальну архітектуру клієнт-серверної системи, спроєктовано структуру бази даних та основні модулі програмного продукту.

Для реалізації програмного продукту використано мову програмування Kotlin, фреймворк Jetpack Compose, середовище розробки Android Studio, а також серверну частину на базі Node.js, Express.js, TypeScript, PostgreSQL та Prisma ORM. За допомогою цих засобів розроблено мобільний застосунок, який забезпечує створення спільного простору для пари користувачів, ведення побутових задач, планування подій у календарі, облік фінансів.

Практичне значення результатів роботи полягає в тому, що впровадження розробленого застосунку дозволяє автоматизувати процес розподілу побутових обов'язків між членами сім'ї, покращити організацію спільного планування та підвищити прозорість фінансових взаєморозрахунків.

29.05.2026

Дата

Підпис

ЗМІСТ

					КвРІПЗс.230129.01.02.ПЗ		
		№ докум.	Підпис				
Розроб.		Вандоляк Р.А.			Мобільний застосунок для автоматизації розподілу обов'язків в сім'ї Пояснювальна записка	Літ.	Арк. 4 / Аркушів 77
Перевір.		Бедратюк Г.І.					
Реценз.							
Н. Контр.		Форкун Ю.В.				ХНУ, ІПЗс–23–	
Затверд.		Бедратюк Л.П.					

ПЕРЕЛІК СКОРОЧЕНЬ

ACID – Atomicity, Consistency, Isolation, Durability
API – Application Programming Interface
APK – Android Package Kit
AR – Augmented Reality
CAD – Computer-Aided Design
CI/CD – Continuous Integration / Continuous Deployment
CRUD – Create, Read, Update, Delete
DAP – Debug Adapter Protocol
DXF – Drawing Exchange Format
FPS – Frames Per Second
GPU – Graphics Processing Unit
HUD – Head-Up Display
IDE – Integrated Development Environment
JSON – JavaScript Object Notation
KMP – Kotlin Multiplatform
LiDAR – Light Detection and Ranging
LSP – Language Server Protocol
MAU – Monthly Active Users
MVC – Model-View-Controller
MVI – Model-View-Intent
MVP – Model-View-Presenter
MVVM – Model-View-ViewModel
PDF – Portable Document Format
PNG – Portable Network Graphics
SDK – Software Development Kit
SOC – System on a Chip
SQL – Structured Query Language
UI – User Interface
UX – User Experience
VIO – Visual-Inertial Odometry
VPS – Visual Positioning System
XML – Extensible Markup Language
СКБД – Система керування базами даних

ВСТУП

Створення цифрових засобів для організації спільного побуту є актуальним напрямом сучасного розроблення програмного забезпечення. З кожним роком мобільні застосунки дедалі активніше використовуються для планування особистого часу, керування задачами, ведення фінансів та автоматизації повсякденних процесів. Водночас значна частина існуючих програмних рішень орієнтована переважно на індивідуальне використання і не враховує специфіку спільного проживання двох людей, де важливими є розподіл обов'язків, узгодження дій, контроль виконання задач і прозоре ведення спільних витрат.

Організація спільного побуту передбачає постійну взаємодію між членами сім'ї або партнерами. До таких процесів належать планування домашніх справ, розподіл побутових задач, ведення спільного календаря, контроль витрат, планування покупок і підтримка домовленостей між користувачами. За відсутності єдиного зручного інструменту такі домовленості часто залишаються неформалізованими: частина задач обговорюється усно, частина — у месенджерах, фінанси можуть вестися у таблицях або взагалі не фіксуватися, а спільні події — зберігатися в окремих календарях.

Особливо актуальною є розробка мобільного застосунку, який об'єднує в одному середовищі задачі, календар, фінансовий облік, wishlist та елементи гейміфікації. Такий підхід дозволяє не лише зберігати інформацію про спільні справи, але й формувати прозору систему відповідальності між партнерами. Наприклад, користувачі можуть створювати побутові задачі, призначати їх на конкретні дати, підтверджувати виконання, отримувати бали та використовувати їх у внутрішньому магазині винагород.

Важливим чинником розробки цього проєкту є відсутність достатньої кількості комплексних рішень, орієнтованих саме на пару або невелику сім'ю. Існуючі застосунки часто вирішують лише окремі задачі: календарі забезпечують планування подій, таск-менеджери дозволяють створювати списки справ, фінансові сервіси допомагають обліковувати витрати, а wishlist-застосунки

використовуються для планування покупок. Однак такі інструменти не формують єдиного інформаційного простору, у якому всі ці процеси пов’язані між собою.

Важливим є також підвищення мотивації користувачів до виконання побутових обов’язків. Для цього в застосунку доцільно використовувати елементи гейміфікації: систему балів, банк задач, підтвердження виконання партнером, спільні задачі з розподілом винагороди та внутрішній магазин бонусів. Такий підхід дозволяє зробити виконання побутових задач більш прозорим, контрольованим і менш конфліктним.

Саме тому розроблення мобільного застосунку для автоматизації розподілу обов’язків у сім’ї є доцільним і практично значущим. Якісне проєктування та програмна реалізація такого програмного продукту дозволяє поглибити знання з мобільної розробки, клієнт-серверної архітектури, роботи з базами даних, проєктування інтерфейсів, авторизації користувачів, синхронізації даних і побудови складної бізнес-логіки. У підсумку це свідчить про достатній рівень підготовки фахівця та може стати основою для подальшого розвитку повноцінного програмного продукту.

Метою кваліфікаційної роботи є розроблення мобільного застосунку для автоматизації розподілу обов’язків у сім’ї, який забезпечує створення спільного простору для двох користувачів, ведення побутових задач, календарне планування, фінансовий облік, wishlist, систему балів, внутрішній магазин винагород та персоналізацію профілю користувача.

Для успішного досягнення мети роботи необхідно розв’язати такі завдання:

- провести змістовий аналіз предметної області автоматизації розподілу побутових обов’язків у сім’ї;
- проаналізувати наявне програмне забезпечення для планування, задач, фінансів, wishlist і гейміфікації;
- сформувати перелік функціональних та нефункціональних вимог до розроблюваного застосунку;

- здійснити вибір архітектурних підходів, шаблонів проєктування та засобів зберігання даних;
- виконати декомпозицію системи на функціональні модулі та описати їхні залежності;
- спроєктувати структуру бази даних і основні моделі предметної області;
- проаналізувати та обрати технології реалізації клієнтської і серверної частин застосунку;
- реалізувати Android-застосунок з використанням Kotlin та Jetpack Compose;
- реалізувати backend API для авторизації, пар, задач, календаря, фінансів, wishlist, профілів, винагород і повідомлень;
- забезпечити синхронізацію даних між двома користувачами через серверну частину;
- реалізувати підтримку локалізації, тем оформлення та вибору валюти;
- виконати тестування основних сценаріїв роботи застосунку;
- проаналізувати отримані результати та сформулювати висновки.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Змістовий аналіз предметної області, її структурних та функціональних особливостей

Дослідження предметної області спрямоване на виявлення наявних проблем та технологічних прогалин у процесах організації спільного побуту, розподілу домашніх обов'язків, планування спільного часу та ведення сімейних фінансів. На основі проведеного аналізу формується обґрунтування для розробки мобільного застосунку, що дозволяє автоматизувати взаємодію між членами сім'ї та підвищити ефективність організації повсякденного життя.

Сучасний рівень розвитку мобільних технологій перетворює смартфони на універсальні інструменти для планування, комунікації, фінансового обліку та контролю виконання задач. У побутовому середовищі такі інструменти можуть застосовуватися для координації дій між членами сім'ї, фіксації домовленостей, розподілу відповідальності та відстеження виконання спільних справ.

Предметна область ґрунтується на розгляді сім'ї або пари, що проживає разом, як невеликої соціально-побутової системи, у якій існують спільні задачі, плани, фінансові зобов'язання та потреба у взаємній координації. Ключовими об'єктами дослідження є користувачі системи, побутові задачі, календарні події, фінансові записи, бажані покупки, система балів та винагород.

Розподіл обов'язків у сім'ї — це процес визначення, закріплення та контролю виконання побутових задач між членами сім'ї. До таких задач можуть належати прибирання, приготування їжі, закупівля продуктів, оплата рахунків, догляд за житлом, планування подій або виконання регулярних домашніх справ. У разі відсутності прозорого механізму розподілу обов'язків можуть виникати непорозуміння, нерівномірне навантаження та зниження відповідальності за виконання спільних задач.

Автоматизація розподілу обов’язків полягає у створенні цифрового середовища, в якому користувачі можуть створювати задачі, призначати їх собі або партнеру, встановлювати дату виконання, підтверджувати результат та відстежувати прогрес. Додаткове використання системи балів дозволяє застосувати елементи гейміфікації, що підвищує мотивацію користувачів до виконання побутових задач.

Типовий процес взаємодії користувача з подібною системою полягає у реєстрації, створенні спільного простору для пари, додаванні задач або подій, визначенні відповідального користувача, контролі виконання та фіксації результату. У випадку фінансового модуля користувачі також можуть додавати доходи, витрати, визначати категорії транзакцій та переглядати баланс між партнерами.

Потреба в автоматизації цього процесу зумовлена тим, що значна частина побутових домовленостей у повсякденному житті залишається неформалізованою. Задачі можуть обговорюватися усно або в месенджерах, фінансові витрати — фіксуватися нерегулярно, а спільні плани — зберігатися в різних календарях або взагалі не документуватися. Це ускладнює контроль виконання домовленостей і створює передумови для конфліктів або дублювання дій.

Наявність єдиного мобільного застосунку для організації спільного побуту дозволяє об’єднати кілька важливих функцій:

- створення та розподіл побутових задач;
- планування подій у спільному календарі;
- ведення обліку доходів і витрат;
- формування списку бажаних покупок;
- використання системи балів і винагород;
- персоналізацію профілю користувача;
- підтримку взаємодії між двома партнерами.

Особливістю предметної області є те, що більшість дій одного користувача впливають на дані іншого. Наприклад, створена задача може бути призначена партнеру, фінансова транзакція може потребувати підтвердження, а покупка винагороди за бали може створювати нові очікування або домовленості між користувачами. Тому система повинна забезпечувати синхронізацію даних, контроль доступу та чітке розмежування дій між користувачами.

Прикладом практичного використання такої системи може бути організація щотижневого прибирання квартири. Один із користувачів створює повторювану задачу, яка автоматично з’являється у календарі. Після виконання задачі партнер підтверджує результат, а користувачі отримують відповідну кількість балів. Надалі ці бали можуть бути використані у внутрішньому магазині винагород, наприклад для вибору спільного дозвілля.

Класичним підходом до розв’язання подібних задач є використання окремих інструментів: календаря для подій, нотаток або месенджерів для задач, банківських застосунків або таблиць для фінансів. Проте такий підхід є фрагментованим, оскільки дані зберігаються в різних системах і не утворюють єдиного інформаційного простору для пари.

Сучасні програмні рішення частково покривають окремі аспекти предметної області. Наприклад, календарні застосунки дозволяють планувати події, таск-менеджери — створювати списки справ, а фінансові сервіси — вести облік витрат. Однак більшість таких рішень не орієнтовані саме на спільний побут пари та не поєднують задачі, календар, фінанси й гейміфікацію в межах однієї системи.

Розроблюваний застосунок спрямований на вирішення цієї проблеми шляхом створення єдиного мобільного середовища для автоматизації розподілу обов’язків у сім’ї. Система забезпечує не лише створення і контроль задач, а й підтримує календарне планування, фінансовий облік, wishlist, профілі користувачів, систему балів, винагороди та внутрішні повідомлення.

Застосування мобільного застосунку на базі операційної системи Android дозволяє забезпечити доступність рішення для широкого кола користувачів. Android-пристрої є поширеними, мають достатні обчислювальні можливості для роботи з клієнт-серверними застосунками та підтримують сучасні засоби створення зручного інтерфейсу користувача.

Вхідними даними в межах предметної області є дії користувачів, зокрема створення задач, подій, фінансових записів, wishlist-елементів, вибір налаштувань, підтвердження виконання задач або фінансових операцій. Вихідними даними є оновлений стан спільного простору пари: список актуальних задач, календар подій, баланс фінансів, кількість балів, придбані винагороди та повідомлення для користувачів.

Таким чином, предметна область розроблюваного застосунку охоплює комплексну організацію спільного побуту пари. Її структурні особливості полягають у взаємозв'язку користувачів, задач, подій, фінансів і системи мотивації, а функціональні особливості — у необхідності автоматизованого розподілу обов'язків, синхронізації даних та підтримки зручної взаємодії між партнерами.

1.2 Аналіз наявного програмно-технічного забезпечення предметної області

Сучасний ринок програмного забезпечення для організації спільного побуту, планування задач та ведення сімейних фінансів представлений значною кількістю мобільних і вебзастосунків. Проте більшість таких рішень орієнтовані на розв'язання окремих задач: ведення календаря, керування списком справ, фінансовий облік або планування покупок. Комплексні системи, які одночасно враховують потреби пари або сім'ї у розподілі обов'язків, спільному календарі, фінансах та мотиваційній системі, представлені значно менше.

Наявні програмні засоби цієї предметної області можна умовно поділити на кілька груп:

- застосунки для керування задачами та списками справ;
- календарні застосунки для планування подій;
- сервіси для спільного фінансового обліку;
- сімейні органайзери;
- застосунки для списків покупок і wishlist;
- гейміфіковані системи продуктивності.

Для розуміння необхідних функцій розроблюваної системи, якості їх реалізації та обмежень наявних рішень доцільно провести аналіз популярного програмного забезпечення, яке частково перетинається з предметною областю розробки.

Застосунок Google Calendar є одним із найпоширеніших інструментів для планування подій і керування календарем. Він дозволяє створювати події, налаштовувати нагадування, запрошувати інших користувачів та вести спільні календарі.

Переваги:

- зручне відображення подій у денному, тижневому та місячному форматах;
- підтримка спільного доступу до календаря;
- можливість створення повторюваних подій;
- інтеграція з обліковим записом Google;
- стабільна синхронізація між пристроями.

Недоліки:

- відсутність повноцінного механізму розподілу побутових задач;
- відсутність системи підтвердження виконання задач партнером;
- відсутність фінансового модуля;
- відсутність системи балів, винагород або гейміфікації;

– застосунок не орієнтований саме на організацію спільного побуту пари.

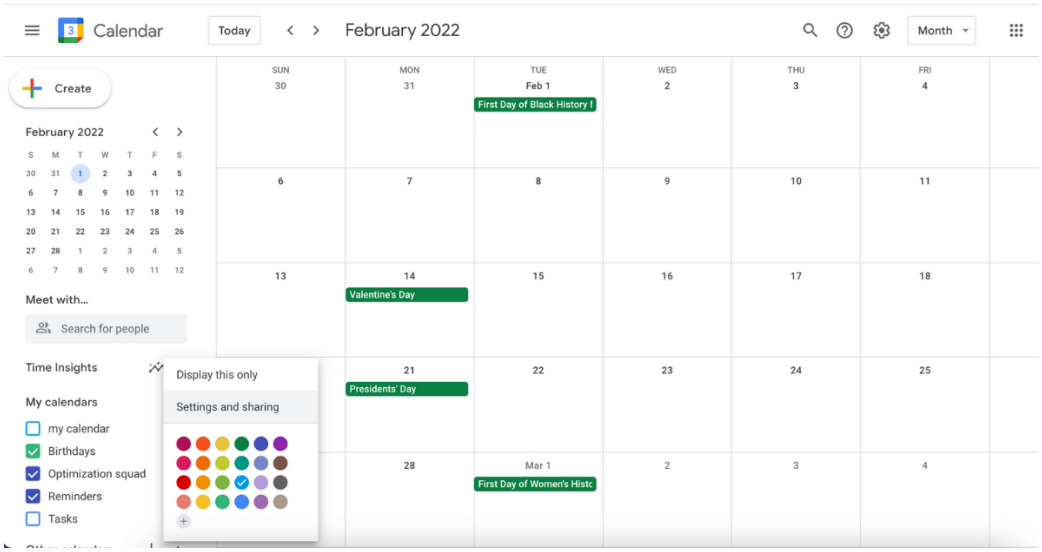


Рисунок 1.2 – Інтерфейс «GoogleCalendar»

Застосунок Todoist призначений для керування задачами, списками справ і проєктами. Він дозволяє створювати задачі, встановлювати терміни, пріоритети, повторення та організовувати задачі за проєктами.

Переваги:

– зручне створення та структурування задач;

– підтримка повторюваних задач;

– можливість спільної роботи над проєктами;

– наявність пріоритетів, міток і фільтрів;

– доступність на різних платформах.

Недоліки:

– орієнтація переважно на індивідуальну продуктивність або робочі проєкти;

– відсутність механіки взаємного підтвердження виконання задач;

– відсутність спільного побутового контексту;

– відсутність фінансового модуля;

– гейміфікація обмежена і не пов’язана з взаємодією між партнерами.

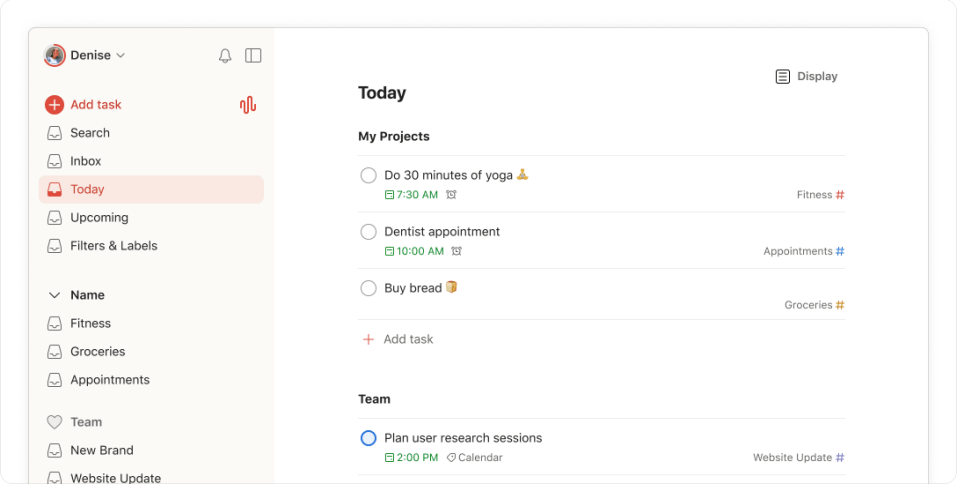


Рисунок 1.3 – Інтерфейс «Todoist»

Застосунок Splitwise використовується для обліку спільних витрат і розрахунку боргів між кількома користувачами. Він є популярним рішенням для подорожей, спільного проживання або групових витрат.

Переваги:

– зручне додавання спільних витрат;

– автоматичний розрахунок того, хто кому винен;

– підтримка груп користувачів;

– історія фінансових операцій;

– зручний механізм розділення витрат.

Недоліки:

– відсутність календаря та планування подій;

– відсутність таск-менеджера;

– відсутність wishlist або планування майбутніх покупок;

– відсутність гейміфікації;

– застосунок охоплює лише фінансову частину спільного побуту.

Застосунок Cozi Family Organizer орієнтований на організацію сімейного життя. Він поєднує календар, списки справ, списки покупок та інші засоби сімейного планування.

Переваги:

– наявність сімейного календаря;

– підтримка списків справ і покупок;

– орієнтація саме на сімейне використання;

– можливість спільного доступу для кількох користувачів;

– поєднання кількох побутових функцій в одному застосунку.

Недоліки:

– інтерфейс і функціональність більше орієнтовані на сім'ї з дітьми, а не на пари;

– відсутність розвиненої системи балів і винагород;

– відсутність механіки підтвердження задач партнером;

– недостатня гнучкість для побудови ігрової системи мотивації.

Рисунок 1.4 – Інтерфейс «Cozi Family Organizer»

Застосунок OurHome є прикладом гейміфікованого підходу до виконання домашніх задач. Він дозволяє створювати завдання, призначати їх членам сім'ї та нараховувати бали за виконання.

Переваги:

– підтримка домашніх задач;

– наявність системи балів;

– можливість призначення задач користувачам;

– мотивація користувачів через винагороди.

Недоліки:

– застосунок переважно орієнтований на сім’ї з дітьми;

– обмежені можливості фінансового обліку;

– відсутність повноцінного wishlist;

– календарна складова є менш розвиненою;

– відсутня специфічна логіка для взаємодії саме двох партнерів.

Рисунок 1.5 – Інтерфейс «OurHome»

Застосунок Tody призначений для організації прибирання та регулярних домашніх справ. Основна увага приділяється контролю стану побутових задач і нагадуванням про необхідність їх виконання.

Переваги:

– зручний облік регулярних домашніх справ;

– візуалізація стану виконання задач;

– підтримка повторюваних побутових дій;

– орієнтація на домашнє середовище;
– можливість структурування задач за приміщеннями або категоріями.

Недоліки:

– вузька спеціалізація на прибиранні;
– відсутність фінансового модуля;
– відсутність календаря спільних подій у широкому розумінні;
– відсутність wishlist;
– обмежена взаємодія між партнерами.

Рисунок 1.6 – Інтерфейс «Tody»

Застосунок AnyList орієнтований на створення спільних списків покупок і планування придбань. Він може використовуватися парами або сім'ями для ведення спільних списків.

Переваги:

– зручне створення списків покупок;
– можливість спільного доступу;
– синхронізація між користувачами;
– підтримка категорій товарів

Недоліки:

– відсутність задач і календаря;

– відсутність фінансової аналітики;

– відсутність системи балів;

– немає механізму підтвердження витрат або задач;

– застосунок вирішує лише окрему задачу планування покупок.

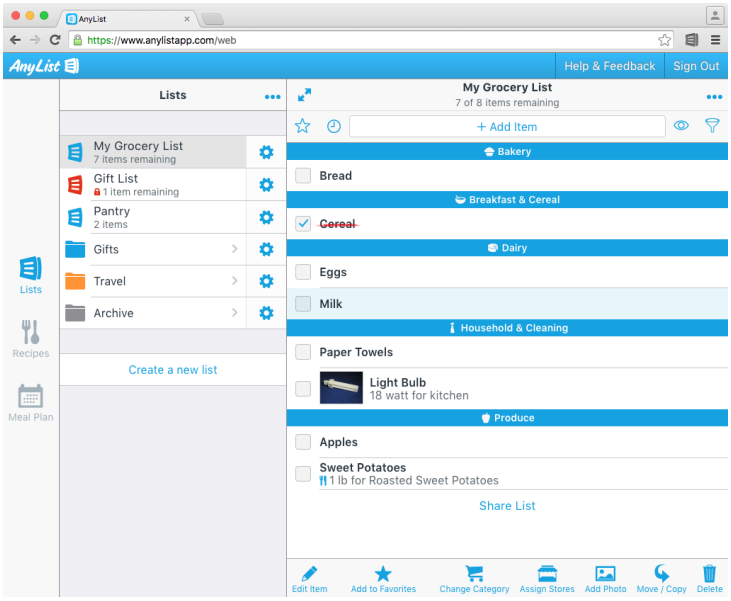


Рисунок 1.7 – Інтерфейс «AnyList»

Застосунок Habitica є гейміфікованим таск-менеджером, який перетворює виконання задач на рольову гру. Користувач отримує досвід, винагороди та штрафи залежно від виконання завдань.

Переваги:

– сильна гейміфікація;

– мотивація користувача через винагороди;

– підтримка щоденних і повторюваних задач;

– можливість спільної активності;

– цікавий ігровий підхід до продуктивності.

Недоліки:

– орієнтація на індивідуальну продуктивність, а не на спільний побут;

– відсутність фінансового обліку;

– відсутність побутового календаря для пари;

– складність інтерфейсу для простих побутових задач;

– ігрова модель не враховує реальний розподіл сімейних обов’язків.

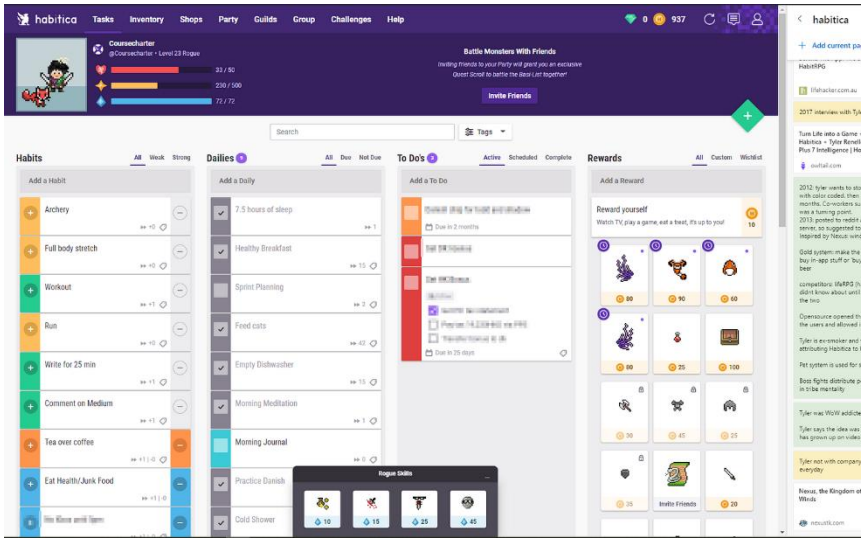


Рисунок 1.8 – Інтерфейс «Habitica»

Порівняння розглянутих застосунків подано у таблиці 1.

Таблиця 1 – Порівняння застосунків

Назва застосунку	**Основне призначення**	**Спільне використання**	**Календар**	**Фінанси**	**Гейміфікація**
Google Calendar	Планування подій	Так	Так	Ні	Ні
Todoist	Керування задачами	Так	Частково	Ні	Частково
Splitwise	Облік витрат	Так	Ні	Так	Ні
Cozi Family Organizer	Сімейний органайзер	Так	Так	Частково	Ні
OurHome	Домашні задачі з балами	Так	Частково	Ні	Так

Продовження Таблиця 1 – Порівняння застосунків

Назва застосунку	Основне призначення	Спільне використання	Календар	Фінанси	Гейміфікація
Tody	Організація прибирання	Частково	Частково	Ні	Ні
AnyList	Списки покупок	Так	Ні	Ні	Ні
Habitica	Гейміфікована продуктивність	Частково	Частково	Ні	Так
Розроблюваний застосунок	Організація спільного побуту пари	Так	Так	Так	Так

Аналіз наявного програмного забезпечення показує, що кожне з розглянутих рішень покриває лише окрему частину потреб користувачів. Календарні застосунки забезпечують планування подій, таск-менеджери дозволяють створювати списки справ, фінансові сервіси допомагають розраховувати спільні витрати, а гейміфіковані застосунки мотивують користувачів до виконання задач.

Водночас більшість наявних рішень не забезпечують єдиного середовища, у якому пара може одночасно розподіляти побутові обов’язки, планувати спільні події, вести фінансовий облік, створювати wishlist та використовувати систему балів і винагород. Окремою проблемою є відсутність механізму взаємного підтвердження дій, що є важливим для задач, фінансових операцій та спільних домовленостей.

Отже, існує потреба у створенні нового мобільного застосунку, який поєднує основні функції організації спільного побуту в межах однієї системи. Розроблюване програмне забезпечення має усунути фрагментованість наявних рішень і забезпечити зручну взаємодію між двома користувачами, що проживають разом.

1.3 Визначення функціональних та нефункціональних вимог до програмного забезпечення

Наявність на ринку численних програмних рішень для планування подій, ведення списків задач, обліку фінансів і організації сімейного побуту зумовлює недоцільність повного копіювання функцій наявних застосунків. З огляду на обмеженість часових та технічних ресурсів, розробка зосереджена на створенні спеціалізованого мобільного застосунку для автоматизації розподілу обов'язків у сім'ї, який поєднує кілька ключових напрямів в одному інформаційному середовищі.

Аналіз предметної області та наявних програмних засобів підтвердив необхідність створення застосунку, орієнтованого саме на спільний побут двох користувачів. На відміну від універсальних таск-менеджерів, календарів або фінансових сервісів, розроблювана система має враховувати взаємну участь партнерів у виконанні задач, підтвердженні дій, веденні спільних витрат та використанні мотиваційної системи.

Основною метою розроблюваного програмного забезпечення є автоматизація процесів розподілу побутових обов'язків, планування спільного часу, ведення спільного фінансового обліку та підвищення мотивації користувачів за допомогою системи балів і винагород.

Функціональні вимоги. До функціональних вимог розроблюваного мобільного застосунку належать:

- реєстрація нового користувача;
- авторизація користувача за допомогою email та пароля;
- збереження сесії користувача за допомогою JWT-токена;
- автоматичний перехід до відповідного екрана залежно від стану авторизації та наявності пари;
- створення пари користувачів;
- підключення партнера до пари за кодом;
- розформування пари користувачів;

- вихід користувача з облікового запису;
- перегляд профілю користувача;
- редагування нікнейму користувача;
- вибір або завантаження аватарки користувача;
- перегляд профілю партнера без можливості його редагування;
- відображення кількості балів користувача та партнера;
- щотижневе нарахування балів користувачам;
- створення побутових задач;
- створення гейміфікованих задач-викликів;
- створення спільних задач;
- призначення задачі на конкретну дату;
- створення повторюваних задач;
- зупинка повторення задачі;
- редагування та видалення задач;
- підтвердження виконання задачі партнером;
- повернення задачі партнеру в межах гейміфікованої логіки;
- фіксація провалу або відхилення задачі;
- розподіл банку балів у спільних задачах за погодженням партнерів;
- перегляд задач у календарі;
- створення календарних подій;
- створення повторюваних подій;
- редагування та видалення подій;
- перегляд подій і задач у місячному календарі;
- відкриття детальної інформації про день календаря;
- створення шаблонів задач і подій;
- використання шаблонів для швидкого створення записів;
- створення фінансових транзакцій;
- поділ транзакцій на доходи та витрати;

– вибір категорії фінансової транзакції;
– вибір типу транзакції: для себе, для партнера або спільна;
– підтвердження фінансових транзакцій партнером для спільних витрат;
– редагування та видалення фінансових записів;
– розрахунок загального бюджету пари;
– розрахунок балансу між партнерами;
– відображення фінансової історії;
– створення елементів wishlist;
– додавання посилання, ціни, пріоритету та призначення wishlist-елемента;
– відкриття посилань wishlist у браузері;
– редагування та видалення wishlist-елементів;
– позначення wishlist-елемента як придбаного;
– автоматичне створення фінансової транзакції після покупки з wishlist;
– вибір категорії транзакції під час покупки з wishlist;
– перегляд внутрішнього магазину винагород;
– покупка винагород за бали;
– створення внутрішнього повідомлення для партнера після покупки винагороди;
– перегляд внутрішніх повідомлень;
– позначення повідомлень як прочитаних;
– вибір теми інтерфейсу;
– вибір мови застосунку;
– вибір валюти для відображення фінансових даних.

Детальний опис варіантів використання може бути подано у додатку до кваліфікаційної роботи у вигляді діаграми варіантів використання.

Нефункціональні вимоги. До нефункціональних вимог розроблюваного програмного забезпечення належать вимоги до захисту даних, зручності

використання, продуктивності, портативності, масштабованості та підтримуваності системи.

Вимоги до захисту даних: застосунок повинен забезпечувати доступ до персональних та спільних даних лише авторизованим користувачам. Усі захищені запити до серверної частини мають виконуватися з використанням JWT-токена. Користувач повинен мати доступ лише до даних власної пари. У разі завершення строку дії токена система повинна очищати локальну сесію та надавати можливість повторного входу.

Вимоги до цілісності даних: операції, пов’язані з нарахуванням або списанням балів, підтвердженням задач, фінансовими транзакціями та покупками винагород, повинні виконуватися узгоджено. Система не повинна допускати від’ємного балансу балів користувача або некоректного стану задачі чи транзакції.

Вимоги до зручності використання: інтерфейс застосунку має бути зрозумілим для користувачів без спеціальної технічної підготовки. Основні дії — створення задачі, події, фінансового запису або wishlist-елемента — повинні виконуватися за мінімальну кількість кроків. Форми створення записів мають відкриватися лише після натискання відповідної кнопки, щоб не перевантажувати основні екрани.

Вимоги до інтерфейсу користувача: інтерфейс має бути адаптований до мобільних пристроїв, підтримувати світлу і темну тему, а також коректно працювати у книжковій та альбомній орієнтації. Календар має відображати дні у вигляді сітки з можливістю перегляду задач і подій за обраний день. Вибір параметрів у формах повинен відображатися візуально, а не через текстові написи типу “selected”.

Вимоги до локалізації: застосунок повинен підтримувати українську та англійську мови. Усі текстові елементи інтерфейсу мають зберігатися у ресурсах локалізації, що забезпечує можливість подальшого додавання інших мов.

Вимоги до портативності: клієнтська частина застосунку повинна працювати на мобільних пристроях з операційною системою Android версії 8.0 або вище. Серверна частина має бути придатною для розгортання на хмарній платформі та взаємодії з віддаленою базою даних PostgreSQL.

Вимоги до продуктивності: застосунок повинен забезпечувати коректне відображення основних екранів без помітних затримок. Завантаження списків задач, подій, транзакцій і wishlist-елементів має виконуватися у прийнятний для користувача час за умови стабільного інтернет-з'єднання.

Вимоги до надійності: система повинна коректно обробляти помилки мережі, помилки авторизації та некоректні дії користувача. У разі невдалого запиту користувач має отримувати зрозуміле повідомлення про помилку без аварійного завершення роботи застосунку.

Вимоги до масштабованості: архітектура застосунку повинна дозволяти подальше розширення функціоналу, зокрема додавання push-сповіщень, розширеної фінансової аналітики, нових типів винагород, додаткових мов локалізації та покращених механізмів синхронізації.

Вимоги до підтримуваності: програмний код має бути структурований за модулями, з розділенням клієнтської частини, серверної логіки та роботи з базою даних. На мобільній стороні доцільно застосовувати архітектурний підхід MVVM, а на серверній стороні — розділення маршрутів, контролерів, middleware та моделей бази даних.

Вимоги до відображення даних: задачі та події повинні відображатися укалендарі за відповідними датами. Фінансові записи мають відображатися з урахуванням типу операції, категорії, валюти та статусу підтвердження. Кількість балів повинна відображатися у вигляді ігрової валюти або спеціального візуального елемента, а не лише текстовим написом.

Вимоги до взаємодії користувачів: система повинна враховувати, що застосунок використовується двома користувачами, об'єднаними в пару. Дії одного користувача можуть потребувати підтвердження партнера, зокрема

у випадку виконання задач, розподілу балів у спільних задачах або створення спільних фінансових транзакцій.

1.4 Висновки. Постановка задачі

У цьому розділі розглянуто предметну область автоматизації розподілу побутових обов’язків у сім’ї, а також вивчено наявні підходи до організації спільного планування, керування задачами, фінансового обліку та мотивації користувачів.

У результаті аналізу встановлено, що наявні програмні рішення здебільшого охоплюють лише окремі аспекти спільного побуту. Календарні застосунки забезпечують планування подій, таск-менеджери дозволяють створювати списки справ, фінансові сервіси допомагають обліковувати витрати, а гейміфіковані застосунки мотивують користувача до виконання задач. Проте більшість таких рішень не забезпечують комплексного підходу до автоматизації побутової взаємодії саме між двома партнерами.

Аналіз предметної області дозволив сформувати перелік функціональних та нефункціональних вимог до розроблюваного мобільного застосунку. На основі цих вимог необхідно розробити клієнт-серверну систему, яка забезпечує створення спільного простору для пари, розподіл і підтвердження побутових задач, ведення календаря, фінансовий облік, wishlist, систему балів, винагороди та персоналізацію користувацького профілю.

Для досягнення поставленої мети необхідно розв’язати такі задачі:

- провести аналіз архітектурних підходів і шаблонів проєктування, придатних для реалізації мобільного клієнт-серверного застосунку;
- спроєктувати структуру системи, виконати її декомпозицію та визначити взаємозв’язки між основними модулями;
- розробити модель даних для зберігання користувачів, пар, задач, подій, фінансових транзакцій, wishlist-елементів, винагород і повідомлень;

- розробити макетне представлення інтерфейсу користувача;
- провести аналіз і вибір технологій для реалізації клієнтської та серверної частин системи;
- реалізувати backend API для авторизації, роботи з парами, задачами, календарем, фінансами, wishlist, профілями та винагородами;
- реалізувати мобільний застосунок для Android з використанням сучасних засобів побудови інтерфейсу;
- забезпечити синхронізацію даних між партнерами через серверну частину;
- реалізувати механізми підтвердження задач і фінансових операцій;
- реалізувати підтримку локалізації, тем оформлення та вибору валюти;
- виконати тестування основних сценаріїв роботи застосунку;
- сформулювати висновки на основі результатів виконаної кваліфікаційної роботи.

2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Вибір типу архітектури та шаблонів проєктування

У процесі проєктування мобільного застосунку для автоматизації розподілу обов'язків у сім'ї необхідно визначити архітектурний підхід, який забезпечить зручну підтримку коду, синхронізацію даних між користувачами та можливість подальшого розширення функціоналу. Розроблювана система складається з мобільного Android-застосунку, серверної частини та бази даних, тому архітектура повинна враховувати як особливості клієнтської розробки, так і вимоги до backend API.

Класичним підходом до побудови Android-застосунків є шаблон Model-View-Controller. У цьому шаблоні Model відповідає за дані та бізнес-логіку, View — за відображення інтерфейсу, а Controller — за обробку дій користувача і керування взаємодією між моделлю та представленням.

Основними перевагами MVC є простота реалізації, зрозумілий розподіл відповідальності та невелика кількість шаблонного коду. Проте в Android-застосунках цей підхід часто призводить до надмірного навантаження Activity або Fragment, що ускладнює підтримку коду, тестування та керування станом інтерфейсу.

Перевагою MVP є краща тестованість порівняно з MVC, оскільки логіка представлення відокремлюється від Android-компонентів. Недоліком є значна кількість шаблонного коду, необхідність створення інтерфейсів для View та Presenter, а також ускладнення структури проєкту при збільшенні кількості екранів.

Шаблон Model-View-ViewModel є одним із найпоширеніших підходів у сучасній Android-розробці. У цьому шаблоні View відповідає за відображення інтерфейсу, ViewModel — за стан екрана та обробку дій користувача, а Model або Repository — за отримання й обробку даних. ViewModel не має прямого

посилання на View, що знижує зв’язаність компонентів і спрощує керування життєвим циклом.

Шаблон Model-View-Intent базується на односпрямованому потоці даних. У ньому всі дії користувача розглядаються як наміри, які обробляються централізовано і створюють новий стан інтерфейсу. Такий підхід забезпечує високу передбачуваність стану, однак потребує більш складної структури коду та значної кількості шаблонних конструкцій.

Для розроблюваного застосунку було обрано шаблон MVVM, оскільки він добре поєднується з Jetpack Compose, дозволяє зручно керувати станом екранів, відокремлює бізнес-логіку від інтерфейсу та забезпечує можливість подальшого масштабування. Застосунок містить багато екранів: авторизацію, створення пари, календар, задачі, фінанси, wishlist, профіль, магазин винагород і налаштування. Тому використання MVVM дозволяє підтримувати зрозумілу структуру коду та уникати надмірного ускладнення UI-компонентів.

Важливим етапом проєктування є також вибір загальної архітектури системи. Оскільки застосунок повинен синхронізувати дані між двома користувачами, зберігати спільні задачі, фінансові записи та повідомлення, локального зберігання даних на одному пристрої недостатньо. Тому для реалізації системи обрано клієнт-серверну архітектуру.

Клієнт і сервер взаємодіють через REST API. Такий підхід є зрозумілим, поширеним і достатнім для реалізації основних сценаріїв застосунку. Кожен модуль системи має набір відповідних API-маршрутів: авторизація, пари, задачі, події, фінанси, wishlist, профілі, винагороди та повідомлення.

Серверна частина реалізована за модульним принципом. Вона містить маршрути, контролери, middleware, утиліти та модулі роботи з базою даних. Такий підхід можна розглядати як модульну монолітну архітектуру backend-частини: усі функціональні модулі розгортаються як один застосунок, але логічно розділені за предметними областями.

На серверній стороні використовується багатошаровий підхід:

- рівень маршрутів — приймає HTTP-запити та передає їх до відповідних контролерів;
- рівень контролерів — реалізує бізнес-логіку, перевірки та обробку даних;
- рівень middleware — відповідає за авторизацію, перевірку токена та доступ до захищених маршрутів;
- рівень доступу до даних — забезпечує взаємодію з PostgreSQL через Prisma ORM.

Для захисту API використовується JWT-авторизація. Після входу користувача сервер формує токен, який мобільний застосунок зберігає локально. Надалі токен додається до запитів, що дозволяє серверу визначати користувача та перевіряти його права доступу до даних пари.

Таким чином, поєднання клієнт-серверної архітектури, MVVM на мобільній стороні та модульної структури backend-частини забезпечує зрозумілу організацію коду, можливість розширення функціоналу, синхронізацію між користувачами та контроль доступу до спільних даних.

2.2 Проектування бази даних

Для проєктування системи необхідно визначити структуру даних, які вона зберігає та обробляє. Оскільки розроблюваний застосунок охоплює авторизацію, пари користувачів, задачі, календар, фінансовий облік, wishlist, профілі, магазин винагород і повідомлення, база даних повинна забезпечувати зберігання взаємопов’язаних сутностей і підтримувати цілісність даних.

Для реалізації застосунку обрано реляційну базу даних PostgreSQL. Такий вибір обумовлений тим, що більшість сутностей системи мають чітко визначені зв’язки між собою. Наприклад, користувач належить до пари, пара містить задачі, події, транзакції та wishlist-елементи, а задачі та фінансові записи пов’язані з конкретними користувачами.

Об’єктами предметної області, необхідними для реалізації проєкту, є:

– користувач;
– пара користувачів;
– задача;
– серія повторюваних задач;
– подія календаря;
– фінансова транзакція;

Основною сутністю системи є User. Вона зберігає дані користувача, необхідні для авторизації та персоналізації застосунку. До властивостей користувача належать ідентифікатор, email, пароль у захешованому вигляді, nickname, avatarKey або avatarUrl, кількість балів, winStreak, pairId, а також службові дані, пов’язані з датою створення та щотижневим нарахуванням балів.

Сутність Task описує побутову задачу або виклик між партнерами. Властивостями задачі є ідентифікатор, назва, опис, тип задачі, статус, банк балів, дата виконання, ідентифікатори автора та відповідального користувача, pairId, а також поля, пов’язані з підтвердженням виконання або розподілом балів у спільних задачах. Задачі можуть бути звичайними викликами або спільними задачами, що мають іншу логіку нарахування балів.

Сутність Event використовується для зберігання календарних подій і нагадувань. Вона містить назву, опис, дату, pairId, автора та параметри повторення. Події разом із задачами відображаються у спільному календарі.

Сутність Transaction описує фінансовий запис. Транзакція може бути доходом або витратою. До її властивостей належать назва, сума, тип операції, категорія, область застосування операції, статус підтвердження, автор, pairId і дата створення. Статус підтвердження використовується для випадків, коли фінансова операція стосується обох партнерів або партнера і повинна бути підтверджена іншим користувачем.

У системі доцільно використовувати реляційну модель даних, оскільки більшість об’єктів мають чіткі зв’язки та обмеження цілісності. Застосування

PostgreSQL дозволяє забезпечити надійне зберігання даних, підтримку транзакцій і коректну роботу з взаємопов’язаними сутностями. Використання Prisma ORM спрощує опис моделей, виконання міграцій та взаємодію серверної частини з базою даних.

Особливу увагу під час проєктування бази даних приділено полю pairId, яке є ключовим для ізоляції даних між різними парами користувачів. Завдяки цьому кожна пара має власний спільний простір, а користувачі не можуть отримати доступ до задач, подій, фінансів або wishlist інших пар.

Таким чином, спроєктована база даних забезпечує зберігання всіх основних об’єктів предметної області, підтримує зв’язки між користувачами та спільними даними, а також створює основу для синхронізованої роботи мобільного застосунку між двома партнерами.

2.3 Вибір системи керування базами даних

Для будь-якого програмного забезпечення, що зберігає та обробляє користувацькі дані, вибір системи керування базами даних є важливим етапом проєктування.

Оскільки застосунок передбачає взаємодію між партнерами, локальне сховище на одному Android-пристрої не може бути основним джерелом даних. У разі зберігання інформації лише локально неможливо забезпечити актуальність задач, подій, фінансів і повідомлень на пристрої другого користувача. Тому база даних повинна бути розміщена на серверній стороні та бути доступною через backend API.

Під час вибору СКБД для розроблюваної системи доцільно розглянути як реляційні, так і нереляційні бази даних.

До нереляційних баз даних належать, зокрема, MongoDB, Firebase Firestore та інші документно-орієнтовані рішення. Їх перевагами є гнучка структура документів, зручність зберігання вкладених об’єктів і можливість швидкої розробки прототипів. Нереляційні бази даних можуть бути ефективними

для систем, у яких структура даних часто змінюється або більшість даних зберігається у вигляді незалежних документів.

Іншим варіантом є використання локальної бази даних на Android-пристрої, наприклад Room. Room є бібліотекою абстракції над SQLite, яка входить до екосистеми Android Jetpack. Вона надає зручний API для локального зберігання даних і дозволяє використовувати SQL-запити через анотації та DAO-інтерфейси.

Основними перевагами Room є перевірка SQL-запитів під час компіляції, інтеграція з ViewModel і Kotlin Coroutines, зменшення кількості шаблонного коду та зручність роботи з локальними даними. Room доцільно використовувати у застосунках, де основні дані зберігаються локально або де потрібне кешування.

Недоліком Room у межах цього проєкту є те, що він не вирішує задачу синхронізації між двома партнерами. Оскільки дані мають бути спільними й однаково доступними для обох користувачів, локальна база даних може використовуватися лише як допоміжний кеш, але не як основне сховище.

Firebase Firestore також може бути використаний для синхронізації даних між користувачами. Його перевагами є вбудована синхронізація, простота інтеграції з мобільними застосунками та підтримка realtime-оновлень. Проте використання Firestore ускладнює реалізацію складної бізнес-логіки на сервері, контроль транзакцій із балами, перевірку фінансових підтверджень і гнучке керування доступом через власні правила. Крім того, застосунок має власну серверну частину, тому доцільніше централізувати бізнес-логіку саме на backend-рівні.

З огляду на структуру даних і потребу в узгоджених операціях було обрано реляційну СКБД PostgreSQL. PostgreSQL забезпечує підтримку зв’язків між таблицями, транзакцій, обмежень цілісності, індексів і стабільну роботу зі складними запитами. Це важливо для операцій, де потрібно одночасно змінювати кілька сутностей, наприклад під час завершення задачі, розподілу балів, створення фінансової транзакції або покупки винагороди.

Для взаємодії серверної частини з PostgreSQL використовується Prisma ORM. Prisma дозволяє описувати моделі даних у декларативному вигляді, автоматично генерувати типізований клієнт для TypeScript, виконувати міграції бази даних і зменшувати кількість помилок під час роботи з SQL-запитами.

Основними перевагами PostgreSQL у межах розроблюваної системи є:

- підтримка реляційної структури даних;
- надійна робота з транзакціями;
- зменшення кількості шаблонного коду;
- інтеграція з TypeScript;
- спрощення підтримки структури бази даних.

Таким чином, для реалізації розроблюваного програмного забезпечення обрано PostgreSQL як основну СКБД та Prisma ORM як інструмент доступу доданих. Таке рішення відповідає потребам клієнт-серверного застосунку, забезпечує цілісність даних і створює основу для подальшого масштабування системи.

2.4 Опис декомпозиції

Модульна декомпозиція полягає у поділі програмного комплексу на автономні функціональні блоки з чітко визначеними зонами відповідальності. Такий підхід дозволяє зменшити зв’язаність компонентів, спростити розробку та тестування, а також забезпечити можливість подальшого розширення функціоналу мобільного застосунку для автоматизації розподілу обов’язків у сім’ї.

Розроблювана система є клієнт-серверною, тому її декомпозиція охоплює як мобільний Android-застосунок, так і серверну частину. Мобільний застосунок відповідає за інтерфейс користувача, локальне збереження налаштувань івзаємодію з backend API. Серверна частина реалізує бізнес-логіку, авторизацію, перевірку прав доступу та роботу з базою даних.

Модуль задач і викликів відповідає за створення побутових задач, гейміфікованих задач-викликів і спільних задач. Він реалізує логіку банку балів, підтвердження виконання, повернення задачі, провалу задачі, повторення задач і розподілу балів у спільних задачах. Цей модуль є одним із центральних у системі, оскільки безпосередньо пов'язаний із автоматизацією розподілу обов'язків.

Модуль календаря забезпечує відображення задач і подій за датами. Він дозволяє користувачам переглядати місячну сітку календаря, відкривати деталі конкретного дня, створювати події та задачі на обрану дату, а також працювати з повторюваними записами.

Модуль фінансів відповідає за створення доходів і витрат, вибір категорій, визначення типу транзакції, підтвердження спільних фінансових записів партнером, обчислення загального бюджету пари та балансу між партнерами. Цей модуль забезпечує прозорість фінансової взаємодії у спільному побуті.

Модуль wishlist призначений для планування майбутніх покупок. Він дозволяє створювати записи з назвою, посиланням, ціною, пріоритетом і категорією призначення. Після позначення покупки як придбаної модуль може ініціювати створення фінансової транзакції.

Модуль повідомлень забезпечує інформування користувачів про важливі події в системі. Зокрема, він використовується для повідомлення партнера про покупку винагороди або інші дії, що можуть потребувати уваги.

Модуль налаштувань відповідає за вибір мови, теми оформлення, валюти, вихід з акаунту та розформування пари. Цей модуль працює переважно з локальним сховищем налаштувань, але також може ініціювати серверні запити, наприклад для виходу з пари.

У результаті проведеного аналізу було визначено такі основні модулі проєкту:

- модуль авторизації;
- модуль керування парою користувачів;

– модуль задач і викликів;
– модуль календаря;
– модуль фінансового обліку;
– модуль wishlist;
– модуль профілю користувача;
– модуль винагород і внутрішнього магазину;
– модуль доступу до бази даних.

Така декомпозиція дозволяє розділити систему на логічні блоки, кожен із яких відповідає за окрему частину предметної області. Завдяки цьому спрощується підтримка коду, локалізація помилок і подальше розширення функціональності застосунку.

2.5 Опис залежностей

Проєктування архітектури мобільного застосунку для автоматизації розподілу обов'язків у сім'ї потребує аналізу залежностей між функціональними компонентами системи. Систематизація цих зв'язків дозволяє визначити напрями передачі даних, забезпечити цілісність інформації та зменшити ризик виникнення помилок під час взаємодії модулів.

Розроблювана система складається з трьох основних рівнів: мобільного клієнта, серверної частини та бази даних. Мобільний клієнт взаємодіє із сервером через REST API. Серверна частина виконує перевірку запитів, реалізує бізнес-логіку та звертається до бази даних через Prisma ORM. База даних PostgreSQL зберігає всі основні сутності системи та забезпечує їхню цілісність.

Модуль авторизації є початковою точкою взаємодії користувача із системою. Після успішного входу він отримує JWT-токен, який зберігається у локальному сховищі DataStore. Надалі модуль взаємодії з API використовує цей токен для захищених запитів. У разі завершення строку дії токена модуль сесії очищає локальні дані та повертає користувача до екрана входу.

Модуль пари користувачів має базову залежність для більшості інших модулів. Задачі, події, фінанси, wishlist, повідомлення та винагороди прив’язуються до pairId. Це означає, що всі спільні дані системи залежать від наявності пари. Якщо користувач не входить до пари, доступ до основних функцій застосунку обмежується.

Модуль задач залежить від модуля користувачів і модуля пари, оскільки кожна задача має автора, може мати відповідального користувача та належить до конкретної пари. Також модуль задач взаємодіє з модулем балів, оскільки створення, завершення або підтвердження задач може змінювати кількість балів користувачів. Повторювані задачі додатково залежать від модуля серій повторення.

Модуль календаря залежить від модуля задач і модуля подій. Він не є самостійним джерелом даних, а виконує функцію агрегатора, який відображає задачі та події за датами. Тому зміни у задачах або подіях мають відображатися у календарі після оновлення даних.

Модуль фінансів залежить від модуля користувачів і пари. Кожна транзакція створюється певним користувачем і належить до пари. Фінансовий модуль також залежить від механізму підтвердження дій партнером, оскільки транзакції для партнера або для обох користувачів повинні бути погоджені перед тим, як впливати на бюджет і баланс.

Модуль wishlist залежить від модуля фінансів, оскільки придбання wishlist-елемента може автоматично створювати фінансову транзакцію. Також він залежить від модуля пари, адже список бажаних покупок є спільним для двох користувачів.

Серверна частина має власну структуру залежностей. Маршрути приймають HTTP-запити та передають їх до контролерів. Контролери використовують middleware для авторизації та Prisma Client для роботи з базою даних. Усі запити, що працюють зі спільними даними, повинні перевіряти, чи належать ці дані до pairId поточного користувача.

Систематизований опис основних міжмодульних залежностей можна подати таким чином:

- модуль авторизації створює JWT-токен і забезпечує доступ до захищених API-запитів;
- модуль сесії зберігає токен і налаштування у DataStore;
- модуль пари визначає область доступу до спільних даних;
- модуль задач залежить від користувачів, пари, балів і календаря;
- модуль календаря агрегує задачі та події за датами;
- модуль фінансів залежить від користувачів, пари та підтверджень партнера;
- модуль wishlist взаємодіє з фінансовим модулем під час покупки;

Сформована структура залежностей демонструє чітке розділення між рівнем інтерфейсу користувача, рівнем бізнес-логіки та рівнем зберігання даних. Це створює передумови для подальшого розширення функціональності застосунку без порушення стабільності основної архітектури.

2.6 Опис інтерфейсу модулів

Програмні інтерфейси компонентів мобільного застосунку для автоматизації розподілу обов’язків у сім’ї мають забезпечувати стандартизовану взаємодію між клієнтською частиною, серверною частиною та базою даних. Оскільки розроблювана система є клієнт-серверною, основним інтерфейсом обміну даними між Android-застосунком і backend є REST API, а дані передаються у форматі JSON.

Клієнтська частина застосунку не взаємодіє з базою даних напряму. Усі операції з користувачами, парами, задачами, календарем, фінансами, wishlist, профілем, магазином винагород і повідомленнями виконуються через серверні маршрути. Такий підхід дозволяє централізувати бізнес-логіку, перевірку авторизації, контроль доступу до даних пари та обробку помилок.

Модулі авторизації, пари користувачів, задач, календаря, фінансів, wishlist, профілю, винагород, повідомлень і налаштувань мають власні інтерфейси взаємодії. На мобільній стороні ці інтерфейси реалізовано через API-класи Retrofit, ViewModel та екрани Jetpack Compose. На серверній стороні вони представлені маршрутами Express, контролерами та middleware.

Модуль авторизації забезпечує реєстрацію та вхід користувача. Його інтерфейс передбачає передачу email і пароля на сервер, отримання JWT-токена та базової інформації про користувача. Отриманий токен зберігається у локальному сховищі Android-застосунку та використовується для виконання захищених запитів.

Модуль пари користувачів надає інтерфейс створення пари, приєднання до пари за кодом і розформування пари. Він є базовим для більшості інших модулів, оскільки всі спільні дані застосунку прив’язуються до ідентифікатора пари.

Модуль задач і викликів має інтерфейси для створення, редагування, видалення та виконання задач. Окремі методи відповідають за підтвердження виконання партнером, повернення задачі, провал задачі, створення повторюваних задач і погодження розподілу балів у спільних задачах.

Модуль календаря забезпечує інтерфейс отримання подій і задач за датами. Він об’єднує дані з модуля задач і модуля подій, після чого відображає їх у вигляді календарної сітки. Для подій передбачено створення, редагування, видалення та повторення.

Модуль фінансів надає інтерфейси для створення доходів і витрат, редагування та видалення фінансових записів, підтвердження спільних транзакцій партнером, отримання загального бюджету пари та балансу між партнерами.

Модуль wishlist забезпечує створення, редагування, видалення та позначення бажаних покупок як придбаних. Якщо wishlist-елемент має ціну, система може створити відповідну фінансову транзакцію з вибраною категорією.

Модуль профілю надає інтерфейси для отримання власного профілю, редагування нікнейму, вибору або завантаження аватарки, а також перегляду профілю партнера. Модуль винагород забезпечує отримання списку доступних винагород, купівлю винагород за бали та перегляд історії покупок.

Модуль повідомлень призначений для отримання внутрішніх повідомлень, позначення їх як прочитаних і відображення кількості непрочитаних повідомлень. Він використовується, зокрема, для інформування партнера про покупку винагороди.

Узагальнені специфікації визначених інтерфейсів модулів проєкту наведено в таблиці 2.

Таблиця 2 – Модулі та їх інтерфейси

Модуль	Інтерфейс
Модуль авторизації	register(), login(), getCurrentUser(), saveToken(), clearSession()
Модуль пари користувачів	createPair(), joinPair(), leavePair(), getPairInfo()
Модуль задач і викликів	getTasks(), createTask(), updateTask(), deleteTask(), requestCompletion(), confirmCompletion(), rejectCompletion(), returnTask(), failTask()
Модуль спільних задач	proposeSplit(), acceptSplit(), counterSplit()
Модуль повторюваних задач	getRecurringTasks(), stopRecurringTask(), generateRecurringInstances()
Модуль календаря та подій	getEvents(), createEvent(), updateEvent(), deleteEvent(), getCalendarItems()
Модуль фінансів	getTransactions(), createTransaction(), updateTransaction(), deleteTransaction(), confirmTransaction(), rejectTransaction(), getFinanceSummary()
Модуль wishlist	getWishlistItems(), createWishlistItem(), updateWishlistItem(), deleteWishlistItem(), purchaseWishlistItem()
Модуль профілю	getMyProfile(), updateProfile(), uploadAvatar(), getPartnerProfile()

На рівні Android-застосунку зазначені інтерфейси використовуються у ViewModel відповідних екранів. ViewModel отримує події від користувацького інтерфейсу, викликає методи API-класів, обробляє результат і оновлює стан екрана. У свою чергу, екрани Jetpack Compose відображають актуальний стан та передають дії користувача до ViewModel.

Систематизація зазначених інтерфейсів створює передумови для стабільної взаємодії між модулями застосунку. Чітке розділення відповідальності дозволяє змінювати окремі частини системи без порушення роботи інших модулів, а використання REST API забезпечує стандартизований обмін даними між клієнтом і сервером.

Після реалізації всіх макетів інтерфейсу, стає можливим створення відповідних елементи інтерфейсу під час реалізації застосунку.

2.7 Детальне проєктування модулів

Визначення основних модулів програмного забезпечення було проведено у попередніх розділах. Було визначено їх функціональне призначення, взаємозв’язки та загальну роль у структурі клієнт-серверної системи. Наступним етапом детального проєктування є поділ модулів на окремі класи, компоненти, контролери, моделі та інтерфейси взаємодії.

Оскільки розроблюваний застосунок складається з Android-клієнта, backend API та бази даних, деталізацію модулів доцільно виконувати окремо для клієнтської та серверної частин.

Деталізація модулів Android-застосунку

Мобільний застосунок реалізовано з використанням архітектурного підходу MVVM. Тому більшість функціональних модулів на клієнтській стороні складаються з екрана Jetpack Compose, ViewModel та API-інтерфейсу для взаємодії з backend.

Модуль авторизації відповідає за реєстрацію, вхід користувача та збереження сесії. До нього належать екрани LoginScreen і RegisterScreen, а також відповідні ViewModel. Вони отримують email і пароль від користувача,

передають ці дані до backend API та після успішної авторизації зберігають JWT-токен за допомогою локального сховища. Для взаємодії із сервером використовується API-інтерфейс AuthApi.

Модуль керування сесією та локальними налаштуваннями реалізовано за допомогою класу TokenManager або аналогічного компонента, що працює з DataStore. Він відповідає за збереження JWT-токена, pairId, вибраної мови, теми оформлення та валюти. Цей модуль використовується іншими частинами застосунку для визначення стану авторизації та локальних параметрів користувача.

Модуль пари користувачів забезпечує створення спільного простору для двох партнерів. Він містить PairScreen, PairViewModel і PairApi. Користувач може створити пару або приєднатися до вже наявної пари за кодом. Після успішного створення або приєднання інформація про пару зберігається у стані користувача, а застосунок переходить до основних екранів.

Модуль навігації реалізовано через AppNavigation та головний контейнер застосунку. Він визначає переходи між екранами авторизації, створення пари, основними вкладками, профілем, магазином винагород і налаштуваннями. Основна частина застосунку використовує нижнє навігаційне меню, що дозволяє швидко перемикатися між календарем, задачами, фінансами, wishlist і профілем.

Модуль задач і викликів є одним із центральних модулів системи. На клієнтській стороні він містить екран задач, ViewModel для керування станом і API-інтерфейс TaskApi. Модуль забезпечує створення задач, відображення активних задач, виконання, підтвердження, повернення, провал задач, а також роботу зі спільними задачами. У випадку гейміфікованих задач застосунок відображає банк балів, статус виконання та доступні дії залежно від ролі користувача.

Модуль повторюваних задач деталізується через окремий API та ViewModel або через розширення модуля задач. Він відповідає за отримання активних серій повторення, зупинку повторення та відображення повторюваних

задач у календарі. Для спільних і звичайних задач повторення має працювати за однаковою календарною логікою, але з різною бізнес-логікою нарахування балів.

Модуль календаря містить CalendarScreen, відповідну ViewModel та API-інтерфейси для задач і подій. Його основне завдання — агрегувати задачі та події за датами й відображати їх у вигляді місячної сітки. Після вибору конкретного дня користувач отримує детальну інформацію про всі записи на цю дату та може створити нову задачу або подію.

Модуль подій календаря реалізує створення, редагування, видалення та повторення подій. Він взаємодіє з backend через EventApi. На відміну від задач, події не мають банку балів і використовуються для звичайних планів або нагадувань.

Модуль фінансів містить FinanceScreen, FinanceViewModel і TransactionApi. Він відповідає за створення доходів і витрат, вибір категорії, підтвердження спільних фінансових операцій партнером, розрахунок загального бюджету пари та балансу між користувачами. У цьому модулі також реалізовано редагування та видалення фінансових записів.

Модуль wishlist реалізує планування майбутніх покупок. Він містить WishlistScreen, WishlistViewModel і WishlistApi. Користувач може створювати записи з посиланням, ціною, пріоритетом і категорією призначення. Після позначення товару як придбаного модуль може ініціювати створення фінансової транзакції з вибором відповідної категорії.

Модуль профілю відповідає за відображення та редагування даних користувача. До нього належать ProfileScreen, ProfileViewModel і ProfileApi. У профілі відображаються нікнейм, аватарка, кількість балів, winStreak, інформація про партнера та придбані винагороди. Також із профілю користувач може перейти до налаштувань і магазину винагород.

Модуль магазину винагород реалізовано через ShopScreen, RewardViewModel і RewardApi. Він дозволяє переглядати список доступних

винагород, купувати їх за бали, перевіряти доступність винагород за streak і створювати повідомлення для партнера після покупки.

Модуль повідомлень забезпечує отримання внутрішніх повідомлень, відображення кількості непрочитаних повідомлень і позначення повідомлень як прочитаних. Він може бути реалізований окремим екраном або діалоговим вікном, що відкривається з верхньої частини застосунку.

Модуль налаштувань містить SettingsScreen і відповідну ViewModel. Він відповідає за вибір мови, теми оформлення, валюти, вихід з акаунту та розформування пари. Частина даних цього модуля зберігається локально, а частина дій виконується через backend API.

Деталізація модулів backend

Серверна частина реалізована на основі Node.js, Express.js та TypeScript. Вона має модульну структуру, у якій кожен функціональний напрям представлений окремими маршрутами та контролерами.

Модуль авторизації backend складається з auth.routes, auth.controller, middleware для перевірки токена та утиліт для генерації JWT. Він відповідає за реєстрацію користувача, хешування пароля, вхід, формування токена та повернення даних користувача.

Модуль пари користувачів складається з pair.routes і pair.controller. Він реалізує створення пари, приєднання до пари, вихід із пари та перевірку належності користувача до певного pairId.

Модуль задач складається з task.routes і task.controller. У ньому зосереджено бізнес-логіку створення задач, списання або нарахування балів, підтвердження виконання, повернення задачі, провалу задачі та погодження розподілу балів у спільних задачах. Для операцій із балами використовуються транзакції бази даних, щоб уникнути некоректних станів.

Модуль повторюваних задач може бути винесений в окремий controller або реалізований як допоміжний сервіс у межах модуля задач. Він відповідає

за створення серій повторюваних задач, генерацію екземплярів на майбутні дати та зупинку повторення.

Модуль подій складається з event.routes і event.controller. Він відповідає за створення, редагування, видалення та отримання подій календаря. Для подій також може підтримуватися повторення за встановленим правилом.

Модуль фінансів складається з transaction.routes і transaction.controller. Він реалізує створення транзакцій, підтвердження або відхилення транзакцій партнером, редагування, видалення, отримання історії, розрахунок бюджету пари та балансу між партнерами.

Модуль wishlist складається з wishlist.routes і wishlist.controller. Він відповідає за створення, редагування, видалення wishlist-елементів, відкриття посилань на клієнтській стороні та позначення покупки як придбаної.

Модуль профілю складається з profile.routes і profile.controller. Він реалізує отримання власного профілю, профілю партнера, оновлення нікнейму та аватарки, а також обробку завантаження зображень.

Модуль винагород складається з reward.routes і reward.controller. Він відповідає за отримання списку винагород, перевірку доступності винагород, покупку за бали та створення запису про покупку.

Модуль повідомлень складається з notification.routes і notification.controller. Він забезпечує отримання повідомлень, підрахунок непрочитаних повідомлень і зміну їхнього статусу.

На основі виконаної деталізації модулів може бути створена діаграма класів або діаграма компонентів системи, що демонструє зв’язки між клієнтськими ViewModel, API-інтерфейсами, backend-контролерами та моделями бази даних.

2.8 Аналіз та вибір технологій та методів реалізації застосунку

Створення мобільного застосунку для автоматизації розподілу обов’язків у сім’ї є комплексним процесом, що потребує вибору відповідних технологій для

клієнтської частини, серверної частини, бази даних, авторизації та взаємодії між компонентами. Основна мета цього етапу — визначити стек технологій, який дозволить реалізувати застосунок із підтримкою синхронізації між партнерами, зручним інтерфейсом, стабільною серверною логікою та можливістю подальшого розширення.

Першим необхідно обрати технологію для реалізації мобільного застосунку. Для Android-розробки основними варіантами є нативна розробка з використанням Kotlin або Java, а також кросплатформні рішення, наприклад Flutter або React Native.

Кросплатформні рішення мають перевагу у вигляді можливості одночасної розробки для Android та iOS. Однак вони можуть ускладнювати роботу знативними можливостями Android, збільшувати залежність від сторонніх бібліотек і не завжди забезпечувати таку ж глибоку інтеграцію з платформою. Оскільки основною цільовою платформою розроблюваного застосунку є Android, було обрано нативний підхід.

Для реалізації інтерфейсу користувача розглянуто два основні підходи: традиційна XML-розмітка та Jetpack Compose.

Jetpack Compose є сучасним декларативним інструментарієм для створення нативного Android-інтерфейсу. Він дозволяє описувати UI безпосередньо у Kotlin-коді, будувати інтерфейс на основі стану та швидко створювати адаптивні екрани. Для застосунку, у якому багато екранів, діалогових вікон, карток, списків, chips і динамічних станів, Jetpack Compose є доцільним вибором.

Основними перевагами Jetpack Compose є:

- декларативний підхід до побудови інтерфейсу;
- зручне керування станом екранів;
- зручна реалізація адаптивних екранів, списків і діалогів;
- підтримка Material Design.

Основними недоліками Jetpack Compose є довший час компіляції, потреба у вивченні нового підходу до побудови інтерфейсу та можливі труднощі

з оптимізацією складних екранів. Незважаючи на це, для розроблюваного застосунку було обрано Jetpack Compose, оскільки він дозволяє швидко реалізувати сучасний, адаптивний і візуально цілісний інтерфейс.

Наступним етапом є вибір мови програмування для Android-застосунку. Основними варіантами є Java та Kotlin.

Java є стабільною об’єктно-орієнтованою мовою програмування, яка тривалий час була основною мовою Android-розробки. Її перевагами є велика кількість навчальних матеріалів, зрілість екосистеми та сумісність із багатьма бібліотеками.

Kotlin є сучасною статично типізованою мовою програмування, офіційно рекомендованою для Android-розробки. Вона забезпечує лаконічніший синтаксис, null-безпеку, зручну роботу з корутинами та повну сумісність із Java. Kotlin також є основною мовою для Jetpack Compose.

Основними перевагами Kotlin є:

– зменшення кількості шаблонного коду;
– вбудована null-безпека;
– зручна асинхронність через coroutines;

Основними недоліками Kotlin є вищий поріг входу порівняно з Java та іноді довший час компіляції. Оскільки проєкт використовує Jetpack Compose і сучасну Android-архітектуру, мовою мобільної частини обрано Kotlin.

Для реалізації серверної частини розглянуто кілька можливих підходів: Node.js з Express.js, NestJS, Spring Boot та Firebase. Spring Boot є потужним рішенням для корпоративних систем, однак потребує більшої кількості конфігурації та часу на розробку. Firebase дозволяє швидко створити backend для мобільного застосунку, але обмежує гнучкість складної бізнес-логіки та контроль над структурою даних.

Для розроблюваного застосунку обрано Node.js з Express.js. Node.js дозволяє швидко створювати серверні застосунки, а Express.js надає простий і зрозумілий механізм побудови REST API. У поєднанні з TypeScript це дозволяє

реалізувати backend із типізацією, зрозумілою структурою та достатньою гнучкістю для бізнес-логіки задач, фінансів, винагород і повідомлень.

Основними перевагами Node.js та Express.js є:

– швидка розробка REST API;
– велика кількість бібліотек;
– простота налаштування маршрутів і middleware;
– зручність інтеграції з Prisma;
– достатня гнучкість для реалізації бізнес-логіки застосунку.

Мовою серверної частини обрано TypeScript. Він додає статичну типізацію до JavaScript, зменшує кількість помилок і полегшує підтримку коду. Для системи з великою кількістю моделей, статусів і API-відповідей типізація є важливою перевагою.

Для авторизації користувачів обрано JWT. Такий підхід дозволяє реалізувати stateless-авторизацію, коли сервер перевіряє токен у кожному захищеному запиті. На мобільній стороні токен зберігається локально у DataStore.

Для взаємодії Android-застосунку з backend API використовується Retrofit. Він дозволяє зручно описувати HTTP-запити у вигляді Kotlin-інтерфейсів і автоматично перетворювати JSON-відповіді у моделі даних.

для завантаження аватарок користувачів у мобільному застосунку може використовуватися бібліотека Coil. Вона добре інтегрується з Jetpack Compose і дозволяє відображати зображення з URL або локального джерела.

Останнім аспектом є вибір моделі життєвого циклу програмного забезпечення. Для розроблюваного проєкту доцільною є ітеративна модель, близька до Agile-підходу. Під час розробки функціонал створювався поступово: спочатку авторизація і пари, потім задачі, календар, фінанси, wishlist, профіль, магазин винагород і налаштування. Такий підхід дозволив регулярно перевіряти працездатність окремих модулів, виправляти помилки та уточнювати вимоги.

Перевагами ітеративного підходу для цього проєкту є:

– можливість швидкого отримання робочого MVP;
– поступове додавання функцій;
– раннє виявлення помилок;
– можливість адаптації вимог у процесі розробки;
– зручність тестування окремих модулів.

Недоліком ітеративного підходу є складність точного прогнозування фінального обсягу робіт, оскільки вимоги можуть уточнюватися під час реалізації. Однак для дипломного проєкту з великою кількістю взаємопов’язаних модулів такий підхід є більш придатним, ніж жорстка каскадна модель.

У результаті аналізу для реалізації застосунку було обрано такі технології:

– Kotlin — мова програмування мобільної частини;
– Jetpack Compose — побудова інтерфейсу користувача;
– MVVM — архітектурний підхід клієнтської частини;
– Node.js — середовище виконання серверної частини;
– Express.js — фреймворк для побудови REST API;
– TypeScript — мова реалізації backend;
– PostgreSQL — реляційна база даних;
– Prisma ORM — робота з базою даних і міграціями;
– JWT — механізм авторизації;
– Retrofit — HTTP-клієнт Android-застосунку;

Обраний стек технологій забезпечує можливість реалізації повноцінного клієнт-серверного мобільного застосунку з підтримкою синхронізації між користувачами, збереженням даних у PostgreSQL, сучасним інтерфейсом і можливістю подальшого розширення функціональності.

2.9 Висновки

Пройшовши всі етапи проєктування, необхідно оцінити результати проведених робіт та зробити підсумковий висновок. Під час проєктування було проведено аналіз наявних архітектурних рішень, шаблонів проєктування та підходів до побудови мобільних клієнт-серверних застосунків.

Зважаючи на потребу в синхронізації даних між двома користувачами, розмежуванні доступу до спільних даних і підтримці великої кількості взаємопов'язаних модулів, для розроблюваного застосунку обрано клієнт-серверну архітектуру. На стороні Android-застосунку використано шаблон MVVM. Серверна частина спроєктована за модульним принципом із використанням маршрутів, контролерів, middleware та окремого рівня доступу до бази даних.

Здійснено проєктування структури збереження інформації. Було визначено основні сутності предметної області: користувач, пара користувачів, задача, подія, фінансова транзакція, wishlist-елемент, винагорода, повідомлення та шаблон. З огляду на значну кількість зв'язків між даними та потребу в підтримці транзакцій, як основну систему керування базами даних обрано PostgreSQL, а для взаємодії з нею — Prisma ORM.

Паралельно було виконано детальне проєктування клієнтської та серверної частин системи. Для Android-застосунку визначено основні екрани, ViewModel та API-інтерфейси, а для backend-частини — відповідні маршрути, контролери та моделі даних. Це дозволило сформувати цілісну структуру програмного продукту та забезпечити логічний зв'язок між інтерфейсом користувача, бізнес-логікою і базою даних.

Результатом цього розділу стала технічна основа для реалізації програмного забезпечення. Проведене проєктування забезпечує чітке розуміння внутрішньої структури системи, взаємодії її модулів і способів зберігання даних, що створює надійне підґрунтя для переходу до етапу програмної реалізації мобільного застосунку для автоматизації розподілу обов'язків у сім'ї.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ЗАСТОСУНКУ

3.1 Особливості програмної реалізації Android застосунків з використанням Kotlin, Jetpack Compose та клієнт-серверної архітектури

Програмна реалізація мобільного застосунку для автоматизації розподілу обов'язків у сім'ї базується на використанні нативного Android-стека розробки, мови програмування Kotlin, декларативного інструментарію Jetpack Compose та клієнт-серверної архітектури. Такий підхід дозволяє створити застосунок, у якому користувацький інтерфейс, логіка екранів, обмін даними з сервером і локальне збереження сесії розділені між окремими компонентами.

Базовою вимогою для функціонування системи є наявність Android-пристрою з доступом до мережі Інтернет. Оскільки застосунок передбачає синхронізацію даних між двома користувачами, основні дані зберігаються не локально на пристрої, а у віддаленій базі даних PostgreSQL, доступ до якої здійснюється через backend API. Локально на Android-пристрої зберігаються лише службові дані: JWT-токен, ідентифікатор пари, обрана мова, тема оформлення та валюта.

Для реалізації клієнтської частини використано середовище Android Studio та систему збірки Gradle. У конфігурації проєкту підключено необхідні залежності для Jetpack Compose, Navigation Compose, Retrofit, DataStore, роботи з ViewModel та завантаження зображень. Це забезпечує можливість створення сучасного інтерфейсу, навігації між екранами, взаємодії з backend API та локального збереження налаштувань.

Важливим етапом реалізації Android-застосунку є конфігурація файлу AndroidManifest.xml. У ньому визначаються основні параметри застосунку, головна Activity, а також дозвіл на доступ до мережі Інтернет. Оскільки застосунок взаємодіє з backend API, дозвіл INTERNET є обов'язковим для виконання HTTP-запитів.

Користувацький інтерфейс застосунку реалізовано за допомогою Jetpack Compose. На відміну від класичного підходу з XML-розміткою, Compose дозволяє описувати екрани декларативно, безпосередньо у Kotlin-коді. Кожен екран формується як набір composable-функцій, що відображають поточний стан ViewModel. У разі зміни стану інтерфейс автоматично перемальовується, що спрощує роботу зі списками задач, календарем, фінансовими записами, профілем і налаштуваннями.

Архітектура клієнтської частини побудована за шаблоном MVVM. Екрани Jetpack Compose відповідають за відображення даних і приймання дій користувача. ViewModel зберігає стан екрана, виконує валідацію даних, ініціює запити до backend API та обробляє відповіді. API-рівень реалізовано через Retrofit-інтерфейси, які описують HTTP-запити до серверної частини.

Для збереження сесії використовується DataStore. Після успішної авторизації JWT-токен зберігається локально та додається до захищених запитів. Якщо сервер повертає помилку авторизації, наприклад через завершення строку дії токена, застосунок очищає локальну сесію та повертає користувача до екрана входу. Це забезпечує коректну обробку ситуацій, коли токен став недійсним.

Навігація між екранами реалізована за допомогою Navigation Compose. Після запуску застосунок перевіряє стан сесії користувача. Якщо токен відсутній або недійсний, користувач переходить до екрана входу. Якщо користувач авторизований, але ще не належить до пари, відкривається екран створення або приєднання до пари. Якщо пара вже створена, користувач переходить до основної частини застосунку, де доступні календар, задачі, фінанси, wishlist і профіль.

Серверна частина реалізована на Node.js з використанням Express.js та TypeScript. Backend API відповідає за авторизацію, керування парами, задачами, подіями, фінансовими транзакціями, wishlist, профілями, винагородами та повідомленнями. Для доступу до бази даних використовується

Prisma ORM, що дозволяє описувати моделі даних, виконувати міграції та працювати з PostgreSQL через типізований клієнт.

Критичним аспектом реалізації є коректна робота з операціями, що змінюють бали користувачів або фінансовий баланс. Такі дії мають виконуватися узгоджено, щоб уникнути некоректного стану даних. Наприклад, під час завершення задачі необхідно змінити її статус і нарахувати бали користувачу. Під час покупки винагороди потрібно перевірити достатність балів, списати їх і створити запис про покупку. Для таких операцій у backend-частині використовуються транзакції бази даних.

Окрему увагу приділено локалізації та темам оформлення. Усі основні текстові елементи інтерфейсу винесено у ресурсні файли, що дозволяє підтримувати українську та англійську мови. Вибір мови, теми та валюти зберігається локально за допомогою DataStore. Застосунок підтримує світлу та темну тему, що дозволяє адаптувати інтерфейс до вподобань користувача.

Таким чином, програмна реалізація застосунку поєднує сучасні засоби Android-розробки, клієнт-серверну взаємодію, локальне збереження сесії та централізовану backend-логіку. Це дозволяє забезпечити синхронну роботу двох користувачів у межах одного спільного простору.

3.2 Програмна реалізація модулів

Типовий план реалізації модулів для мобільного застосунку автоматизації розподілу обов’язків у сім’ї включає такі етапи:

- розроблення backend-структури, моделей бази даних та механізму авторизації;
- реалізація клієнтської авторизації та локального збереження сесії;
- створення модуля пари користувачів;
- реалізація модуля задач і гейміфікованої логіки;
- реалізація календаря та подій;
- розроблення фінансового модуля;

– створення wishlist та інтеграції з фінансами;
– реалізація профілю користувача, магазину винагород і повідомлень;
– додавання налаштувань, локалізації, тем і валюти;
– тестування основних користувацьких сценаріїв.

Розроблення backend-структури є базовим етапом створення системи, оскільки саме серверна частина забезпечує збереження даних, синхронізацію між партнерами та перевірку прав доступу. Сервер реалізовано на основі Express.js. Структура backend поділена на маршрути, контролери, middleware, утиліти та Prisma-моделі.

Моделі бази даних описано у файлі schema.prisma. Основними сутностями є User, Pair, Task, RecurringTaskSeries, Event, Transaction, WishlistItem, Reward, RewardPurchase, Notification та Blueprint. Така структура дозволяє зберігати користувачів, спільні задачі, події, фінансові записи, покупки, винагороди та повідомлення в межах конкретної пари.

Модуль авторизації реалізує реєстрацію, вхід користувача та формування JWT-токена. Під час реєстрації пароль користувача хешується перед збереженням у базі даних. Після успішного входу сервер повертає токен і дані користувача. На стороні Android токен зберігається в DataStore та використовується для захищених запитів.

Модуль пари користувачів реалізує створення спільного простору для двох партнерів. Один користувач може створити пару та отримати код приєднання, а другий — приєднатися за цим кодом. Після створення пари всі спільні записи — задачі, події, фінансові операції та wishlist — прив’язуються до pairId.

Реалізація модуля задач є одним із ключових етапів, оскільки саме він відповідає за автоматизацію розподілу побутових обов’язків. Задачі можуть бути двох типів: challenge-задачі та спільні задачі. У challenge-задачах один користувач створює завдання та вкладає певну кількість балів у банк задачі. Партнер може виконати задачу, повернути її або відхилити відповідно до правил гейміфікованої

логіки. У спільних задачах банк балів формується системою, а користувачі після виконання погоджують розподіл винагороди між собою.

Для задач реалізовано механізм підтвердження виконання. Користувач, який виконав задачу, надсилає запит на підтвердження, після чого партнер може підтвердити або відхилити виконання. Бали нараховуються лише після підтвердження, що запобігає односторонньому отриманню винагороди.

Окремо реалізовано повторювані задачі. Для цього використовується модель серії повторень, яка зберігає тип повторення, інтервал, дату початку та стан активності. На основі серії система генерує екземпляри задач на майбутні дати. Користувач може зупинити повторення, після чого нові екземпляри задачі більше не створюються.

Модуль календаря об’єднує задачі та події за датами. На клієнтській стороні календар відображається у вигляді місячної сітки. У кожній клітинці дня показуються короткі позначки задач і подій. Після вибору конкретної дати відкривається детальна панель, де користувач може переглянути всі записи, створити нову задачу або подію, а також використати шаблон.

Модуль подій реалізує створення, редагування, видалення та повторення подій. На відміну від задач, події не мають банку балів і використовуються для звичайних планів або нагадувань. Події також відображаються у календарі разом із задачами.

Фінансовий модуль відповідає за створення доходів і витрат, вибір категорій, підтвердження спільних транзакцій партнером, підрахунок загального бюджету пари та балансу між партнерами. Транзакції мають тип, категорію, область застосування та статус підтвердження. Якщо транзакція створена як спільна або для партнера, вона не впливає на бюджет і баланс до моменту підтвердження іншим користувачем.

Модуль wishlist дозволяє користувачам планувати майбутні покупки. Кожен запис містить назву, посилання, ціну, пріоритет і призначення. Якщо користувач позначає wishlist-елемент як придбаний, застосунок може

створити фінансову транзакцію. При цьому користувач обирає категорію витрат, до якої буде віднесено покупку.

Модуль профілю реалізує перегляд і редагування даних користувача. У профілі відображається нікнейм, аватарка, кількість балів, winStreak, придбані винагороди та профіль партнера. Користувач може змінити нікнейм, вибрати preset-аватар або завантажити власне зображення.

Модуль магазину винагород дозволяє витрачати бали на внутрішні бонуси. Перед покупкою система перевіряє, чи має користувач достатню кількість балів і чи виконано умови доступу до винагороди. Після покупки з користувача списуються бали, створюється запис про покупку, а партнер отримує внутрішнє повідомлення.

Модуль повідомлень реалізує внутрішні повідомлення між партнерами. Він використовується для інформування про важливі дії, наприклад покупку винагороди. Повідомлення мають статус прочитання, а застосунок може відображати кількість непрочитаних повідомлень.

Модуль налаштувань забезпечує вибір теми, мови та валюти. Також він містить функції виходу з акаунту та розформування пари. Налаштування теми, мови й валюти зберігаються локально, оскільки вони стосуються конкретного пристрою.

Для взаємодії Android-застосунку з backend API використовується Retrofit. Кожен функціональний модуль має відповідний API-інтерфейс. Наприклад, авторизація працює через AuthApi, задачі — через TaskApi, фінанси — через TransactionApi, wishlist — через WishlistApi, профіль — через ProfileApi.

На рівні користувацького інтерфейсу основні екрани реалізовано за допомогою Jetpack Compose. Для кожного екрана створено ViewModel, що зберігає стан, обробляє дії користувача та викликає відповідні API-методи. Такий підхід забезпечує розділення інтерфейсу та бізнес-логіки.

Фрагмент коду, що демонструє типову взаємодію Android-застосунку з backend API, може бути наведено нижче:

```
interface TaskApi {
    @GET("api/tasks")
    suspend fun getTasks(): List<TaskDto>

    @POST("api/tasks/create")
    suspend fun createTask(
        @Body request: CreateTaskRequest
    ): TaskDto

    @POST("api/tasks/request-completion/{id}")
    suspend fun requestCompletion(
        @Path("id") taskId: String
    ): TaskDto
}
```

Повний код програми наведено у додатку В.

3.3 Реалізація інтерфейсу користувача

Реалізація інтерфейсу користувача мобільного застосунку виконана у середовищі Android Studio з використанням Jetpack Compose. На відміну від класичного підходу з XML-розміткою, Jetpack Compose дозволяє створювати інтерфейс декларативно, безпосередньо мовою Kotlin. У такому підході екран описується як набір composable-функцій, а його вигляд автоматично оновлюється при зміні стану.

Основним елементом побудови інтерфейсу є composable-функція. Вона може представляти як увесь екран, так і окремий компонент, наприклад кнопку, картку, поле введення, список або діалогове вікно. Такий підхід дозволяє повторно використовувати компоненти інтерфейсу в різних частинах застосунку та підтримувати єдиний стиль оформлення.

Навігація між основними екранами реалізована за допомогою Navigation Compose. Після запуску застосунок перевіряє наявність збереженої сесії користувача. Якщо користувач не авторизований, відкривається екран входу. Якщо користувач авторизований, але ще не входить до пари, відкривається екран

створення або підключення до пари. Якщо пара вже створена, користувач переходить до основної частини застосунку.

Основна частина інтерфейсу побудована навколо нижнього навігаційного меню, яке містить основні розділи застосунку: календар, задачі, фінанси, wishlist та профіль. Така структура дозволяє користувачу швидко переходити між ключовими модулями програми.

Реалізація екрана авторизації базується на використанні текстових полів для введення email та пароля, кнопки входу і переходу до реєстрації. Логіка введення даних і виконання запиту винесена у ViewModel, а сам екран відповідає лише за відображення стану та передачу дій користувача.

Реалізація екрана реєстрації побудована аналогічно до екрана входу. Він містить поля введення email, пароля та підтвердження пароля. Після успішної реєстрації користувач автоматично переходить до наступного етапу роботи із застосунком.

Екран створення або приєднання до пари містить кнопки для створення нового спільного простору та поле введення коду для підключення до вже існуючої пари. Після успішного створення або підключення pairId зберігається у локальному стані користувача, а застосунок відкриває основний інтерфейс.

Екран календаря реалізовано у вигляді місячної сітки з сімома днями в рядку. Кожна клітинка календаря відображає число дня та короткі позначки задач або подій. Для збереження компактності у клітинці показується лише обмежена кількість записів, а повний список відкривається після натискання на відповідний день. Детальна інформація про день виводиться у нижній панелі або діалоговому вікні, де користувач може переглянути задачі й події, створити новий запис або використати шаблон.

Екран задач і викликів реалізує роботу з гейміфікованими побутовими задачами. У верхній частині екрана відображаються бали користувача та партнера у вигляді спеціального візуального елемента. Основна частина екрана містить список активних задач, задач, що очікують підтвердження,

повторюваних задач і спільних задач. Форми створення задач не відображаються постійно, а відкриваються після натискання кнопки створення, що дозволяє не перевантажувати інтерфейс.

Екран фінансів містить блок загального бюджету пари, інформацію про баланс між партнерами, категорії доходів і витрат, а також список фінансових транзакцій. Для створення нового фінансового запису використовується окрема форма, що відкривається після натискання кнопки додавання. У формі користувач обирає тип операції, суму, категорію та призначення транзакції. Для спільних витрат або витрат на партнера передбачено підтвердження з боку другого користувача.

Екран wishlist реалізує список бажаних покупок. Кожен елемент містить назву, посилання, ціну, пріоритет і призначення. Посилання у записах відкриваються у браузері. Після позначення товару як придбаного користувач може створити фінансову транзакцію, вибравши відповідну категорію витрат.

Екран профілю відображає аватарку, нікнейм, кількість балів, winStreak, інформацію про партнера та придбані винагороди. Користувач може редагувати власний профіль, змінювати нікнейм і аватарку. Профіль партнера доступний лише для перегляду, що запобігає несанкціонованій зміні чужих даних.

Екран магазину винагород містить перелік бонусів, які можна придбати за бали. Кожна винагорода відображається у вигляді картки з назвою, описом, вартістю та умовами доступу. Після покупки винагороди створюється відповідний запис і внутрішнє повідомлення для партнера.

Екран налаштувань містить вибір мови, теми оформлення, валюти, а також кнопки виходу з акаунту та розформування пари. Для вибору параметрів використовуються візуально виділені елементи, що покращує зрозумілість інтерфейсу та дозволяє уникнути текстових повідомлень на кшталт “selected”.

Для створення та редагування записів у різних модулях застосунку використовуються діалогові вікна або нижні панелі. Такий підхід дозволяє залишати основні екрани чистими й відображати форми лише тоді, коли

користувач виконує відповідну дію. Це особливо важливо для мобільного інтерфейсу, де простір екрана є обмеженим.

Окрему увагу приділено підтримці світлої та темної теми. Кольори інтерфейсу визначаються через MaterialTheme, що дозволяє уникнути некоректного відображення елементів при перемиканні теми. Текстові елементи інтерфейсу винесено у ресурсні файли, що забезпечує підтримку української та англійської мов.

Таким чином, інтерфейс користувача реалізовано з урахуванням принципів мобільної зручності, адаптивності та повторного використання компонентів. Використання Jetpack Compose дозволило створити динамічний інтерфейс, який реагує на зміну стану застосунку та забезпечує зручну взаємодію з усіма основними модулями системи.

3.4 Вимоги до технічних та програмних засобів

Визначення вимог до технічних та програмних засобів є важливим етапом розробки, оскільки дозволяє встановити мінімальні умови для стабільної роботи застосунку та визначити рекомендоване середовище для комфортного використання системи. Оскільки розроблюваний застосунок є клієнт-серверним, вимоги необхідно розглядати окремо для мобільного клієнта, серверної частини та середовища розробки.

Для роботи мобільного застосунку користувачу необхідний смартфон або планшет на базі операційної системи Android. Оскільки застосунок виконує мережеві запити до backend API та синхронізує дані між партнерами, обов’язковою умовою є наявність стабільного доступу до мережі Інтернет.

Мобільний застосунок не потребує спеціалізованого обладнання, датчиків або високопродуктивної графічної підсистеми. Основними операціями є відображення інтерфейсу, робота зі списками, календарем, формами введення, HTTP-запитами та локальним збереженням налаштувань. Тому застосунок може працювати на більшості сучасних Android-пристроїв.

Основними характеристиками пристрою, до яких встановлюються вимоги, є версія операційної системи, обсяг оперативної пам’яті, продуктивність процесора, наявність вільного місця на диску та доступ до мережі Інтернет.

Аналізуючи розроблений застосунок, було визначено мінімальні та рекомендовані системні вимоги, подані у таблиці 3.1.

Таблиця 3 – системі вимоги застосунку

Характеристика	**Мінімальні вимоги**	**Рекомендовані вимоги**
Операційна система	Android 8.0	Android 12 або вище
Процесор	4-ядерний ARM-процесор	8-ядерний ARM-процесор
Оперативна пам’ять	3 ГБ	6 ГБ або більше
Вільне місце на диску	200 МБ	500 МБ або більше
Інтернет з’єднання	Обов’язкове	
Камера	Необов’язкова	

3.5 Тестування застосунку

Тестування програмного забезпечення є важливим етапом розробки, оскільки дозволяє перевірити коректність реалізації функціональних вимог, стабільність роботи окремих модулів та правильність взаємодії між клієнтською і серверною частинами системи.

Розроблюваний застосунок є клієнт-серверною системою, що складається з Android-клієнта, backend API та бази даних PostgreSQL. Через це тестування повинно охоплювати як роботу інтерфейсу користувача, так і коректність серверної логіки, авторизації, синхронізації даних між партнерами та операцій із базою даних.

Оскільки значна частина логіки застосунку пов’язана з взаємодією двох користувачів, повністю ізольоване модульне тестування не дозволяє перевірити всі важливі сценарії. Наприклад, підтвердження виконання задачі, підтвердження фінансової транзакції або отримання повідомлення після покупки винагороди потребують участі двох облікових записів, що належать до однієї пари. Тому основним видом тестування для цього проєкту обрано інтеграційне та ручне функціональне тестування.

Інтеграційне тестування дозволяє перевірити взаємодію між Android-застосунком, backend API та базою даних. У межах такого тестування перевіряється, чи коректно мобільний клієнт формує запити, чи правильно сервер обробляє бізнес-логіку, чи зберігаються дані у базі даних і чи відображаються вони в інтерфейсі другого користувача.

Автоматизоване тестування є більш повторюваним і дозволяє швидко перевіряти базові сценарії після змін у коді. Проте частина функціоналу застосунку залежить від складної взаємодії двох користувачів, стану backend-сервера та даних у базі, тому в межах цієї роботи основну увагу приділено ручному інтеграційному тестуванню ключових сценаріїв.

Автоматизованій або напівавтоматизованій перевірці можуть підлягати такі сценарії:

- запуск Android-застосунку;
- відкриття екрана входу;
- перевірка переходу між екранами входу та реєстрації;
- перевірка валідації порожніх полів;
- перевірка переходу між основними вкладками застосунку;
- перевірка відображення основних елементів інтерфейсу;
- перевірка перемикання теми;
- перевірка наявності локалізованих ресурсів для основних екранів.

Ручному інтеграційному тестуванню підлягають такі сценарії:

- реєстрація нового користувача;
- вхід у вже створений обліковий запис;
- збереження сесії користувача після перезапуску застосунку;
- створення пари користувачів;
- приєднання другого користувача до пари за кодом;
- розформування пари;
- вихід з облікового запису;
- створення задачі;
- створення задачі на конкретну дату;
- створення повторюваної задачі;
- зупинка повторення задачі;
- редагування та видалення задачі;
- запит на підтвердження виконання задачі;
- підтвердження або відхилення виконання задачі партнером;
- створення спільної задачі;
- погодження розподілу балів у спільній задачі;
- відображення задач у календарі;
- створення календарної події;
- створення повторюваної події;
- редагування та видалення події;
- створення шаблону задачі або події;
- використання шаблону для швидкого створення запису;
- створення фінансової транзакції;
- створення транзакції для себе;
- створення транзакції для партнера або для обох;
- підтвердження або відхилення фінансової транзакції партнером;
- редагування та видалення фінансового запису;

– перевірка розрахунку загального бюджету пари;
– перевірка балансу між партнерами;
– створення wishlist-елемента;
– відкриття посилання з wishlist;
– редагування та видалення wishlist-елемента;
– позначення wishlist-елемента як придбаного;
– створення фінансової транзакції після покупки з wishlist;
– перегляд профілю користувача;
– редагування нікнейму та аватарки;
– перегляд профілю партнера;
– покупка винагороди у внутрішньому магазині;
– списання балів після покупки винагороди;
– створення повідомлення для партнера після покупки винагороди;
– перегляд і позначення повідомлень як прочитаних;
– зміна мови інтерфейсу;
– зміна теми оформлення;
– зміна валюти фінансового модуля;
– обробка завершення строку дії JWT-токена.

Тестування сценарію реєстрації полягає у введенні email та пароля, натисканні кнопки реєстрації, перевірці створення користувача на сервері та переходу до наступного екрана застосунку. Додатково перевіряється неможливість створення двох користувачів з однаковим email.

Тестування сценарію авторизації полягає у введенні правильних облікових даних, отриманні JWT-токена, збереженні його у локальному сховищі та переході до основної частини застосунку. Також перевіряється сценарій введення неправильного пароля або неіснуючого email.

Тестування створення пари полягає у створенні спільного простору першим користувачем, отриманні коду пари та підключенні другого користувача

за цим кодом. Після цього перевіряється, що обидва користувачі мають однаковий pairId і бачать спільні дані.

Тестування сценарію створення задачі полягає у відкритті форми створення, введенні назви, виборі типу задачі, встановленні дати та кількості балів. Після створення перевіряється поява задачі у списку задач і в календарі на відповідну дату.

Тестування сценарію виконання задачі полягає у тому, що користувач надсилає запит на підтвердження виконання, після чого партнер бачить відповідний статус і може підтвердити або відхилити виконання. У разі підтвердження перевіряється зміна статусу задачі та коректне нарахування балів.

Тестування спільної задачі полягає у створенні задачі з системним банком балів, запиті на розподіл винагороди та підтвердженні або зміні цього розподілу партнером. Перевіряється, що сума розподілених балів дорівнює банку задачі, а бали нараховуються лише після погодження обома партнерами.

Тестування повторюваних задач полягає у створенні задачі з правилом повторення, наприклад щотижня. Після створення перевіряється, що екземпляри задачі з’являються на відповідних датах календаря протягом заданого періоду. Також перевіряється зупинка повторення, після якої нові екземпляри не повинні створюватися.

Тестування календаря полягає у перевірці відображення задач і подій у місячній сітці. Для кожного дня перевіряється відкриття детальної панелі, відображення повного списку записів, а також можливість створення задачі або події на вибрану дату.

Тестування фінансового модуля полягає у створенні доходів і витрат із різними категоріями та призначеннями. Для транзакцій, що стосуються партнера або обох користувачів, перевіряється механізм підтвердження. До моменту підтвердження такі транзакції не повинні впливати на загальний бюджет і баланс між партнерами.

Тестування wishlist полягає у створенні запису з назвою, посиланням, ціною та пріоритетом. Перевіряється можливість відкриття посилання у браузері, редагування запису, видалення запису та позначення його як придбаного. Якщо під час покупки створюється транзакція, перевіряється правильність її категорії та відображення у фінансовому модулі.

Тестування профілю полягає у зміні нікнейму, виборі або завантаженні аватарки, перегляді кількості балів і winStreak. Окремо перевіряється, що профіль партнера доступний лише для перегляду і не може бути змінений іншим користувачем.

Тестування магазину винагород полягає у перегляді доступних винагород, перевірці достатності балів, покупці винагороди та списанні балів. Після покупки перевіряється створення запису про покупку та поява повідомлення у партнера.

Тестування налаштувань полягає у зміні теми, мови та валюти. Для теми перевіряється коректне відображення екранів у світлому та темному режимі. Для локалізації перевіряється, що після вибору мови та перезапуску застосунку інтерфейс відображається відповідною мовою. Для валюти перевіряється зміна символу валюти у фінансовому модулі без виконання конвертації сум.

Окремо було перевірено сценарій завершення строку дії JWT-токена. У разі отримання відповіді сервера з помилкою авторизації застосунок повинен очистити локальну сесію та повернути користувача до екрана входу, замість відображення помилки на всіх екранах.

Для аналізу результатів тестування було складено перелік основних сценаріїв і очікуваних результатів. Узагальнені результати подано у таблиці 3.2.

Таблиця 3.2 – Результати тестування основних сценаріїв

Сценарій	Очікуваний результат	Результат тестування
Реєстрація користувача	Створення нового облікового запису	Виконано успішно
Авторизація користувача	Отримання JWT-токена та перехід у застосунок	Виконано успішно
Створення пари	Формування pairId і коду приєднання	Виконано успішно
Приєднання партнера	Другий користувач підключається до тієї ж пари	Виконано успішно
Створення задачі	Задача з'являється у списку та календарі	Виконано успішно
Підтвердження задачі	Бали нараховуються після підтвердження партнером	Виконано успішно
Повторювана задача	Екземпляри створюються на майбутні дати	Виконано успішно після виправлення логіки повторення
Спільна задача	Бали розподіляються після погодження партнерів	Виконано успішно
Створення події	Подія відображається у календарі	Виконано успішно
Фінансова транзакція	Запис відображається в історії	Виконано успішно
Підтвердження транзакції	Спільна транзакція впливає на баланс після підтвердження	Виконано успішно
Wishlist-покупка	Покупка може створити фінансову транзакцію	Виконано успішно
Покупка винагороди	Бали списуються, партнер отримує повідомлення	Виконано успішно
Зміна мови	Після перезапуску застосунок відображає вибрану мову	Виконано успішно після виправлення ресурсів локалізації
Зміна теми	Інтерфейс коректно змінює світлу/темну тему	Виконано успішно після виправлення кольорів
Завершення строку дії токена	Сесія очищується, користувач повертається на екран входу	Виконано успішно

У процесі тестування було виявлено низку помилок. Частина з них стосувалася інтерфейсу користувача, зокрема некоректних відступів, дублювання відображення балів і неправильного відображення окремих елементів у темній темі. Також було виявлено проблеми з локалізацією, коли частина екранів залишалась однією мовою незалежно від вибору користувача. Окрему увагу було приділено обробці завершення строку дії токена, оскільки без цього користувач міг залишатися у застосунку з недійсною сесією та отримувати помилки на різних екранах.

Після внесення виправлень основні сценарії роботи застосунку виконуються коректно. Тестування підтвердило працездатність ключових модулів: авторизації, пари користувачів, задач, календаря, фінансів, wishlist, профілю, магазину винагород, повідомлень і налаштувань. Результати тестування засвідчили, що застосунок може бути використаний для подальшого практичного тестування двома користувачами на окремих Android-пристроях.

3.6 Висновки

Для реалізації мобільного застосунку для автоматизації розподілу обов'язків у сім'ї було проведено комплекс етапів роботи, що базувалися на результатах попереднього проєктування системи.

Першим важливим етапом початку реалізації застосунку була підготовка середовища розробки та конфігурація проєкту. Було налаштовано Android-проєкт у середовищі Android Studio, підключено необхідні залежності для Jetpack Compose, Navigation Compose, Retrofit, DataStore та ViewModel. Також було підготовлено backend-частину на основі Node.js, Express.js, TypeScript, PostgreSQL та Prisma ORM.

Після налаштування базової архітектури було реалізовано спроєктовані раніше модулі. На стороні Android-застосунку було створено екрани авторизації, реєстрації, створення пари, календаря, задач, фінансів, wishlist, профілю, магазину винагород і налаштувань. Для кожного з основних екранів було

реалізовано відповідні ViewModel, що відповідають за стан інтерфейсу, обробку дій користувача та взаємодію з backend API.

Окремим важливим етапом стала програмна реалізація серверної частини. Було створено REST API для авторизації користувачів, керування парами, задачами, календарними подіями, фінансовими транзакціями, wishlist-елементами, профілями, винагородами та повідомленнями. Для роботи з базою даних використано Prisma ORM, що дозволило описати моделі предметної області та виконати міграції структури бази даних.

Найскладнішим з точки зору бізнес-логіки етапом стала реалізація модуля задач. У ньому було реалізовано гейміфіковану систему балів, банк задачі, підтвердження виконання партнером, повторювані задачі та спільні задачі з погодженням розподілу винагороди. Така логіка потребувала узгодженої роботи клієнтської частини, backend-контролерів і бази даних.

Також було реалізовано календарний модуль, який об’єднує задачі та події в єдиному представленні. Календар дозволяє переглядати записи за датами, створювати задачі й події на конкретний день, а також працювати з повторюваними записами. Це забезпечило зручну інтеграцію планування з модулем побутових задач.

Фінансовий модуль було реалізовано з підтримкою доходів, витрат, категорій, загального бюджету пари та балансу між партнерами. Для спільних транзакцій передбачено підтвердження партнером, що підвищує прозорість фінансових взаєморозрахунків. Додатково було реалізовано інтеграцію wishlist із фінансами, завдяки якій придбана покупка може автоматично створювати фінансовий запис.

Створення елементів користувацького інтерфейсу за допомогою Jetpack Compose стало наступним важливим кроком розробки. Інтерфейс застосунку було побудовано на основі Material Design, з використанням карток, нижньої навігації, діалогових вікон, нижніх панелей і візуально виділених елементів

вибору. Було реалізовано підтримку світлої та темної теми, української та англійської локалізації, а також вибір валюти для фінансового модуля.

Разом зі створенням модулів відбувалося тестування застосунку. Зважаючи на клієнт-серверну архітектуру та необхідність взаємодії двох користувачів, основна увага приділялася ручному інтеграційному тестуванню. Було перевірено сценарії реєстрації, авторизації, створення пари, створення й підтвердження задач, роботу календаря, фінансів, wishlist, профілю, магазину винагород, повідомлень і налаштувань.

У процесі тестування було виявлено та виправлено помилки, пов’язані з відображенням інтерфейсу, дублюванням даних, локалізацією, темною темою, обробкою завершення строку дії токена та логікою повторюваних задач. Після внесення виправлень основні сценарії роботи застосунку виконуються коректно.

Результатом цього розділу став реалізований і протестований мобільний застосунок, призначений для автоматизації розподілу обов’язків у сім’ї. Застосунок забезпечує створення спільного простору для пари, розподіл задач, календарне планування, фінансовий облік, wishlist, систему балів, внутрішній магазин винагород і персоналізацію профілю користувача. Реалізована система відповідає поставленим функціональним вимогам і створює основу для подальшого практичного тестування та розширення функціональності.

4 ВИСНОВКИ

Узагальнення результатів розробки мобільного застосунку для автоматизації розподілу обов'язків у сім'ї дало змогу сформулювати підсумкові висновки щодо виконаної кваліфікаційної роботи.

Виконання роботи розпочалося з дослідження предметної області та постановки задачі. Було проаналізовано особливості організації спільного побуту, розподілу домашніх обов'язків, планування спільного часу та ведення сімейних фінансів. У результаті встановлено, що наявні програмні рішення переважно охоплюють лише окремі аспекти цієї предметної області: календарне планування, задачі, фінансовий облік або списки покупок. Це підтвердило актуальність створення комплексного мобільного застосунку, орієнтованого саме на взаємодію двох партнерів у межах спільного побуту.

Під час аналізу наявного програмного забезпечення було розглянуто застосунки для календарного планування, таск-менеджменту, обліку витрат, сімейної організації та гейміфікованої продуктивності. Порівняння аналогів показало, що більшість із них не поєднують в одному середовищі розподіл побутових задач, календар, фінанси, wishlist та систему мотивації. На основі цього було обґрунтовано доцільність розроблення нового програмного продукту.

У межах постановки задачі було визначено функціональні та нефункціональні вимоги до застосунку. До основних функціональних вимог належали авторизація користувачів, створення пари, розподіл задач, підтвердження виконання, календарне планування, фінансовий облік, wishlist, профілі користувачів, магазин винагород, повідомлення, локалізація та налаштування інтерфейсу. До нефункціональних вимог віднесено захист даних, зручність користування, підтримку світлої й темної теми, української та англійської мов, стабільну роботу з backend API та можливість подальшого розширення системи.

Після визначення вимог було проведено проєктування програмного забезпечення. Для реалізації системи обрано клієнт-серверну архітектуру,

оскільки застосунок повинен синхронізувати дані між двома користувачами. На стороні Android-застосунку використано архітектурний підхід MVVM, що забезпечує розділення інтерфейсу користувача, стану екранів і логіки роботи з даними. Серверну частину спроєктовано за модульним принципом із використанням маршрутів, контролерів, middleware та окремого рівня доступу до бази даних.

Було спроєктовано структуру бази даних, що охоплює основні сутності предметної області: користувача, пару, задачу, серію повторюваних задач, календарну подію, фінансову транзакцію, wishlist-елемент, винагороду, покупку винагороди, повідомлення та шаблон. Для зберігання даних обрано PostgreSQL, оскільки система має значну кількість взаємопов'язаних сутностей і потребує підтримки транзакцій. Для взаємодії backend-частини з базою даних використано Prisma ORM.

На основі визначеної архітектури було проведено декомпозицію системи на функціональні модулі. Було виокремлено модулі авторизації, керування парою, задач і викликів, календаря, фінансів, wishlist, профілю, магазину винагород, повідомлень, налаштувань, локального збереження сесії та взаємодії з backend API. Для кожного модуля описано зони відповідальності, залежності та програмні інтерфейси.

На етапі вибору технологій для мобільної частини було обрано Kotlin, Jetpack Compose, Navigation Compose, Retrofit, DataStore та ViewModel. Для серверної частини використано Node.js, Express.js, TypeScript, PostgreSQL, Prisma ORM та JWT. Такий стек дозволив реалізувати повноцінний клієнт-серверний застосунок із сучасним інтерфейсом, захищеною авторизацією, синхронізацією даних і можливістю подальшого розвитку.

Після завершення етапу проєктування було виконано програмну реалізацію застосунку. На Android-стороні реалізовано екрани авторизації, реєстрації, створення календаря, задач, фінансів, wishlist, профілю, магазину винагород і налаштувань.

Серверна частина реалізована як REST API. Було створено маршрути та контролери для авторизації, роботи з парами, задачами, подіями, фінансовими транзакціями, wishlist, профілями, винагородами та повідомленнями. Для захищених маршрутів використано JWT middleware, що дозволяє перевіряти користувача та обмежувати доступ до даних лише в межах його пари.

Особливу увагу під час реалізації приділено модулю задач. У ньому реалізовано гейміфіковану логіку з банком балів, підтвердженням виконання партнером, повторюваними задачами та спільними задачами з погодженням розподілу винагороди. Це дозволило зробити розподіл побутових обов'язків не лише формальним, а й мотиваційним.

Також було реалізовано календарний модуль, що об'єднує задачі та події в одному представленні. Фінансовий модуль забезпечує облік доходів і витрат, категорії, загальний бюджет пари та баланс між партнерами. Wishlist дозволяє планувати майбутні покупки та за потреби створювати фінансові записи після придбання товару. Профіль і магазин винагород забезпечують персоналізацію, відображення балів та використання внутрішньої системи мотивації.

У процесі тестування було виявлено та виправлено низку помилок, зокрема проблеми з локалізацією, темною темою, дублюванням відображення балів, логікою повторюваних задач і обробкою недійсного токена. Після внесення виправлень основні сценарії роботи застосунку виконуються коректно.

Отже, можна стверджувати, що мета кваліфікаційної роботи, яка полягала у розробленні мобільного застосунку для автоматизації розподілу обов'язків у сім'ї, була досягнута. Розроблений застосунок забезпечує створення спільного простору для пари, розподіл задач, календарне планування, фінансовий облік, wishlist, систему балів, магазин винагород, профілі користувачів і налаштування інтерфейсу. Результати виконання роботи підтверджують відповідність програмного продукту встановленим функціональним і нефункціональним вимогам, а також створюють основу для подальшого практичного використання та розвитку системи.

5 ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Android Developers. Guide to app architecture [Електронний ресурс] – Режим доступу: https://developer.android.com/topic/architecture (дата звернення: 21.03.2026).
2. Android Developers. DataStore [Електронний ресурс] – Режим доступу: https://developer.android.com/topic/libraries/architecture/datastore (дата звернення: 14.03.2026).
3. Android Developers. Get started with Jetpack Compose [Електронний ресурс] – Режим доступу: https://developer.android.com/develop/ui/compose/documentation (дата звернення: 26.03.2026).
4. Android Developers. Jetpack Compose UI App Development Toolkit [Електронний ресурс] – Режим доступу: https://developer.android.com/compose (дата звернення: 05.04.2026).
5. Android Developers. State and Jetpack Compose [Електронний ресурс] – Режим доступу: https://developer.android.com/develop/ui/compose/state (дата звернення: 03.04.2026).
6. Android Developers. Kotlin and Android [Електронний ресурс] – Режим доступу: https://developer.android.com/kotlin (дата звернення: 13.04.2026).
7. Kotlin Documentation. Kotlin for Android [Електронний ресурс] – Режим доступу: https://kotlinlang.org/docs/android-overview.html (дата звернення: 28.03.2026).
8. Square. Retrofit Introduction [Електронний ресурс] – Режим доступу: https://square.github.io/retrofit/ (дата звернення: 08.04.2026).
9. Express.js. Express – Node.js web application framework [Електронний ресурс] – Режим доступу: https://expressjs.com/ (дата звернення: 01.04.2026).

10. Express.js. Using middleware [Електронний ресурс] – Режим доступу: https://expressjs.com/en/guide/using-middleware.html (дата звернення: 01.04.2026).
11. Express.js. Routing [Електронний ресурс] – Режим доступу: https://expressjs.com/en/guide/routing.html (дата звернення: 03.04.2026).
12. TypeScript Documentation [Електронний ресурс] – Режим доступу: https://www.typescriptlang.org/docs/ (дата звернення: 16.04.2026).
13. Node.js Documentation [Електронний ресурс] – Режим доступу: https://nodejs.org/en/docs (дата звернення: 01.05.2026).
14. PostgreSQL Documentation [Електронний ресурс] – Режим доступу: https://www.postgresql.org/docs/ (дата звернення: 02.05.2026).
15. PostgreSQL. About PostgreSQL [Електронний ресурс] – Режим доступу: https://www.postgresql.org/about/ (дата звернення: 21.03.2026).
16. Prisma ORM. Quickstart with PostgreSQL [Електронний ресурс] – Режим доступу: https://www.prisma.io/docs/prisma-orm/quickstart/postgresql (дата звернення: 13.04.2026).
17. Prisma ORM. Prisma Migrate [Електронний ресурс] – Режим доступу: https://www.prisma.io/docs/orm/prisma-migrate (дата звернення: 11.03.2026).
18. JSON Web Token. Introduction [Електронний ресурс] – Режим доступу: https://jwt.io/introduction (дата звернення: 08.04.2026).
19. Google Material Design. Material Design 3 [Електронний ресурс] – Режим доступу: https://m3.material.io/ (дата звернення: 18.04.2026).
20. Android Developers. App resources overview [Електронний ресурс] – Режим доступу: https://developer.android.com/guide/topics/resources/providing-resources (дата звернення: 11.05.2026).

21. Android Developers. App localization [Електронний ресурс] – Режим доступу: https://developer.android.com/guide/topics/resources/localization (дата звернення: 17.04.2026).
22. Android Developers. Navigation with Compose [Електронний ресурс] – Режим доступу: https://developer.android.com/develop/ui/compose/navigation (дата звернення: 17.04.2026).
23. Android Developers. ViewModel overview [Електронний ресурс] – Режим доступу: https://developer.android.com/topic/libraries/architecture/viewmodel (дата звернення: 09.03.2026).
24. Neon. PostgreSQL database platform documentation [Електронний ресурс] – Режим доступу: https://neon.com/docs (дата звернення: 17.05.2026).
25. Render. Deploy a Node.js app [Електронний ресурс] – Режим доступу: https://render.com/docs/deploy-node-express-app (дата звернення: 14.04.2026).
26. Android Developers. Kotlin coroutines on Android [Електронний ресурс] – Режим доступу: https://developer.android.com/kotlin/coroutines (дата звернення: 07.05.2026).
27. Android Developers. Lifecycle-aware components [Електронний ресурс] – Режим доступу: https://developer.android.com/topic/libraries/architecture/lifecycle (дата звернення: 08.05.2026).
28. Android Developers. Save data using DataStore [Електронний ресурс] – Режим доступу: https://developer.android.com/topic/libraries/architecture/datastore (дата звернення: 09.05.2026).
29. Android Developers. Navigation with Compose [Електронний ресурс] – Режим доступу: https://developer.android.com/develop/ui/compose/navigation (дата звернення: 10.05.2026).
30. Android Developers. Material Design in Compose [Електронний ресурс] – Режим доступу: https://developer.android.com/develop/ui/compose/designsystems/material3 (дата звернення: 11.05.2026).

31. Android Developers. ViewModel overview [Електронний ресурс] – Режим доступу: https://developer.android.com/topic/libraries/architecture/viewmodel (дата звернення: 12.05.2026).
32. Android Developers. App localization [Електронний ресурс] – Режим доступу: https://developer.android.com/guide/topics/resources/localization (дата звернення: 13.05.2026).
33. Kotlin Documentation. Kotlin language documentation [Електронний ресурс] – Режим доступу: https://kotlinlang.org/docs/home.html (дата звернення: 14.05.2026).
34. Kotlin Documentation. Kotlin for Android [Електронний ресурс] – Режим доступу: https://kotlinlang.org/docs/android-overview.html (дата звернення: 15.05.2026).
35. Square. Retrofit documentation [Електронний ресурс] – Режим доступу: https://square.github.io/retrofit/ (дата звернення: 16.05.2026).
36. Coil. Image loading for Android and Compose [Електронний ресурс] – Режим доступу: https://coil-kt.github.io/coil/ (дата звернення: 18.05.2026).
37. Prisma ORM. Prisma ORM documentation [Електронний ресурс] – Режим доступу: https://www.prisma.io/docs (дата звернення: 20.05.2026).
38. Prisma ORM. Prisma Migrate documentation [Електронний ресурс] – Режим доступу: https://www.prisma.io/docs/orm/prisma-migrate (дата звернення: 22.05.2026).
39. Render. Deploy a Node Express App [Електронний ресурс] – Режим доступу: https://render.com/docs/deploy-node-express-app (дата звернення: 24.05.2026).
40. Neon Docs. Use Neon Postgres with Render [Електронний ресурс] – Режим доступу: https://neon.com/docs/guides/render (дата звернення: 25.05.2026).

ДОДАТОК А
(обов’язковий)

ГРАФІЧНІ МАТЕРІАЛИ

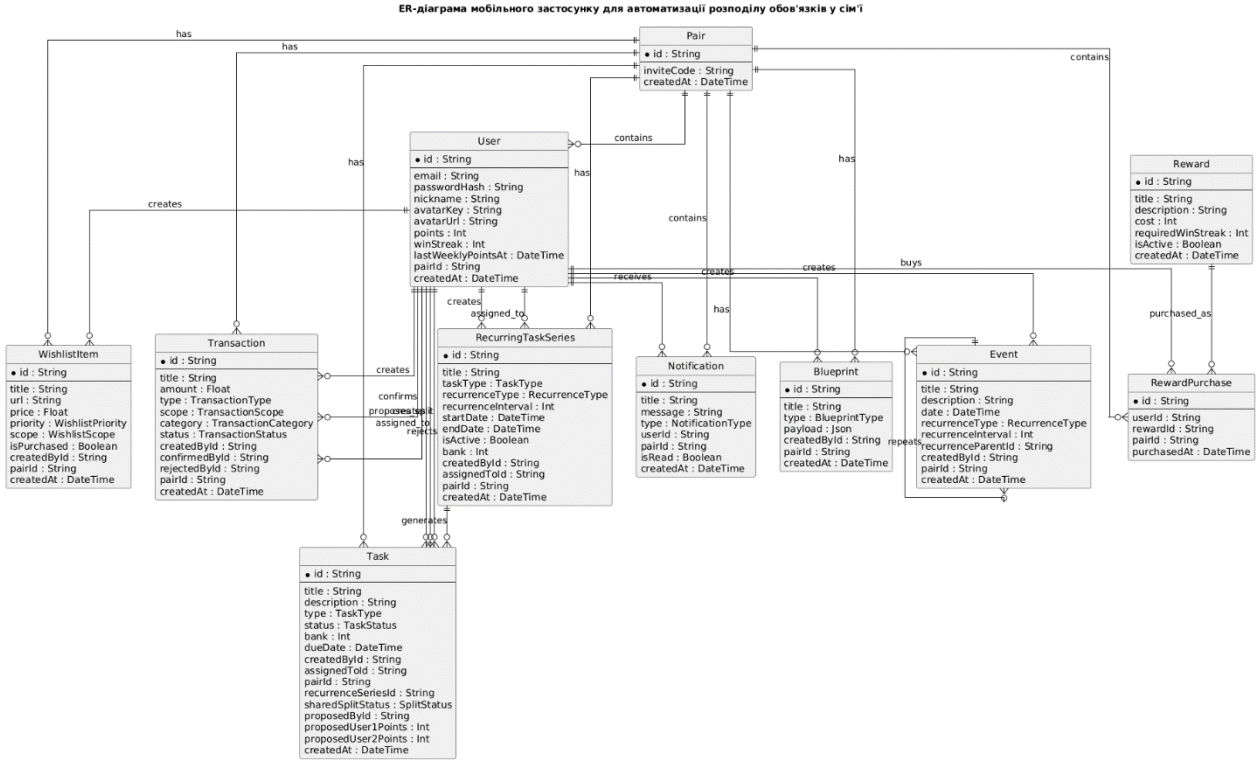


Рисунок А.1 – діаграма сутність-зв'язок

Рисунок А.2 – інтерфейс екрану календаря

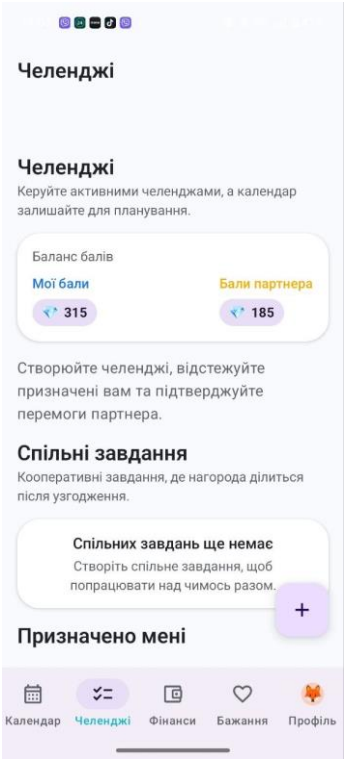


Рисунок А.3 – інтерфейс екрану челенджів

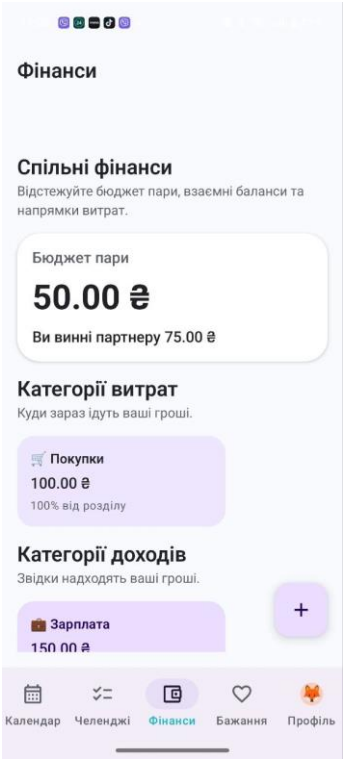


Рисунок А.4 – інтерфейс екрану фінансів

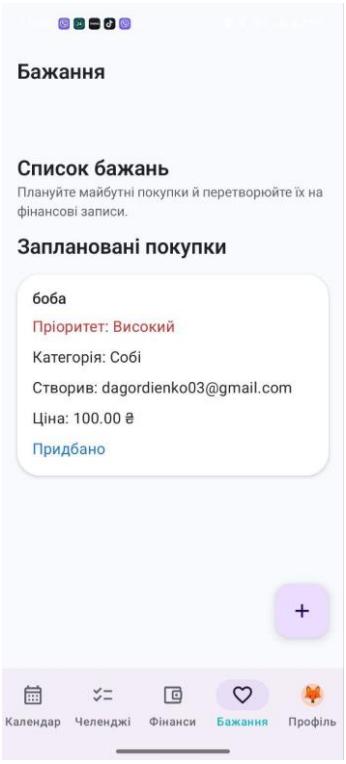


Рисунок А.5 – інтерфейс екрану вішлисту

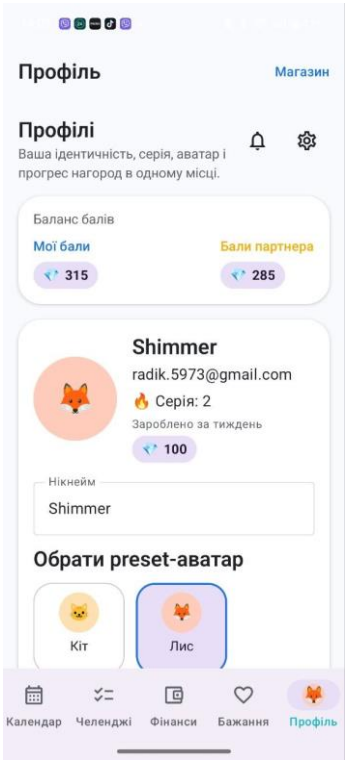


Рисунок А.6 – інтерфейс екрану профілю

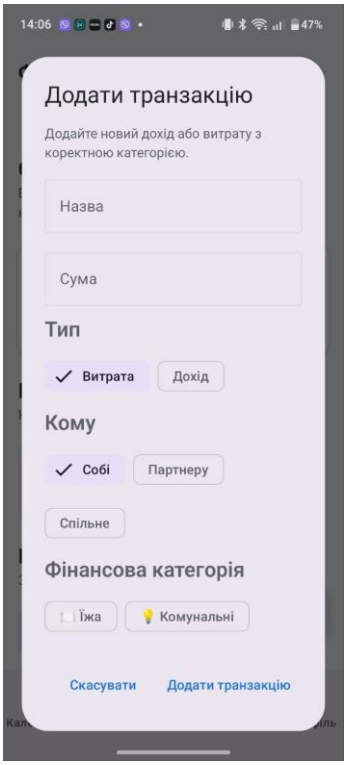


Рисунок А.7 – інтерфейс екрану створення транзакції

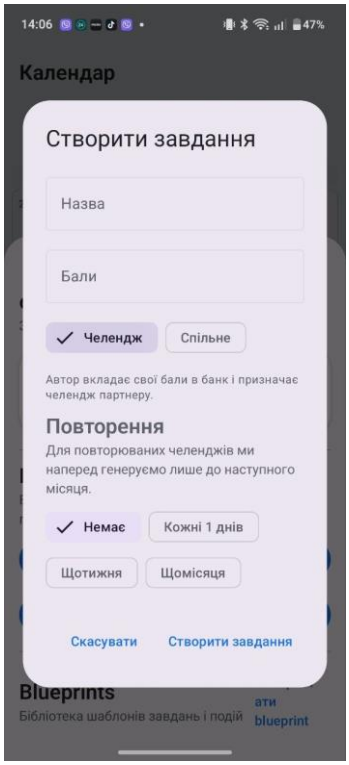


Рисунок А.8 – інтерфейс екрану створення завдання

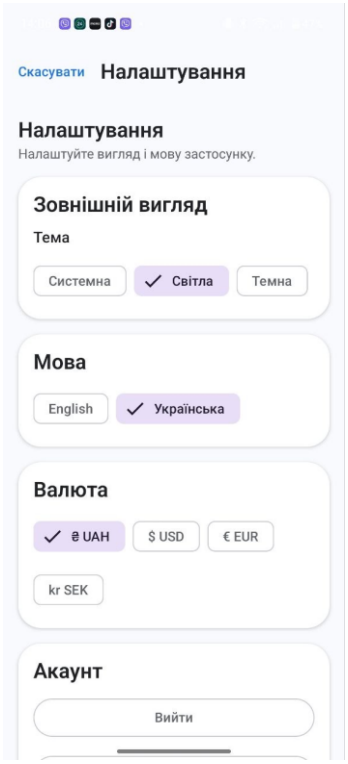


Рисунок А.9 – інтерфейс екрану налаштувань

ДОДАТОК Б
(обов'язковий)

ДЕТАЛЬНИЙ ОПИС ВАРІАНТІВ ВИКОРИСТАННЯ

Варіант використання: реєстрація користувача

Передумови:

- користувач відкрив мобільний застосунок;
- користувач ще не має облікового запису;
- пристрій має доступ до мережі Інтернет;
- серверна частина застосунку доступна для обробки запитів.

Тригер: користувач натискає кнопку переходу до екрана реєстрації.

Основний потік:

- система відображає екран реєстрації;
- користувач вводить адресу електронної пошти;
- користувач вводить пароль;
- користувач підтверджує створення облікового запису;
- система перевіряє коректність заповнення полів;
- мобільний застосунок надсилає запит на сервер;
- сервер перевіряє, чи не існує користувач із такою адресою електронної пошти;
- сервер створює новий обліковий запис;
- сервер формує JWT-токен;
- застосунок зберігає токен у локальному сховищі;
- система перенаправляє користувача до екрана створення або приєднання до пари.

Альтернативні потоки:

- користувач може повернутися до екрана входу без створення облікового запису;
- якщо користувач уже має акаунт, він може перейти до авторизації;
- якщо введені дані некоректні, система дозволяє виправити їх без перезапуску процесу.
- Заборонні потоки:
- якщо email уже використовується іншим користувачем, система не створює новий обліковий запис;
- якщо пароль або email не відповідають вимогам, запит не виконується;
- якщо сервер недоступний, користувач отримує повідомлення про помилку.

Гарантії:

– мінімальна гарантія: у разі помилки новий користувач не створюється, а локальна сесія не зберігається;

– гарантія успіху: користувача зареєстровано, JWT-токен збережено, користувач переходить до наступного етапу налаштування застосунку.

Діаграму станів для варіанту використання «реєстрація користувача» зображено на рисунку Б.1.

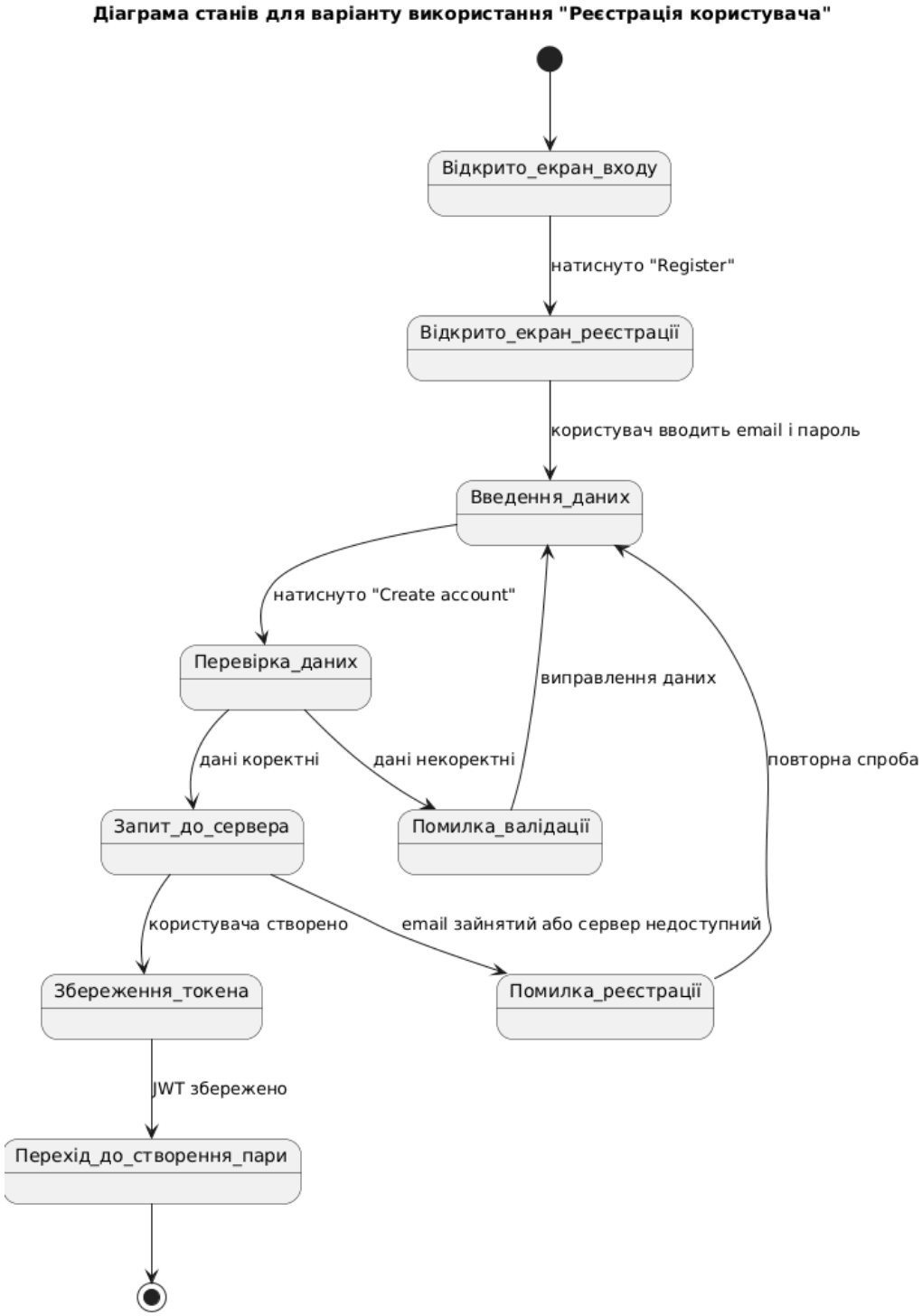


Рисунок Б.1 - Діаграма станів для варіанту використання «Реєстрація користувача»

Варіант використання «Авторизація користувача»

Передумови:

- користувач відкрив мобільний застосунок;
- користувач має раніше створений обліковий запис;
- пристрій має доступ до мережі Інтернет;
- серверна частина застосунку доступна.

Тригер: користувач вводить email і пароль та натискає кнопку входу.

Основний потік:

- система відображає екран входу;
- користувач вводить email;
- користувач вводить пароль;
- користувач натискає кнопку авторизації;
- система перевіряє заповнення полів;
- застосунок надсилає дані на сервер;
- сервер перевіряє наявність користувача;
- сервер порівнює введений пароль із збереженим хешем;
- сервер формує JWT-токен;
- застосунок зберігає токен у локальному сховищі;
- система перевіряє, чи належить користувач до пари;
- якщо користувач має пару, відкривається основний екран;
- якщо користувач не має пари, відкривається екран створення або підключення до пари.

Альтернативні потоки:

- користувач може перейти до екрана реєстрації;
- користувач може повторити введення даних після помилки;
- якщо користувач уже має збережену активну сесію, система може автоматично перейти до відповідного екрана.

Заборонні потоки:

- якщо email або пароль неправильні, авторизація не виконується;
- якщо токен не може бути збережений локально, система не переходить до основного екрана;
- якщо сервер недоступний, користувач отримує повідомлення про помилку.

Гарантії:

– мінімальна гарантія: у разі помилки користувач залишається на екрані входу, а стан застосунку не змінюється;

– гарантія успіху: користувач авторизований, токен збережено, застосунок відкриває екран відповідно до стану користувача.

Діаграму станів для варіанту використання «авторизація користувача» зображено на рисунку Б.2.

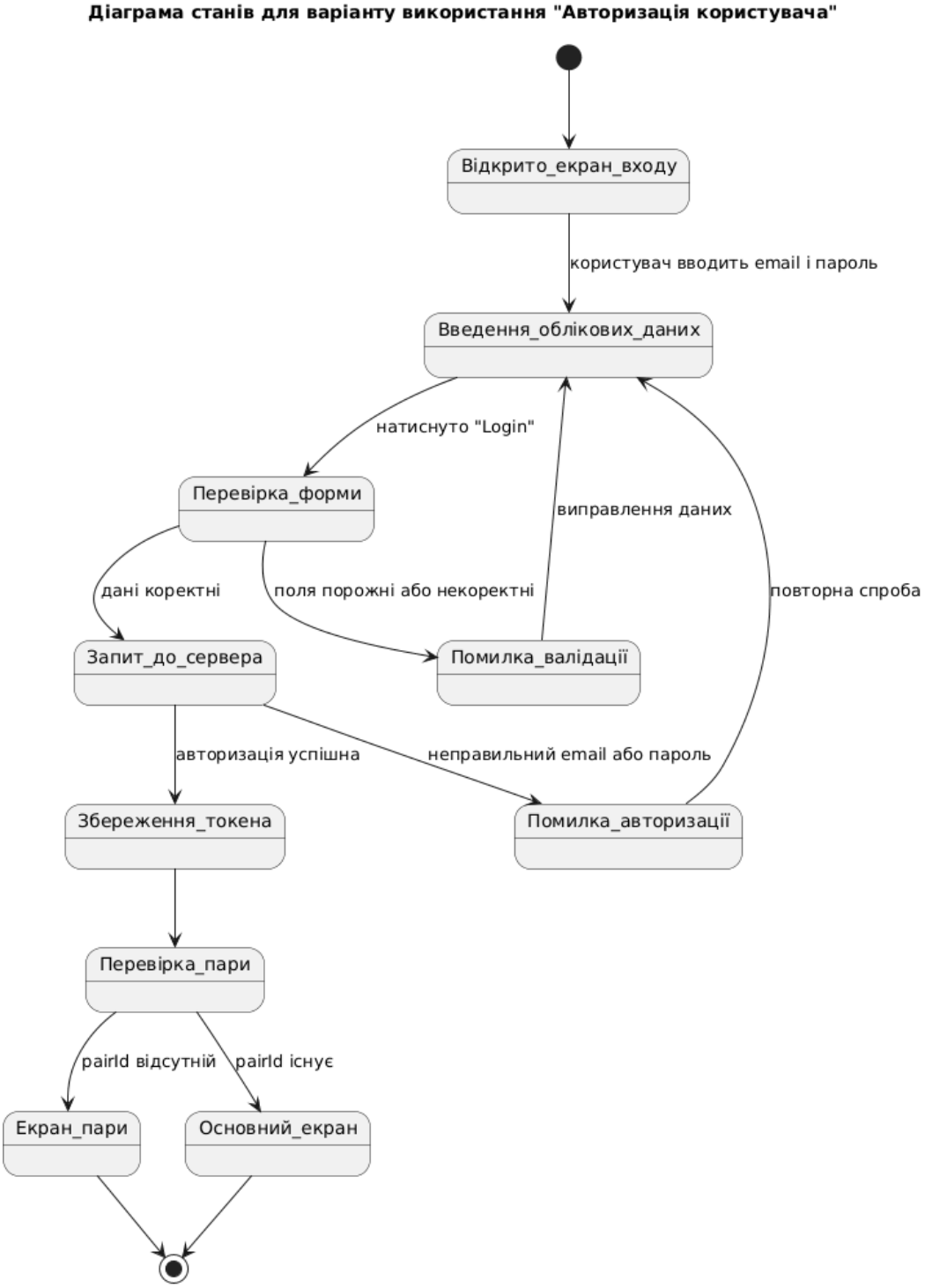


Рисунок Б.2 - Діаграма станів для варіанту використання «Авторизація користувача»

Варіант використання: створення або приєднання до пари

Передумови:

– користувач авторизований;
– користувач ще не належить до пари;
– серверна частина застосунку доступна;
– пристрій має доступ до мережі Інтернет.

Тригер: користувач натискає кнопку створення пари або вводить код існуючої пари.

Основний потік:

– система відкриває екран керування парою;
– користувач обирає створення нової пари або приєднання до вже існуючої;
– у разі створення пари застосунок надсилає відповідний запит на сервер;
– сервер перевіряє JWT-токен користувача;
– сервер створює нову пару та генерує код приєднання;
– сервер прив’язує користувача до створеної пари;
– застосунок зберігає ідентифікатор пари;
– система відкриває основний екран застосунку.

Для приєднання до пари:

– користувач вводить код пари;
– застосунок надсилає код на сервер;
– сервер перевіряє існування пари;
– сервер приєднує користувача до пари;
– застосунок зберігає ідентифікатор пари;
– система відкриває основний екран.

Альтернативні потоки:

– користувач може залишитися на екрані пари без виконання дії;
– користувач може повторно ввести код у разі помилки.

Заборонні потоки:

– користувач не може створити нову пару, якщо вже належить до іншої;
– користувач не може приєднатися до пари з некоректним кодом;
– у разі недійсного токена система очищає сесію та повертає користувача до екрана входу.

Гарантії:

– мінімальна гарантія: у разі помилки користувач залишається без пари, а локальні дані не змінюються;

– гарантія успіху: користувач прив’язаний до пари та отримує доступ до спільного простору застосунку.

Діаграму станів для варіанту використання «створення або приєднання до пари» зображено на рисунку Б.3.

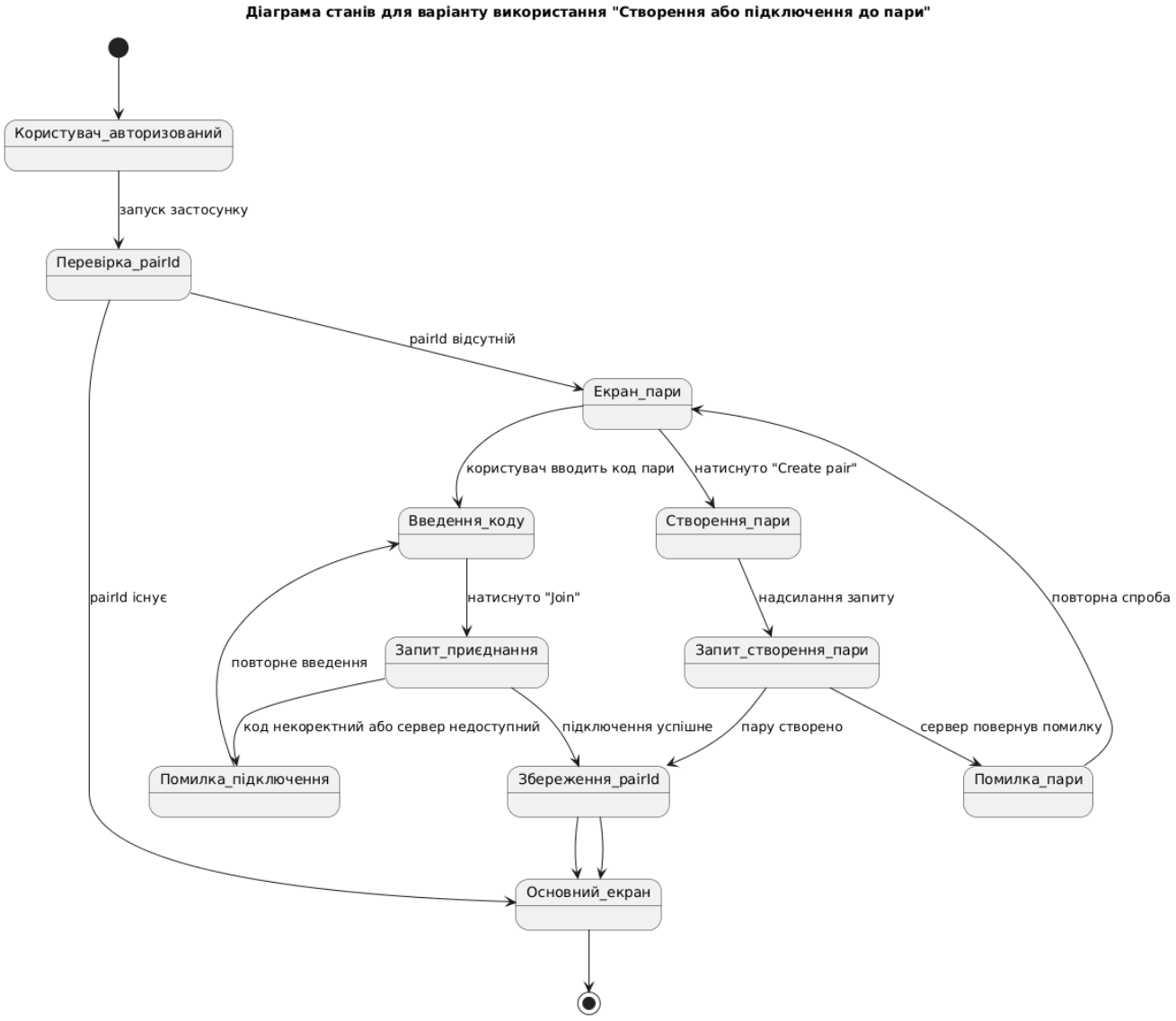


Рисунок Б.3 - Діаграма станів для варіанту використання «Створення або підключення до пари»

Варіант використання: створення задачі

Передумови:

- користувач авторизований;
- користувач належить до пари;
- користувач перебуває на екрані задач або календаря;
- серверна частина доступна.

Тригер: користувач натискає кнопку створення задачі.

Основний потік:

- система відкриває форму створення задачі;
- користувач вводить назву задачі;
- користувач обирає тип задачі;
- користувач вказує дату виконання;
- користувач вводить кількість балів;
- користувач за потреби обирає повторення задачі;
- користувач підтверджує створення;
- система перевіряє коректність введених даних;
- застосунок надсилає запит на сервер;
- сервер перевіряє авторизацію та належність користувача до пари;
- сервер створює задачу;
- для challenge-задачі сервер списує бали з автора та формує банк задачі;
- для спільної задачі банк формується системою без списання балів користувача;
- якщо обрано повторення, сервер створює серію повторюваних задач;
- застосунок оновлює список задач і календар.

Альтернативні потоки:

- користувач може створити задачу без повторення;
- користувач може створити задачу без прив’язки до конкретної дати, якщо це дозволено інтерфейсом;
- користувач може закрити форму без створення задачі.

Заборонні потоки:

- задача не створюється без назви;
- кількість балів не може бути від’ємною або некоректною;
- challenge-задача не створюється, якщо користувачу не вистачає балів;
- користувач без пари не може створювати спільні задачі.

Гарантії:

– мінімальна гарантія: у разі помилки задача не створюється, а бали користувача не змінюються;

– гарантія успіху: задачу створено, вона відображається у списку задач і календарі.

Діаграму станів для варіанту використання «створення задачі» зображено на рисунку Б.4.

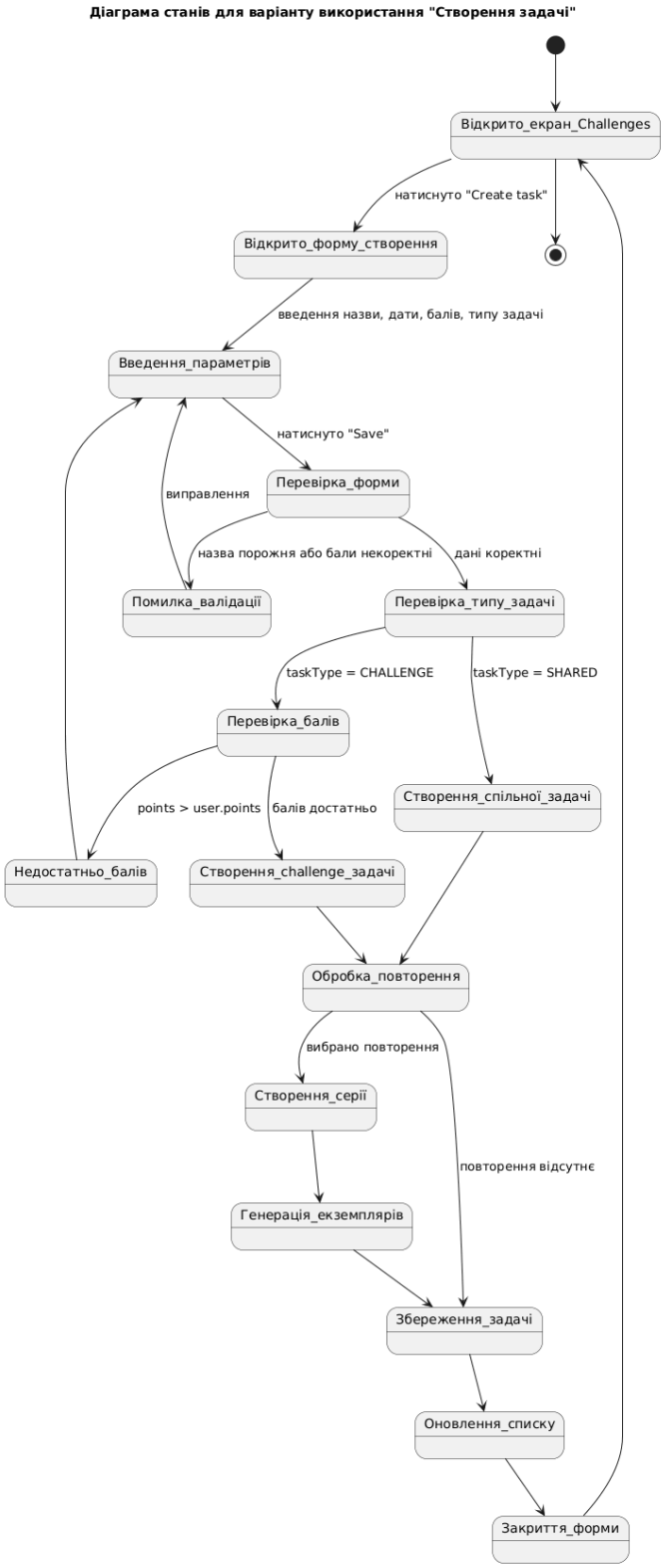


Рисунок Б.4 - Діаграма станів для варіанту використання «Створення задачі»

Варіант використання: редагування або видалення задачі

Передумови:

- користувач авторизований;
- користувач належить до пари;
- задача існує в системі;
- задача не має завершеного або проваленого статусу;
- користувач має право редагувати або видаляти задачу.

Тригер: користувач відкриває задачу та натискає кнопку редагування або видалення.

Основний потік редагування:

- система відкриває деталі задачі;
- система перевіряє права користувача;
- користувач натискає кнопку редагування;
- система відкриває форму редагування;
- користувач змінює назву або дату задачі;
- користувач підтверджує зміни;
- система перевіряє коректність введених даних;
- застосунок надсилає запит на сервер;
- сервер оновлює задачу;
- застосунок оновлює список задач і календар.

Основний потік видалення:

- користувач натискає кнопку видалення;
- система відображає діалог підтвердження;
- користувач підтверджує видалення;
- застосунок надсилає запит на сервер;
- сервер перевіряє права користувача;
- сервер видаляє задачу;
- застосунок оновлює список задач і календар.

Альтернативні потоки:

- користувач може скасувати редагування;
- користувач може скасувати видалення у діалоговому вікні;
- якщо задача є частиною повторюваної серії, користувач може окремо зупинити повторення.

Заборонні потоки:

– користувач не може редагувати або видаляти задачу, якщо не є її автором або не має відповідних прав;
– виконана або провалена задача не може бути змінена;
– недійсний токен призводить до очищення сесії.

Гарантії:

– мінімальна гарантія: у разі помилки задача залишається без змін;
– гарантія успіху: задачу оновлено або видалено, а інтерфейс відображає актуальні дані.

Діаграму станів для варіанту використання «редагування або видалення задачі» зображено на рисунку Б.5.

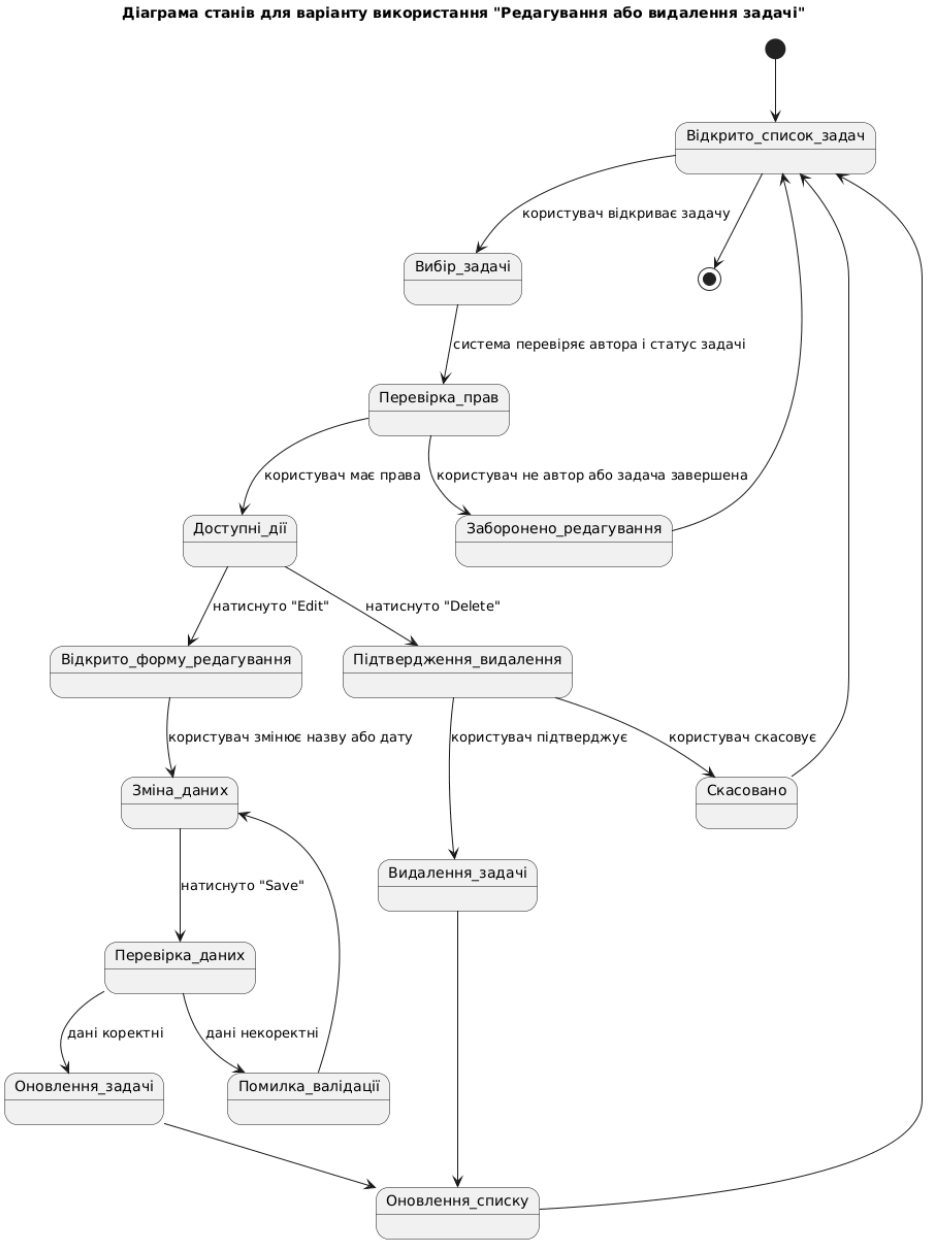


Рисунок Б.5 - Діаграма станів для варіанту використання «Редагування або видалення задачі»

Варіант використання: підтвердження виконання challenge-задачі

Передумови:

– користувач і партнер належать до однієї пари;
– задача існує та має активний статус;
– задача призначена одному з користувачів;
– задача має банк балів.

Тригер: виконавець натискає кнопку запиту підтвердження виконання.

Основний потік:

– користувач відкриває задачу;
– користувач натискає кнопку запиту підтвердження виконання;
– застосунок надсилає запит на сервер;
– сервер перевіряє, чи має користувач право виконувати задачу;
– сервер змінює статус задачі на очікування підтвердження;
– партнер бачить задачу у відповідному статусі;
– партнер відкриває задачу;
– партнер підтверджує виконання;
– сервер змінює статус задачі на виконану;
– сервер нараховує банк балів виконавцю;
– застосунок оновлює список задач і кількість балів.

Альтернативні потоки:

– партнер може відхилити підтвердження;
– у разі відхилення задача повертається до активного стану;
– виконавець може повторно надіслати запит після фактичного виконання задачі.

Заборонні потоки:

– користувач не може сам підтвердити власне виконання;
– задача не може бути підтверджена, якщо вона вже виконана або провалена;
– користувач поза парою не може змінювати статус задачі.

Гарантії:

– мінімальна гарантія: до підтвердження партнером бали не нараховуються;
– гарантія успіху: задача завершена, статус змінено на виконаний, бали нараховано виконавцю.

Діаграму станів для варіанту використання «підтвердження виконання challenge-задачі» зображено на рисунку Б.6.

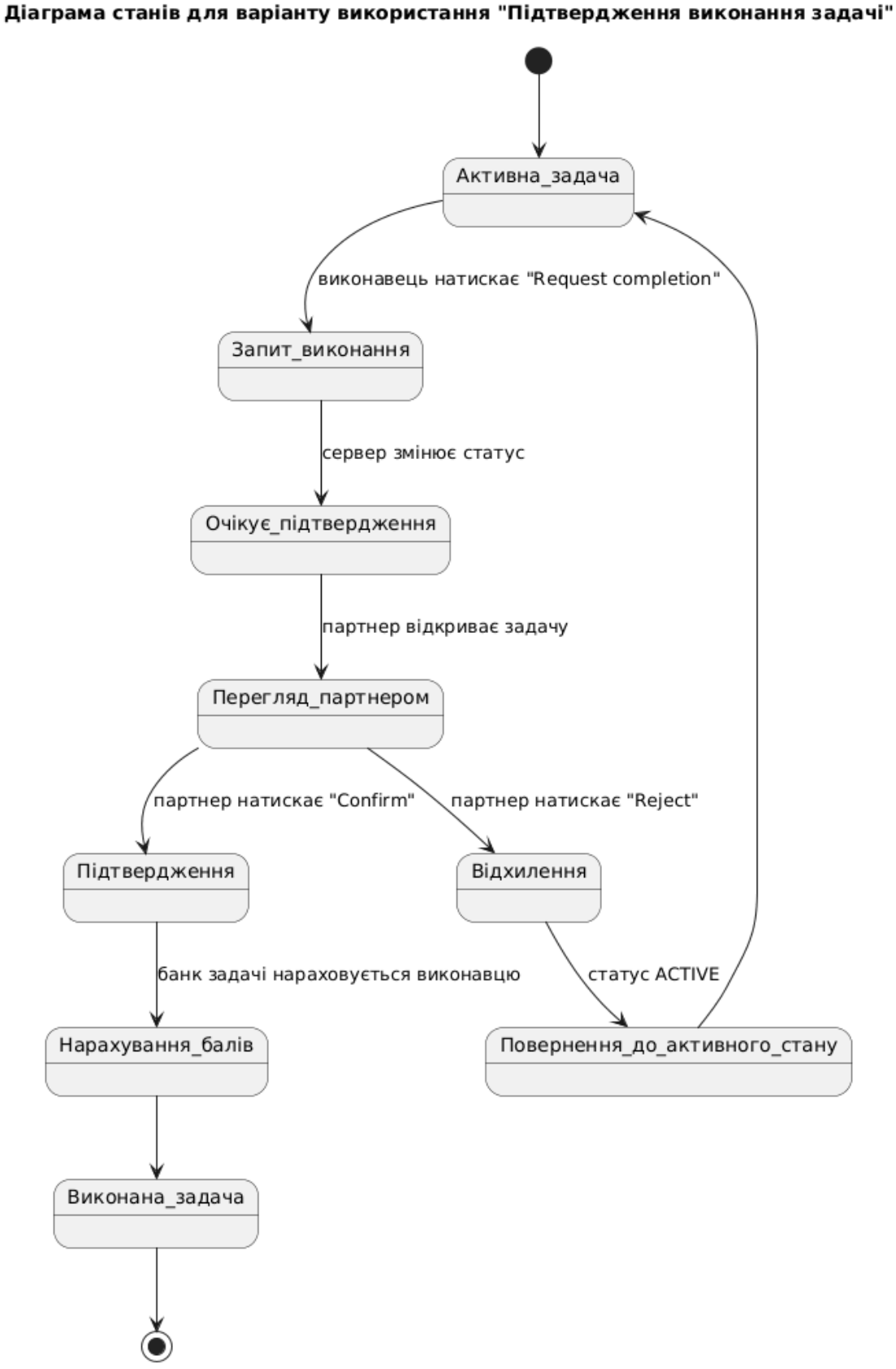


Рисунок Б.6 - Діаграма станів для варіанту використання «Підтвердження виконання задачі»

Варіант використання: спільна задача та погодження розподілу балів

Передумови:

- користувач і партнер належать до однієї пари;
- у системі існує активна спільна задача;
- задача має банк балів, який надається системою;
- обидва користувачі мають доступ до задачі.
- Тригер: один із користувачів натискає кнопку пропозиції розподілу балів після виконання спільної задачі.

Основний потік:

- користувач відкриває спільну задачу;
- користувач відкриває форму розподілу балів;
- система пропонує стандартний розподіл порівну;
- користувач вводить кількість балів для себе та партнера;
- система перевіряє, що сума балів дорівнює банку задачі;
- застосунок надсилає пропозицію на сервер;
- сервер зберігає пропозицію та переводить задачу в стан очікування погодження;
- партнер переглядає запропонований розподіл;
- партнер погоджується з розподілом;
- сервер нараховує бали обом користувачам;
- задача отримує статус виконаної.

Альтернативні потоки:

- партнер може не погодитися з розподілом;
- партнер може створити контрпропозицію;
- процес погодження може повторюватися, доки користувачі не дійдуть згоди.

Заборонні потоки:

- сума розподілених балів не може відрізнятися від банку задачі;
- значення балів не можуть бути від’ємними;
- задача не може бути завершена без погодження другого партнера;
- користувач поза парою не може брати участь у погодженні.

Гарантії:

- мінімальна гарантія: без погодження партнера бали не нараховуються;
- гарантія успіху: задача завершена, а банк балів розподілено між користувачами відповідно до погодженого рішення.

Діаграму станів для варіанту використання «спільна задача та погодження розподілу балів» зображено на рисунку Б.7.

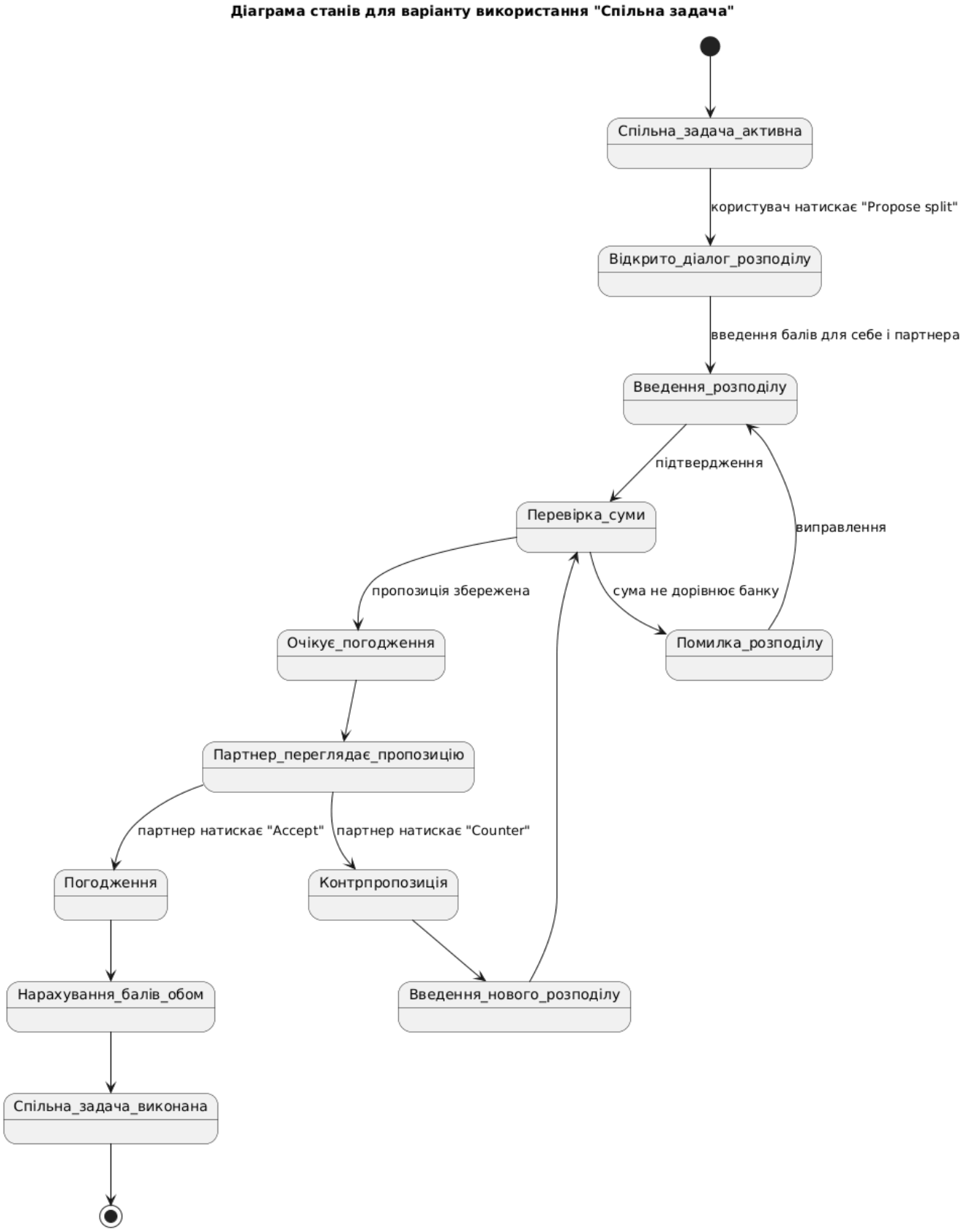


Рисунок Б.7 - Діаграма станів для варіанту використання «Спільна задача»

Варіант використання: створення календарної події

Передумови:

- користувач авторизований;
- користувач належить до пари;
- користувач перебуває на екрані календаря;
- серверна частина доступна.

Тригер: користувач обирає день у календарі та натискає кнопку створення події.

Основний потік:

- система відкриває календар;
- користувач обирає конкретну дату;
- система відкриває панель з деталями обраного дня;
- користувач натискає кнопку створення події;
- система відкриває форму створення події;
- користувач вводить назву події;
- користувач за потреби додає опис;
- користувач обирає або підтверджує дату;
- користувач за потреби обирає повторення події;
- система перевіряє коректність введених даних;
- застосунок надсилає запит на сервер;
- сервер створює подію;
- якщо вибрано повторення, сервер створює майбутні екземпляри події;
- застосунок оновлює календар.

Альтернативні потоки:

- користувач може створити одноразову подію без повторення;
- користувач може закрити форму без збереження;
- користувач може редагувати або видалити подію після створення.

Заборонні потоки:

- подія не створюється без назви;
- некоректна дата не приймається системою;
- користувач без пари не може створювати спільні події.

Гарантії:

- мінімальна гарантія: у разі помилки подія не створюється;
- гарантія успіху: подію створено, вона відображається в календарі на відповідній даті.

Діаграму станів для варіанту використання «створення календарної події» зображено на рисунку Б.8.

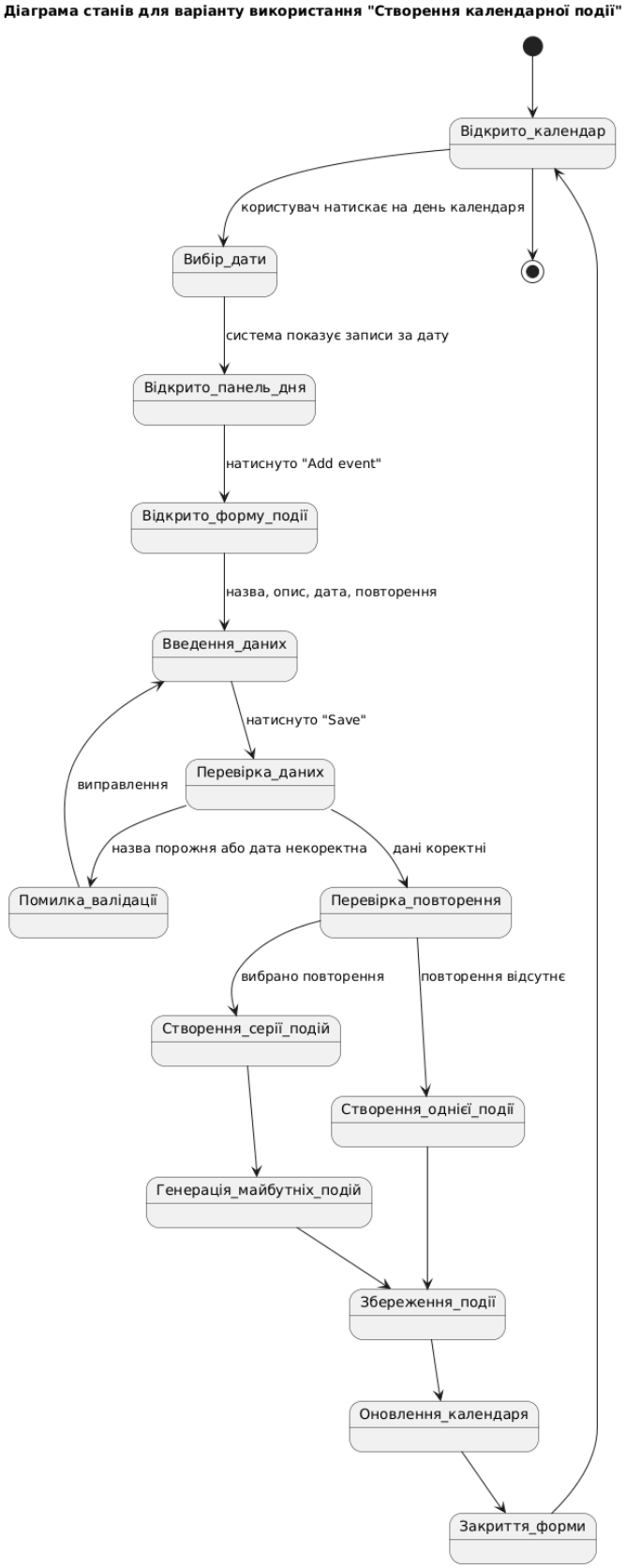


Рисунок Б.8 - Діаграма станів для варіанту використання «Створення календарної події»

Варіант використання: створення фінансової транзакції

Передумови:

- користувач авторизований;
- користувач належить до пари;
- користувач відкрив екран фінансів;
- серверна частина доступна.

Тригер: користувач натискає кнопку додавання фінансового запису.

Основний потік:

- система відкриває форму створення транзакції;
- користувач вводить назву транзакції;
- користувач вводить суму;
- користувач обирає тип операції: дохід або витрата;
- користувач обирає фінансову категорію;
- користувач обирає призначення транзакції: для себе, для партнера або для обох;
- користувач підтверджує створення;
- система перевіряє коректність введених даних;
- застосунок надсилає запит на сервер;
- сервер створює транзакцію;
- якщо транзакція створена для себе, вона одразу отримує підтверджений статус;
- якщо транзакція створена для партнера або для обох, вона отримує статус очікування підтвердження;
- застосунок оновлює історію транзакцій.

Альтернативні потоки:

- користувач може скасувати створення транзакції;
- користувач може змінити тип операції, після чого система оновить доступні категорії;
- користувач може відредагувати транзакцію після створення, якщо має відповідні права.

Заборонні потоки:

- сума транзакції не може бути меншою або рівною нулю;
- транзакція не створюється без назви або категорії;
- користувач без пари не може створювати спільні фінансові записи;
- недійсний токен призводить до виходу із сесії.

Гарантії:

– мінімальна гарантія: у разі помилки фінансовий запис не створюється;

– гарантія успіху: транзакція створена та відображається в історії фінансового модуля.

Діаграму станів для варіанту використання «створення фінансової транзакції» зображено на рисунку Б.9.

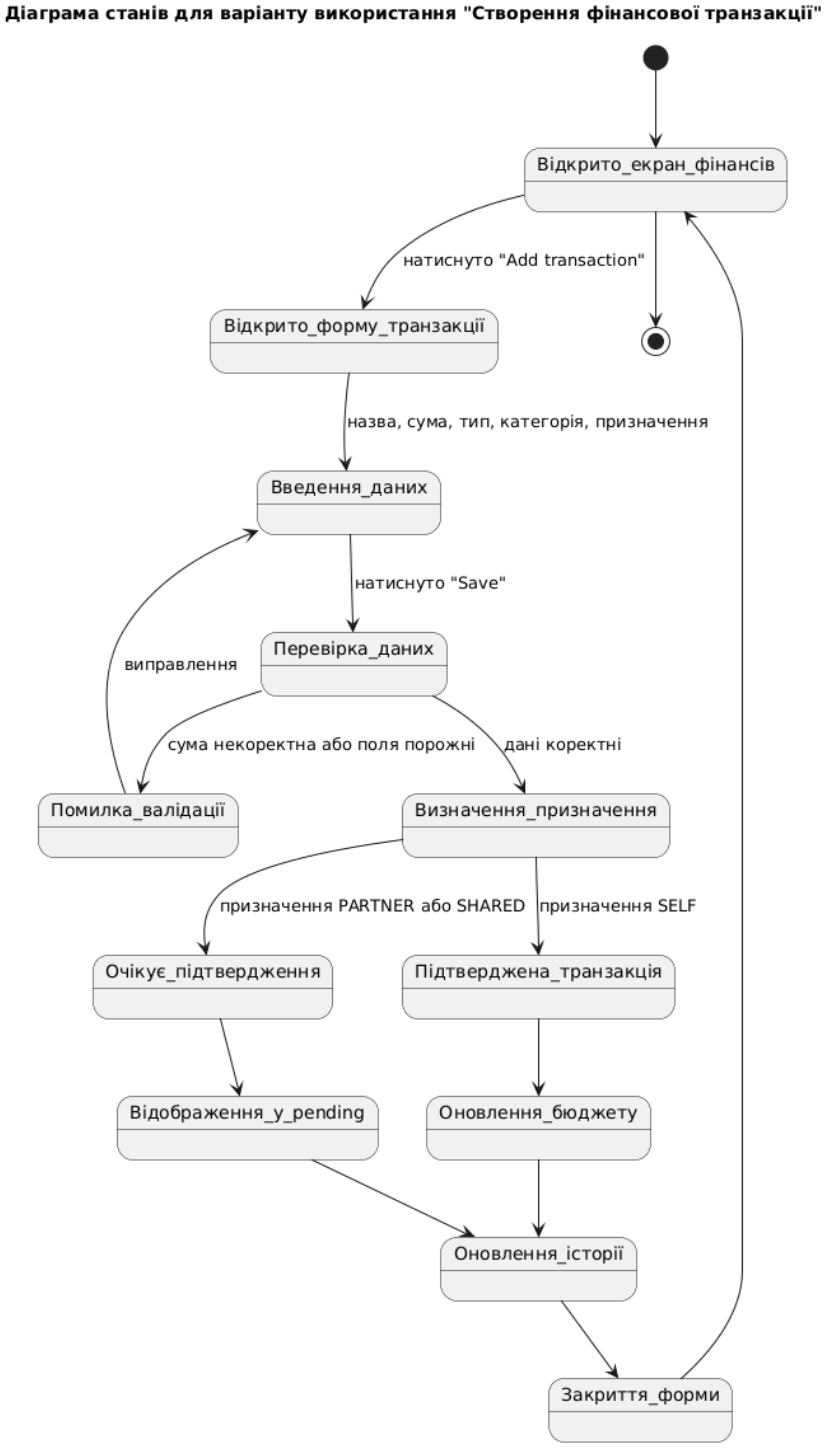


Рисунок Б.9 - Діаграма станів для варіанту використання

«Створення фінансової транзакції»

Варіант використання: підтвердження фінансової транзакції партнером

Передумови:

- користувач і партнер належать до однієї пари;
- існує фінансова транзакція зі статусом очікування підтвердження;
- транзакція створена одним користувачем і потребує підтвердження іншого.

Тригер: партнер натискає кнопку підтвердження або відхилення транзакції.

Основний потік:

- партнер відкриває екран фінансів;
- система відображає транзакції, що очікують підтвердження;
- партнер переглядає суму, категорію та призначення транзакції;
- партнер натискає кнопку підтвердження;
- застосунок надсилає запит на сервер;
- сервер перевіряє, що дію виконує не автор транзакції;
- сервер змінює статус транзакції на підтверджений;
- транзакція починає враховуватися у бюджеті пари;
- система оновлює баланс між партнерами;
- застосунок оновлює фінансовий екран.
- Альтернативні потоки:
- партнер може відхилити транзакцію;
- у разі відхилення транзакція не враховується у бюджеті та балансі;
- автор може створити нову транзакцію з виправленими даними.

Заборонні потоки:

- автор транзакції не може самостійно підтвердити власну спільну транзакцію;
- підтверджена або відхилена транзакція не може бути повторно підтверджена;
- користувач поза парою не може переглядати або змінювати транзакцію.

Гарантії:

- мінімальна гарантія: до підтвердження транзакція не впливає на бюджет і баланс;
- гарантія успіху: транзакція підтверджена та врахована у фінансових розрахунках.

Діаграму станів для варіанту використання «підтвердження фінансової транзакції партнером» зображено на рисунку Б.10.

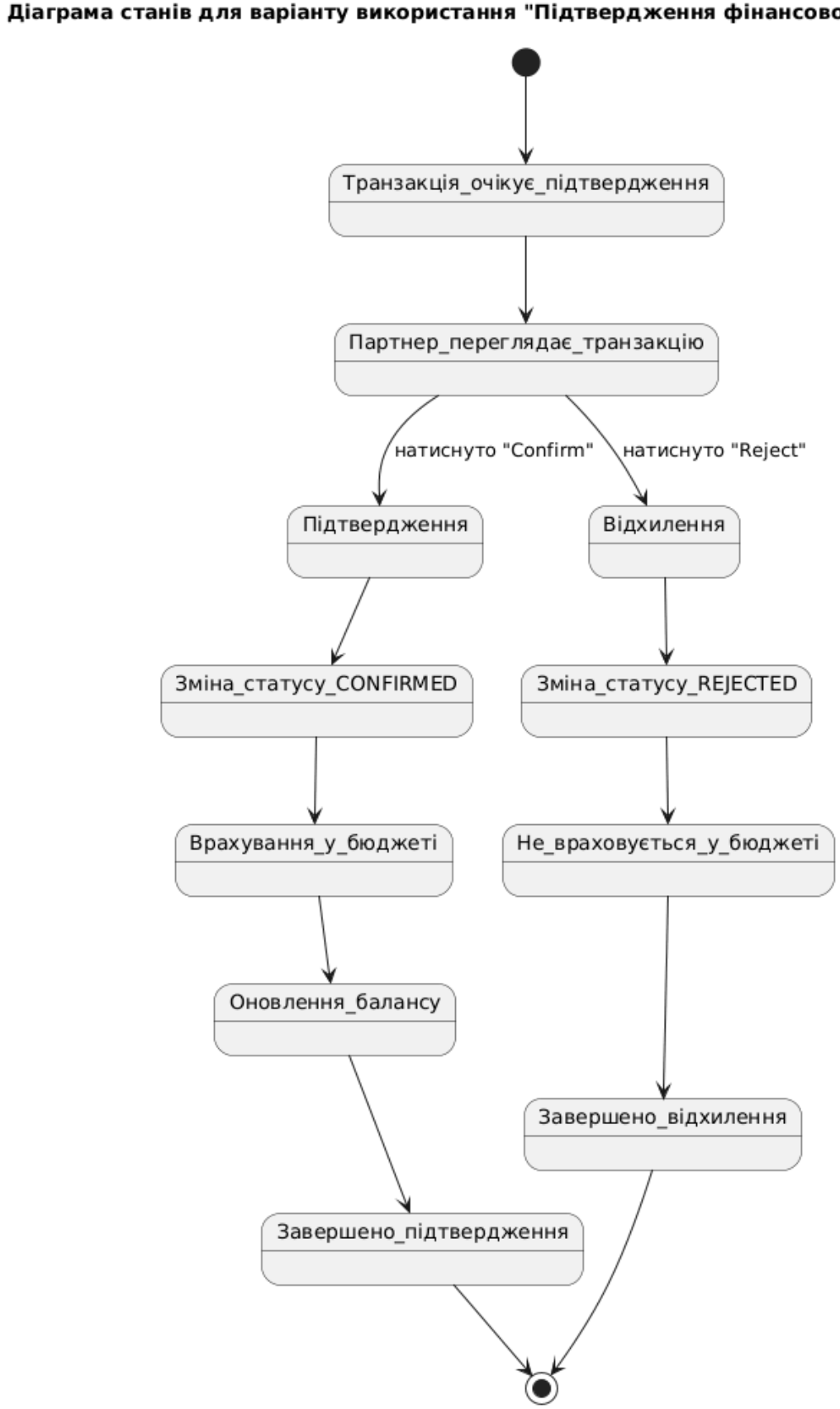


Рисунок Б.10 - Діаграма станів для варіанту використання «Підтвердження фінансової транзакції»

Варіант використання: створення wishlist-елемента

Передумови:

- користувач авторизований;
- користувач належить до пари;
- користувач відкрив екран wishlist;
- серверна частина доступна.

Тригер: користувач натискає кнопку додавання нового wishlist-елемента.

Основний потік:

- система відкриває форму створення wishlist-елемента;
- користувач вводить назву бажаної покупки;
- користувач за потреби додає посилання на товар;
- користувач вводить орієнтовну ціну;
- користувач обирає пріоритет;
- користувач обирає призначення покупки;
- користувач підтверджує створення;
- система перевіряє коректність введених даних;
- якщо посилання не містить схеми, система додає її автоматично або приводить посилання до коректного формату;
- застосунок надсилає запит на сервер;
- сервер створює wishlist-елемент;
- застосунок оновлює список wishlist.

Альтернативні потоки:

- користувач може створити елемент без посилання;
- користувач може створити елемент без ціни, якщо точна вартість невідома;
- користувач може закрити форму без створення запису.
- Заборонні потоки:
- wishlist-елемент не створюється без назви;
- ціна не може бути від’ємною;
- користувач без пари не може створювати спільні wishlist-записи.

Гарантії:

- мінімальна гарантія: у разі помилки wishlist-елемент не створюється;
- гарантія успіху: wishlist-елемент створено та відображено у списку бажаних покупок.

Діаграму станів для варіанту використання «створення wishlist-елемента» зображено на рисунку Б.11.

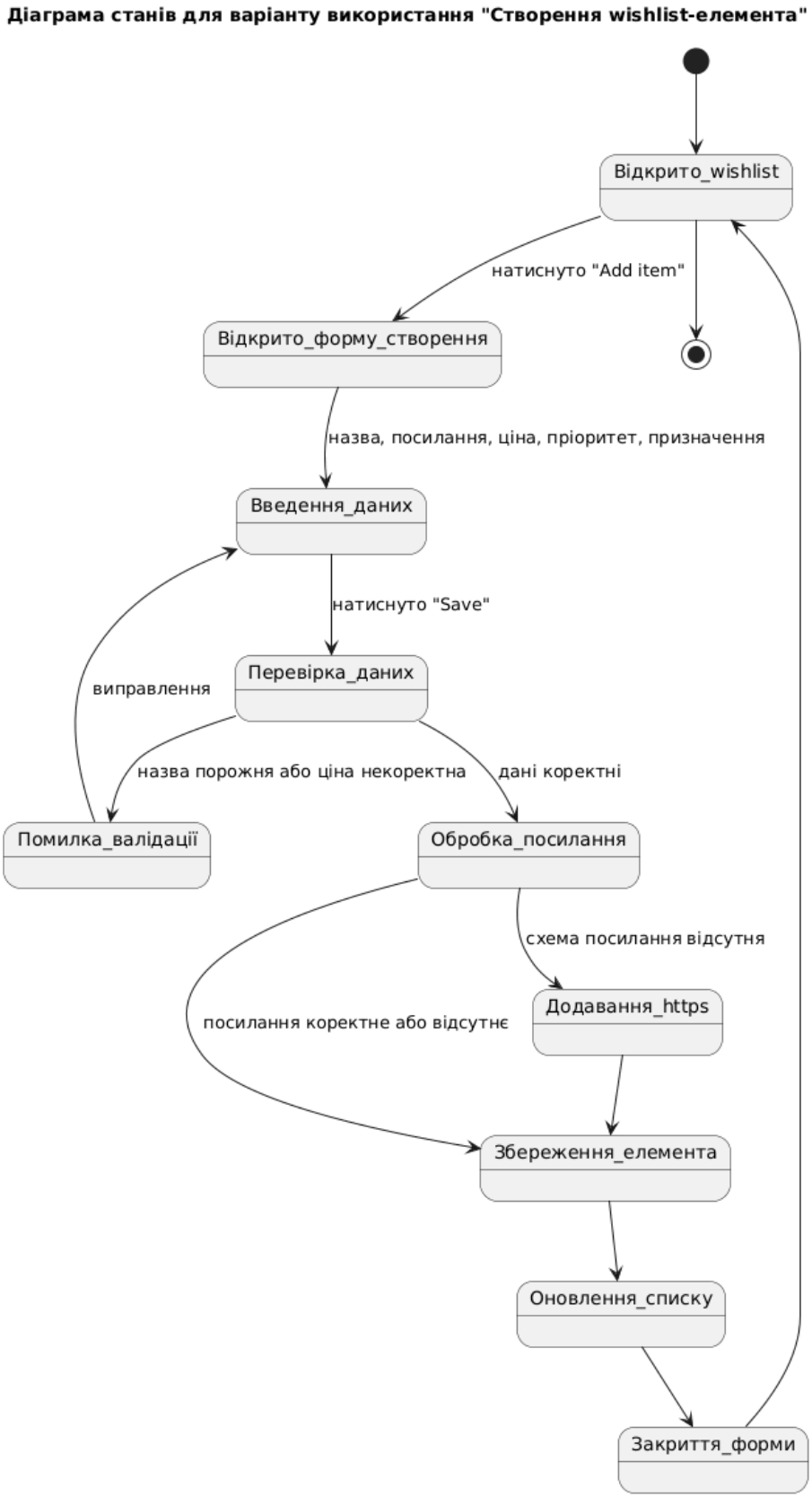


Рисунок Б.11 - Діаграма станів для варіанту використання «Створення wishlist-елемента»

Варіант використання: покупка з wishlist зі створенням транзакції

Передумови:

- користувач авторизований;
- користувач належить до пари;
- у wishlist існує хоча б один елемент;
- елемент ще не позначено як придбаний.

Тригер: користувач натискає кнопку позначення wishlist-елемента як придбаного.

Основний потік:

- користувач відкриває wishlist;
- користувач обирає потрібний елемент;
- користувач натискає кнопку позначення покупки як придбаної;
- система перевіряє, чи має елемент зазначену ціну;
- якщо ціна вказана, система пропонує створити фінансову транзакцію;
- користувач обирає категорію;
- застосунок надсилає запит на сервер;
- сервер позначає wishlist-елемент як придбаний;
- сервер створює фінансову транзакцію;
- залежно від призначення транзакція може бути підтвердженою або очікувати погодження партнера;
- застосунок оновлює wishlist і фінансову історію.

Альтернативні потоки:

- користувач може позначити покупку як придбану без створення транзакції;
- користувач може скасувати дію до підтвердження.

Заборонні потоки:

- уже придбаний wishlist-елемент не може бути повторно придбаний;
- транзакція не створюється без вибору категорії, якщо користувач погодився її створити;
- користувач поза парою не може змінювати wishlist-елемент.

Гарантії:

- мінімальна гарантія: у разі помилки статус wishlist-елемента не змінюється;
- гарантія успіху: wishlist-елемент позначено як придбаний, а за потреби створено фінансову транзакцію.

Діаграму станів для варіанту використання «покупка з wishlist зі створенням транзакції» зображено на рисунку Б.12.

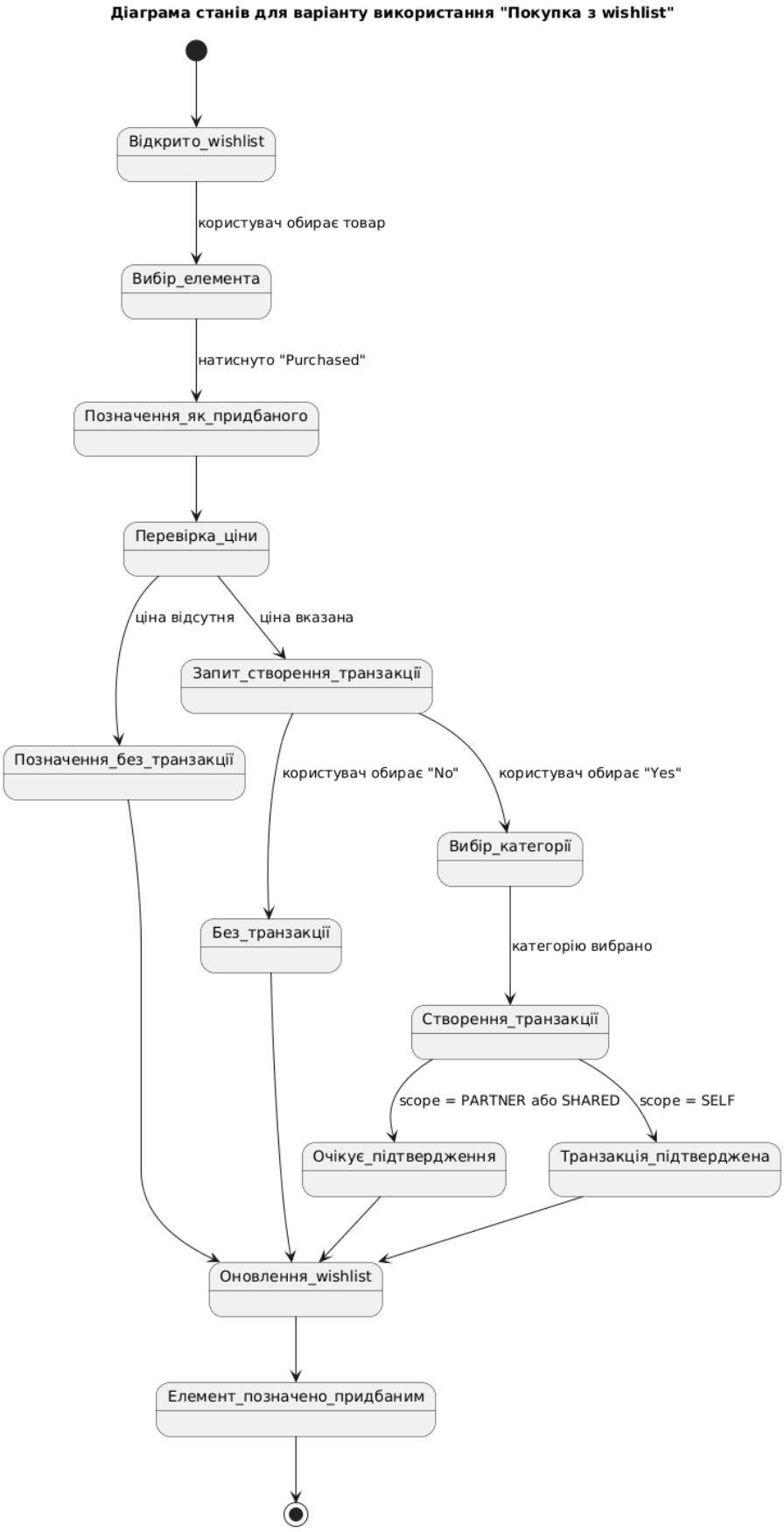


Рисунок Б.12 - Діаграма станів для варіанту використання «Покупка з wishlist»

Варіант використання: покупка винагороди у внутрішньому магазині

Передумови:

- користувач авторизований;
- користувач належить до пари;
- у магазині є активні винагороди;
- користувач має достатню кількість балів або виконує умови доступу до винагороди.

Тригер: користувач натискає кнопку покупки винагороди.

Основний потік:

- система відображає список винагород;
- користувач обирає винагороду;
- система перевіряє активність винагороди;
- система перевіряє вимоги до winStreak;
- система перевіряє кількість балів користувача;
- користувач підтверджує покупку;
- застосунок надсилає запит на сервер;
- сервер списує бали з користувача;
- сервер створює запис про покупку винагороди;
- сервер створює повідомлення для партнера;
- застосунок оновлює кількість балів і список покупок.

Альтернативні потоки:

- користувач може скасувати покупку;
- винагорода може бути заблокована через недостатній streak;
- користувач може повернутися до профілю або магазину без покупки.
- Заборонні потоки:
- покупка не виконується, якщо користувачу не вистачає балів;
- неактивну або заблоковану винагороду не можна придбати;
- у разі помилки сервера бали не списуються.

Гарантії:

- мінімальна гарантія: якщо покупка не завершена, бали користувача не змінюються;
- гарантія успіху: винагороду придбано, бали списано, покупку збережено, партнер отримує повідомлення.

Діаграму станів для варіанту використання «покупка винагороди у внутрішньому магазині» зображено на рисунку Б.13.

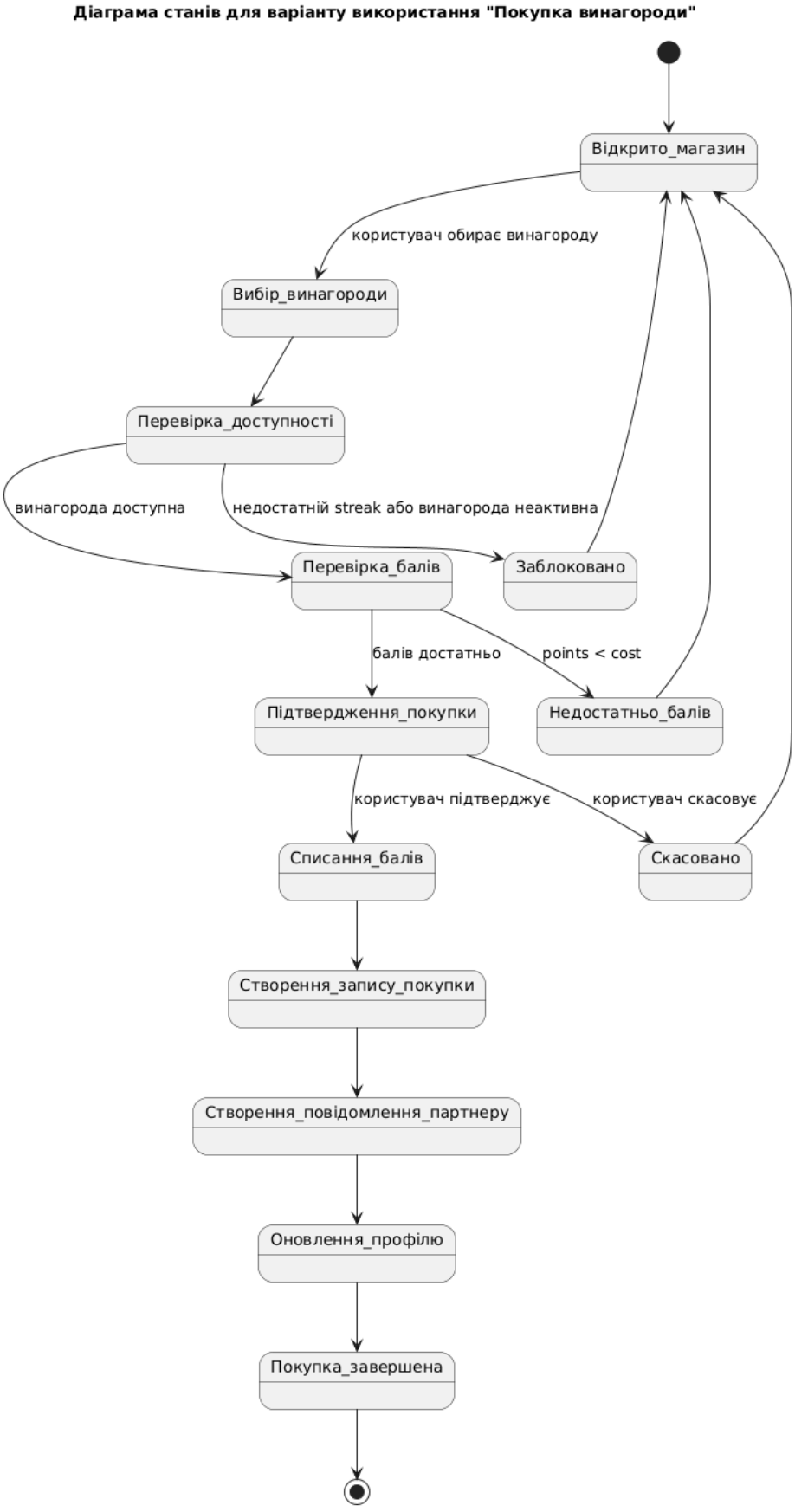


Рисунок Б.13 - Діаграма станів для варіанту використання «Покупка винагороди»

Варіант використання: зміна налаштувань застосунку

Передумови:

- користувач авторизований;
- користувач перебуває в основній частині застосунку;
- локальне сховище налаштувань доступне.

Тригер: користувач відкриває екран налаштувань із профілю.

Основний потік:

- користувач відкриває профіль;
- користувач натискає кнопку налаштувань;
- система відкриває екран налаштувань;
- користувач змінює тему оформлення;
- система зберігає вибрану тему локально;
- інтерфейс оновлюється відповідно до вибраної теми;
- користувач змінює мову застосунку;
- система зберігає вибрану мову локально;
- система повідомляє, що для повного застосування мови може знадобитися перезапуск;
- користувач змінює валюту;
- система зберігає вибрану валюту;
- фінансові значення відображаються з новим символом валюти.

Альтернативні потоки:

- користувач може змінити лише один параметр;
- користувач може вийти з налаштувань без змін;
- користувач може виконати вихід з акаунту через екран налаштувань.

Заборонні потоки:

- якщо локальне сховище недоступне, налаштування не зберігаються;
- некоректне значення теми, мови або валюти не приймається системою;
- у разі виходу з акаунту користувач втрачає доступ до основної частини застосунку до повторної авторизації.

Гарантії:

- мінімальна гарантія: у разі помилки попередні налаштування залишаються активними;
- гарантія успіху: вибрані параметри збережено, а застосунок використовує їх під час подальшої роботи.

Діаграму станів для варіанту використання «зміна налаштувань застосунку» зображено на рисунку Б.14.

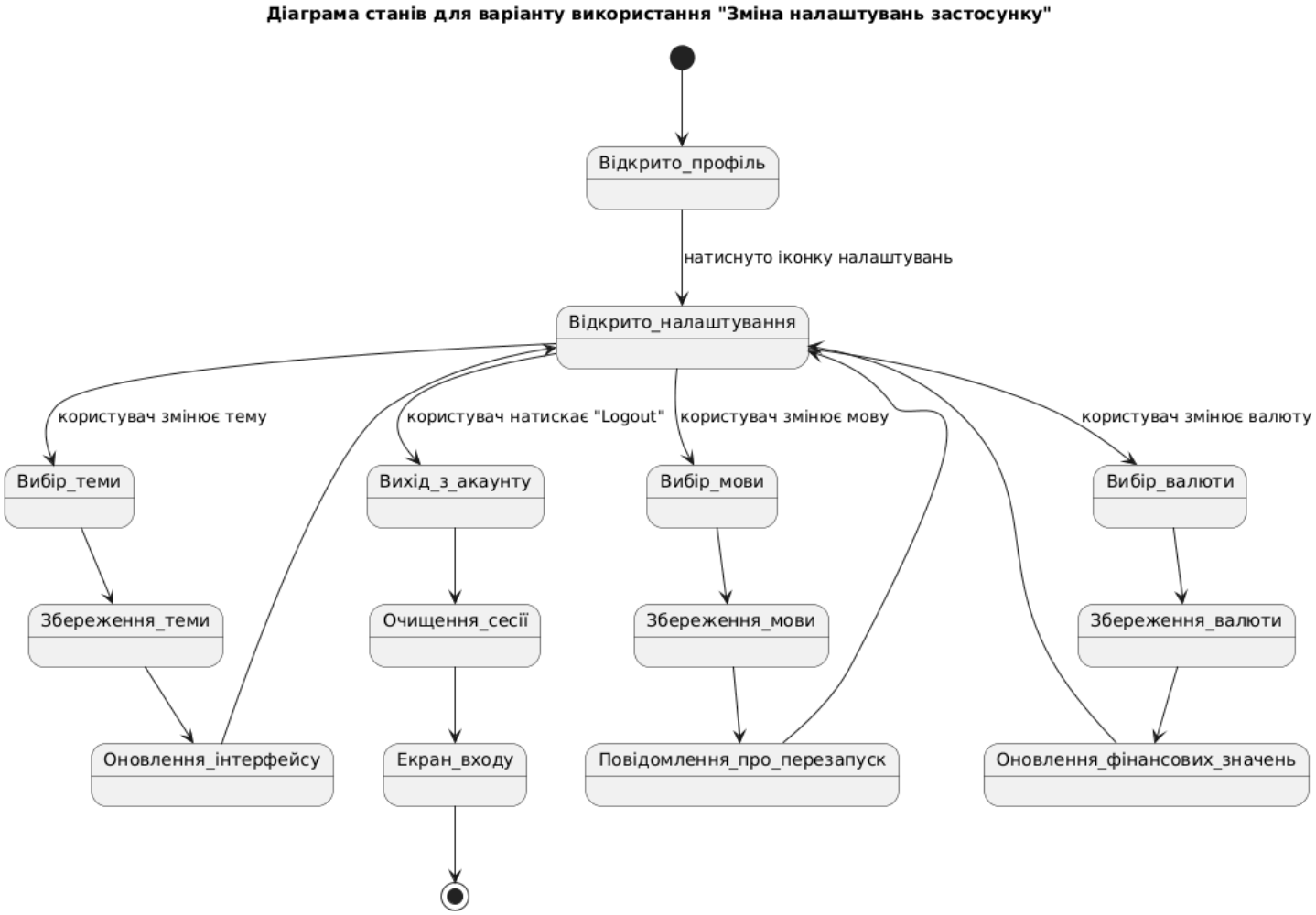


Рисунок Б.14 - Діаграма станів для варіанту використання «Зміна налаштувань застосунку»

ДОДАТОК В
(обов'язковий)

ЛІСТИНГ КОДУ ПРОГРАМИ

В1. Код автоматизованих тестів.

```
p
jest.mock("../src/config/prisma", () => {
  const { prismaMock } = require("./helpers/prisma-mock");
  return { prisma: prismaMock };
});

jest.mock("../src/utils/weekly-points", () => ({
  grantWeeklyPointsIfNeeded: jest.fn().mockResolvedValue(undefined),
  recordPointEarning: jest.fn().mockResolvedValue(undefined),
}));

import request from "supertest";
import bcrypt from "bcrypt";
import { app } from "../src/app";
import { prismaMock, resetPrismaMock } from "./helpers/prisma-mock";

describe("Auth routes", () => {
  beforeEach(() => {
    resetPrismaMock();
  });

  it("registers a user and returns JWT + sanitized user", async () => {

prismaMock.user.findUnique.mockResolvedValue(null);

prismaMock.user.create.mockResolvedValue({
      id: "user-1",
      email: "test@example.com",
      pairId: null,
      points: 100,
      nickname: null,
      avatarKey: null,
      avatarUrl: null,
      winStreak: 0,
    });

    const response = await request(app).post("/api/auth/register").send({
      email: "Test@Example.com",
      password: "secret123",
    });

    expect(response.status).toBe(201);

expect(response.body.token).toEqual(expect.any(String));
    expect(response.body.user).toMatchObject({
      id: "user-1",
      email: "test@example.com",
      pairId: null,
      points: 100,
    });

expect(prismaMock.user.create).toHaveBeenCalledWith(
      expect.objectContaining({
        data: expect.objectContaining({
          email: "test@example.com",
          password: expect.any(String),
        }),
      })
    );
  });

  it("logs in an existing user", async () => {
    const passwordHash = await bcrypt.hash("secret123", 10);
```

```
prismaMock.user.findUnique.mockResolvedValue({
        id: "user-1",
        email: "test@example.com",
        password: passwordHash,
        pairId: "pair-1",
        points: 140,
        nickname: "Tester",
        avatarKey: "cat",
        avatarUrl: null,
        winStreak: 2,
      });

      const response = await request(app).post("/api/auth/login").send({
        email: "test@example.com",
        password: "secret123",
      });

      expect(response.status).toBe(200);

      expect(response.body.token).toEqual(expect.any(String));
      expect(response.body.user).toMatchObject({
        id: "user-1",
        email: "test@example.com",
        pairId: "pair-1",
        points: 140,
        nickname: "Tester",
        avatarKey: "cat",
        winStreak: 2,
      });
    });

    it("rejects invalid credentials", async () => {
      const passwordHash = await bcrypt.hash("secret123", 10);

      prismaMock.user.findUnique.mockResolvedValue({
        id: "user-1",
        email: "test@example.com",
        password: passwordHash,
        pairId: null,
        points: 100,
        nickname: null,
        avatarKey: null,
        avatarUrl: null,
        winStreak: 0,
      });

      const response = await request(app).post("/api/auth/login").send({
        email: "test@example.com",
        password: "wrong-password",
      });

      expect(response.status).toBe(400);
      expect(response.body).toEqual({ message: "Invalid credentials" });
    });

    it("rejects missing token on protected route", async () => {
      const response = await request(app).post("/api/pair/create").send({});

      expect(response.status).toBe(401);
      expect(response.body).toEqual({ message: "Authorization token is missing" });
    });

    it("rejects invalid token on protected route", async () => {
      const response = await request(app)
        .post("/api/pair/create")
        .set("Authorization", "Bearer invalid-token")
        .send({});

      expect(response.status).toBe(401);
      expect(response.body).toEqual({ message: "Invalid or expired token" });
    });
```

```
});
jest.mock("../src/config/prisma", () => {
  const { prismaMock } = require("./helpers/prisma-mock");
  return { prisma: prismaMock };
});

jest.mock("../src/utils/weekly-points", () => ({
  grantWeeklyPointsIfNeeded: jest.fn().mockResolvedValue(undefined),
  recordPointEarning: jest.fn().mockResolvedValue(undefined),
}));

import request from "supertest";
import {
  TransactionCategory,
  TransactionScope,
  TransactionStatus,
  TransactionType,
} from "@prisma/client";
import { app } from "../src/app";
import { generateToken } from "../src/utils/jwt";
import { prismaMock, resetPrismaMock } from "./helpers/prisma-mock";

type TestUser = {
  id: string;
  email: string;
  pairId: string;
};

type TestTransaction = {
  id: string;
  title: string;
  amount: number;
  type: TransactionType;
  category: TransactionScope;
  transactionCategory: TransactionCategory;
  status: TransactionStatus;
  createdById: string;
  confirmedById: string | null;
  rejectedById: string | null;
  pairId: string;
  createdAt: Date;
};

const createFinanceHarness = () => {
  const users: TestUser[] = [
    { id: "user-1", email: "one@example.com", pairId: "pair-1" },
    { id: "user-2", email: "two@example.com", pairId: "pair-1" },
  ];
  const transactions: TestTransaction[] = [];
  let transactionCounter = 1;

  const includeTransaction = (transaction: TestTransaction) => ({
    ...transaction,
    createdBy: users.find((user) => user.id === transaction.createdById)!,
    confirmedBy: transaction.confirmedById
      ? users.find((user) => user.id === transaction.confirmedById)!
      : null,
    rejectedBy: transaction.rejectedById
      ? users.find((user) => user.id === transaction.rejectedById)!
      : null,
  });


prismaMock.user.findUnique.mockImplementation(async ({ where, select }: any) => {
    const user = users.find((entry) =>
      where.id ? entry.id === where.id : entry.email === where.email
    );
    if (!user) {
      return null;
    }
    if (!select) {
      return { ...user };
```

```
        }
        return
Object.fromEntries(Object.keys(select).map((key)    =>
[key, (user as any)[key]]));
      });


prismaMock.transaction.create.mockImplementation(as
ync ({ data, include }: any) => {
        const transaction: TestTransaction = {
         id: `transaction-${transactionCounter++}`,
         title: data.title,
         amount: data.amount,
         type: data.type,
         category: data.category,
         transactionCategory:
data.transactionCategory,
         status: data.status,
         createdById: data.createdById,
         confirmedById:   data.confirmedById   ??
null,
         rejectedById: data.rejectedById ?? null,
         pairId: data.pairId,
         createdAt:  new  Date(Date.UTC(2026,  4,
transactionCounter)),
        };
        transactions.push(transaction);
        return                 include                 ?
includeTransaction(transaction) : { ...transaction };
      });


prismaMock.transaction.findUnique.mockImplementati
on(async ({ where, include }: any) => {
        const transaction = transactions.find((entry)
=> entry.id === where.id);
        if (!transaction) {
         return null;
        }
        return                 include                 ?
includeTransaction(transaction) : { ...transaction };
      });

prismaMock.transaction.findMany.mockImplementation
(async ({ where, select, include, orderBy }: any) => {
        let result = transactions.filter((transaction) =>
transaction.pairId === where.pairId);
        if (orderBy?.createdAt === "desc") {
         result  =  result.sort((left,  right)  =>
right.createdAt.getTime() - left.createdAt.getTime());
        }
        if (include) {
         return result.map(includeTransaction);
        }
        if (select) {
         return result.map((transaction) =>

Object.fromEntries(Object.keys(select).map((key)    =>
[key, (transaction as any)[key]]))
         );
        }
        return       result.map((transaction)       =>
({ ...transaction }));
      });


prismaMock.transaction.update.mockImplementation(as
ync ({ where, data, include }: any) => {
        const  transaction  =  transactions.find((entry)
=> entry.id === where.id)!;
        Object.assign(transaction, data);
        return                 include                 ?
includeTransaction(transaction) : { ...transaction };
      });


prismaMock.transaction.delete.mockImplementation(as
ync ({ where }: any) => {
        const  index  =  transactions.findIndex((entry)
=> entry.id === where.id);
        const  [deleted]  =  transactions.splice(index,
1);
        return deleted;
```

```
  });

  return { transactions };
};

describe("Finance routes", () => {
  beforeEach(() => {
    resetPrismaMock();
  });

  it("creates SELF transaction as confirmed immediately", async () => {
    const harness = createFinanceHarness();

    const response = await request(app)
      .post("/api/transactions/create")
      .set("Authorization", `Bearer ${generateToken("user-1")}`)
      .send({
        title: "Salary",
        amount: 1000,
        type: "INCOME",
        scope: "SELF",
        transactionCategory: "SALARY",
      });

    expect(response.status).toBe(201);

    expect(response.body.transaction.status).toBe("CONFIRMED");

    expect(harness.transactions).toHaveLength(1);
  });

  it("creates SHARED transaction as pending and excludes it from summary until confirmed", async () => {
    createFinanceHarness();

    const createResponse = await request(app)
      .post("/api/transactions/create")
      .set("Authorization", `Bearer ${generateToken("user-1")}`)
      .send({
        title: "Groceries",
        amount: 120,
        type: "EXPENSE",
        scope: "SHARED",
        transactionCategory: "FOOD",
      });

    expect(createResponse.status).toBe(201);

    expect(createResponse.body.transaction.status).toBe("PENDING_CONFIRMATION");

    const pendingSummary = await request(app)
      .get("/api/transactions/summary")
      .set("Authorization", `Bearer ${generateToken("user-1")}`);

    expect(pendingSummary.status).toBe(200);

    expect(pendingSummary.body.totalBudget).toBe(0);

    expect(pendingSummary.body.balance.direction).toBe("SETTLED");

    const confirmResponse = await request(app)
      .post(`/api/transactions/confirm/${createResponse.body.transaction.id}`)
      .set("Authorization", `Bearer ${generateToken("user-2")}`)
      .send({});

    expect(confirmResponse.status).toBe(200);

    expect(confirmResponse.body.transaction.status).toBe("CONFIRMED");

    const confirmedSummary = await request(app)
      .get("/api/transactions/summary")
```

```
            .set("Authorization", `Bearer ${generateToken("user-1")}`);


expect(confirmedSummary.body.totalBudget).toBe(-120);

expect(confirmedSummary.body.balance).toEqual({
              amount: 60,
              direction: "PARTNER_OWES",
            });
          });

          it("rejects pending transaction through partner action", async () => {
            createFinanceHarness();

            const createResponse = await request(app)
              .post("/api/transactions/create")
              .set("Authorization", `Bearer ${generateToken("user-1")}`)
              .send({
                title: "Taxi",
                amount: 80,
                type: "EXPENSE",
                scope: "PARTNER",
                transactionCategory: "TRANSPORT",
              });

            const rejectResponse = await request(app)
              .post(`/api/transactions/reject/${createResponse.body.transaction.id}`)
              .set("Authorization", `Bearer ${generateToken("user-2")}`)
              .send({});

            expect(rejectResponse.status).toBe(200);

expect(rejectResponse.body.transaction.status).toBe("REJECTED");

            const summaryResponse = await request(app)
              .get("/api/transactions/summary")
              .set("Authorization", `Bearer ${generateToken("user-1")}`);


expect(summaryResponse.body.totalBudget).toBe(0);

expect(summaryResponse.body.balance.direction).toBe("SETTLED");
          });
        });
```

B2. Код автентифікації та призначення jwt-токенів.

```
/**
 * Documentation sample:
 * Authentication controller, JWT utility, and auth middleware.
 * This file is a copy-only code sample for thesis documentation.
 */

import { Request, Response, NextFunction } from "express";
import bcrypt from "bcrypt";
import jwt, { JwtPayload } from "jsonwebtoken";
import { prisma } from "../../couple-app-backend/src/config/prisma";
import { AuthenticatedRequest } from "../../couple-app-backend/src/types/auth-request";
import { grantWeeklyPointsIfNeeded } from "../../couple-app-backend/src/utils/weekly-points";

export type AuthTokenPayload = JwtPayload & {
  userId: string;
};
```

```
const getJwtSecret = () => {
  const secret = process.env.JWT_SECRET;

  if (!secret) {
    throw new Error("JWT_SECRET is not configured");
  }

  return secret;
};

export const generateToken = (userId: string) => {
  return jwt.sign({ userId }, getJwtSecret(), {
    expiresIn: "7d",
  });
};

export const verifyToken = (token: string) => {
  return jwt.verify(token, getJwtSecret()) as AuthTokenPayload;
};

const sanitizeUser = (user: {
  id: string;
  email: string;
  pairId: string | null;
  points: number;
  nickname: string | null;
  avatarKey: string | null;
  avatarUrl: string | null;
  winStreak: number;
}) => ({
  id: user.id,
  email: user.email,
  pairId: user.pairId,
  points: user.points,
  nickname: user.nickname,
  avatarKey: user.avatarKey,
  avatarUrl: user.avatarUrl,
  winStreak: user.winStreak,
});

export const register = async (req: Request, res: Response) => {
  try {
    const email = String(req.body.email ?? "").trim().toLowerCase();
    const password = String(req.body.password ?? "");

    if (!email || !password) {
      return res.status(400).json({ message: "Email and password are required" });
    }

    const existing = await prisma.user.findUnique({ where: { email } });

    if (existing) {
      return res.status(400).json({ message: "User already exists" });
    }

    const hashedPassword = await bcrypt.hash(password, 10);

    const user = await prisma.user.create({
      data: {
        email,
        password: hashedPassword,
      },
      select: {
        id: true,
        email: true,
        pairId: true,
        points: true,
        nickname: true,
        avatarKey: true,
        avatarUrl: true,
        winStreak: true,
      },
    });
```

```
    const token = generateToken(user.id);

    return res.status(201).json({ token, user: sanitizeUser(user) });
  } catch (error) {
    console.error("Register error:", error);
    return res.status(500).json({ message: "Internal server error" });
  }
};

export const login = async (req: Request, res: Response) => {
  try {
    const email = String(req.body.email ?? "").trim().toLowerCase();
    const password = String(req.body.password ?? "");

    if (!email || !password) {
      return res.status(400).json({ message: "Email and password are required" });
    }

    const user = await prisma.user.findUnique({ where: { email } });

    if (!user) {
      return res.status(400).json({ message: "Invalid credentials" });
    }

    const isMatch = await bcrypt.compare(password, user.password);

    if (!isMatch) {
      return res.status(400).json({ message: "Invalid credentials" });
    }

    const token = generateToken(user.id);

    return res.json({
      token,
      user: sanitizeUser(user),
    });
  } catch (error) {
    console.error("Login error:", error);
    return res.status(500).json({ message: "Internal server error" });
  }
};

export const authenticate = async (req: AuthenticatedRequest, res: Response, next: NextFunction) => {
  const authorization = req.headers.authorization;

  if (!authorization?.startsWith("Bearer ")) {
    return res.status(401).json({ message: "Authorization token is missing" });
  }

  const token = authorization.replace("Bearer ", "").trim();

  try {
    const payload = verifyToken(token);
    req.userId = payload.userId;

    try {
      await grantWeeklyPointsIfNeeded(payload.userId);
    } catch (error) {
      console.error("Weekly points grant error:", error);
    }

    return next();
  } catch (error) {
    console.error("Auth middleware error:", error);
```

```
        return res.status(401).json({ message: "Invalid or expired token" });
      }
    };
```

В3. Код системи парування користувачів.

```
import { Router } from "express";
import { Response } from "express";
import { prisma } from "../../couple-app-backend/src/config/prisma";
import { AuthenticatedRequest } from "../../couple-app-backend/src/types/auth-request";
import { generateUniqueJoinCode } from "../../couple-app-backend/src/utils/pair";
import { authenticate } from "../../couple-app-backend/src/middleware/auth.middleware";

const normalizeJoinCode = (value: string) => value.trim().toUpperCase();

export const createPair = async (req: AuthenticatedRequest, res: Response) => {
  try {
    const userId = req.userId;

    if (!userId) {
      return res.status(401).json({ message: "Unauthorized" });
    }

    const pair = await prisma.$transaction(async (tx) => {
      const user = await tx.user.findUnique({
        where: { id: userId },
        select: {
          id: true,
          pairId: true,
        },
      });

      if (!user) {
        return null;
      }

      if (user.pairId) {
        throw new Error("USER_ALREADY_IN_PAIR");
      }

      const joinCode = await generateUniqueJoinCode(async (candidate) => {
        const existingPair = await tx.pair.findUnique({
          where: { joinCode: candidate },
          select: { id: true },
        });

        return Boolean(existingPair);
      });

      const createdPair = await tx.pair.create({
        data: { joinCode },
      });

      await tx.user.update({
        where: { id: userId },
        data: { pairId: createdPair.id },
      });

      return createdPair;
    });

    if (!pair) {
      return res.status(404).json({ message: "User not found" });
    }

    return res.status(201).json({
      pairId: pair.id,
      joinCode: pair.joinCode,
```

```
    });
  } catch (error) {
    if (error instanceof Error && error.message
=== "USER_ALREADY_IN_PAIR") {
      return res.status(400).json({ message: "User
is already connected to a pair" });
    }

    console.error("Create pair error:", error);
    return res.status(500).json({ message:
"Internal server error" });
  }
};

export const joinPair = async (req:
AuthenticatedRequest, res: Response) => {
  try {
    const userId = req.userId;
    const joinCode =
normalizeJoinCode(String(req.body.joinCode ?? ""));

    if (!userId) {
      return res.status(401).json({ message:
"Unauthorized" });
    }

    if (!joinCode) {
      return res.status(400).json({ message: "Join
code is required" });
    }

    const pair = await prisma.$transaction(async
(tx) => {
      const user = await tx.user.findUnique({
        where: { id: userId },
        select: {
          id: true,
          pairId: true,
        },
      });

      if (!user) {
        return null;
      }

      if (user.pairId) {
        throw new
Error("USER_ALREADY_IN_PAIR");
      }

      const existingPair = await
tx.pair.findUnique({
        where: { joinCode },
        select: {
          id: true,
          joinCode: true,
        },
      });

      if (!existingPair) {
        throw new Error("PAIR_NOT_FOUND");
      }

      const membersCount = await tx.user.count({
        where: { pairId: existingPair.id },
      });

      if (membersCount >= 2) {
        throw new Error("PAIR_IS_FULL");
      }

      await tx.user.update({
        where: { id: userId },
        data: { pairId: existingPair.id },
      });

      return existingPair;
    });

    if (!pair) {
      return res.status(404).json({ message: "User
not found" });
    }
```

```
            return res.json({
              pairId: pair.id,
              joinCode: pair.joinCode,
            });
          } catch (error) {
            if (error instanceof Error) {
              if (error.message === "USER_ALREADY_IN_PAIR") {
                return res.status(400).json({ message: "User is already connected to a pair" });
              }

              if (error.message === "PAIR_NOT_FOUND") {
                return res.status(404).json({ message: "Pair with this join code was not found" });
              }

              if (error.message === "PAIR_IS_FULL") {
                return res.status(400).json({ message: "This pair already has two users" });
              }
            }

            console.error("Join pair error:", error);
            return res.status(500).json({ message: "Internal server error" });
          }
        };

        export const leavePair = async (req: AuthenticatedRequest, res: Response) => {
          try {
            const userId = req.userId;

            if (!userId) {
              return res.status(401).json({ message: "Unauthorized" });
            }

            const result = await prisma.$transaction(async (tx) => {
              const user = await tx.user.findUnique({
                where: { id: userId },
                select: {
                  id: true,
                  pairId: true,
                },
              });

              if (!user) {
                return null;
              }

              if (!user.pairId) {
                throw new Error("PAIR_REQUIRED");
              }

              const pairId = user.pairId;

              await tx.user.update({
                where: { id: userId },
                data: {
                  pairId: null,
                },
              });

              const remainingMembers = await tx.user.count({
                where: { pairId },
              });

              if (remainingMembers === 0) {
                await tx.pair.delete({
                  where: { id: pairId },
                });
              }

              return { pairId };
            });

            if (!result) {
              return res.status(404).json({ message: "User not found" });
```

```
    }

    return res.json({
     leftPair: true,
    });
   } catch (error) {
    if (error instanceof Error && error.message
=== "PAIR_REQUIRED") {
     return res.status(400).json({ message: "You
are not connected to a pair" });
    }

    console.error("Leave pair error:", error);
    return res.status(500).json({ message:
"Internal server error" });
   }
  };

  const router = Router();

  router.post("/create", authenticate, createPair);
  router.post("/join", authenticate, joinPair);
  router.post("/leave", authenticate, leavePair);

  export default router;
```

В4. Код гейміфікованої системи завдань.

```
  import {
   PointEarningSource,
   Prisma,
   SharedSplitStatus,
   TaskRecurrenceType,
   TaskStatus,
   TaskType,
  } from "../../node_modules/.prisma/client";
  import { Response } from "express";
  import { prisma } from "../../couple-app-
backend/src/config/prisma";
  import { AuthenticatedRequest } from
"../../couple-app-backend/src/types/auth-request";
  import { parseOptionalDate } from "../../couple-
app-backend/src/utils/date";
  import { recordPointEarning } from
"../../couple-app-backend/src/utils/weekly-points";

  export const taskDetailsInclude = {
   createdBy: { select: { id: true, email: true } },
   assignedTo: { select: { id: true, email: true } },
   completionRequestedBy: { select: { id: true,
email: true } },
   proposedBy: { select: { id: true, email: true } },
  } satisfies Prisma.TaskInclude;

  const normalizeTitle = (value: string) =>
value.trim();

  const taskContextSelect = {
   id: true,
   title: true,
   bank: true,
   type: true,
   status: true,
   sharedSplitStatus: true,
   pairId: true,
   assignedToId: true,
   createdById: true,
   completionRequestedById: true,
   proposedById: true,
   proposedUser1Points: true,
   proposedUser2Points: true,
   dueDate: true,
   recurrenceType: true,
   recurrenceInterval: true,
   recurrenceParentId: true,
  } satisfies Prisma.TaskSelect;

  type PairUser = {
   id: string;
```

```
  email: string;
  nickname: string | null;
  avatarKey: string | null;
  points: number;
};

type TaskCreationInput = {
  userId: string;
  title: string;
  taskType: TaskType;
  points: number;
  dueDate: Date | null;
  recurrenceType: TaskRecurrenceType;
  recurrenceInterval: number | null;
};

type TaskContext = Prisma.TaskGetPayload<{ select: typeof taskContextSelect }>;

const getUserContext = async (tx: Prisma.TransactionClient, userId: string) => {
  const user = await tx.user.findUnique({
    where: { id: userId },
    select: { id: true, pairId: true, points: true },
  });

  if (!user) {
    throw new Error("USER_NOT_FOUND");
  }
  if (!user.pairId) {
    throw new Error("PAIR_REQUIRED");
  }

  return user;
};

const getPairUsers = async (tx: Prisma.TransactionClient, pairId: string) => {
  const pairUsers = await tx.user.findMany({
    where: { pairId },
    select: {
      id: true,
      email: true,
      nickname: true,
      avatarKey: true,
      points: true,
    },
  });

  if (pairUsers.length !== 2) {
    throw new Error("PARTNER_REQUIRED");
  }

  return pairUsers.sort((left, right) => left.id.localeCompare(right.id));
};

const getPartner = (pairUsers: PairUser[], userId: string) => {
  const partner = pairUsers.find((pairUser) => pairUser.id !== userId);
  if (!partner) {
    throw new Error("PARTNER_REQUIRED");
  }
  return partner;
};

const ensureChallengeTask = (task: TaskContext) => {
  if (task.type !== TaskType.CHALLENGE) {
    throw new Error("CHALLENGE_TASK_REQUIRED");
  }
};

const ensureSharedTask = (task: TaskContext) => {
  if (task.type !== TaskType.SHARED) {
    throw new Error("SHARED_TASK_REQUIRED");
  }
```

```
        };

        const getGenerationHorizon = () => {
          const now = new Date();
          return                                    new
Date(Date.UTC(now.getUTCFullYear(),
now.getUTCMonth() + 2, 0, 23, 59, 59, 999));
        };

        const addRecurrence = (date: Date, type:
TaskRecurrenceType, interval: number | null) => {
          const next = new Date(date);

          switch (type) {
            case
TaskRecurrenceType.EVERY_X_DAYS:
              next.setUTCDate(next.getUTCDate()      +
(interval ?? 1));
              return next;
            case TaskRecurrenceType.WEEKLY:
              next.setUTCDate(next.getUTCDate() + 7);
              return next;
            case TaskRecurrenceType.MONTHLY:
              next.setUTCMonth(next.getUTCMonth()  +
1);
              return next;
            default:
              return next;
          }
        };

        const createTaskInstance = async (
          tx: Prisma.TransactionClient,
          input: {
            title: string;
            bank: number;
            type: TaskType;
            dueDate: Date | null;
            pairId: string;
            createdById: string;
            assignedToId: string | null;
            recurrenceType: TaskRecurrenceType;
            recurrenceInterval: number | null;
            recurrenceParentId: string | null;
          }
        ) => {
          return tx.task.create({
            data: {
              title: input.title,
              bank: input.bank,
              type: input.type,
              status: TaskStatus.ACTIVE,
              sharedSplitStatus:
SharedSplitStatus.NONE,
              assignedToId:      input.assignedToId      ??
undefined,
              createdById: input.createdById,
              completionRequestedById: null,
              proposedById: null,
              proposedUser1Points: null,
              proposedUser2Points: null,
              pairId: input.pairId,
              dueDate: input.dueDate,
              recurrenceType: input.recurrenceType,
              recurrenceInterval:
input.recurrenceInterval,
              recurrenceParentId:
input.recurrenceParentId ?? undefined,
            },
            include: taskDetailsInclude,
          });
        };

        const generateFutureRecurringTasks = async (
          tx: Prisma.TransactionClient,
          options: {
            rootTaskId: string;
            title: string;
            bank: number;
            type: TaskType;
            pairId: string;
            createdById: string;
            assignedToId: string | null;
            dueDate: Date;
```

```
    recurrenceType: TaskRecurrenceType;
    recurrenceInterval: number | null;
   }
  ) => {
   if (options.recurrenceType === TaskRecurrenceType.NONE) {
    return;
   }

   const horizon = getGenerationHorizon();
   const relatedTasks = await tx.task.findMany({
    where: {
     OR: [{ id: options.rootTaskId }, { recurrenceParentId: options.rootTaskId }],
    },
    select: { dueDate: true },
    orderBy: { dueDate: "asc" },
   });

   const datedTasks = relatedTasks
    .map((task) => task.dueDate)
    .filter((date): date is Date => date instanceof Date)
    .sort((left, right) => left.getTime() - right.getTime());

   let lastDate = datedTasks.length > 0 ? datedTasks[datedTasks.length - 1] : options.dueDate;

   while (true) {
    const nextDate = addRecurrence(lastDate, options.recurrenceType, options.recurrenceInterval);
    if (nextDate.getTime() > horizon.getTime()) {
     break;
    }

    if (options.type === TaskType.CHALLENGE) {
     const creator = await tx.user.findUnique({
      where: { id: options.createdById },
      select: { points: true },
     });

     if (!creator || creator.points < options.bank) {
      break;
     }

     await tx.user.update({
      where: { id: options.createdById },
      data: { points: { decrement: options.bank } },
     });
    }

    await createTaskInstance(tx, {
     title: options.title,
     bank: options.bank,
     type: options.type,
     dueDate: nextDate,
     pairId: options.pairId,
     createdById: options.createdById,
     assignedToId: options.assignedToId,
     recurrenceType: options.recurrenceType,
     recurrenceInterval: options.recurrenceInterval,
     recurrenceParentId: options.rootTaskId,
    });

    lastDate = nextDate;
   }
  };

  export const createTaskForUser = async (tx: Prisma.TransactionClient, input: TaskCreationInput) => {
   const { userId, title, taskType, points, dueDate, recurrenceType, recurrenceInterval } = input;
   const user = await getUserContext(tx, userId);
   const pairUsers = await getPairUsers(tx, user.pairId!);

   if (recurrenceType !== TaskRecurrenceType.NONE && !dueDate) {
```

```
            throw new Error("RECURRENCE_DATE_REQUIRED");
          }

          let assignedToId: string | null = null;
          let currentUserPoints = user.points;

          if (taskType === TaskType.CHALLENGE) {
            if (user.points < points) {
              throw new Error("INSUFFICIENT_POINTS");
            }

            const partner = getPartner(pairUsers, userId);
            assignedToId = partner.id;

            const updatedUser = await tx.user.update({
              where: { id: userId },
              data: { points: { decrement: points } },
              select: { points: true },
            });

            currentUserPoints = updatedUser.points;
          }

          const task = await createTaskInstance(tx, {
            title,
            bank: points,
            type: taskType,
            dueDate,
            pairId: user.pairId!,
            createdById: userId,
            assignedToId,
            recurrenceType,
            recurrenceInterval,
            recurrenceParentId: null,
          });

          if (recurrenceType !== TaskRecurrenceType.NONE && dueDate) {
            await generateFutureRecurringTasks(tx, {
              rootTaskId: task.id,
              title,
              bank: points,
              type: taskType,
              pairId: user.pairId!,
              createdById: userId,
              assignedToId,
              dueDate,
              recurrenceType,
              recurrenceInterval,
            });
          }

          return {
            task,
            currentUserPoints,
          };
        };

        export const createTask = async (req: AuthenticatedRequest, res: Response) => {
          try {
            const userId = req.userId;
            const title = normalizeTitle(String(req.body.title ?? ""));
            const taskType = String(req.body.type ?? TaskType.CHALLENGE).trim().toUpperCase() as TaskType;
            const points = Number(req.body.points);
            const dueDate = parseOptionalDate(req.body.dueDate);
            const recurrenceType = String(req.body.recurrenceType ?? "NONE").trim().toUpperCase() as TaskRecurrenceType;
            const recurrenceInterval =
              recurrenceType === TaskRecurrenceType.EVERY_X_DAYS ? Number(req.body.recurrenceInterval) : null;

            if (!userId) {
              return res.status(401).json({ message: "Unauthorized" });
            }
```

```
            if (!title) {
              return res.status(400).json({ message: "Task
title is required" });
            }

            if (!Number.isInteger(points) || points <= 0) {
              return res.status(400).json({ message:
"Points must be a positive integer" });
            }

            const result = await prisma.$transaction((tx)
=>
              createTaskForUser(tx, {
                userId,
                title,
                taskType,
                points,
                dueDate,
                recurrenceType,
                recurrenceInterval,
              })
            );

            return res.status(201).json(result);
          } catch (error) {
            console.error("Task create error:", error);
            return res.status(500).json({ message:
"Internal server error" });
          }
        };

        export const confirmCompletion = async (req:
AuthenticatedRequest, res: Response) => {
          try {
            const userId = req.userId!;
            const taskId = String(req.params.id ?? "");

            const result = await
prisma.$transaction(async (tx) => {
              const user = await getUserContext(tx,
userId);
              const task = await tx.task.findUnique({
                where: { id: taskId },
                select: taskContextSelect,
              });

              if (!task || task.pairId !== user.pairId) {
                throw new
Error("TASK_NOT_FOUND");
              }

              ensureChallengeTask(task);

              if (task.status !==
TaskStatus.WAITING_CONFIRMATION
|| !task.completionRequestedById) {
                throw new
Error("WAITING_CONFIRMATION_REQUIRED");
              }

              await tx.user.update({
                where: { id:
task.completionRequestedById },
                data: {
                  points: { increment: task.bank },
                  winStreak: { increment: 1 },
                },
              });

              await recordPointEarning(tx, {
                userId: task.completionRequestedById,
                pairId: user.pairId!,
                source:
PointEarningSource.CHALLENGE_REWARD,
                amount: task.bank,
              });

              const updatedTask = await tx.task.update({
                where: { id: task.id },
                data: {
                  status: TaskStatus.COMPLETED,
                  completionRequestedById: null,
                },
```

```
            include: taskDetailsInclude,
          });

          return { task: updatedTask };
        });

        return res.json(result);
      } catch (error) {
        console.error("Confirm completion error:", error);
        return res.status(500).json({ message: "Internal server error" });
      }
    };

    export const proposeSharedSplit = async (req: AuthenticatedRequest, res: Response) => {
      try {
        const userId = req.userId!;
        const taskId = String(req.params.id ?? "");
        const myPoints = Number(req.body.myPoints);
        const partnerPoints = Number(req.body.partnerPoints);

        const result = await prisma.$transaction(async (tx) => {
          const user = await getUserContext(tx, userId);
          const pairUsers = await getPairUsers(tx, user.pairId!);
          const task = await tx.task.findUnique({
            where: { id: taskId },
            select: taskContextSelect,
          });

          if (!task || task.pairId !== user.pairId) {
            throw new Error("TASK_NOT_FOUND");
          }

          ensureSharedTask(task);

          if (myPoints + partnerPoints !== task.bank) {
            throw new Error("INVALID_SHARED_SPLIT");
          }

          const partner = getPartner(pairUsers, userId);
          const pointsById = new Map<string, number>([
            [userId, myPoints],
            [partner.id, partnerPoints],
          ]);

          const updatedTask = await tx.task.update({
            where: { id: task.id },
            data: {
              status: TaskStatus.WAITING_CONFIRMATION,
              sharedSplitStatus: SharedSplitStatus.PROPOSED,
              proposedById: userId,
              proposedUser1Points: pointsById.get(pairUsers[0].id) ?? 0,
              proposedUser2Points: pointsById.get(pairUsers[1].id) ?? 0,
            },
            include: taskDetailsInclude,
          });

          return { task: updatedTask };
        });

        return res.json(result);
      } catch (error) {
        console.error("Shared split proposal error:", error);
        return res.status(500).json({ message: "Internal server error" });
      }
    };
```

В5. Код фінансової логіки.

```
import {
  Prisma,
  TransactionCategory,
  TransactionScope,
  TransactionStatus,
  TransactionType,
} from "../../node_modules/.prisma/client";
import { Response } from "express";
import { prisma } from "../../couple-app-backend/src/config/prisma";
import { AuthenticatedRequest } from "../../couple-app-backend/src/types/auth-request";

const EXPENSE_CATEGORIES: TransactionCategory[] = [
  TransactionCategory.FOOD,
  TransactionCategory.UTILITIES,
  TransactionCategory.TRANSPORT,
  TransactionCategory.HOME,
  TransactionCategory.ENTERTAINMENT,
  TransactionCategory.HEALTH,
  TransactionCategory.SHOPPING,
  TransactionCategory.SUBSCRIPTIONS,
  TransactionCategory.OTHER,
];

const INCOME_CATEGORIES: TransactionCategory[] = [
  TransactionCategory.SALARY,
  TransactionCategory.BONUS,
  TransactionCategory.GIFT,
  TransactionCategory.REFUND,
  TransactionCategory.SIDE_JOB,
  TransactionCategory.OTHER,
];

const transactionInclude = {
  createdBy: { select: { id: true, email: true } },
  confirmedBy: { select: { id: true, email: true } },
  rejectedBy: { select: { id: true, email: true } },
} satisfies Prisma.TransactionInclude;

const normalizeTitle = (value: string) => value.trim();

const parseTransactionType = (value: unknown) => {
  const normalized = String(value ?? "").trim().toUpperCase();
  if (normalized === TransactionType.EXPENSE || normalized === TransactionType.INCOME) {
    return normalized;
  }
  throw new Error("INVALID_TRANSACTION_TYPE");
};

const parseTransactionScope = (value: unknown) => {
  const normalized = String(value ?? "").trim().toUpperCase();
  if (
    normalized === TransactionScope.SELF ||
    normalized === TransactionScope.PARTNER ||
    normalized === TransactionScope.SHARED
  ) {
    return normalized;
  }
  throw new Error("INVALID_TRANSACTION_SCOPE");
};

const parseTransactionCategory = (value: unknown) => {
  const normalized = String(value ?? "").trim().toUpperCase();
```

```
        if (!normalized) {
          return TransactionCategory.OTHER;
        }

        if (
          normalized === TransactionCategory.FOOD
||
          normalized                                  ===
TransactionCategory.UTILITIES ||
          normalized                                  ===
TransactionCategory.TRANSPORT ||
          normalized === TransactionCategory.HOME
||
          normalized                                  ===
TransactionCategory.ENTERTAINMENT ||
          normalized                                  ===
TransactionCategory.HEALTH ||
          normalized                                  ===
TransactionCategory.SHOPPING ||
          normalized                                  ===
TransactionCategory.SUBSCRIPTIONS ||
          normalized                                  ===
TransactionCategory.SALARY ||
          normalized                                  ===
TransactionCategory.BONUS ||
          normalized === TransactionCategory.GIFT ||
          normalized                                  ===
TransactionCategory.REFUND ||
          normalized                                  ===
TransactionCategory.SIDE_JOB ||
          normalized                                  ===
TransactionCategory.OTHER
        ) {
          return normalized;
        }

        throw                                         new
Error("INVALID_TRANSACTION_CATEGORY");
      };

      const assertCategoryMatchesType = (type:
TransactionType, category: TransactionCategory) => {
        if (type === TransactionType.EXPENSE
&& !EXPENSE_CATEGORIES.includes(category)) {
          throw                                       new
Error("CATEGORY_TYPE_MISMATCH");
        }

        if (type === TransactionType.INCOME
&& !INCOME_CATEGORIES.includes(category)) {
          throw                                       new
Error("CATEGORY_TYPE_MISMATCH");
        }
      };

      const getInitialTransactionStatus = (scope:
TransactionScope) =>
        scope === TransactionScope.SELF
          ? TransactionStatus.CONFIRMED
          :
TransactionStatus.PENDING_CONFIRMATION;

      const isConfirmedTransaction = (status:
TransactionStatus) =>
        status === TransactionStatus.CONFIRMED;

      const getBalanceContribution = (
        transaction: {
          amount: number;
          type: TransactionType;
          category: TransactionScope;
          createdById: string;
          status: TransactionStatus;
        },
        currentUserId: string
      ) => {
        if
(!isConfirmedTransaction(transaction.status)) {
          return 0;
        }

        const isCreator = transaction.createdById ===
currentUserId;
```

```
            if (transaction.category === TransactionScope.SELF) {
              return 0;
            }
            if (transaction.type === TransactionType.EXPENSE && transaction.category === TransactionScope.PARTNER) {
              return isCreator ? transaction.amount : -transaction.amount;
            }
            if (transaction.type === TransactionType.EXPENSE && transaction.category === TransactionScope.SHARED) {
              return isCreator ? transaction.amount / 2 : -(transaction.amount / 2);
            }
            if (transaction.type === TransactionType.INCOME && transaction.category === TransactionScope.PARTNER) {
              return isCreator ? -transaction.amount : transaction.amount;
            }
            if (transaction.type === TransactionType.INCOME && transaction.category === TransactionScope.SHARED) {
              return isCreator ? -(transaction.amount / 2) : transaction.amount / 2;
            }

            return 0;
          };

          const buildCategorySummary = (
            transactions: Array<{
              amount: number;
              type: TransactionType;
              transactionCategory: TransactionCategory;
              status: TransactionStatus;
            }>,
            type: TransactionType
          ) => {
            const matching = transactions.filter(
              (transaction) => transaction.type === type && isConfirmedTransaction(transaction.status)
            );
            const total = matching.reduce((sum, transaction) => sum + transaction.amount, 0);
            const grouped = new Map<TransactionCategory, number>();

            matching.forEach((transaction) => {
              grouped.set(
                transaction.transactionCategory,
                (grouped.get(transaction.transactionCategory) ?? 0) + transaction.amount
              );
            });

            return Array.from(grouped.entries())
              .map(([category, categoryTotal]) => ({
                category,
                total: Number(categoryTotal.toFixed(2)),
                percentage: total > 0 ? Number(((categoryTotal / total) * 100).toFixed(2)) : 0,
              }))
              .sort((left, right) => right.total - left.total);
          };

          const buildBalancePayload = (rawBalance: number) => {
            if (Math.abs(rawBalance) < 0.005) {
              return { amount: 0, direction: "SETTLED" as const };
            }

            return {
              amount: Number(Math.abs(rawBalance).toFixed(2)),
              direction: rawBalance < 0 ? ("YOU_OWE" as const) : ("PARTNER_OWES" as const),
            };
          };
```

```
const parseTransactionPayload = (body: Record<string, unknown>) => {
  const title = normalizeTitle(String(body.title ?? ""));
  const amount = Number(body.amount);
  const type = parseTransactionType(body.type);
  const category = parseTransactionScope(body.scope ?? body.category);
  const transactionCategory = parseTransactionCategory(body.transactionCategory);

  if (!title) {
   throw new Error("TITLE_REQUIRED");
  }
  if (!Number.isFinite(amount) || amount <= 0) {
   throw new Error("AMOUNT_INVALID");
  }

  assertCategoryMatchesType(type, transactionCategory);

  return { title, amount, type, category, transactionCategory };
 };

export const createTransaction = async (req: AuthenticatedRequest, res: Response) => {
  try {
   const userId = req.userId;
   if (!userId) {
    return res.status(401).json({ message: "Unauthorized" });
   }

   const payload = parseTransactionPayload(req.body as Record<string, unknown>);
   const user = await prisma.user.findUnique({
    where: { id: userId },
    select: { id: true, pairId: true },
   });

   if (!user?.pairId) {
    return res.status(400).json({ message: "You need to be connected to a pair first" });
   }

   const transaction = await prisma.transaction.create({
    data: {
     ...payload,
     status: getInitialTransactionStatus(payload.category),
     createdById: userId,
     confirmedById: payload.category === TransactionScope.SELF ? userId : null,
     rejectedById: null,
     pairId: user.pairId,
    },
    include: transactionInclude,
   });

   return res.status(201).json({ transaction });
  } catch (error) {
   console.error("Create transaction error:", error);
   return res.status(500).json({ message: "Internal server error" });
  }
 };

export const getTransactionSummary = async (req: AuthenticatedRequest, res: Response) => {
  try {
   const userId = req.userId;
   if (!userId) {
    return res.status(401).json({ message: "Unauthorized" });
   }

   const user = await prisma.user.findUnique({
    where: { id: userId },
    select: { pairId: true },
   });
```

```
            const transactions = await prisma.transaction.findMany({
              where: { pairId: user!.pairId! },
              select: {
                amount: true,
                type: true,
                category: true,
                transactionCategory: true,
                createdById: true,
                status: true,
              },
            });

            const totalIncome = transactions
              .filter((transaction) => transaction.type === TransactionType.INCOME && isConfirmedTransaction(transaction.status))
              .reduce((sum, transaction) => sum + transaction.amount, 0);
            const totalExpense = transactions
              .filter((transaction) => transaction.type === TransactionType.EXPENSE && isConfirmedTransaction(transaction.status))
              .reduce((sum, transaction) => sum + transaction.amount, 0);
            const rawBalance = transactions.reduce(
              (sum, transaction) => sum + getBalanceContribution(transaction, userId),
              0
            );

            return res.json({
              totalBudget: Number((totalIncome - totalExpense).toFixed(2)),
              balance: buildBalancePayload(rawBalance),
              expenseByCategory: buildCategorySummary(transactions, TransactionType.EXPENSE),
              incomeByCategory: buildCategorySummary(transactions, TransactionType.INCOME),
            });
          } catch (error) {
            console.error("Summary error:", error);
            return res.status(500).json({ message: "Internal server error" });
          }
        };

        export const confirmTransaction = async (req: AuthenticatedRequest, res: Response) => {
          try {
            const userId = req.userId!;
            const transactionId = String(req.params.id ?? "");

            const transaction = await prisma.transaction.findUnique({
              where: { id: transactionId },
              include: transactionInclude,
            });

            if (!transaction || transaction.createdById === userId) {
              return res.status(403).json({ message: "Only the partner can confirm or reject this transaction" });
            }

            if (transaction.status !== TransactionStatus.PENDING_CONFIRMATION) {
              return res.status(400).json({ message: "Transaction is not waiting for confirmation" });
            }

            const updatedTransaction = await prisma.transaction.update({
              where: { id: transaction.id },
              data: {
                status: TransactionStatus.CONFIRMED,
                confirmedById: userId,
```

```
            rejectedById: null,
          },
          include: transactionInclude,
        });

        return res.json({ transaction: updatedTransaction });
      } catch (error) {
        console.error("Confirm transaction error:", error);
        return res.status(500).json({ message: "Internal server error" });
      }
    };
```

В7. Код сутності viewmodel.

```
package docs.samples

import android.app.Application
import androidx.compose.runtime.mutableStateOf
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.viewModelScope
import com.vandoliak.coupleapp.R
import com.vandoliak.coupleapp.data.local.TokenManager
import com.vandoliak.coupleapp.data.remote.RetrofitInstance
import com.vandoliak.coupleapp.data.remote.SharedSplitRequest
import com.vandoliak.coupleapp.data.remote.TaskActionResponse
import com.vandoliak.coupleapp.data.remote.TaskCreateRequest
import com.vandoliak.coupleapp.data.remote.TaskDto
import com.vandoliak.coupleapp.data.remote.TaskListResponse
import com.vandoliak.coupleapp.data.remote.TaskPartnerDto
import com.vandoliak.coupleapp.data.remote.extractErrorMessage
import com.vandoliak.coupleapp.presentation.util.appString
import com.vandoliak.coupleapp.presentation.util.dateInputToApiDate
import com.vandoliak.coupleapp.presentation.util.sanitizeDateInput
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.launch
import retrofit2.Response

class TaskViewModel(app: Application) : AndroidViewModel(app) {

    private val tokenManager = TokenManager(app)

    var taskTitle = mutableStateOf("")
        private set

    var taskPoints = mutableStateOf("")
        private set

    var dueDate = mutableStateOf("")
        private set

    var taskType = mutableStateOf("CHALLENGE")
        private set

    var recurrenceType = mutableStateOf("NONE")
        private set

    var recurrenceInterval = mutableStateOf("")
        private set

    var tasks = mutableStateOf<List<TaskDto>>(emptyList())
        private set

    var currentUserId = mutableStateOf<String?>(null)
        private set

    var currentUserPoints = mutableStateOf(0)
        private set

    var partner = mutableStateOf<TaskPartnerDto?>(null)
        private set

    var isLoading = mutableStateOf(false)
        private set

    var isSubmitting = mutableStateOf(false)
        private set
```

```
    var error =
mutableStateOf<String?>(null)
        private set

    var successMessage =
mutableStateOf<String?>(null)
        private set

    fun onTaskTitleChange(value: String)
{
        taskTitle.value = value
    }

    fun onTaskPointsChange(value: String)
{
        taskPoints.value = value.filter
{ it.isDigit() }
    }

    fun onDueDateChange(value: String) {
        dueDate.value =
sanitizeDateInput(value)
    }

    fun onTaskTypeChange(value: String) {
        taskType.value = value
    }

    fun onRecurrenceTypeChange(value:
String) {
        recurrenceType.value = value
        if (value != "EVERY_X_DAYS") {
            recurrenceInterval.value = ""
        }
    }

    fun onRecurrenceIntervalChange(value:
String) {
        recurrenceInterval.value =
value.filter { it.isDigit() }
    }

    fun loadTasks() {
        viewModelScope.launch {
            fetchTasks(showLoader = true)
        }
    }

    fun createTask(onSuccess: (() ->
Unit)? = null) {
        val points =
taskPoints.value.toIntOrNull()
        val apiDueDate = if
(dueDate.value.isBlank()) null else
dateInputToApiDate(dueDate.value)
        val recurrenceIntervalValue =
recurrenceInterval.value.toIntOrNull()

        successMessage.value = null

        if (taskTitle.value.isBlank()) {
            error.value =
appString(R.string.task_title_required)
            return
        }

        if (points == null || points <=
0) {
            error.value =
appString(R.string.points_positive)
            return
        }

        if (dueDate.value.isNotBlank() &&
apiDueDate == null) {
            error.value =
appString(R.string.due_date_format_requir
ed)
            return
        }

        if (recurrenceType.value !=
"NONE" && apiDueDate == null) {
            error.value =
appString(R.string.recurring_tasks_requir
e_due_date)
            return
        }

        if (recurrenceType.value ==
"EVERY_X_DAYS" &&
            (recurrenceIntervalValue ==
null || recurrenceIntervalValue <= 0)
        ) {
            error.value =
appString(R.string.recurring_interval_pos
itive)
            return
        }

        viewModelScope.launch {
            executeTaskMutation(
                successText = if
(taskType.value == "SHARED") {

appString(R.string.shared_task_created_su
ccessfully)
                } else {

appString(R.string.challenge_created_succ
essfully)
                }
            ) { authorization ->

RetrofitInstance.taskApi.createTask(
                    authorization =
authorization,
                    request =
TaskCreateRequest(
                        title =
taskTitle.value.trim(),
                        type =
taskType.value,
                        points = points,
                        dueDate =
apiDueDate,
                        recurrenceType =
recurrenceType.value,

recurrenceInterval =
recurrenceIntervalValue
                    )
                )
```

```
            }

            if (error.value == null) {
                resetCreateForm()
                onSuccess?.invoke()
            }
        }
    }

    fun requestCompletion(taskId: String)
{
        runTaskAction(taskId,
appString(R.string.waiting_for_partner_co
nfirmation)) { authorization, id ->

RetrofitInstance.taskApi.requestCompletio
n(authorization, id)
        }
    }

    fun confirmCompletion(taskId: String)
{
        runTaskAction(taskId,
appString(R.string.task_completed_success
fully)) { authorization, id ->

RetrofitInstance.taskApi.confirmCompletio
n(authorization, id)
        }
    }

    fun returnTask(taskId: String) {
        runTaskAction(taskId,
appString(R.string.task_returned_successf
ully)) { authorization, id ->

RetrofitInstance.taskApi.returnTask(autho
rization, id)
        }
    }

    fun failTask(taskId: String) {
        runTaskAction(taskId,
appString(R.string.task_failed_success))
{ authorization, id ->

RetrofitInstance.taskApi.failTask(authori
zation, id)
        }
    }

    fun proposeSharedSplit(taskId:
String, myPoints: Int, partnerPoints:
Int, onSuccess: (() -> Unit)? = null) {
        runTaskAction(taskId,
appString(R.string.shared_split_proposed)
, onSuccess) { authorization, id ->

RetrofitInstance.taskApi.proposeSharedSpl
it(
                authorization =
authorization,
                taskId = id,
                request =
SharedSplitRequest(myPoints = myPoints,
partnerPoints = partnerPoints)
            )
        }
    }

    private fun runTaskAction(
        taskId: String,
        successText: String,
        onSuccess: (() -> Unit)? = null,
        action: suspend (String, String)
-> Response<TaskActionResponse>
    ) {
        viewModelScope.launch {

executeTaskMutation(successText)
{ authorization ->
                action(authorization,
taskId)
            }
            if (error.value == null) {
                onSuccess?.invoke()
            }
        }
    }

    private suspend fun
fetchTasks(showLoader: Boolean) {
        val token =
tokenManager.tokenFlow.first()

        if (token.isNullOrBlank()) {
            error.value =
appString(R.string.session_expired_login)
            return
        }

        try {
            if (showLoader) {
                isLoading.value = true
            }

            val response =
RetrofitInstance.taskApi.getTasks("Bearer
$token")

            if (response.isSuccessful) {
                val body =
response.body()
                if (body != null) {
                    applyTaskList(body)
                    error.value = null
                } else {
                    error.value =
appString(R.string.unexpected_server_resp
onse)
                }
            } else {
                error.value =
response.extractErrorMessage()
            }
        } catch (e: Exception) {
            error.value =
e.localizedMessage ?:
appString(R.string.network_error)
        } finally {
            isLoading.value = false
        }
    }
```

```
    private fun applyTaskList(body:
TaskListResponse) {
        tasks.value = body.tasks
        currentUserId.value =
body.currentUserId
        currentUserPoints.value =
body.currentUserPoints
        partner.value = body.partner
    }
}
```

В8. Код інтерфейсу користувача на екрані календаря. /*

* Documentation sample:

* Core Compose calendar UI with monthly grid and day cells.

* This file is a copy-only code sample for thesis documentation.

*/

package docs.samples

import androidx.compose.foundation.background

import androidx.compose.foundation.border

import androidx.compose.foundation.clickable

import androidx.compose.foundation.layout.Arrangement

import androidx.compose.foundation.layout.Box

import androidx.compose.foundation.layout.BoxWithConstraints

import androidx.compose.foundation.layout.PaddingValues

import androidx.compose.foundation.layout.Row

import androidx.compose.foundation.layout.fillMaxSize

import androidx.compose.foundation.layout.fillMaxWidth

import androidx.compose.foundation.layout.height

import androidx.compose.foundation.layout.padding

import androidx.compose.foundation.lazy.grid.GridCells

import androidx.compose.foundation.lazy.grid.GridItemSpan

import androidx.compose.foundation.lazy.grid.LazyVerticalGrid

import androidx.compose.foundation.lazy.grid.items

import androidx.compose.foundation.shape.CircleShape

import androidx.compose.foundation.shape.RoundedCornerShape

import androidx.compose.material.icons.Icons

import androidx.compose.material.icons.automirrored.outlined.KeyboardArrowLeft

import androidx.compose.material.icons.automirrored.outlined.KeyboardArrowRight

import androidx.compose.material3.Icon

import androidx.compose.material3.MaterialTheme

import androidx.compose.material3.ModalBottomSheet

import androidx.compose.material3.Surface

import androidx.compose.material3.Text

import androidx.compose.material3.TextButton

import androidx.compose.runtime.Composable

import androidx.compose.runtime.LaunchedEffect

import androidx.compose.runtime.getValue

import androidx.compose.runtime.mutableStateOf

import androidx.compose.runtime.remember

import androidx.compose.runtime.setValue

import androidx.compose.ui.Alignment

import androidx.compose.ui.Modifier

import androidx.compose.ui.draw.clip

```
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.Dp
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import androidx.lifecycle.viewmodel.compose.viewModel
import com.vandoliak.coupleapp.R
import com.vandoliak.coupleapp.presentation.components.EmptyState
import com.vandoliak.coupleapp.presentation.components.PointBadge
import com.vandoliak.coupleapp.presentation.components.RefreshContainer
import com.vandoliak.coupleapp.presentation.util.formatMonthTitle
import com.vandoliak.coupleapp.presentation.viewmodel.BlueprintViewModel
import com.vandoliak.coupleapp.presentation.viewmodel.CalendarDayUi
import com.vandoliak.coupleapp.presentation.viewmodel.CalendarItemType
import com.vandoliak.coupleapp.presentation.viewmodel.CalendarItemUi
import com.vandoliak.coupleapp.presentation.viewmodel.CalendarViewModel

@androidx.compose.material3.ExperimentalMaterial3Api
@Composable
fun CalendarScreen(
    modifier: Modifier = Modifier
) {
    val calendarViewModel: CalendarViewModel = viewModel()
    val blueprintViewModel: BlueprintViewModel = viewModel()

    LaunchedEffect(Unit) {
        calendarViewModel.loadCalendar()
        blueprintViewModel.loadBlueprints()
    }

    if (calendarViewModel.isDaySheetVisible.value) {
        ModalBottomSheet(
            onDismissRequest = calendarViewModel::dismissDaySheet
        ) {
            // Day sheet omitted here; main thesis interest is the calendar grid.
        }
    }

    RefreshContainer(
        isRefreshing = calendarViewModel.isLoading.value,
        onRefresh = calendarViewModel::loadCalendar,
        modifier = modifier.fillMaxSize()
    ) {
        BoxWithConstraints(
            modifier = Modifier.fillMaxSize()
        ) {
            val cellHeight = if (maxWidth > maxHeight || maxWidth >= 600.dp) 120.dp else 96.dp
            val days = calendarViewModel.monthDays.value
            val currentUserId = calendarViewModel.currentUserId.value
            val weekdayLabels = listOf(
```

```
stringResource(R.string.weekday_mon),
stringResource(R.string.weekday_tue),
stringResource(R.string.weekday_wed),
stringResource(R.string.weekday_thu),
stringResource(R.string.weekday_fri),
stringResource(R.string.weekday_sat),
stringResource(R.string.weekday_sun)
                )

                LazyVerticalGrid(
                    columns = GridCells.Fixed(7),
                    modifier = Modifier.fillMaxSize(),
                    horizontalArrangement = Arrangement.spacedBy(2.dp),
                    verticalArrangement = Arrangement.spacedBy(2.dp),
                    contentPadding = PaddingValues(start = 8.dp, top = 8.dp, end = 8.dp, bottom = 16.dp)
                ) {
                    item(span = { GridItemSpan(7) }) {
                        CalendarHeader(
                            monthTitle = formatMonthTitle(calendarViewModel.visibleMonth.value),
                            onPrevious = calendarViewModel::goToPreviousMonth,
                            onNext = calendarViewModel::goToNextMonth
                        )
                    }

                    items(weekdayLabels, key = { it }) { label ->
                        Box(
                            modifier = Modifier
                                .fillMaxWidth()
                                .clip(RoundedCornerShape(6.dp))
                                .background(MaterialTheme.colorScheme.surfaceVariant.copy(alpha = 0.55f))
                                .padding(vertical = 4.dp),
                            contentAlignment = Alignment.Center
                        ) {
                            Text(
                                text = label,
                                style = MaterialTheme.typography.labelSmall,
                                color = MaterialTheme.colorScheme.onSurfaceVariant
                            )
                        }
                    }

                    if (calendarViewModel.isLoading.value && days.isEmpty()) {
                        item(span = { GridItemSpan(7) }) {
                            EmptyState(
                                title = stringResource(R.string.calendar_loading_title),
                                modifier = Modifier.padding(top = 12.dp)
                            )
                        }
                    } else {
                        items(days, key = { it.date.toString() }) { day ->
                            CalendarDayCell(
                                day = day,
                                currentUserId = currentUserId,
                                cellHeight = cellHeight,
                                onClick = { calendarViewModel.onDaySelected(day.date) }
                            )
```

```
                }
            }
        }
    }
}

@Composable
private fun CalendarHeader(
    monthTitle: String,
    onPrevious: () -> Unit,
    onNext: () -> Unit
) {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(bottom = 4.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement = Arrangement.SpaceBetween
    ) {
        TextButton(onClick = onPrevious) {
            Icon(
                imageVector = Icons.AutoMirrored.Outlined.KeyboardArrowLeft,
                contentDescription = stringResource(R.string.previous_month)
            )
        }

        Text(
            text = monthTitle,
            style = MaterialTheme.typography.titleMedium,
            fontWeight = FontWeight.SemiBold
        )

        TextButton(onClick = onNext) {
            Icon(
                imageVector = Icons.AutoMirrored.Outlined.KeyboardArrowRight,
                contentDescription = stringResource(R.string.next_month)
            )
        }
    }
}

@Composable
private fun CalendarDayCell(
    day: CalendarDayUi,
    currentUserId: String?,
    cellHeight: Dp,
    onClick: () -> Unit
) {
    val colors = MaterialTheme.colorScheme
    val borderColor = when {
        day.isToday -> colors.primary.copy(alpha = 0.45f)
        else -> colors.outlineVariant.copy(alpha = 0.75f)
    }

    Box(
        modifier = Modifier
            .fillMaxWidth()
            .height(cellHeight)
            .clip(RoundedCornerShape(8.dp))
            .background(
                if (day.isInCurrentMonth) colors.surface else colors.surfaceVariant.copy(alpha = 0.28f)
            )
            .border(
                width = 1.dp,
                color = borderColor,
                shape = RoundedCornerShape(8.dp)
            )
            .clickable(onClick = onClick)
            .padding(horizontal = 4.dp, vertical = 5.dp)
    ) {
```

```
androidx.compose.foundation.layout.Column(
                modifier = Modifier.fillMaxSize(),
                verticalArrangement =
Arrangement.spacedBy(3.dp)
            ) {
                Box(
                    modifier = Modifier.fillMaxWidth()
                ) {
                    val dayNumberColor = when {
                        day.isSelected -> colors.onPrimary
                        day.isToday -> colors.primary
                        day.isInCurrentMonth ->
colors.onSurface
                        else -> colors.onSurfaceVariant
                    }

                    Surface(
                        shape = CircleShape,
                        color = if (day.isSelected)
colors.primary else
androidx.compose.ui.graphics.Color.Transparent,
                        modifier =
Modifier.align(Alignment.TopStart)
                    ) {
                        Text(
                            text = day.dayLabel,
                            modifier =
Modifier.padding(horizontal = 6.dp, vertical = 2.dp),
                            color = dayNumberColor,
                            style =
MaterialTheme.typography.labelSmall.copy(
                                fontSize = 11.sp,
                                fontWeight =
FontWeight.SemiBold
                            )
                        )
                    }
                }

                val previews = day.items.take(2)
                previews.forEach { item ->
                    CalendarPreviewChip(
                        item = item,
                        currentUserId = currentUserId
                    )
                }

                val hiddenItems = day.items.size -
previews.size
                if (hiddenItems > 0) {
                    Text(
                        text = "+$hiddenItems",
                        style =
MaterialTheme.typography.labelSmall.copy(fontSize =
8.sp),
                        color = colors.onSurfaceVariant,
                        maxLines = 1,
                        overflow = TextOverflow.Ellipsis
                    )
                }
            }
        }
    }

    @Composable
    private fun CalendarPreviewChip(
        item: CalendarItemUi,
        currentUserId: String?
    ) {
        val colors = MaterialTheme.colorScheme
        val isMine = item.createdById ==
currentUserId || item.assignedToId == currentUserId
        val backgroundColor = when (item.type) {
            CalendarItemType.EVENT ->
colors.secondaryContainer
            CalendarItemType.TASK -> if
(item.taskType == "SHARED") {
                colors.tertiaryContainer
            } else if (isMine) {
                colors.primaryContainer
            } else {
                colors.secondaryContainer
            }
```

```
            }
            val textColor = when (item.type) {
                CalendarItemType.EVENT -> colors.onSecondaryContainer
                CalendarItemType.TASK -> if (item.taskType == "SHARED") {
                    colors.onTertiaryContainer
                } else if (isMine) {
                    colors.onPrimaryContainer
                } else {
                    colors.onSecondaryContainer
                }
            }

            Surface(
                shape = RoundedCornerShape(5.dp),
                color = backgroundColor,
                modifier = Modifier.fillMaxWidth()
            ) {
                Row(
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(horizontal = 4.dp, vertical = 2.dp),
                    verticalAlignment = Alignment.CenterVertically,
                    horizontalArrangement = Arrangement.spacedBy(3.dp)
                ) {
                    Text(
                        text = item.title,
                        modifier = Modifier.weight(1f, fill = false),
                        maxLines = 1,
                        overflow = TextOverflow.Ellipsis,
                        color = textColor,
                        style = MaterialTheme.typography.labelSmall.copy(fontSize = 8.sp)
                    )

                    if (item.type == CalendarItemType.TASK) {
                        PointBadge(points = item.points, compact = true)
                    }
                }
            }
        }
```

ДОДАТОК Г
(обов’язковий)

КЕРІВНИЦТВО КОРИСТУВАЧА

Дане керівництво користувача описує порядок роботи з мобільним застосунком, призначеним для автоматизації розподілу обов’язків у сім’ї. Програмний засіб дозволяє створювати спільний простір для двох користувачів, вести побутові задачі, планувати події у календарі, обліковувати спільні фінанси, формувати wishlist, використовувати систему балів та купувати внутрішні винагороди.

Застосунок працює за клієнт-серверною моделлю, тому для повноцінного використання необхідне підключення до мережі Інтернет. Дані користувачів, задач, подій, фінансових записів та винагород синхронізуються між партнерами через серверну частину.

Після запуску застосунку користувач потрапляє на екран входу. Якщо обліковий запис уже створено, необхідно ввести адресу електронної пошти та пароль, після чого натиснути кнопку входу. У разі успішної авторизації система зберігає сесію користувача та автоматично переходить до наступного етапу.

Якщо користувач не має облікового запису, необхідно перейти на екран реєстрації, ввести електронну пошту, пароль та підтвердити створення акаунту. Після успішної реєстрації система автоматично авторизує користувача.

Після входу в акаунт користувач має створити пару або приєднатися до вже існуючої. Для створення нового спільного простору використовується кнопка створення пари. У такому разі система формує код, який може бути переданий партнеру для підключення. Якщо користувач отримав код від партнера, він вводить його у відповідне поле та підтверджує приєднання.

Після створення або підключення до пари користувач отримує доступ до основних розділів застосунку.

Основна частина застосунку побудована навколо нижнього навігаційного меню. За його допомогою користувач може швидко переходити між головними модулями системи:

- календарем;
- задачами та викликами;
- фінансами;
- wishlist;
- профілем.

Кожна вкладка відповідає за окрему частину спільного побуту. Календар використовується для перегляду задач і подій за датами, задачі — для розподілу побутових обов’язків, фінанси — для обліку доходів і витрат, wishlist — для планування покупок, а профіль — для перегляду особистих даних, балів, винагород і налаштувань.

Форми створення записів не відображаються постійно на екрані. Для додавання нового запису користувач натискає відповідну кнопку створення, після чого відкривається окрема форма, діалогове вікно або нижня панель.

Модуль задач призначений для створення, виконання та контролю побутових обов’язків. Користувач може створювати звичайні гейміфіковані задачі-виклики або спільні задачі.

Для створення задачі необхідно відкрити вкладку задач або обрати конкретний день у календарі та натиснути кнопку створення. У формі створення потрібно ввести назву задачі, вибрати її тип, дату виконання та кількість балів. За потреби можна встановити повторення задачі: кожні кілька днів, щотижня абощомісяця.

У challenge-задачі один користувач створює завдання та вкладає певну кількість власних балів у банк задачі. Партнер може виконати задачу, після чого виконання має бути підтверджене другим користувачем. Бали нараховуються лише після підтвердження.

Для спільних задач банк балів формується системою, а не списується з користувачів. Після виконання такої задачі партнери погоджують розподіл

винагороди між собою. Стандартним варіантом є поділ балів порівну, однак користувачі можуть запропонувати інший розподіл. Задача вважається завершеною лише після погодження обома сторонами.

Також користувач може редагувати або видаляти задачі, якщо має відповідні права. Для повторюваних задач передбачено можливість зупинити подальше повторення.

Календар призначений для перегляду задач, подій і нагадувань за конкретними датами. Він відображається у вигляді місячної сітки, де кожен день може містити короткі позначки створених записів.

Для перегляду детальної інформації необхідно натиснути на потрібний день. Після цього відкривається панель із повним переліком задач і подій за обрану дату. З цієї панелі користувач може створити нову задачу, додати календарну подію або використати раніше збережений шаблон.

Події використовуються для спільного планування та нагадувань. Для створення події необхідно ввести назву, опис, дату та за потреби встановити повторення. Події, як і задачі, можуть повторюватися щотижня, щомісяця абочерез заданий інтервал днів.

Календар дозволяє об’єднати планування побутових задач і звичайних подій в одному розділі, що спрощує координацію спільного часу.

Фінансовий модуль призначений для обліку доходів, витрат і взаєморозрахунків між партнерами. У верхній частині екрана фінансів відображається загальний бюджет пари. Нижче розміщується інформація про баланс між партнерами, тобто хто кому винен.

Для створення нового фінансового запису необхідно натиснути кнопку додавання транзакції. У формі потрібно ввести назву, суму, тип операції — дохід або витрата, категорію та призначення. Призначення може бути одним із таких: для себе, для партнера або для обох.

Якщо транзакція стосується лише самого користувача, вона одразу враховується у фінансовій історії. Якщо запис створено для партнера або для обох, він отримує статус очікування підтвердження. У такому разі партнер має

підтвердити або відхилити транзакцію. До моменту підтвердження така операція не впливає на загальний бюджет і баланс.

Користувач може переглядати історію транзакцій, редагувати або видаляти власні записи, а також аналізувати витрати й доходи за категоріями. Символ валюти відображається відповідно до вибору в налаштуваннях.

Модуль wishlist призначений для збереження бажаних або запланованих покупок. Для створення нового запису користувач натискає кнопку додавання, після чого вводить назву товару, посилання, орієнтовну ціну, пріоритет і призначення покупки.

Посилання у wishlist-елементах є клікабельними. Якщо користувач натискає на посилання, воно відкривається у браузері пристрою. Це дозволяє швидко переходити до сторінки товару.

Коли покупку виконано, користувач може позначити wishlist-елемент як придбаний. Якщо для елемента вказана ціна, система пропонує створити фінансову транзакцію. У такому разі користувач обирає категорію витрат, після чого система створює відповідний фінансовий запис.

Wishlist-елементи можна редагувати або видаляти, якщо вони не актуальні.

Профіль користувача

Екран профілю призначений для перегляду особистої інформації користувача. У ньому відображається аватарка, нікнейм, кількість балів, win streak, інформація про партнера та придбані винагороди.

Користувач може редагувати власний профіль: змінювати нікнейм, вибирати стандартну аватарку або завантажувати власне зображення. Профіль партнера доступний лише для перегляду, без можливості редагування.

Бали користувача відображаються як внутрішня ігрова валюта. Вони використовуються для створення задач, участі в гейміфікованій системі тапокупки винагород у магазині.

Внутрішній магазин винагород дозволяє користувачу витрачати накопичені бали на бонуси. Кожна винагорода має назву, опис, вартість у балах і, за потреби, додаткову умову доступу.

Для покупки винагороди користувач відкриває магазин, обирає потрібний бонус і підтверджує покупку. Якщо балів достатньо, система списує відповідну кількість балів, створює запис про покупку та надсилає партнеру внутрішнє повідомлення.

Цей механізм дозволяє використовувати бали не лише як показник активності, а і як елемент мотивації та домовленостей між партнерами.

Система внутрішніх повідомлень використовується для інформування користувача про важливі події. Наприклад, партнер може отримати повідомлення після покупки винагороди у магазині.

Користувач може переглядати список повідомлень і позначати їх як прочитані. Непрочитані повідомлення можуть відображатися у вигляді індикатора або лічильника біля відповідного елемента інтерфейсу.

Повідомлення не є повноцінним чатом, а використовуються для коротких системних сповіщень про важливі дії партнера.

Екран налаштувань відкривається з профілю користувача. У цьому розділі доступні параметри мови, теми оформлення, валюти, а також дії виходу з акаунту та розформування пари.

Користувач може обрати світлу, темну або системну тему. Тема застосовується до інтерфейсу застосунку й змінює його зовнішній вигляд відповідно до вибору користувача.

У налаштуваннях також доступний вибір мови інтерфейсу: української або англійської. Після зміни мови система може повідомити користувача про необхідність перезапуску застосунку для повного застосування нового параметра.

Вибір валюти використовується для фінансового модуля. Доступні варіанти: українська гривня, шведська крона, долар США та євро. Зміна валюти має косметичний характер і впливає лише на відображення символу, без автоматичної конвертації сум.

Кнопка виходу з акаунту очищає локальну сесію та повертає користувача до екрана входу. Функція розформування пари дозволяє припинити зв’язок із партнером і повернутися до екрана створення або приєднання до пари.

Якщо строк дії JWT-токена завершується, система автоматично очищає локальні дані сесії та повертає користувача до екрана входу. Це дозволяє уникнути ситуації, коли на різних екранах застосунку постійно відображається помилка недійсного або простроченого токена.

Після повторної авторизації користувач знову отримує доступ до свого профілю, пари та спільних даних.

Загальний порядок роботи із застосунком

Загальний сценарій використання застосунку складається з таких етапів:

- користувач реєструється або входить в акаунт;
- створює пару або приєднується до партнера за кодом;
- додає побутові задачі, події та фінансові записи;
- використовує календар для перегляду спільних планів;
- веде wishlist для майбутніх покупок;
- виконує задачі та отримує бали;
- витрачає бали у магазині винагород;
- керує профілем і налаштуваннями.

Таким чином, застосунок забезпечує зручне цифрове середовище для організації спільного побуту пари, поєднуючи планування, розподіл обов’язків, фінансовий облік і мотиваційну систему в одному мобільному інтерфейсі.

Протокол аналізу звіту подібності науковим керівником

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Родіон ВАНДОЛЯК

Співавтор:

Назва: Мобільний застосунок для автоматизації розподілу обов'язків у сім'ї

Науковий керівник: старший викладач Ганна БЕДРАТЮК

Підрозділ: Кафедра інженерії програмного забезпечення

Коефіцієнт подібності 1:3%

Коефіцієнт подібності 2:0.93%

Мікропробіли: 218

Заміна букв: 3

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2026-05-25 11:10:13.0

Після аналізу Звіту подібності констатую наступне:

☑ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

Дата 25.05.2026

експерт

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатами звіту/звітів перевірки роботи, продукованими програмно-технічним засобом (ами), на наявність текстових збігів.
Назва кваліфікаційної роботи: «Мобільний застосунок для автоматизації розподілу обов'язків в сім'ї»
Автор: Вандоляк Родіон Анатолійович
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Спеціальність: 121 – Інженерія програмного забезпечення
Науковий керівник: Бедратюк Ганна Іванівна, старший викладач
Після аналізу звіту/звітів зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається до захисту.	**відповідає**
2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована.	
3	Виявлені запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Виявлені запозичення частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнуті. Робота може бути допущена до захисту після того, як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:
Запозичення, виявлені в роботі є законними і не є плагіатом, оскільки:

1) Запозиченя розміщені в розділах аналізу існуючих аналогів та прототипів, які не описують безпосередньо авторське дослідження і не стосуються результатів роботиж
2) Усі запозичення фрагментарні, або мають належним чином оформленні посилання;
3) Окремі виявлені збіги є загальновживаними фразами або виразами;
4) Всі зафіксовані системою ознаки моифікації тесту відносяться до комбінування латинських символів з україномовними скороченнями індексів в формулах, що не є модифікацією тексту

   Сумарний обсяг всіх запозичень, визначений системо виявлення збігів/ідентичності/схожості Anti-Plagiarism, складає 5%, що, з урахуванням наведених обгрунтувань,відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

Дата 25.05.26

Завідувач кафедри _____ Леонід БЕДРАТЮК
Гарант освітньої програми _____ Леонід БЕДРАТЮК
Керівник кваліфікаційної роботи _____ Ганна БЕДРАТЮК

Mon May 25 10:28:25 EEST 2026, Форкун Юрій Вікторович, Хмельницький національний університет, ХНУ

Anti-Plagiarism (http://ap.km.ua) v-16.718

Максимальне співпадіння з одним документом 4.0%

Словники перевірки: UA, US, RU. **Помилок в документах: 14%**

ID: 272139 Назва: Мобільний застосунок для автоматизації розподілу обов'язків у сім'ї Додано в БД: 2026-05-25 Автора: Родіон ВАНДОЛЯК Керівники: старший викладач Ганна БЕДРАТЮК Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	114663	880	5551 (5%)	67 (8%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

Завідувачу кафедри інженерії програмного
забезпечення проф. Леоніду БЕДРАТЮКУ
здобувача вищої освіти
Вандоляка Родіона Анатолійовича
факультет ІТ, ІІІ курс, група ІПЗс-23-1

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності в Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений та надаю свою згоду на обробку й збереження університетом моєї роботи в інституційному репозитарії Хмельницького національного університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-обчислювального комплексу StrikePlagiarism та/або програмно-технічного засобу AntiPlagiarism і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення текстових збігів у роботах.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

02.09.2025
дата

підпис

Завідувачу кафедри
інженерії програмного забезпечення
проф. Бедратюку Л. П.
студента групи ІПЗс-23-1
Вандоляка Р.А.
Прізвище, ініціали

ЗАЯВА

Прошу закріпити за мною тему кваліфікаційної роботи освітнього ступеня «бакалавр» за спеціальністю 121 «Інженерія програмного забезпечення»: «Мобільний застосунок для автоматизації розподілу обов'язків в сім'ї»

(керівник роботи – старший викладач. Бедратюк Ганна Іванівна)
Прізвище, ім'я, по батькові

02.01.2026р.
Дата

Підпис студента

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ
освітнього ступеня «Бакалавр»

Дипломник Вандоляк Родіон Анатолійович

Тема Мобільний застосунок для автоматизації розподілу обов'язків в сім'ї

Спеціальність 121 – Інженерія програмного забезпечення

Обсяг кваліфікаційної роботи:

Кількість листів креслень 3 кількість сторінок записки 78

1.Короткий зміст пояснювальної записки та прийнятих рішень У кваліфікаційній роботі досліджено і проаналізовано предметну область, визначено усі функціональні та нефункціональні вимоги. Був проведений аналіз існуючих програм на ринку, розглянуто їх переваги і недоліки, та доведено актуальність розробки нового програмного забезпечення. Розглянуто інструменти для реалізації спроектованих рішень, в результаті чого створено програмне забезпечення. Також було проведено тестування програми, за результатами якого доведено, що розроблене програмне забезпечення працює коректно та готове до експлуатації.

2. Висновок про відповідність роботи поставленому завданню Кваліфікаційна робота виконана відповідно до поставленого завдання та з дотриманням всіх вимог.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки та передових методів роботи У вступі доведено актуальність теми, визначено мету та завдання дипломного проектування. У першому розділі проведено аналіз предметної області, розглянуто існуючі рішення та визначені функціональні і нефункціональні вимоги до розроблюваного програмного забезпечення. У другому розділі проведено аналіз сучасних архітектур, розглянуто їх переваги і недоліки та визначено, що система буде відповідати монолітній архітектурі та моделі клієнт-сервер. У третьому розділі підготовлено всі залежності для написання коду та виконано практичну розробку програмних модулів і описано їх особливості, в результаті чого створено програмний продукт. Також у цьому розділі виконано модульне тестування системи та проведено його у відповідності до функціональних вимог, в результаті було підтверджено коректну роботу програми.

4. Позитивні сторони роботи Тематика кваліфікаційної роботи є актуальною, оскільки організація спільного побуту, розподіл домашніх обов'язків, планування спільного часу та ведення сімейних фінансів є важливими аспектами повсякденного життя сім'ї або пари.
У роботі запропоновано комплексне програмне рішення, яке поєднує в одному мобільному застосунку таск-менеджер, календар, фінансовий облік, wishlist, профілі користувачів та систему гейміфікації. Позитивною стороною є також використання сучасних технологій розробки, зокрема Kotlin, Jetpack Compose, Node.js, Express.js, TypeScript, PostgreSQL та Prisma ORM.

5. Негативні сторони роботи У роботі реалізовано основні функції мобільного застосунку, проте окремі можливості можуть бути розширені в майбутньому. Зокрема, доцільно додати push-сповіщення для оперативного інформування користувачів про нові задачі, підтвердження транзакцій або покупки винагород.

6. Оцінка графічного оформлення та пояснювальної записки Графічне оформлення виконано відповідно до теми кваліфікаційної роботи та подано у вигляді діаграм, таблиць і рисунків, що відображають архітектуру системи, структуру бази даних, варіанти використання та логіку роботи окремих модулів.
Пояснювальна записка оформлена згідно з вимогами чинних стандартів. Матеріал структуровано послідовно: від аналізу предметної області та постановки задачі до проєктування, програмної реалізації, тестування та формування висновків.

7. Відгук про кваліфікаційну роботу в цілому Кваліфікаційна робота заслуговує позитивної оцінки. Матеріал пояснювальної записки структурований, послідовний, чіткий та простий, що дозволяє чітко зрозуміти викладений матеріал у рамках тематики проектування. Графічний матеріал дає можливість наочно побачити деталі проектування системи.

8. Інші зауваження

9. Оцінка кваліфікаційної роботи Кваліфікаційна робота виконана у повному обсязі, відповідає поставленій задачі та заслуговує на оцінку «добре».

РЕЦЕНЗЕНТ Киселя Світлана Миколаївна к.ф.-м.н.
доц. кафедри КІС, ХНУ

"25" 05 2026 р. _____
(підпис)

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ДЕКЛАРАЦІЯ УЧАСНИКА ОСВІТНЬОГО ПРОЦЕСУ

щодо дотримання академічної доброчесності

Цією декларацією я, Вандоляк Родіон Анатолійович,

студент III курсу спеціальності 121 – Інженерія програмного забезпечення, група ІПЗс-23-1

здобувач вищої освіти (шифр та назва спец-ті, курс, академічна група)

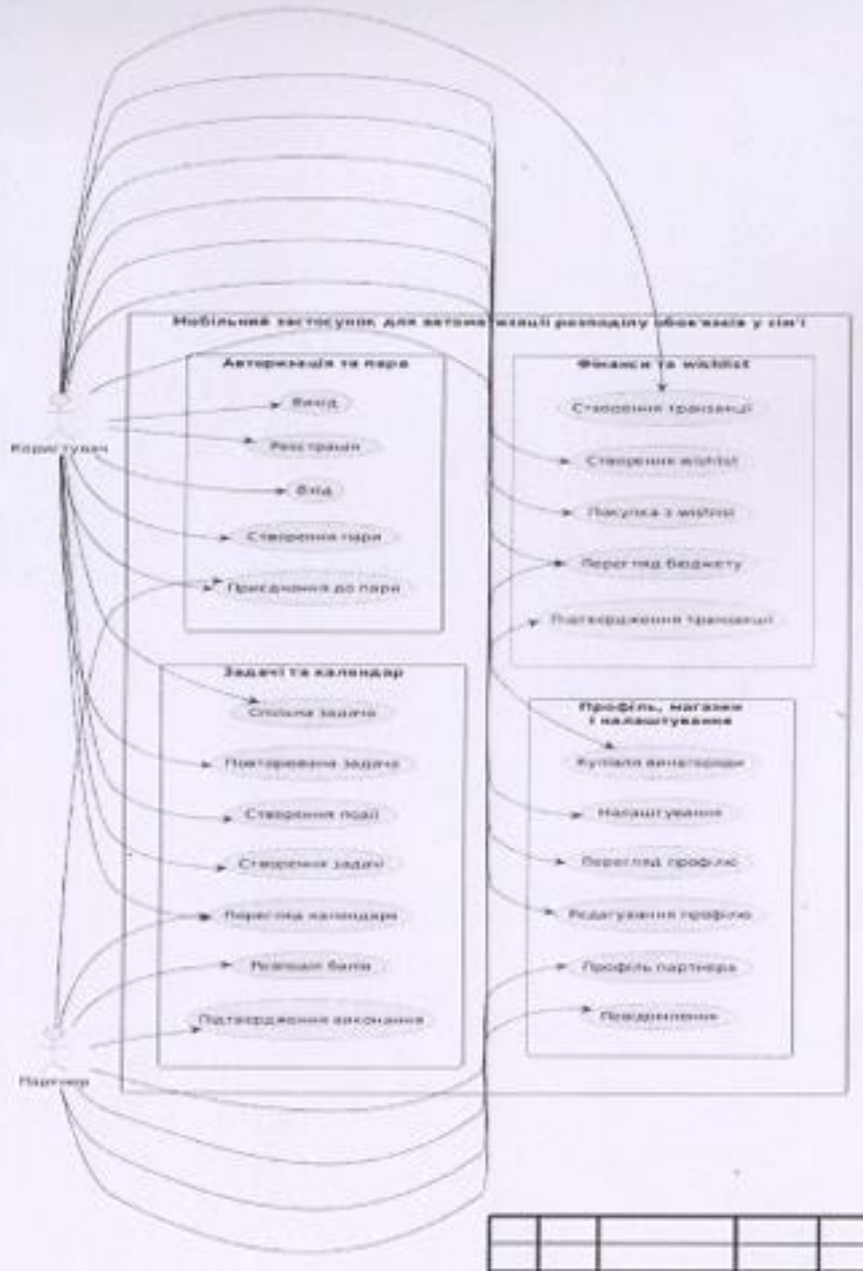
підтверджую, що ознайомився (-лась) з Положенням про систему забезпечення академічної доброчесності у Хмельницькому національному університеті та Кодексом академічної доброчесності і **зобов'язуюсь** дотримуватися їх вимог під час освітнього процесу, проведення наукової діяльності, виконання організаційно-адміністративних функцій тощо.

Усвідомлюю, що у разі порушення мною принципів академічної доброчесності нестиму відповідальність перед академічною спільнотою ХНУ згідно з нормами, визначеними Положенням про систему забезпечення академічної доброчесності у Хмельницькому національному університеті, законодавства України.

02 09. 2025 р. _____

Підпис

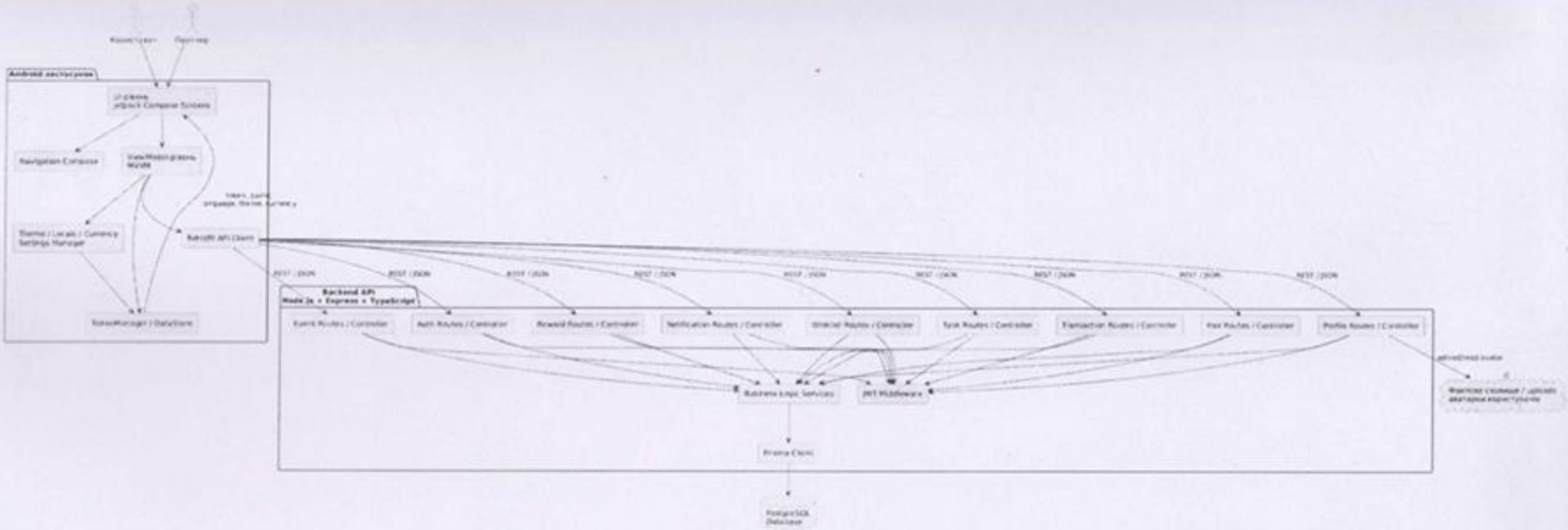
ГРАФІЧНІ МАТЕРІАЛИ



КвРІПЗс.230129.01.02.ПЗ

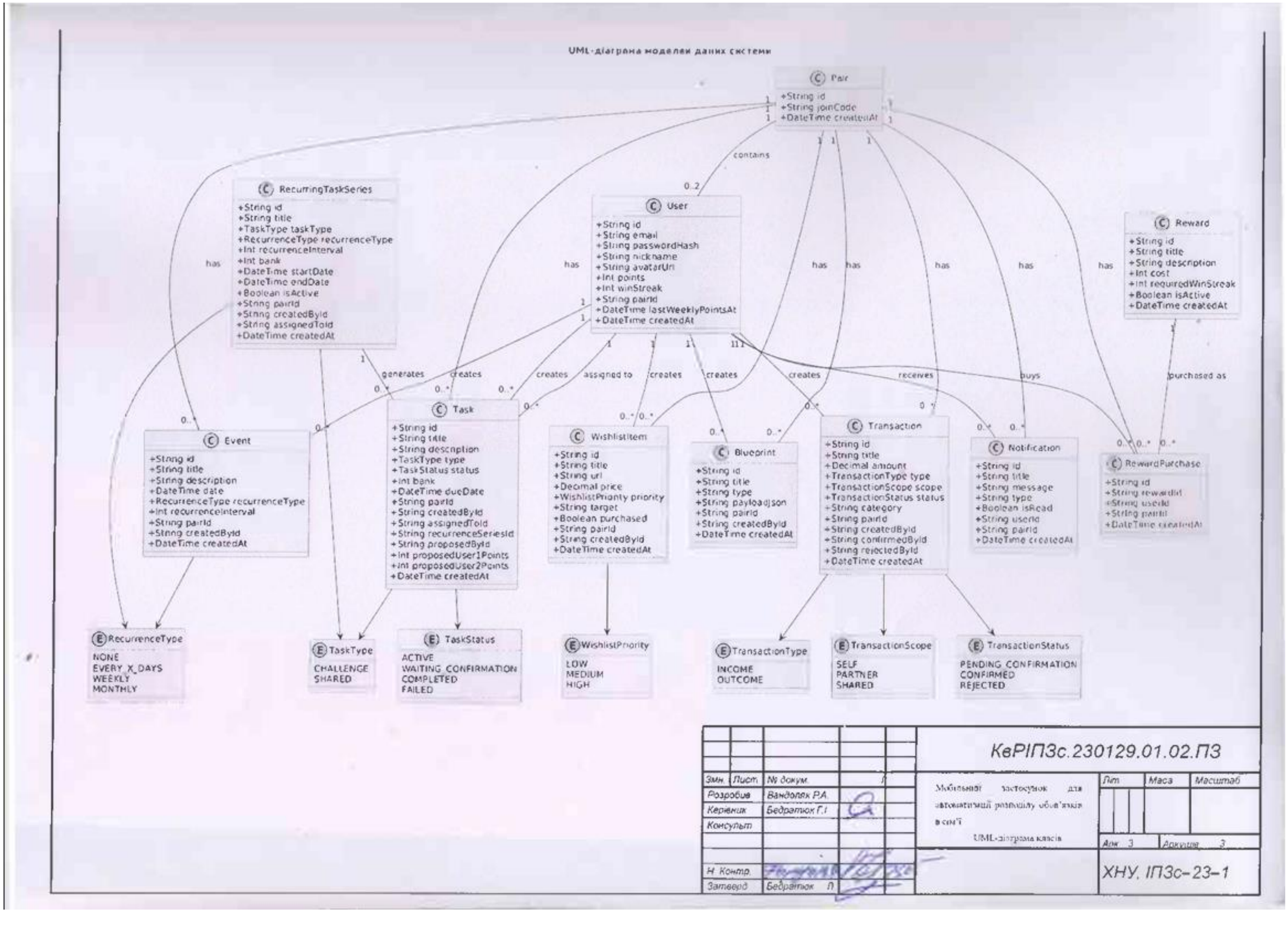
		№ докум.	Підпис	Дата		
Розробив	Вандолах Р.А.				Мобільний застосунок для автоматизації розподілу обов'язків в сім'ї. UML-діаграма варіантів використання	Літ.	Маса	Масштаб
Керівник	Бебешко Г.І.							
Консульт.						Арк. 1	Аркушів 3	
Н. Контр.	Форсун Ю.В.					ХНУ, ІПЗс–23–1		

Діаграма компонентів системи



										КвРІПЗс.230129.01.02.ПЗ	
Змн.	Лист	№ докум.	Підпис								
Розробив		Вандолюк Р.А.			Мобільний застосунок для автоматизації розподілу обов'язків в сім'ї			Літ.	Маса	Масштаб	
Керівник		Бедратюк Г.І.									
Консульт.					Діаграма зв'язків модулів			Арк. 2		Аркушів 3	
Н. Контр.								ХНУ. ІПЗс–23–1			
Затвердв.		Бедратюк Г.І.									

UML-діаграма моделей даних системи



КвРІПЗс.230129.01.02.ПЗ

Змін.	Лист.	№ докум.		
Розробив		Вандолях Р.А.		
Керівник		Бедратюк Г.І.		
Консульт				
Н. Контр.				
Затвердив		Бедратюк Л.		

Мобільний застосунок для автоматизації розподілу обов'язків в сім'ї

UML-діаграма класів

Літ.	Маса	Масштаб
Арк. 3	Аркушів 3	

ХНУ, ІПЗс-23-1