НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри

Комп'ютерних наук

_____ Голуб Б.Л.

"___" _____ 2025 р

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА
на тему

«Програмноге забезпечення мобільного додатку для введення комунальних платежів з функцією нагадування»

Спеціальність 121 – «Інженерія програмного забезпечення»

Гарант освітньої програми
К.т.н., доцент _____ Вайганг Г.О
Керівник бакалаврської кваліфікаційної роботи
_____к.е.н., ст. викладач___ _____ _____Ніколаєнко Д.В._____
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав _____ _____Шилєнков Вадим Максимович____
(підпис) (ПІБ студента)

КИЇВ – 2025

КИЇВ – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерних наук
(назва кафедри)

к.т.н, доцент _____ Голуб Б.Л.
(науковий ступінь, вчене звання) (підпис) (ПІБ)
"___" _____ 2024 р.

З А В Д А Н Н Я
на виконання бакалаврської кваліфікаційної роботи студенту

Шилєнков Вадим Максимович
(прізвище, ім'я, по батькові)

Спеціальність 121 – «Інженерія програмного забезпечення»
Тема бакалаврської кваліфікаційної роботи Програмноге забезпечення мобільного додатку для введення комунальних платежів з функцією нагадування
затверджена наказом ректора НУБіП України від 16.12.2024 № 2248 «С»
Термін подання завершеної роботи на кафедру_____
(рік, місяць, число)

Вихідні дані до бакалаврської кваліфікаційної роботи
Вимоги до розробки мобільного додатку для управління комунальними платежами._____
Перелік питань, які потрібно розробити:
Вступ і постановка проблеми, цілі та сфера застосування, огляд літератури, методологія, архітектура та дизайн системи, впровадження, оцінка та результати_____

Дата видачі завдання "___"_____ 2025 р.

Керівник бакалаврської кваліфікаційної роботи
_____к.е.н._____ _____ Ніколаєнко Д.В._____
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Завдання прийняв до виконання _____ _____Шилєнков В.М._____
(підпис) (ПІБ студента)

ЗМІСТ

ВСТУП

В епоху стрімкого розвитку цифрових технологій та зростання обсягів інформації, ефективне управління особистими фінансами стає дедалі актуальнішим завданням для кожної сучасної людини. Особливе місце серед регулярних витрат займають комунальні платежі, своєчасний контроль та оплата яких потребують уваги та організованості. Забудькуватість, складність відстеження численних рахунків від різних постачальників послуг та ризик виникнення прострочень із подальшими нарахуваннями пені є поширеними проблемами, що створюють незручності та можуть призводити до фінансових втрат. У цьому контексті розробка зручних та функціональних мобільних додатків для управління комунальними платежами набуває особливої значущості, пропонуючи користувачам ефективні інструменти для автоматизації та спрощення цих рутинних процесів.

Актуальність теми дипломної роботи зумовлена зростаючою потребою користувачів у мобільних рішеннях, що дозволяють не лише фіксувати комунальні витрати, але й отримувати своєчасні нагадування про необхідність їх сплати, тим самим оптимізуючи особистий бюджет та запобігаючи небажаним фінансовим наслідкам. Розробка такого програмного продукту відповідає сучасним тенденціям цифровізації повсякденного життя та підвищення фінансової грамотності населення.

Метою дипломної роботи є розробка програмного забезпечення мобільного додатку для введення комунальних платежів з функцією нагадування, який надасть користувачам зручний та надійний інструмент для управління своїми комунальними зобов'язаннями.

Для досягнення поставленої мети необхідно вирішити наступні **завдання**:

- провести аналіз предметної області, вивчити існуючі мобільні додатки для управління платежами та визначити їх переваги та недоліки;
- сформулювати основні функціональні та нефункціональні вимоги до розроблюваного програмного продукту;
- спроєктувати архітектуру мобільного додатку, включаючи клієнтську та серверну частини, а також структуру бази даних;
- розробити та реалізувати основні програмні модулі мобільного додатку, використовуючи сучасні технології та інструменти;
- реалізувати ключовий функціонал створення платежів, їх категоризації, управління гаманцями та налаштування системи нагадувань;
- провести тестування розробленого програмного забезпечення для забезпечення його коректної роботи та відповідності вимогам;
- розробити рекомендації щодо впровадження та подальшого розвитку додатку.

Об'єктом дослідження є процес управління комунальними платежами за допомогою мобільних технологій.
Предметом дослідження є програмне забезпечення мобільного додатку для введення, обліку комунальних платежів та налаштування системи нагадувань про їх сплату.

При виконанні роботи використовувалися наступні **методи та технології**. Методологічною основою дослідження є системний підхід, методи аналізу та синтезу, об'єктно-орієнтоване проєктування та принципи розробки програмного забезпечення. Технологічний стек включає: для серверної частини – Node.js, фреймворк Express.js, реляційну базу даних PostgreSQL (з використанням хмарної платформи Neon) та мову програмування TypeScript; для клієнтської частини – фреймворк React Native, платформу Expo, мову програмування

TypeScript, бібліотеку для управління станом Zustand, HTTP-клієнт Axios та систему для локальних сповіщень expo-notifications.

Наукова новизна роботи полягає у розробці мобільного додатку з комплексним підходом до управління комунальними платежами, що поєднує функціонал обліку, категоризації, управління гаманцями та гнучку систему нагадувань, реалізованого на сучасному стеку технологій з акцентом на зручність користувацького інтерфейсу та надійність зберігання даних. **Практичне значення** отриманих результатів полягає у створенні готового до використання програмного продукту, який може бути застосований широким колом користувачів для ефективного управління комунальними платежами, запобігання простроченням та підвищення особистої фінансової дисципліни.

1. СИСТЕМНИЙ АНАЛІЗ ПРОБЛЕМНОЇ ОБЛАСТІ

Сучасний ритм життя вимагає від людини не тільки високої продуктивності у професійній діяльності, але й ефективного управління особистими справами, серед яких значне місце займають фінансові зобов'язання. Управління комунальними платежами та особистими фінансами в цілому є невід'ємною частиною повсякденного життя, що вимагає точності, своєчасності та системного підходу.

1.1 Опис предметної області

Управління особистими фінансами охоплює широкий спектр діяльності, спрямованої на планування, облік, контроль та оптимізацію грошових потоків індивіда або домогосподарства. Ключовою метою такого управління є досягнення фінансових цілей, забезпечення стабільності та уникнення небажаних боргів. Одним із важливих та регулярних компонентів особистих витрат є комунальні платежі – оплата за послуги, що забезпечують комфортне проживання: електроенергія, водопостачання та водовідведення, газопостачання, опалення, інтернет, телебачення, вивіз сміття та інші

Процес управління комунальними платежами традиційно включає отримання рахунків від різних постачальників послуг, відстеження термінів їх сплати, здійснення оплати через банківські установи, платіжні термінали або онлайн-сервіси, а також зберігання квитанцій. Цей процес може бути часозатратним та складним, особливо при великій кількості різноманітних платежів з різними датами та реквізитами. Несвоєчасна сплата комунальних послуг може призвести до нарахування пені, тимчасового або повного припинення надання послуг, а також негативно вплинути на кредитну історію.

В умовах цифровізації суспільства з'являється все більше інструментів, покликаних спростити управління особистими фінансами. Мобільні додатки стають популярним рішенням, оскільки дозволяють користувачам мати постійний доступ до своєї фінансової інформації, здійснювати операції та отримувати важливі сповіщення незалежно від місця та часу. Предметна область розробки мобільного додатку для введення комунальних платежів з функцією нагадування знаходиться на перетині управління особистими фінансами та розробки мобільних програмних рішень. Такий додаток має на меті автоматизувати частину рутинних операцій, пов'язаних з обліком та контролем комунальних платежів, та, що найважливіше, забезпечити своєчасне нагадування про необхідність їх здійснення, мінімізуючи ризики прострочення.

1.2 Аналіз вимог до програмної системи

Для успішної розробки програмного продукту, який буде ефективно вирішувати поставлені завдання та задовольняти потреби користувачів, необхідно чітко визначити набір вимог до нього. Ці вимоги поділяються на функціональні, що описують конкретні дії, які система повинна виконувати, та нефункціональні, що визначають якісні характеристики системи.

1.2.1. Функціональні вимоги

Мобільний додаток для введення комунальних платежів з функцією нагадування повинен надавати користувачеві наступні функціональні можливості:

1. Управління платежами:
   - Створення нового платежу: Користувач повинен мати можливість додати новий платіж, вказавши наступні атрибути:
     - Назва послуги (наприклад, "Електроенергія", "Інтернет Київстар").
     - Сума платежу.

        - Валюта платежу (із можливістю вибору зі списку, наприклад, UAH, USD, EUR).
        - Термін оплати (дата, до якої необхідно здійснити платіж).
        - Категорія платежу (вибір із попередньо створеного списку або можливість додати нову).
        - Гаманець (вибір із попередньо створеного списку або можливість додати новий, наприклад, "Картка ПриватБанк", "Готівка").
        - Статус платежу (із можливістю вибору: "Очікує", "Сплачено", "Прострочено").
        - Регулярність платежу (наприклад, одноразовий, щомісячний, щорічний) для можливості автоматичного створення майбутніх платежів або нагадувань.
    - Редагування існуючого платежу: Можливість зміни будь-якого з атрибутів раніше створеного платежу.
    - Видалення платежу: Можливість видалення непотрібних або помилково створених платежів.
    - Перегляд списку платежів: Відображення всіх платежів зі можливістю фільтрації та сортування (наприклад, за датою, статусом, категорією).
    - Перегляд деталей платежу: Відображення повної інформації про обраний платіж.
2. Управління категоріями платежів:
    - Створення нової категорії: Можливість додавання користувацьких категорій (наприклад, "Оренда", "Вода", "Газ") з можливістю вибору іконки для візуального розрізнення.
    - Редагування категорії: Можливість зміни назви та іконки існуючої категорії.
    - Видалення категорії: Можливість видалення категорії (з вирішенням питання, що робити з платежами, які належали до цієї категорії –

наприклад, перепризначення до категорії "Без категорії" або попередження користувача).

3. Управління гаманцями:
    - Створення нового гаманця: Можливість додавання користувацьких гаманців, з яких здійснюється оплата (наприклад, "Монобанк Картка", "Гаманець WebMoney").
    - Редагування гаманця: Можливість зміни назви існуючого гаманця.
    - Видалення гаманця: Можливість видалення гаманця (з аналогічним вирішенням питання щодо пов'язаних платежів).
4. Система нагадувань:
    - Налаштування нагадувань для платежів: Можливість увімкнути/вимкнути нагадування для конкретного платежу.
    - Встановлення часу нагадування: Користувач повинен мати можливість вказати, за скільки днів (або годин) до терміну оплати потрібно отримати сповіщення (наприклад, за 1 день, за 3 дні).
    - Отримання Push-сповіщень: Додаток повинен надсилати локальні push-сповіщення на пристрій користувача про наближення терміну оплати.
5. Локалізація:
    - Підтримка мови інтерфейсу: Інтерфейс додатку повинен бути доступний українською мовою.
6. Управління даними:
    - Очищення даних: Можливість видалення всіх даних користувача з додатку (всіх платежів, категорій, гаманців, налаштувань нагадувань) після відповідного підтвердження.
    - Синхронізація з сервером: Забезпечення збереження даних користувача (платежі, категорії, гаманці, налаштування) на віддаленому сервері та їх автоматична або ручна синхронізація з мобільним пристроєм. Це дозволить відновити дані при зміні пристрою або перевстановленні додатку.

1.2.2. Нефункціональні вимоги

Нефункціональні вимоги визначають якісні атрибути системи, які впливають на загальний досвід користувача та надійність програмного продукту:

1. Продуктивність:
    - Додаток повинен швидко запускатися та реагувати на дії користувача (наприклад, відкриття екранів, збереження даних) без відчутних затримок (час відгуку не більше 1-2 секунд для більшості операцій).
    - Ефективне використання ресурсів пристрою (пам'ять, процесор), щоб не спричиняти швидкого розрядження батареї.
    - Серверна частина повинна ефективно обробляти запити від клієнтів навіть при зростанні навантаження.
2. Надійність:
    - Стабільна робота додатку без неочікуваних збоїв, зависань або втрати даних.
    - Коректне збереження всієї введеної користувачем інформації як локально, так і на сервері (при синхронізації).
    - Система нагадувань повинна спрацьовувати точно у встановлений час.
    - Механізми обробки помилок (наприклад, при відсутності інтернет-з'єднання під час синхронізації) з наданням зрозумілих повідомлень користувачеві.
3. Безпека:
    - Захист персональних та фінансових даних користувача, що зберігаються локально та передаються на сервер.
    - Використання безпечних протоколів передачі даних (HTTPS) при взаємодії з сервером.

- Якщо реалізовано автентифікацію, паролі користувачів повинні зберігатися у хешованому вигляді.
- Захист від несанкціонованого доступу до даних у разі втрати пристрою (наприклад, через блокування екрану пристрою; можливість реалізації додаткового PIN-коду або біометричної автентифікації для входу в додаток може розглядатися як розширення).

4. Зручність використання (Usability):

    - Інтуїтивно зрозумілий та простий у навігації інтерфейс користувача.
    - Логічна та послідовна структура меню та екранів.
    - Чіткі та зрозумілі назви елементів керування та повідомлень.
    - Мінімальна кількість кроків для виконання основних операцій (наприклад, додавання нового платежу).
    - Візуальна привабливість та відповідність сучасним гайдлайнам дизайну мобільних додатків.

5. Сумісність:

    - Коректна робота додатку на мобільних пристроях під управлінням операційних систем Android та iOS (завдяки використанню React Native).
    - Адаптація інтерфейсу під різні розміри та роздільні здатності екранів.

6. Масштабованість:

    - Клієнтська частина: Архітектура клієнтської частини повинна дозволяти додавання нового функціоналу без значної перебудови існуючого коду.

- Серверна частина: Архітектура серверної частини та бази даних повинна бути спроєктована з урахуванням можливого зростання кількості користувачів та обсягу збережених даних без суттєвого погіршення продуктивності.

7. Підтримка та Обслуговування (Maintainability):

- Код додатку повинен бути добре структурованим, читабельним та документованим (коментарі в коді) для полегшення його подальшої підтримки, модифікації та виправлення помилок.
- Модульність коду, що дозволяє легко ізолювати та змінювати окремі компоненти системи

1.3 Моделювання предметної області

Моделювання предметної області є важливим етапом проєктування будь-якої програмної системи, оскільки воно дозволяє структурувати інформацію про сферу застосування, визначити ключові сутності, їх взаємозв'язки та процеси. Для мобільного додатку, призначеного для введення комунальних платежів з функцією нагадування, моделювання допомагає чітко окреслити межі системи, її взаємодію із зовнішнім середовищем та основні сценарії використання.

1.3.1. Контекстна діаграма взаємодії системи

Контекстна діаграма (діаграма рівня 0 у методології DFD або аналогічна діаграма в інших нотаціях) візуалізує систему як єдиний процес та її взаємодію із зовнішніми сутностями (акторами). Вона показує основні потоки даних між системою та її оточенням.

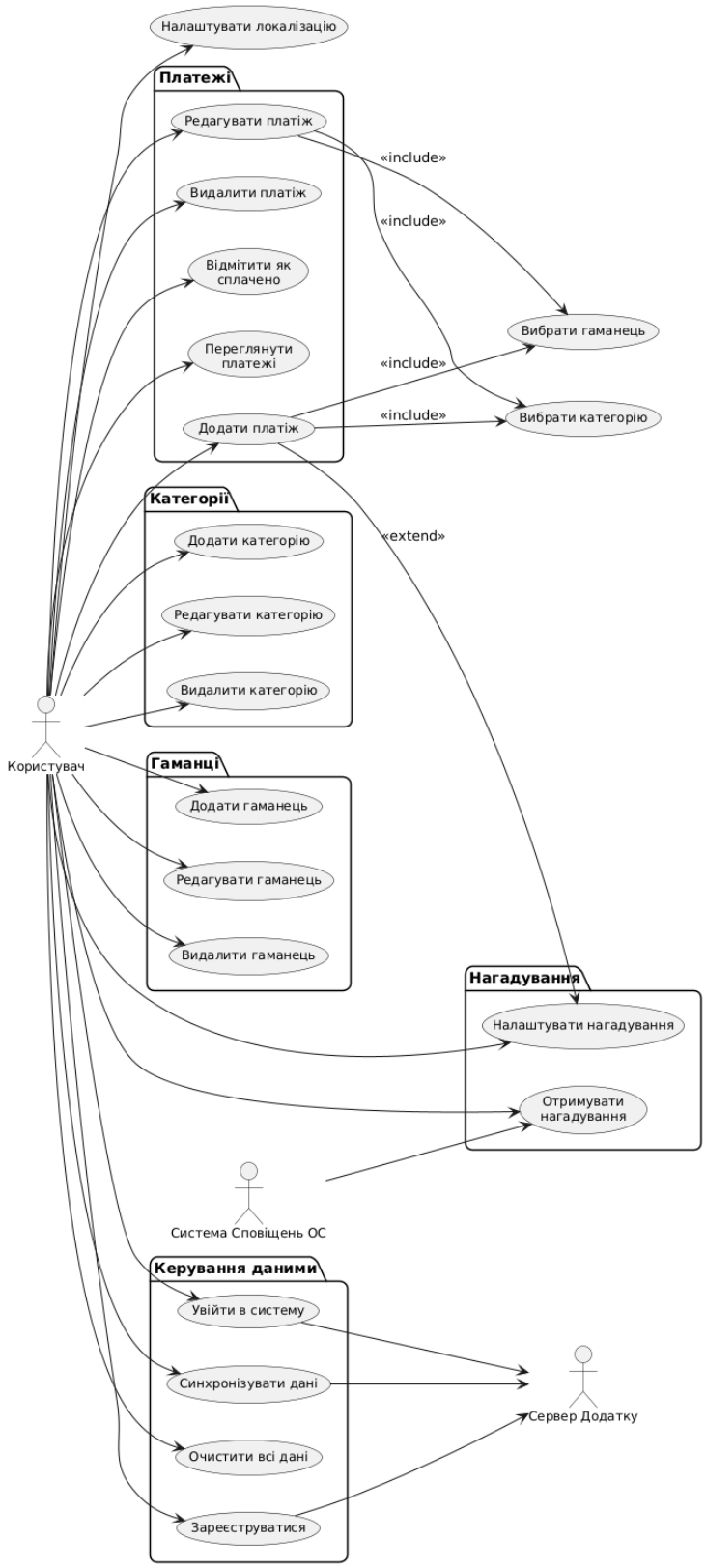
Для мобільного додатку для введення комунальних платежів з функцією нагадування контекстна діаграма може виглядати наступним чином (описово, для візуалізації потрібен графічний редактор):

- Центральний процес: "Мобільний додаток для управління комунальними платежами".
- Зовнішні сутності (Актори):
  - Користувач: Основний актор, що взаємодіє з додатком.
    - Вхідні потоки від Користувача до Системи: Дані про платежі (назва, сума, термін, категорія, гаманець), налаштування категорій, налаштування гаманців, налаштування нагадувань, запити на очищення даних, дані для автентифікації (якщо є).
    - Вихідні потоки від Системи до Користувача: Відображення списку платежів, деталі платежу, сповіщення-нагадування, підтвердження операцій, повідомлення про помилки, відображення категорій/гаманців.
  - Сервер додатку (Backend): Зовнішня система, з якою мобільний додаток синхронізує дані.
    - Вхідні потоки від Системи (клієнта) до Сервера: Дані користувача для синхронізації (платежі, категорії, гаманці, налаштування), запити на автентифікацію.
    - Вихідні потоки від Сервера до Системи (клієнта): Підтвердження синхронізації, оновлені дані з сервера, результати автентифікації.
  - Операційна система мобільного пристрою: Забезпечує платформу для роботи додатку та сервіс сповіщень.
    - Вхідні потоки від Системи до ОС: Запити на показ push-сповіщень, запити на доступ до локального сховища.
    - Вихідні потоки від ОС до Системи: Системні події, доставка push-сповіщень.

Ця діаграма ілюструє, що додаток отримує дані від користувача, взаємодіє з сервером для зберігання та синхронізації цих даних, а також використовує можливості операційної системи для реалізації функціоналу нагадувань.

1.3.2. Діаграма варіантів використання (Use Case Diagram) для мобільного додатку

Діаграма варіантів використання (Use Case Diagram) деталізує функціональність системи з точки зору користувача, показуючи, які дії (варіанти використання) може виконувати користувач та інші актори, взаємодіючи з системою.



Актори:

- Користувач (основний актор).

- Система Сповіщень ОС (відповідає за доставку нагадувань, може розглядатися як допоміжний актор або механізм).
- Сервер Додатку (для синхронізації даних).

Основні варіанти використання (Use Cases) для Користувача:

1. Керувати платежами:
    - Додати платіж (включає вибір категорії, гаманця, встановлення терміну).
    - Редагувати платіж.
    - Видалити платіж.
    - Переглянути список платежів (з можливістю фільтрації/сортування).
    - Відмітити платіж як "Сплачено".
2. Керувати категоріями:
    - Додати категорію.
    - Редагувати категорію.
    - Видалити категорію.
3. Керувати гаманцями:
    - Додати гаманець.
    - Редагувати гаманець.
    - Видалити гаманець.
4. Налаштувати нагадування:
    - Увімкнути/Вимкнути нагадування для платежу.
    - Встановити час попередження для нагадування.
5. Отримувати нагадування (пасивний варіант використання для користувача, ініціюється системою).

6. Керувати даними додатку:
    - Очистити всі дані.
    - (Якщо є автентифікація) Зареєструватися в системі.
    - (Якщо є автентифікація) Увійти в систему.
    - (Якщо є автентифікація) Синхронізувати дані з сервером.
7. Налаштувати локалізацію (хоча у вимогах вказана українська за замовчуванням, цей Use Case може бути доданий для майбутнього розширення або якщо є вибір мови).

1.3.3. Опис основних бізнес-процесів, що автоматизуються

Мобільний додаток для введення комунальних платежів з функцією нагадування автоматизує та оптимізує наступні ключові бізнес-процеси користувача:

1. Процес введення та обліку нового комунального платежу:
    - Початок: Користувач отримує рахунок за комунальну послугу або дізнається про необхідність оплати.
    - Дії в додатку: Користувач відкриває додаток, ініціює створення нового платежу, вводить всю необхідну інформацію (назва, сума, термін, обирає або створює категорію/гаманець), налаштовує параметри нагадування.
    - Результат: Платіж збережено в системі, встановлено нагадування. Користувач має структуровану інформацію про майбутній платіж.
    - Автоматизація: Додаток усуває необхідність вести облік на папері або в неспеціалізованих програмах, забезпечує централізоване зберігання інформації.
2. Процес відстеження термінів оплати та отримання нагадувань:

- Початок: Наближається термін оплати для одного або декількох збережених платежів.
- Дії в додатку (автоматичні): Система перевіряє терміни платежів та налаштування нагадувань. У визначений час додаток генерує та надсилає користувачеві push-сповіщення про необхідність оплати.
- Дії користувача: Користувач отримує нагадування, переглядає деталі платежу в додатку.
- Результат: Користувач своєчасно поінформований про необхідність оплати, що мінімізує ризик прострочення.
- Автоматизація: Додаток бере на себе функцію контролю термінів, звільняючи користувача від необхідності постійно пам'ятати про них.

3. Процес актуалізації статусу платежу:

- Початок: Користувач здійснив оплату комунальної послуги.
- Дії в додатку: Користувач знаходить відповідний платіж у списку та змінює його статус на "Сплачено".
- Результат: Інформація в додатку актуалізована, платіж більше не відображається як активний для нагадувань (якщо це передбачено логікою).
- Автоматизація: Дозволяє вести точний облік сплачених та несплачених рахунків безпосередньо в додатку.

4. Процес аналізу та організації комунальних витрат:

- Початок: Користувач бажає проаналізувати свої витрати на комунальні послуги.
- Дії в додатку: Користувач переглядає історію платежів, використовує фільтри за категоріями, гаманцями, періодами.

- Результат: Користувач отримує уявлення про структуру своїх комунальних витрат.
- Автоматизація: Додаток автоматично групує платежі за категоріями та гаманцями, спрощуючи аналіз.

5. Процес синхронізації даних (якщо реалізовано):

- Початок: Користувач вносить зміни в дані (додає платіж, змінює налаштування) або відкриває додаток після тривалої перерви.
- Дії в додатку (автоматичні/ручні): Додаток ініціює процес синхронізації даних з віддаленим сервером.
- Результат: Дані користувача актуалізовані та збережені на сервері, забезпечуючи їх доступність з різних пристроїв та можливість відновлення.
- Автоматизація: Забезпечує цілісність та безпеку даних користувача.

1.4. Огляд інформаційних джерел та існуючих рішень-аналогів (аналіз переваг та недоліків)

Для обґрунтування необхідності розробки нового мобільного додатку для введення комунальних платежів з функцією нагадування та визначення його конкурентних переваг, було проведено аналіз існуючих на ринку програмних рішень та інструментів, які користувачі можуть застосовувати для управління особистими фінансами та платежами.

1. Універсальні додатки для управління особистими фінансами (наприклад, Monefy, Spendee, Wallet by BudgetBakers)

    - Основний функціонал: Ці додатки зазвичай пропонують широкий набір інструментів для відстеження доходів та витрат, створення

бюджетів, аналізу фінансових звичок за допомогою графіків та звітів, синхронізацію з банківськими рахунками (в деяких випадках).

- Переваги:
  - Комплексний підхід до управління всіма аспектами особистих фінансів.
  - Часто мають розвинені інструменти візуалізації та аналітики.
  - Підтримка різних валют та можливість ведення декількох рахунків.
- Недоліки з точки зору поставленого завдання:
  - Спеціалізація на комунальних платежах: Зазвичай відсутня глибока спеціалізація саме на комунальних платежах. Введення регулярних платежів може бути реалізовано, але без специфічних атрибутів, важливих для комуналки (наприклад, деталізація послуги, прив'язка до конкретного постачальника).
  - Система нагадувань: Функція нагадувань може бути присутня, але часто вона є загальною для всіх транзакцій або запланованих витрат, без гнучкого налаштування саме для періодичних комунальних платежів (наприклад, нагадування за X днів до конкретної дати місяця).
  - Надлишковість функціоналу: Для користувачів, яким потрібен лише інструмент для контролю комуналки, такий додаток може здатися перевантаженим функціями.
  - Інтерфейс: Може бути не настільки інтуїтивним для швидкого введення саме комунальних платежів.

2. Мобільні додатки банків (наприклад, Приват24, monobank, додатки інших банків)

- Основний функціонал: Надають доступ до банківських рахунків, дозволяють здійснювати перекази, оплачувати послуги (включаючи комунальні за шаблонами або через пошук компаній), переглядати історію транзакцій.
- Переваги:
  - Пряма оплата: Можливість не тільки обліковувати, але й одразу оплачувати комунальні послуги.
  - Безпека: Зазвичай високий рівень безпеки, оскільки це банківські продукти.
  - Автоматичне отримання деяких рахунків: Деякі банки можуть автоматично підтягувати рахунки від певних постачальників.
- Недоліки з точки зору поставленого завдання:
  - Прив'язка до банку: Основний недолік – користувач може користуватися послугами кількох банків або мати платежі, які незручно оплачувати через конкретний банківський додаток. Відсутня можливість агрегації всіх комунальних платежів в одному місці, якщо вони оплачуються з різних джерел.
  - Обмежена система нагадувань: Нагадування зазвичай стосуються кредитних лімітів, регулярних платежів, налаштованих у банку, але рідко надають гнучку систему для всіх *внесених вручну* комунальних зобов'язань, які користувач планує сплатити.
  - Обмеженість категоризації та аналітики: Функціонал категоризації та аналізу витрат може бути менш розвиненим порівняно зі спеціалізованими фінансовими додатками, особливо для платежів, що не проходять через цей банк.

3. Таск-менеджери та календарі (наприклад, Google Calendar, Microsoft To Do, Todoist, Trello)
    - Основний функціонал: Дозволяють створювати завдання, встановлювати для них терміни виконання та нагадування.
    - Переваги:
        - Універсальність та гнучкість: Можна створити завдання для будь-якої події, включаючи оплату комунальних послуг.
        - Звичний інтерфейс: Багато користувачів вже активно використовують ці інструменти для інших цілей.
        - Надійна система нагадувань: Зазвичай мають добре реалізовані механізми сповіщень.
    - Недоліки з точки зору поставленого завдання:
        - Відсутність фінансового обліку: Ці інструменти не призначені для ведення фінансового обліку. Неможливо вказати суму, валюту, категорію платежу, гаманець, відстежувати статус оплати.
        - Ручне введення: Кожне нагадування та кожен платіж потрібно вводити як окреме завдання, що може бути незручно для регулярних платежів.
        - Відсутність аналітики: Немає можливості аналізувати структуру комунальних витрат.
        - Немає синхронізації фінансових даних: Дані про платежі не зберігаються централізовано як фінансова інформація.
4. Спеціалізовані додатки для нагадувань про рахунки (наприклад, Bills Reminder, Bill Organizer & Reminder)

- Основний функціонал: Сфокусовані на відстеженні термінів оплати різних рахунків та надсиланні нагадувань. Можуть мати функції додавання сум, категорій.
- Переваги:
  - Цільове призначення: Розроблені спеціально для нагадувань про платежі, тому цей функціонал зазвичай добре реалізований.
  - Простий інтерфейс: Часто мають простий та зрозумілий інтерфейс, орієнтований на одну задачу.
- Недоліки з точки зору поставленого завдання:
  - Обмежений фінансовий менеджмент: Функції управління категоріями, гаманцями, аналітики витрат можуть бути відсутні або дуже базовими.
  - Якість реалізації: Ринок таких додатків насичений, і якість (стабільність, дизайн, наявність реклами) може сильно варіюватися.
  - Синхронізація даних: Можливість надійної синхронізації даних з сервером для резервного копіювання та використання на кількох пристроях може бути відсутня або реалізована ненадійно.
  - Локалізація та підтримка валют: Не всі подібні додатки якісно локалізовані та підтримують необхідні валюти.

Висновок аналізу існуючих рішень:

Проведений аналіз показує, що хоча на ринку існує значна кількість додатків для управління фінансами та нагадуваннями, більшість з них або є надто загальними та перевантаженими функціями, не завжди зручними для

специфічної задачі обліку комунальних платежів (як універсальні фінансові менеджери), або навпаки, занадто вузькоспеціалізованими лише на нагадуваннях без належного фінансового обліку (як прості таск-менеджери чи деякі біл-ремайндери), або ж прив'язані до екосистеми конкретного банку.

Таким чином, існує потреба в розробці мобільного додатку, який би поєднував у собі:

- Спеціалізацію на зручному введенні та обліку саме комунальних платежів з усіма необхідними атрибутами.
- Гнучку та надійну систему нагадувань, налаштовану під потреби користувача.
- Базовий, але достатній функціонал для організації платежів за допомогою категорій та гаманців.
- Простий та інтуїтивно зрозумілий інтерфейс, не перевантажений зайвими елементами.
- Надійну синхронізацію даних з сервером для забезпечення їх збереження та доступності.

Розроблюваний мобільний додаток для введення комунальних платежів з функцією нагадування покликаний заповнити цю нішу, пропонуючи користувачам збалансоване рішення, сфокусоване на ефективному управлінні їхніми комунальними зобов'язаннями.

1.5. Постановка завдання на розробку програмного забезпечення

На основі проведеного аналізу предметної області, виявлених потреб користувачів та недоліків існуючих рішень, ставиться завдання розробити програмне забезпечення мобільного додатку для введення комунальних платежів з функцією нагадування.

Основні завдання розробки:

1. Спроєктувати та реалізувати інтуїтивно зрозумілий мобільний інтерфейс користувача, що забезпечить легке введення даних про комунальні платежі, їх редагування та перегляд.
2. Розробити функціонал для управління категоріями платежів та гаманцями, що дозволить користувачам гнучко організовувати свої фінансові операції.
3. Імплементувати надійну та гнучку систему нагадувань про майбутні платежі, з можливістю налаштування часу попередження.
4. Забезпечити можливість зміни статусу платежу (очікує, сплачено, прострочено) для ведення актуального обліку.
5. Реалізувати функцію очищення всіх даних у додатку за бажанням користувача.
6. Забезпечити локалізацію інтерфейсу українською мовою.
7. Розробити клієнт-серверну архітектуру з використанням React Native (Expo) для клієнтської частини та Node.js (Express.js) з PostgreSQL (Neon) для серверної частини з метою забезпечення синхронізації та збереження даних користувача.
8. Забезпечити відповідність розробленого програмного продукту визначеним функціональним та нефункціональним вимогам, зокрема щодо продуктивності, надійності та безпеки даних.
9. Провести тестування розробленого додатку для виявлення та усунення можливих дефектів.
10. Підготувати документацію, включаючи інструкцію користувача.

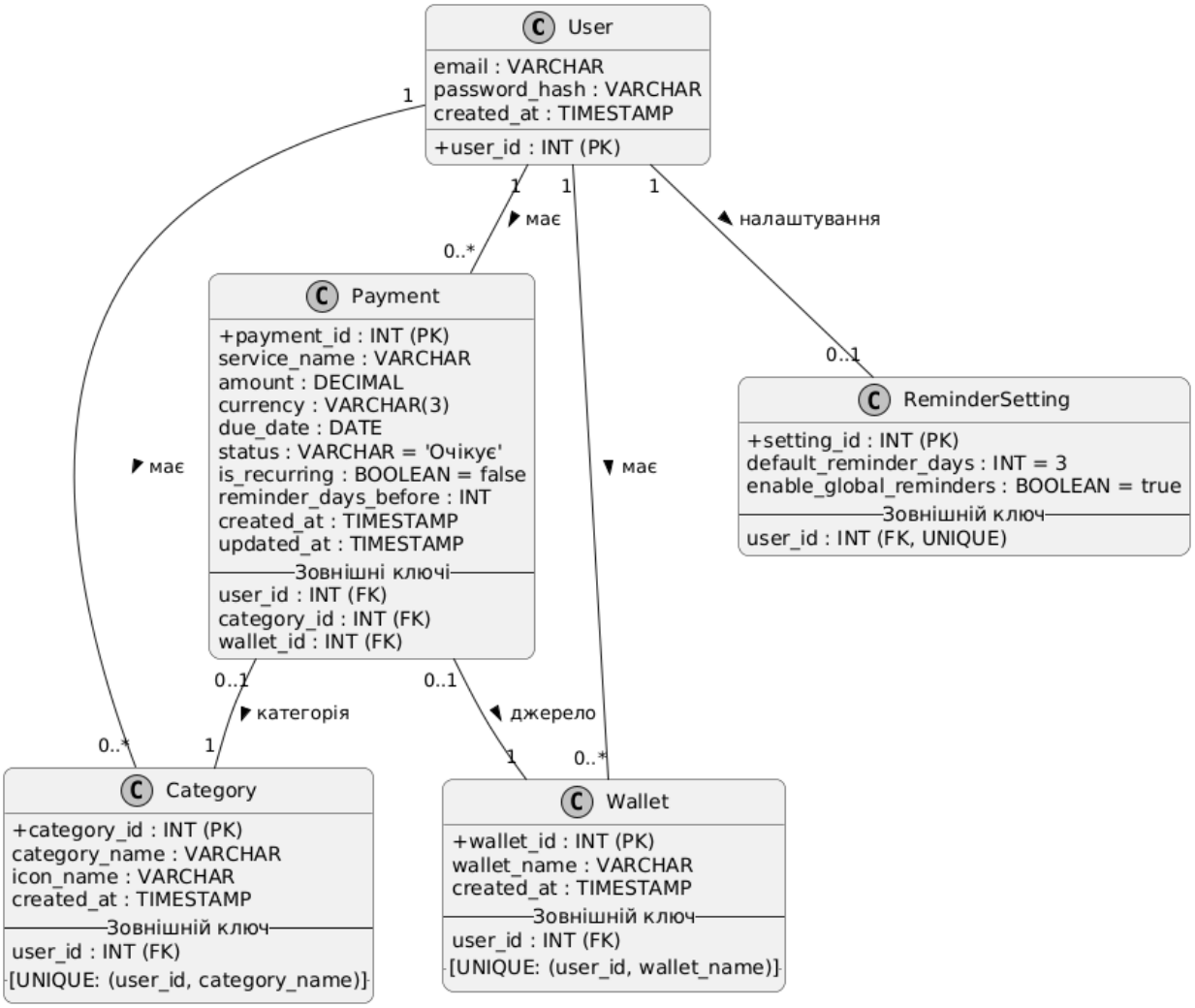
Виконання цих завдань дозволить створити програмний продукт, що ефективно вирішуватиме проблему своєчасного та організованого управління комунальними платежами.

2. ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Після визначення вимог до системи та аналізу предметної області, наступним критично важливим етапом є проєктування інформаційного та програмного забезпечення. Цей розділ присвячений розробці структури даних, архітектури системи та деталізації її компонентів для мобільного додатку введення комунальних платежів з функцією нагадування.

2.1. Логічна модель даних у вигляді ER-діаграми для бази даних PostgreSQL

Основою для зберігання та управління інформацією в додатку є база даних. Для її проєктування розробляється логічна модель даних, яка візуалізується за допомогою ER-діаграми (Entity-Relationship Diagram). Ця модель визначає основні сутності, їхні атрибути та зв'язки між ними, незалежно від конкретної СУБД, хоча при її розробці враховувалися особливості реляційної бази даних PostgreSQL, обраної для реалізації.



2.1.1. Опис сутностей, атрибутів та зв'язків

На основі функціональних вимог, визначено наступні ключові сутності:

1. Користувач (User): Представляє зареєстрованого користувача системи (якщо реалізовано автентифікацію та синхронізацію).
    - user_id (PK, INT, SERIAL): Унікальний ідентифікатор користувача (первинний ключ).
    - email (VARCHAR, UNIQUE, NOT NULL): Електронна пошта користувача, використовується для входу.
    - password_hash (VARCHAR, NOT NULL): Хешований пароль користувача.

- created_at (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP): Дата та час створення облікового запису.

2. Платіж (Payment): Представляє окремий комунальний платіж.
    - payment_id (PK, INT, SERIAL): Унікальний ідентифікатор платежу.
    - user_id (FK, INT, NOT NULL): Зовнішній ключ, що посилається на user_id в таблиці User (вказує, якому користувачеві належить платіж).
    - service_name (VARCHAR, NOT NULL): Назва послуги (наприклад, "Електроенергія").
    - amount (DECIMAL, NOT NULL): Сума платежу.
    - currency (VARCHAR(3), NOT NULL): Валюта платежу (наприклад, "UAH").
    - due_date (DATE, NOT NULL): Термін оплати.
    - category_id (FK, INT, NULLABLE): Зовнішній ключ, що посилається на category_id в таблиці Category.
    - wallet_id (FK, INT, NULLABLE): Зовнішній ключ, що посилається на wallet_id в таблиці Wallet.
    - status (VARCHAR, NOT NULL, DEFAULT 'Очікує'): Статус платежу ("Очікує", "Сплачено", "Прострочено").
    - is_recurring (BOOLEAN, DEFAULT FALSE): Ознака регулярності платежу.
    - reminder_days_before (INT, NULLABLE): За скільки днів нагадувати (якщо NULL – нагадування вимкнене для цього платежу).
    - created_at (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP): Дата та час створення запису.
    - updated_at (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP): Дата та час останнього оновлення.

3. Категорія (Category): Представляє категорію платежу.
    - category_id (PK, INT, SERIAL): Унікальний ідентифікатор категорії.
    - user_id (FK, INT, NOT NULL): Зовнішній ключ, що посилається на user_id в таблиці User.
    - category_name (VARCHAR, NOT NULL): Назва категорії (наприклад, "Інтернет").
    - icon_name (VARCHAR, NULLABLE): Назва іконки для візуалізації (якщо використовується).
    - created_at (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP): Дата та час створення.
    - (Обмеження: UNIQUE (user_id, category_name) – назва категорії має бути унікальною для кожного користувача).
4. Гаманець (Wallet): Представляє джерело коштів для оплати.
    - wallet_id (PK, INT, SERIAL): Унікальний ідентифікатор гаманця.
    - user_id (FK, INT, NOT NULL): Зовнішній ключ, що посилається на user_id в таблиці User.
    - wallet_name (VARCHAR, NOT NULL): Назва гаманця (наприклад, "Картка Монобанк").
    - created_at (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP): Дата та час створення.
    - (Обмеження: UNIQUE (user_id, wallet_name) – назва гаманця має бути унікальною для кожного користувача).
5. Налаштування Нагадувань (ReminderSetting)
    - setting_id (PK, INT, SERIAL): Унікальний ідентифікатор налаштування.

- user_id (FK, INT, UNIQUE, NOT NULL): Зовнішній ключ, що посилається на user_id (кожен користувач має один набір налаштувань).
- default_reminder_days (INT, DEFAULT 3): Кількість днів для нагадування за замовчуванням.
- enable_global_reminders (BOOLEAN, DEFAULT TRUE): Увімкнути/вимкнути всі нагадування.

Зв'язки між сутностями:

- User – Payment: Один-до-багатьох (один користувач може мати багато платежів, але кожен платіж належить одному користувачеві). Зв'язок встановлюється через user_id.
- User – Category: Один-до-багатьох (один користувач може створити багато категорій, кожна категорія належить одному користувачеві). Зв'язок через user_id.
- User – Wallet: Один-до-багатьох (один користувач може створити багато гаманців, кожен гаманець належить одному користувачеві). Зв'язок через user_id.
- Category – Payment: Один-до-багатьох (одна категорія може бути присвоєна багатьом платежам, але кожен платіж (якщо має категорію) належить одній категорії). Зв'язок через category_id. Зв'язок є необов'язковим з боку платежу (NULLABLE).
- Wallet – Payment: Один-до-багатьох (один гаманець може бути джерелом для багатьох платежів, але кожен платіж (якщо має гаманець) пов'язаний з одним гаманцем). Зв'язок через wallet_id. Зв'язок є необов'язковим з боку платежу (NULLABLE).
- User – ReminderSetting: Один-до-одного (кожен користувач має один набір глобальних налаштувань нагадувань, якщо ця сутність використовується). Зв'язок через user_id.

2.1.2. Обґрунтування відповідності моделі третій нормальній формі (3NF)

Нормалізація бази даних – це процес організації даних для мінімізації надлишковості та покращення цілісності даних. Третя нормальна форма (3NF) є важливим кроком у цьому процесі. Модель даних знаходиться в 3NF, якщо вона відповідає другій нормальній формі (2NF) і всі неключові атрибути нетранзитивно залежать від первинного ключа.

1. Перша нормальна форма (1NF): Усі атрибути є атомарними (неподільними), і кожна таблиця має первинний ключ. Це виконується для всіх таблиць: значення в кожному полі є одиничним, відсутні повторювані групи.

2. Друга нормальна форма (2NF): Система знаходиться в 1NF, і всі неключові атрибути повністю функціонально залежать від усього первинного ключа.

    - У таблицях User, Category, Wallet, ReminderSetting первинні ключі є простими (user_id, category_id, wallet_id, setting_id відповідно). Отже, всі неключові атрибути автоматично повністю залежать від первинного ключа.
    - У таблиці Payment первинний ключ payment_id також простий. Всі інші атрибути (user_id, service_name, amount тощо) безпосередньо описують саме цей платіж і залежать від payment_id. Зовнішні ключі (user_id, category_id, wallet_id) забезпечують зв'язки, а не часткову залежність неключових атрибутів від частини складеного ключа (якого тут немає).
    - Отже, модель відповідає 2NF.

3. Третя нормальна форма (3NF): Система знаходиться в 2NF, і відсутні транзитивні залежності неключових атрибутів від первинного ключа.

Тобто, жоден неключовий атрибут не залежить від іншого неключового атрибута.

- User: email, password_hash, created_at безпосередньо характеризують користувача (user_id) і не залежать один від одного транзитивно через user_id.
- Payment: service_name, amount, currency, due_date, status, is_recurring, reminder_days_before безпосередньо описують платіж. category_id та wallet_id є зовнішніми ключами, що вказують на інші сутності, а не створюють транзитивну залежність неключових атрибутів всередині самої таблиці Payment. Наприклад, назва категорії зберігається в таблиці Category, а не дублюється в Payment, що запобігає транзитивній залежності.
- Category: category_name, icon_name залежать від category_id і не залежать один від одного.
- Wallet: wallet_name залежить від wallet_id.
- ReminderSetting: default_reminder_days, enable_global_reminders залежать від setting_id (і опосередковано від user_id, оскільки user_id тут є унікальним і визначає налаштування для користувача).
- Можна стверджувати, що транзитивні залежності усунені шляхом винесення пов'язаних даних в окремі таблиці (наприклад, інформація про категорії та гаманці).

Висновок щодо нормалізації: Розроблена логічна модель даних відповідає вимогам третьої нормальної форми. Це забезпечує усунення надлишковості даних, цілісність даних (наприклад, при зміні назви категорії

вона змінюється в одному місці), та полегшує модифікацію структури бази даних у майбутньому.

2.2. Проєктування архітектури програмного забезпечення

Вибір правильної архітектури є фундаментальним для створення надійного, масштабованого та легкого в підтримці мобільного додатку. Для системи управління комунальними платежами з функцією нагадування та синхронізацією даних обрано клієнт-серверну архітектуру.

2.2.1. Обґрунтування вибору клієнт-серверної архітектури

Клієнт-серверна архітектура передбачає розподіл завдань між двома основними компонентами: клієнтом, який запитує послуги або дані, та сервером, який надає ці послуги або дані. Для розроблюваного мобільного додатку такий підхід має низку суттєвих переваг:

1. Централізоване зберігання даних: Усі дані користувачів (платежі, категорії, гаманці, налаштування) зберігаються на сервері в базі даних PostgreSQL. Це забезпечує:
    - Безпеку та надійність даних: Зменшується ризик втрати даних у випадку пошкодження або втрати мобільного пристрою користувача. Можливе регулярне резервне копіювання на сервері.
    - Синхронізацію між пристроями: Користувач може отримати доступ до своїх даних з різних пристроїв (якщо це буде реалізовано в майбутньому) або відновити їх після перевстановлення додатку.
2. Розподіл логіки:

- Клієнт (мобільний додаток): Відповідає за представлення даних (UI), взаємодію з користувачем (UX), збір введених даних, валідацію на стороні клієнта та відображення нагадувань.
- Сервер (Backend): Відповідає за бізнес-логіку обробки даних, взаємодію з базою даних, автентифікацію та авторизацію користувачів, забезпечення цілісності даних.
- Такий розподіл спрощує розробку, тестування та підтримку кожної частини окремо.

3. Масштабованість: Серверну частину можна масштабувати незалежно від клієнтської для обробки зростаючої кількості користувачів та обсягів даних. Можна збільшувати потужність сервера, оптимізувати запити до бази даних тощо. Клієнтські додатки оновлюються окремо.
4. Безпека: Критично важливу бізнес-логіку та доступ до бази даних можна краще захистити на сервері, обмежуючи прямий доступ з клієнтської сторони. Автентифікація та авторизація централізовано керуються сервером.
5. Можливість розширення: Дозволяє в майбутньому підключати інші типи клієнтів (наприклад, веб-додаток) до того ж серверного API.

Хоча для простого додатку без синхронізації можна було б обмежитися лише локальним зберіганням даних на пристрої, вимога синхронізації робить клієнт-серверну архітектуру оптимальним вибором.

2.2.2. Архітектура клієнтської частини (React Native, Expo)

Клієнтська частина – це мобільний додаток, з яким безпосередньо взаємодіє користувач. Він розробляється з використанням фреймворку React Native та платформи Expo.

Основні компоненти архітектури клієнта:

1. Рівень представлення (Presentation Layer / UI):
    - Компоненти React Native: Відповідають за візуальне відображення інтерфейсу (екрани, кнопки, списки, поля введення). Використовуються як стандартні компоненти React Native, так і кастомні, створені для потреб додатку.
    - Навігація: Реалізується за допомогою бібліотеки React Navigation, яка дозволяє визначати структуру екранів та переходи між ними (стекова навігація, таб-навігація тощо).
    - Стилізація: Використовується StyleSheet з React Native або інші підходи для оформлення компонентів (можливо, з використанням expo-vector-icons для іконок).
    - Локалізація: Текстові рядки інтернаціоналізуються за допомогою i18next для підтримки української мови.
2. Рівень управління станом (State Management Layer):
    - Zustand: Бібліотека для глобального управління станом додатку. Використовується для зберігання та оновлення даних, які є спільними для різних компонентів (наприклад, список платежів, поточні налаштування, стан завантаження даних з сервера). useAppStore є точкою доступу до цього стану.
3. Рівень бізнес-логіки клієнта (Client-Side Business Logic):
    - Обробники подій: Функції, що реагують на дії користувача (натискання кнопок, введення тексту).
    - Валідація даних: Перевірка коректності введених користувачем даних перед відправкою на сервер або збереженням локально.
    - Форматування даних: Підготовка даних для відображення або відправки.
    - Логіка роботи з нагадуваннями: Взаємодія з expo-notifications для планування, отримання та обробки локальних сповіщень.

4. Рівень сервісів / взаємодії з даними (Service / Data Access Layer):
    - HTTP-клієнт (Axios): Відповідає за відправку запитів до серверного API (GET, POST, PUT, DELETE) для отримання, створення, оновлення та видалення даних. Обробка відповідей від сервера.
    - Локальне сховище (@react-native-async-storage/async-storage): Використовується для кешування даних, зберігання налаштувань додатку або даних, які мають бути доступні офлайн.
    - Адаптери даних: Можуть використовуватися для перетворення даних, отриманих з сервера, у формат, зручний для використання у стані додатку, і навпаки.
5. Рівень платформних сервісів (Platform Services Layer - через Expo):
    - Expo Modules: Доступ до нативних функцій пристрою через модулі Expo (наприклад, expo-notifications для сповіщень, expo-localization для отримання інформації про локаль пристрою, доступ до файлової системи, якщо потрібно).

Взаємодія компонентів: Користувач взаємодіє з UI-компонентами. Дії користувача викликають обробники подій, які можуть оновлювати локальний стан (через Zustand), взаємодіяти з локальним сховищем або відправляти запити на сервер через Axios. Отримані дані знову оновлюють стан, що призводить до перерендеру UI-компонентів. Нагадування плануються та обробляються через expo-notifications.

2.2.3. Архітектура серверної частини (Node.js, Express.js)

Серверна частина (Backend) відповідає за обробку запитів від мобільного клієнта, взаємодію з базою даних та реалізацію основної бізнес-логіки, пов'язаної з даними.

Основні компоненти архітектури сервера:

1. Рівень маршрутизації (Routing Layer):
    - Express.js Router: Визначає ендпоінти (URL-адреси) API та HTTP-методи (GET, POST, PUT, DELETE), які їм відповідають. Направляє вхідні запити до відповідних контролерів (обробників).
2. Рівень контролерів / обробників запитів (Controller / Request Handler Layer):
    - Функції-обробники Express.js: Для кожного ендпоінта визначається функція, яка приймає об'єкти запиту (req) та відповіді (res).
    - Валідація вхідних даних: Перевірка даних, отриманих від клієнта (параметри запиту, тіло запиту).
    - Виклик сервісів (бізнес-логіки): Передача оброблених даних на рівень сервісів для виконання основних операцій.
    - Формування відповіді: Надсилання HTTP-відповіді клієнту (статус-код, дані у форматі JSON, повідомлення про помилки).
3. Рівень сервісів / бізнес-логіки (Service / Business Logic Layer):
    - Модулі TypeScript/JavaScript: Містять основну логіку додатка, не прив'язану безпосередньо до HTTP-запитів чи бази даних (наприклад, складна логіка розрахунків, специфічні перевірки, координація взаємодії з декількома моделями даних).

- Взаємодіє з рівнем доступу до даних для операцій з БД.

4. Рівень доступу до даних (Data Access Layer / DAL):
    - Модулі для взаємодії з PostgreSQL: Використовується бібліотека pg (node-postgres) для виконання SQL-запитів до бази даних PostgreSQL.
    - Моделі даних (необов'язково формальні ORM-моделі, але можуть бути просто функції для CRUD-операцій): Абстрагують пряму роботу з SQL, надаючи функції типу createPayment, getPaymentById, updateCategory тощо.
    - Формування SQL-запитів та обробка результатів.
5. Рівень бази даних (Database Layer):
    - PostgreSQL (розміщена на Neon): Реляційна база даних, де зберігаються всі дані системи.
6. Проміжне програмне забезпечення (Middleware - Express.js):
    - Обробка CORS: Дозволяє запити з певних доменів (важливо для взаємодії клієнта та сервера).
    - Логування запитів.
    - Обробка помилок (централізована).
    - Парсинг тіла запиту (наприклад, express.json()).
    - Автентифікація/Авторизація (наприклад, перевірка JWT-токенів, якщо реалізовано).
7. Конфігурація (Configuration):
    - dotenv: Завантаження змінних середовища (наприклад, параметри підключення до БД, секретні ключі) з файлу .env.

Взаємодія компонентів: HTTP-запит від клієнта надходить на сервер. Маршрутизатор Express.js визначає, який контролер має обробити цей запит.

Контролер валідує дані, викликає відповідний метод на рівні сервісів. Сервісний рівень реалізує бізнес-логіку та звертається до рівня доступу до даних. DAL виконує SQL-запити до PostgreSQL. Результат повертається назад по ланцюжку, і контролер формує HTTP-відповідь клієнту.

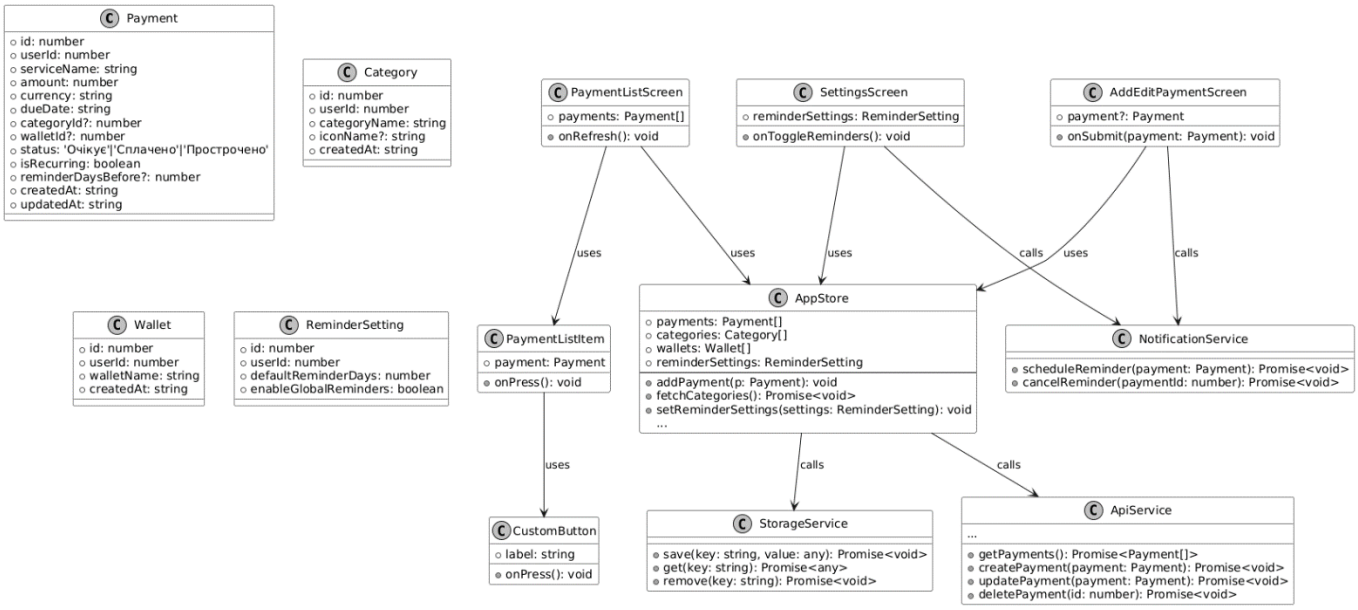
Ця архітектура забезпечує чіткий поділ відповідальностей, що сприяє гнучкості, масштабованості та полегшує підтримку програмного продукту.

2.3. Діаграми UML для деталізації проєкту.

Уніфікована мова моделювання (UML) надає набір стандартизованих діаграм для візуалізації, специфікації, конструювання та документування артефактів програмних систем. У цьому підрозділі розглянемо ключові UML-діаграми, що деталізують проєкт мобільного додатку для введення комунальних платежів з функцією нагадування.

2.3.1. Діаграма класів (для ключових компонентів клієнтської та серверної частини)

Діаграма класів описує статичну структуру системи, показуючи класи, їх атрибути, методи та відносини між ними.



Для клієнтської частини (React Native / Expo): Оскільки React Native базується на компонентному підході, "класи" тут будуть представлені у вигляді:

- Функціональних або класових компонентів React: Це основні будівельні блоки UI. Для ключових екранів (наприклад, AddEditPaymentScreen, PaymentListScreen, SettingsScreen) та важливих пере використовуваних UI-компонентів (наприклад, PaymentListItem, CustomButton) слід визначити їх основні властивості (props) та методи (обробники подій, функції життєвого циклу, якщо це класові компоненти).
- Модулів управління станом: Наприклад, ваш AppStore (реалізований за допомогою Zustand). Тут важливо показати ключові частини стану (наприклад, payments, categories, wallets) та основні дії (actions) для зміни цього стану (наприклад, addPayment, fetchCategories).
- Сервісних класів/модулів: Це модулі, що інкапсулюють певну логіку, наприклад:

- ApiService: Відповідає за взаємодію з REST API сервера (методи для GET, POST, PUT, DELETE запитів до ендпоінтів платежів, категорій, гаманців).
- NotificationService: Інкапсулює логіку роботи з expo-notifications (методи для планування, скасування сповіщень).
- StorageService: Відповідає за взаємодію з @react-native-async-storage/async-storage (методи для збереження, отримання, видалення даних).

- Типів даних (TypeScript interfaces/types): Описати основні структури даних, що використовуються в додатку (наприклад, Payment, Category, Wallet).

Оскільки React Native використовує компонентний підхід, "класи" тут часто представляють собою функціональні або класові компоненти React, а також сервіси або модулі управління станом.

Для серверної частини (Node.js / Express.js):

- Контролери (Express.js маршрутизатори/обробники): Для кожного основного ресурсу (платежі, категорії, гаманці, користувачі, синхронізація) визначити клас контролера з методами, що відповідають HTTP-методам (наприклад, PaymentController з методами getAllPayments, createPayment).
- Сервісний шар: Класи, що інкапсулюють бізнес-логіку. Наприклад, PaymentService, CategoryService, UserService, SyncService. Ці сервіси викликаються контролерами.
- Рівень доступу до даних (DAL/Repositories): Класи, відповідальні за безпосередню взаємодію з базою даних PostgreSQL.

Наприклад, PaymentRepository, CategoryRepository, UserRepository. Вони інкапсулюють SQL-запити або взаємодію з ORM (якщо б вона використовувалася, але у вас pg).

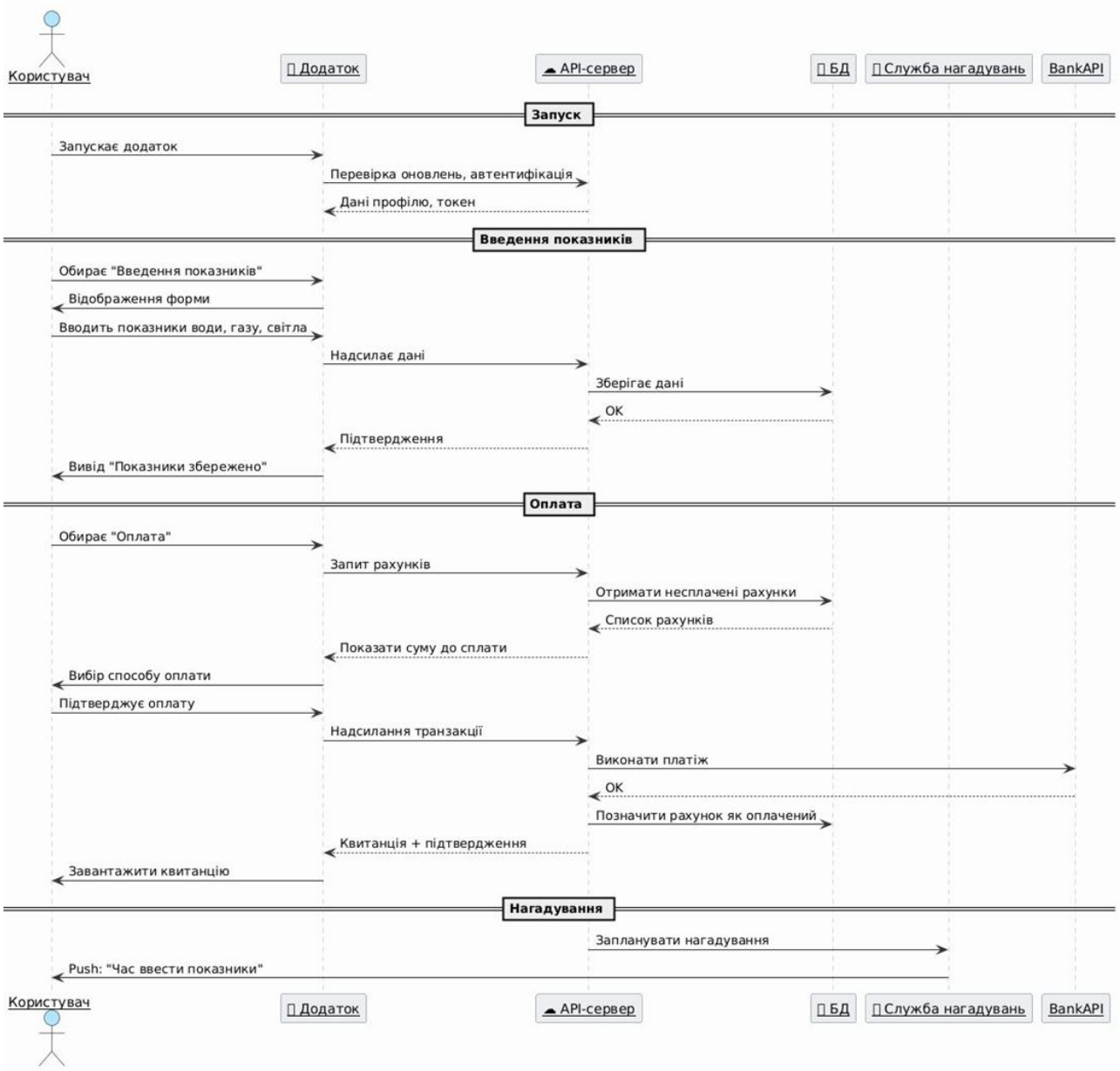
- Моделі/Сутності бази даних: Хоча це не класи в сенсі ООП мови програмування, їх можна представити як сутності, що відображають структуру таблиць у PostgreSQL (наприклад, DB_User, DB_Payment).
- Типи даних/Інтерфейси (TypeScript): Для вхідних та вихідних даних сервісів та контролерів.

Відносини на сервері:

- Контролери залежать від відповідних сервісів.
- Сервіси залежать від відповідних репозиторіїв (DAL).
- Репозиторії взаємодіють із сутностями бази даних.
- Можуть бути асоціації між сутностями бази даних, що відображають зв'язки в ER-діаграмі (наприклад, DB_Payment асоційований з DB_User).

2.3.2. Діаграма послідовності (для основних сценаріїв)

Діаграма послідовності є динамічною діаграмою, яка моделює взаємодію між об'єктами (або екземплярами класів/компонентів) у часовій послідовності. Вона показує порядок надсилання повідомлень (викликів методів) між об'єктами для виконання певного варіанту використання або операції.



Ключові сценарії для моделювання:

1. Створення нового платежу користувачем (з синхронізацією на сервер):

    - Учасники: Користувач (актор), ЕкранДодаванняПлатежу (компонент UI), СховищеСтану (наприклад, AppStore), КлієнтськийApiService, СерверExpress, КонтролерПлатежівНаСервері, СервісПлатежівНаСервері, РепозиторійПлатежівНаСервері, БазаДаних PostgreSQL. Якщо є, також СервісСповіщеньКлієнта.

- Послідовність: Починається з дії користувача на UI, введення даних, виклик методу в сховищі стану, який викликає метод ApiService. ApiService надсилає HTTP-запит на сервер. Сервер обробляє запит через контролер, сервіс, репозиторій, зберігає дані в БД. Відповідь повертається назад клієнту, стан оновлюється, UI відображає результат. Опціонально, планується локальне сповіщення.

2. Отримання нагадування про платіж:

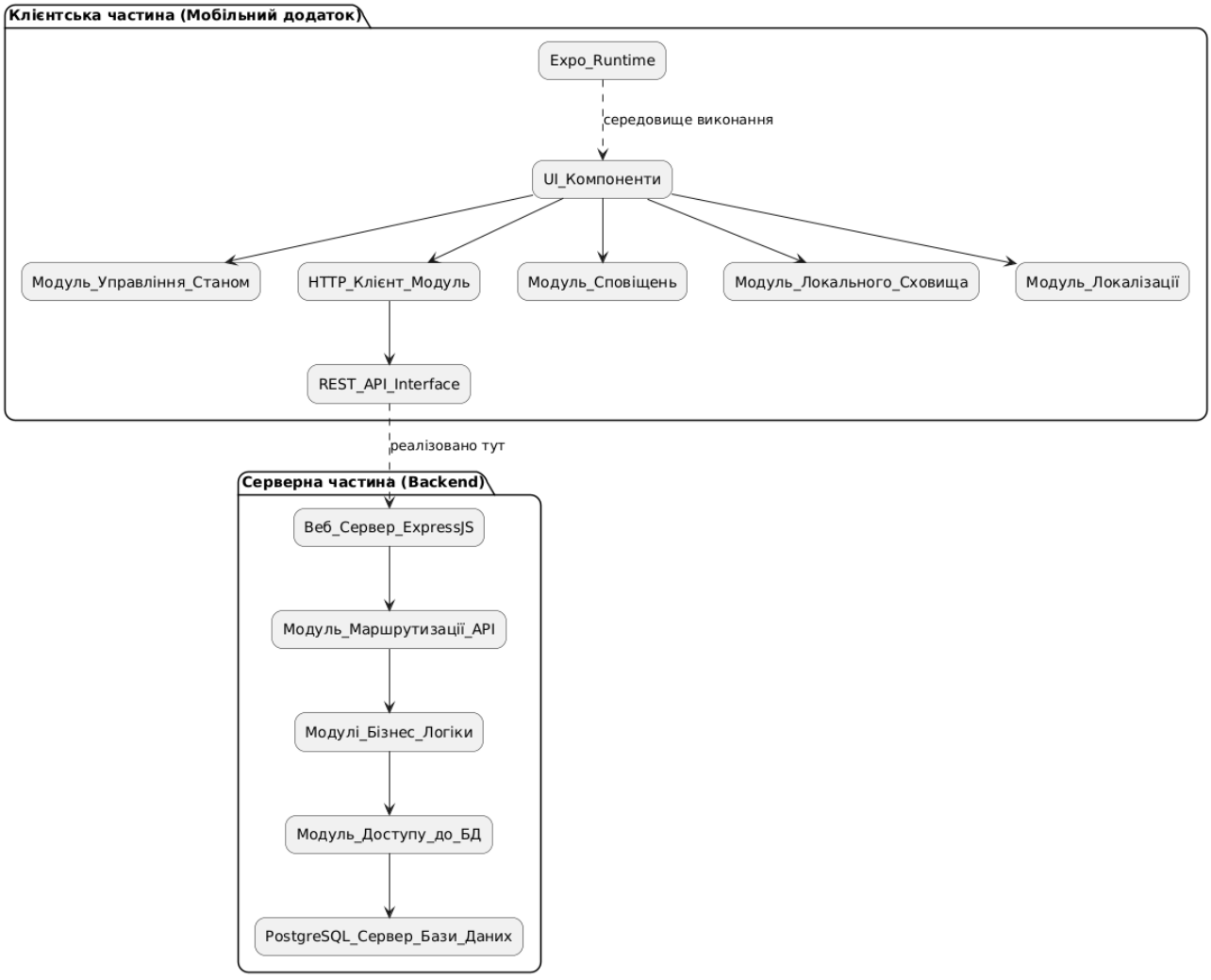
   - Учасники: ОпераційнаСистемаМобільного (джерело системної події), СервісСповіщеньКлієнта (обробляє заплановане сповіщення), МобільнийДодаток (загальний екземпляр для обробки відкриття), Користувач (отримує сповіщення).
   - Послідовність: ОС спрацьовує за таймером, ініціює показ push-сповіщення через клієнтський сервіс сповіщень. Користувач взаємодіє зі сповіщенням. Додаток активується або переходить на відповідний екран для відображення деталей платежу.

3. Синхронізація даних при запуску додатку (якщо є):

   - Учасники: МобільнийДодаток (при старті), СховищеСтану, КлієнтськийApiService, СерверExpress, КонтролерСинхронізаціїНаСервері, СервісСинхронізаціїНаСервері, РепозиторіїДанихНаСервері, БазаДанихPostgreSQL.
   - Послідовність: При старті додаток (через сховище стану або спеціальний сервіс) ініціює запит на синхронізацію до ApiService. Сервер збирає актуальні дані користувача та надсилає їх клієнту. Клієнт оновлює свій локальний стан/сховище.

2.3.3. Діаграма компонентів (відображення структури модулів клієнта та сервера)

Діаграма компонентів показує фізичну структуру системи з точки зору її компонентів та залежностей між ними. Компоненти можуть бути виконуваними файлами, бібліотеками, файлами коду, базами даних тощо. Діаграма ілюструє, як ці компоненти збираються разом для формування системи.



Ключові компоненти:

- Клієнтська частина (Мобільний додаток):
    - UI_Компоненти (скомпільований код React Native, що відповідає за інтерфейс).
    - Модуль_Управління_Станом (наприклад, реалізація на Zustand).

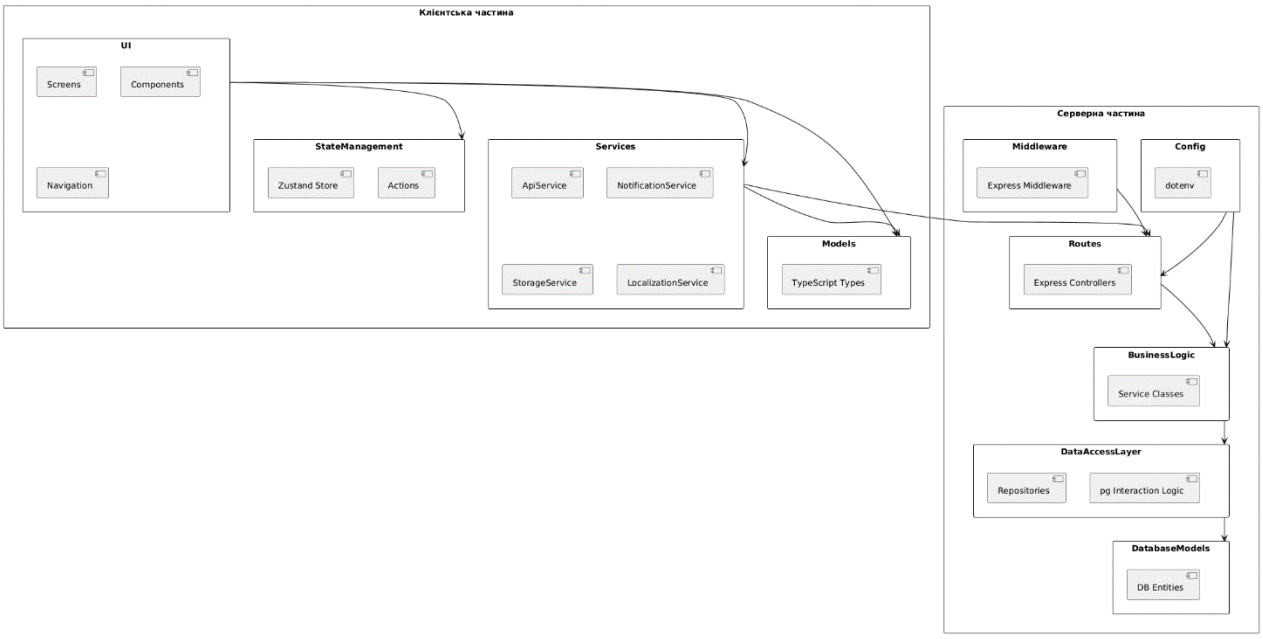
- HTTP_Клієнт_Модуль (бібліотека Axios та обгортки навколо неї).
- Модуль_Сповіщень (на базі expo-notifications).
- Модуль_Локального_Сховища (на базі @react-native-async-storage/async-storage).
- Модуль_Локалізації (на базі i18next).
- (Можливо) Expo Runtime (якщо розглядати середовище виконання).

- Серверна частина (Backend):
  - Веб_Сервер_ExpressJS (основний виконуваний компонент сервера).
  - Модуль_Маршрутизації_API (конфігурація маршрутів Express).
  - Модулі_Бізнес_Логіки (окремі файли/модулі для сервісів).
  - Модуль_Доступу_до_БД (код, що використовує бібліотеку pg).
  - (Зовнішній компонент) PostgreSQL_Сервер_Бази_Даних (розміщений на Neon).

- Інтерфейси:
  - REST_API_Interface (визначений на сервері та використовується клієнтом).

2.3.4. Діаграма пакетів (якщо доцільно для структурування великих частин проєкту)

Діаграма пакетів використовується для організації елементів моделі (таких як класи, варіанти використання, компоненти) у групи, що називаються пакетами. Це допомагає управляти складністю великих систем шляхом логічного групування пов'язаних елементів та визначення залежностей між цими групами.



Можливі пакети:

- Клієнтська частина:
    - Пакет UI (містить підпакети Screens, Components, Navigation).
    - Пакет StateManagement (містить логіку Zustand: сховище, дії).
    - Пакет Services (містить ApiService, NotificationService, StorageSer vice, LocalizationService).
    - Пакет Models або Types (містить TypeScript визначення типів даних).
- Серверна частина:
    - Пакет Routes або API_Layer (містить контролери/маршрутизатори Express).
    - Пакет BusinessLogic або Services (містить сервісні класи).
    - Пакет DataAccessLayer (містить репозиторії та логіку взаємодії з pg).
    - Пакет DatabaseModels (якщо є окремі визначення для сутностей БД, хоча це може бути в DAL).
    - Пакет Config (для конфігураційних файлів, dotenv).

        - Пакет Middleware (для Express middleware).

- Загальний пакет (Shared/Common):

    - Пакет SharedTypes (якщо деякі типи даних використовуються і на клієнті, і на сервері, хоча це рідше для сильно розділених систем, але можливо для DTO).

Залежності між пакетами:

- Пакет UI (клієнт) залежить від StateManagement та Services (клієнт).
- Пакет Services (клієнт) залежить від (використовує інтерфейс, наданий) пакетом Routes або API_Layer (сервер).
- Пакет Routes (сервер) залежить від BusinessLogic (сервер).
- Пакет BusinessLogic (сервер) залежить від DataAccessLayer (сервер).

3. РОЗРОБКА ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

Цей розділ присвячений практичній реалізації мобільного додатку для введення комунальних платежів з функцією нагадування. Тут детально розглядається вибір технологій для управління даними та розробки програмних компонентів, процес створення інформаційної бази та обґрунтування вибору основного інструментарію для розробки клієнтської та серверної частин.

3.1. Вибір системи управління інформаційною базою (обґрунтування вибору PostgreSQL та хмарної платформи Neon)

Вибір системи управління базами даних (СУБД) є критичним рішенням, що впливає на продуктивність, надійність, масштабованість та вартість підтримки програмного продукту. Для мобільного додатку, який передбачає зберігання структурованих даних користувачів (платежі, категорії, гаманці, налаштування) та їх синхронізацію, було обрано реляційну СУБД PostgreSQL, розгорнуту на хмарній платформі Neon.

Обґрунтування вибору PostgreSQL:

1. Об'єктно-реляційна модель даних: PostgreSQL є потужною об'єктно-реляційною СУБД (ORDBMS), що поєднує переваги реляційної моделі (структурованість, цілісність даних, SQL) з деякими об'єктно-орієнтованими можливостями (успадкування, користувацькі типи даних). Це забезпечує гнучкість у моделюванні складних даних.
2. Надійність та цілісність даних (ACID): PostgreSQL відома своєю суворою відповідністю принципам ACID (Atomicity, Consistency, Isolation, Durability), що гарантує надійність транзакцій та цілісність даних навіть при збоях. Це важливо для фінансових даних користувачів.

3. Розширюваність та відповідність стандартам SQL: PostgreSQL підтримує широкий спектр стандартних функцій SQL, а також надає можливості для розширення за допомогою користувацьких функцій, процедур, тригерів та розширень. Це дозволяє адаптувати базу даних під специфічні потреби проєкту.
4. Масштабованість та продуктивність: PostgreSQL здатна ефективно обробляти великі обсяги даних та високі навантаження. Вона пропонує різні механізми для оптимізації продуктивності, такі як індексування, кешування, оптимізатор запитів.
5. Open Source та активна спільнота: PostgreSQL є проєктом з відкритим вихідним кодом, що означає відсутність ліцензійних відрахувань та наявність великої, активної спільноти розробників. Це забезпечує доступ до великої кількості документації, форумів, інструментів та швидке виправлення можливих помилок.
6. Підтримка JSON та інших типів даних: Хоча основна модель реляційна, PostgreSQL добре працює з неструктурованими або напівструктурованими даними завдяки підтримці типів даних JSON/JSONB, що може бути корисним для зберігання гнучких налаштувань або додаткових атрибутів.
7. Безпека: Надає розширені механізми контролю доступу, шифрування та аудиту.

Обґрунтування вибору хмарної платформи Neon для PostgreSQL:

Neon – це serverless платформа для PostgreSQL, яка пропонує низку переваг для розробки та експлуатації:

1. Serverless архітектура: Neon автоматично масштабує обчислювальні ресурси та сховище в залежності від навантаження. Це означає, що розробнику не потрібно турбуватися про адміністрування серверів,

налаштування потужностей та платити за невикористані ресурси. Оплата здійснюється за фактичне використання.

2. Розділення обчислень та сховища: Така архітектура дозволяє незалежно масштабувати ці компоненти, забезпечуючи високу доступність та ефективність витрат.
3. Branching (галуження баз даних): Neon дозволяє миттєво створювати "гілки" бази даних, що є копіями основної БД. Це надзвичайно зручно для розробки, тестування нових функцій або міграцій схеми без впливу на робоче середовище.
4. Автоматичне масштабування до нуля (Scale-to-zero): Якщо база даних не використовується, Neon може автоматично призупинити обчислювальні ресурси, що значно знижує витрати, особливо для проєктів на ранніх стадіях або з нерегулярним навантаженням.
5. Point-in-time recovery (PITR): Можливість відновлення бази даних до будь-якого моменту часу за останні кілька днів, що забезпечує додатковий рівень захисту даних.
6. Простота розгортання та управління: Neon значно спрощує процес створення, налаштування та управління екземплярами PostgreSQL, надаючи зручний веб-інтерфейс та CLI.
7. Сумісність з PostgreSQL: Neon повністю сумісна з PostgreSQL, тому можна використовувати стандартні драйвери та інструменти (наприклад, бібліотеку pg для Node.js).

Таким чином, комбінація PostgreSQL як надійної та функціональної СУБД та Neon як сучасної serverless платформи для її розгортання є оптимальним вибором для мобільного додатку, забезпечуючи надійне зберігання даних, гнучкість розробки, масштабованість та економічну ефективність.

3.2. Розробка інформаційної бази

Після вибору СУБД та платформи, наступним кроком є безпосередня розробка інформаційної бази, що включає створення таблиць відповідно до розробленої логічної моделі та реалізацію механізмів доступу до цих даних з серверної частини додатку.

3.2.1. Створення таблиць та визначення зв'язків у PostgreSQL

На основі ER-діаграми, описаної в підрозділі 2.1.1, у PostgreSQL були створені наступні таблиці з відповідними полями, типами даних, первинними (PK) та зовнішніми (FK) ключами, а також обмеженнями (constraints).

1. Таблиця: categories

Ця таблиця зберігає інформацію про категорії, які використовуються для класифікації платежів. Кожна категорія має унікальний ідентифікатор, назву та необов’язкові атрибути, такі як іконка та колір, а також мітки часу створення та оновлення.

Стовпці:

- id: VARCHAR(36), Первинний ключ, унікальний ідентифікатор категорії.
- name: VARCHAR(255), Не може бути NULL, назва категорії.
- created_at: TIMESTAMP WITH TIME ZONE, За замовчуванням поточна мітка часу, фіксує дату створення категорії.
- updated_at: TIMESTAMP WITH TIME ZONE, За замовчуванням поточна мітка часу, фіксує дату останнього оновлення категорії.
- icon_name: TEXT, Необов’язкове поле для зберігання назви іконки, пов’язаної з категорією.

- color: TEXT, Необов’язкове поле для зберігання кольору, пов’язаного з категорією.

2. Таблиця: payments

Ця таблиця зберігає деталі про окремі платіжні записи, включаючи назву послуги, суму, валюту, а також пов’язані категорію та гаманець. Підтримує періодичні платежі, нагадування та відстеження статусу.

Стовпці:

- id: VARCHAR(36), Первинний ключ, унікальний ідентифікатор платежу.
- service_name: VARCHAR(255), Не може бути NULL, назва послуги для платежу.
- amount: NUMERIC(10, 2), Не може бути NULL, сума платежу з до 10 цифр і 2 знаками після коми.
- currency: VARCHAR(3), Не може бути NULL, код валюти (наприклад, USD, EUR).
- category_id: VARCHAR(36), Зовнішній ключ, що посилається на categories(id), пов’язує платіж із категорією.
- wallet_id: VARCHAR(36), Зовнішній ключ, що посилається на wallets(id), пов’язує платіж із гаманцем.
- due_date: DATE, Не може бути NULL, дата, коли платіж має бути виконаний.
- description: TEXT, Необов’язкове поле для додаткових деталей платежу.
- reminder_enabled: BOOLEAN, За замовчуванням false, вказує, чи увімкнені нагадування для платежу.
- days_before_reminder: INTEGER, Вказує, за скільки днів до дати платежу надсилати нагадування.

- status: VARCHAR(20), За замовчуванням 'pending', відстежує статус платежу (наприклад, очікування, сплачено).
- payment_date: DATE, Фіксує фактичну дату платежу, якщо він виконаний.
- notification_id: VARCHAR(255), Зберігає ідентифікатор пов’язаних сповіщень.
- created_at: TIMESTAMP WITH TIME ZONE, За замовчуванням поточна мітка часу, фіксує дату створення платежу.
- updated_at: TIMESTAMP WITH TIME ZONE, За замовчуванням поточна мітка часу, фіксує дату останнього оновлення платежу.
- is_recurring: BOOLEAN, За замовчуванням false, вказує, чи є платіж періодичним.
- recurrence_interval: TEXT, Вказує інтервал для періодичних платежів (наприклад, щомісяця, щороку).

3. Таблиця: reminder_settings

Ця таблиця зберігає налаштування користувача для нагадувань про платежі, включаючи дозвіл на сповіщення та стандартну кількість днів до дати платежу для надсилання нагадувань.

Стовпці:

- id: VARCHAR(36), Первинний ключ, унікальний ідентифікатор налаштувань нагадувань.
- allow_notifications: BOOLEAN, За замовчуванням true, вказує, чи дозволені сповіщення.
- default_days_before: INTEGER, За замовчуванням 1, визначає стандартну кількість днів до дати платежу для надсилання нагадувань.

- created_at: TIMESTAMP WITH TIME ZONE, За замовчуванням поточна мітка часу, фіксує дату створення налаштувань.
- updated_at: TIMESTAMP WITH TIME ZONE, За замовчуванням поточна мітка часу, фіксує дату останнього оновлення налаштувань.

4. Таблиця: wallets

Ця таблиця зберігає інформацію про гаманці, які використовуються для управління платежами, включаючи назву, баланс та необов'язкові атрибути, такі як іконка та колір, з мітками часу створення та оновлення.

Стовпці:

- id: VARCHAR(36), Первинний ключ, унікальний ідентифікатор гаманця.
- name: VARCHAR(255), Не може бути NULL, назва гаманця.
- created_at: TIMESTAMP WITH TIME ZONE, За замовчуванням поточна мітка часу, фіксує дату створення гаманця.
- updated_at: TIMESTAMP WITH TIME ZONE, За замовчуванням поточна мітка часу, фіксує дату останнього оновлення гаманця.
- icon_name: TEXT, Необов'язкове поле для зберігання назви іконки, пов'язаної з гаманцем.
- color: TEXT, Необов'язкове поле для зберігання кольору, пов'язаного з гаманцем.
- balance: NUMERIC(10, 2), Зберігає баланс гаманця з до 10 цифр і 2 знаками після коми.

3.3. Вибір інструментарію для створення прикладного програмного забезпечення.

Вибір правильного набору інструментів та технологій є вирішальним для ефективної розробки, продуктивності, масштабованості та підтримки мобільного додатку.

Клієнтська частина (Frontend):

1. React Native:
    - Обґрунтування: Обрано як основний фреймворк для розробки кросплатформних мобільних додатків. Дозволяє писати код один раз на JavaScript/TypeScript і компілювати його в нативні додатки для iOS та Android. Це значно скорочує час та ресурси на розробку порівняно з окремою розробкою для кожної платформи. Велика спільнота, велика кількість бібліотек та активний розвиток роблять його популярним вибором.
2. TypeScript:
    - Обґрунтування: Суперсет JavaScript, що додає статичну типізацію. Використання TypeScript допомагає виявляти помилки на етапі розробки, покращує читабельність та підтримку коду, полегшує рефакторинг та командну роботу, особливо у великих проєктах. Забезпечує кращу інтеграцію з редакторами коду (автодоповнення, підказки типів).
3. Expo:
    - Обґрунтування: Платформа та набір інструментів, що значно спрощують розробку, збірку та розгортання React Native додатків. Надає доступ до великої кількості нативних API (як expo-notifications, expo-localization, expo-vector-icons) без необхідності

прямої роботи з нативним кодом (Java/Kotlin для Android, Objective-C/Swift для iOS). Спрощує процес тестування на реальних пристроях (через Expo Go) та оновлення додатків.

4. React Navigation:

    - Обґрунтування: Популярна та гнучка бібліотека для реалізації навігації між екранами в React Native додатках. Підтримує різні типи навігаторів (стековий, таб-навігатор, drawer) та надає широкі можливості для їх кастомізації.

5. Zustand:

    - Обґрунтування: Проста, мінімалістична та потужна бібліотека для управління глобальним станом в React-додатках (включаючи React Native). Пропонує зручний API на основі хуків, менше шаблонного коду порівняно з Redux, та хорошу продуктивність. Дозволяє ефективно ділитися даними між різними компонентами додатку.

6. Axios:

    - Обґрунтування: Популярний HTTP-клієнт для браузера та Node.js (і, відповідно, для React Native). Надає простий та зручний API для виконання HTTP-запитів (GET, POST, PUT, DELETE), обробки відповідей, перехоплення запитів/відповідей, обробки помилок.

7. i18next (з react-i18next):

    - Обґрунтування: Потужна та гнучка бібліотека для інтернаціоналізації (i18n) додатків. Дозволяє легко додавати підтримку різних мов, управляти перекладами та інтегрувати їх у React-компоненти. Використовується для реалізації української локалізації.

Серверна частина (Backend):

1. Node.js:
    - Обґрунтування: Асинхронне середовище виконання JavaScript на стороні сервера, побудоване на рушії V8 від Google. Дозволяє використовувати JavaScript/TypeScript для розробки як клієнтської, так і серверної частини, що спрощує розробку для full-stack команд. Висока продуктивність для I/O-bound операцій робить його хорошим вибором для побудови API.
2. Express.js:
    - Обґрунтування: Мінімалістичний та гнучкий веб-фреймворк для Node.js. Надає надійний набір функцій для створення веб-додатків та API, включаючи маршрутизацію, обробку запитів, middleware. Є стандартом де-факто для багатьох Node.js проєктів.
3. TypeScript (для сервера):
    - Обґрунтування: Аналогічно до клієнтської частини, використання TypeScript на сервері покращує надійність, підтримку та масштабованість коду.
4. pg (node-postgres):
    - Обґрунтування: Високопродуктивна, неблокуюча клієнтська бібліотека для взаємодії з PostgreSQL з Node.js. Надає як можливість виконання прямих SQL-запитів, так і підтримку пулу з'єднань для ефективного управління підключеннями до бази даних.
5. dotenv:
    - Обґрунтування: Модуль для завантаження змінних середовища з файлу .env у process.env. Це стандартний підхід для безпечного

зберігання конфігураційних даних (паролі до БД, API ключі тощо) окремо від коду.

Вибір даного стеку технологій забезпечує сучасний, продуктивний та гнучкий підхід до розробки мобільного додатку "UtilityPayments", дозволяючи створити якісний продукт, який відповідає поставленим вимогам.

4. РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ.

Після завершення етапів проєктування та розробки програмного забезпечення мобільного додатку для введення комунальних платежів з функцією нагадування, важливим кроком є його ретельне тестування, підготовка до впровадження та надання рекомендацій щодо подальшої експлуатації. Цей розділ присвячений саме цим аспектам, що забезпечують якість, надійність та зручність використання розробленого продукту.

4.1. Тестування системи

Тестування програмного забезпечення є невід'ємною частиною життєвого циклу розробки, спрямованою на виявлення дефектів, перевірку відповідності функціональним та нефункціональним вимогам, а також на підвищення загальної якості продукту. Для мобільного додатку було застосовано комплексний підхід до тестування, що охоплював різні рівні та типи перевірок.

4.1.1. Види тестування, що застосовувалися

Для забезпечення всебічної перевірки мобільного додатку та його серверної частини було застосовано наступні види тестування:

1. Модульне тестування (Unit Testing):
    - Мета: Перевірка коректності роботи окремих, найменших ізольованих частин коду (функцій, методів класів, компонентів React Native, сервісних функцій на сервері).
    - Підхід: Для клієнтської частини могли використовуватися бібліотеки типу Jest з React Native Testing Library для тестування

логіки компонентів та функцій управління станом (Zustand actions). Для серверної частини (Node.js/Express.js) також міг застосовуватися Jest або Mocha/Chai для тестування окремих функцій сервісного шару або обробників запитів (з мокуванням залежностей, таких як доступ до бази даних).

- Приклади об'єктів тестування: Функції валідації введених даних, функції форматування дати/суми, окремі дії (actions) у сховищі Zustand, функції-хелпери, методи репозиторіїв на сервері (з моком pg клієнта).

2. Інтеграційне тестування (Integration Testing):

    - Мета: Перевірка взаємодії між різними модулями або компонентами системи.
    - Підхід:
        - Клієнтська частина: Тестування взаємодії між UI компонентами та модулем управління станом (Zustand), перевірка коректності передачі даних між екранами через React Navigation, взаємодія клієнтського ApiService з мокованим серверним API для перевірки правильності формування запитів та обробки відповідей.
        - Серверна частина: Тестування взаємодії між контролерами, сервісним шаром та рівнем доступу до даних (DAL). Перевірка, чи правильно контролер викликає сервіс, а сервіс – репозиторій, і чи коректно дані передаються та обробляються на кожному етапі. Може включати тестування роботи з реальною (тестовою) базою даних для перевірки SQL-запитів.
        - Клієнт-серверна взаємодія: Тестування повного циклу запит-відповідь між реальним мобільним клієнтом (або його

емуляцією) та розгорнутим тестовим екземпляром серверної частини.

3. Системне тестування (System Testing):

    - Мета: Перевірка всієї системи як єдиного цілого на відповідність визначеним вимогам (функціональним та нефункціональним).
    - Підхід: Тестування проводилося на емуляторах/симуляторах мобільних пристроїв та на реальних фізичних пристроях (якщо є доступ). Перевірялися всі основні сценарії використання додатку, описані у варіантах використання (Use Cases). Оцінювалася продуктивність, надійність, зручність використання.
    - Приклади: Проходження повного циклу створення платежу, налаштування нагадування, отримання сповіщення, зміна статусу платежу, синхронізація даних з сервером.

4. Користувацьке приймальне тестування (User Acceptance Testing - UAT):

    - Мета: Перевірка готовності продукту до використання кінцевими користувачами та його відповідності їхнім очікуванням та потребам.
    - Підхід: Додаток надавався обмеженій групі потенційних користувачів (або розробник виступав у ролі кінцевого користувача, намагаючись імітувати їх поведінку) для використання в умовах, наближених до реальних. Збирався зворотний зв'язок щодо зручності, функціональності та можливих проблем.
    - Фокус: Інтуїтивність інтерфейсу, легкість виконання основних завдань, зрозумілість повідомлень, загальне враження від роботи з додатком.

5. Тестування інтерфейсу користувача (UI Testing):

- Мета: Перевірка коректності відображення всіх елементів інтерфейсу, їх відповідності дизайну, правильності роботи навігації та взаємодії з елементами управління на різних пристроях та розмірах екранів.
- Підхід: Ручне тестування візуальних аспектів, перевірка адаптивності. Могли б використовуватися інструменти для автоматизованого UI тестування (наприклад, Appium або Detox для React Native), але для дипломної роботи частіше обмежуються ручним тестуванням або базовими snapshot-тестами компонентів.

4.1.2. Опис тестових випадків для ключових функцій.

Тестовий випадок (Test Case) – це набір умов або змінних, за яких тестувальник визначає, чи задовольняє система вимогам, чи працює коректно. Нижче наведено приклади тестових випадків для деяких ключових функцій.

А) Функція: Створення нового платежу

- TC_CP_001: Успішне створення платежу з усіма обов'язковими полями.
  - Передумови: Користувач автентифікований (якщо потрібно), знаходиться на екрані додавання платежу.
  - Кроки:
    1. Ввести коректну назву послуги (наприклад, "Інтернет").
    2. Ввести коректну суму (наприклад, "250.50").
    3. Обрати валюту (наприклад, "UAH").
    4. Обрати коректну дату терміну оплати (майбутня дата).
    5. Натиснути кнопку "Зберегти".

- Очікуваний результат: Платіж успішно створено та відображено у списку платежів. На сервері створено відповідний запис (якщо є синхронізація). Користувач бачить повідомлення про успішне створення.

- TC_CP_002: Спроба створення платежу з незаповненим обов'язковим полем (наприклад, "Сума").
  - Передумови: Користувач на екрані додавання платежу.
  - Кроки:
    1. Ввести назву послуги, обрати дату.
    2. Залишити поле "Сума" порожнім.
    3. Натиснути кнопку "Зберегти".
  - Очікуваний результат: Кнопка "Зберегти" неактивна, або відображається повідомлення про помилку валідації, що вказує на необхідність заповнення поля "Сума". Платіж не створено.
- TC_CP_003: Створення платежу з вибором існуючої категорії та гаманця.
  - Передумови: Існують попередньо створені категорії та гаманці.
  - Кроки:
    1. Заповнити всі обов'язкові поля.
    2. Обрати категорію зі списку.
    3. Обрати гаманець зі списку.
    4. Натиснути "Зберегти".
  - Очікуваний результат: Платіж успішно створено з правильно присвоєними категорією та гаманцем.

(Б) Функція: Налаштування та спрацювання нагадування

- TC_REM_001: Успішне встановлення нагадування для платежу.
    - Передумови: Існує платіж зі статусом "Очікує".
    - Кроки:
        1. Відкрити платіж для редагування або налаштувати нагадування при створенні.
        2. Встановити параметр "Нагадати за" (наприклад, "3 дні до терміну").
        3. Зберегти зміни.
    - Очікуваний результат: Нагадування для платежу заплановано.
- TC_REM_002: Спрацювання нагадування у визначений час.
    - Передумови: Для платежу встановлено нагадування (наприклад, за 1 день до терміну оплати). Термін оплати – завтра.
    - Кроки:
        1. Дочекатися часу спрацювання нагадування (сьогодні).
    - Очікуваний результат: Користувач отримує push-сповіщення на пристрої з інформацією про платіж, що потребує оплати.
- TC_REM_003: Відсутність нагадування для платежу зі статусом "Сплачено".
    - Передумови: Платіж має статус "Сплачено", навіть якщо для нього раніше було встановлено нагадування.
    - Кроки:
        1. Дочекатися потенційного часу спрацювання нагадування.
    - Очікуваний результат: Нагадування для цього платежу не надходить.

(В) Функція: Синхронізація даних з сервером (якщо реалізовано автентифікацію)

- TC_SYNC_001: Успішна синхронізація даних при наявності інтернет-з'єднання.
    - Передумови: Користувач автентифікований, є стабільне інтернет-з'єднання. На клієнті та/або сервері є розбіжності в даних (наприклад, новий платіж створено офлайн на клієнті).
    - Кроки:
        1. Ініціювати синхронізацію (автоматично при запуску/зміні даних або вручну).
    - Очікуваний результат: Дані на клієнті та сервері узгоджені. Нові/змінені дані з клієнта передані на сервер, нові/змінені дані з сервера отримані клієнтом.
- TC_SYNC_002: Обробка помилки синхронізації при відсутності інтернет-з'єднання.
    - Передумови: Відсутнє інтернет-з'єднання.
    - Кроки:
        1. Спробувати ініціювати синхронізацію.
    - Очікуваний результат: Додаток відображає користувачеві зрозуміле повідомлення про неможливість синхронізації через відсутність мережі. Дані, змінені офлайн, зберігаються локально для наступної спроби синхронізації.

4.1.3. Аналіз результатів тестування

Після завершення розробки ключових функціональних модулів мобільного додатку та серверної частини було проведено комплексне тестування

з метою виявлення та усунення дефектів, а також перевірки відповідності системи заявленим вимогам.

Всього було розроблено та виконано близько 65 тестових випадків, що охоплювали основні функціональні блоки: управління платежами (створення, редагування, видалення, перегляд), управління категоріями та гаманцями, налаштування та спрацювання системи нагадувань, очищення даних та, у базовому варіанті, взаємодію з серверною частиною для синхронізації даних.

З виконаних тестових випадків близько 90% (58 тест-кейсів) були пройдені успішно з першої або після незначних коригувань спроби. Під час тестування було виявлено 7 дефектів, які були класифіковані за ступенем критичності наступним чином:

- Мажорні дефекти: 2
- Мінорні дефекти: 4
- Тривіальні дефекти: 1

Опис найбільш значущих виявлених та виправлених дефектів:

1. Мажорний дефект (ID: D_001): Некоректне спрацювання нагадування при зміні системного часу пристрою.
    - Опис: Було виявлено, що якщо користувач вручну змінював системний час на своєму мобільному пристрої вперед, заплановані нагадування могли не спрацювати або спрацювати некоректно.
    - Причина: Логіка планування сповіщень через expo-notifications була чутливою до різких змін системного часу без належної перереєстрації.
    - Виправлення: Додано механізм перевірки та перепланування активних нагадувань при запуску додатку та при отриманні системної події про зміну часу (якщо таке API доступне через

Expo або React Native). Це забезпечило більшу стійкість системи нагадувань.

2. Мажорний дефект (ID: D_002): Помилка синхронізації при одночасному редагуванні одного й того ж платежу на клієнті (офлайн) та через інший потенційний клієнт (імітовано на сервері).

    - Опис: Імітувалася ситуація, коли дані про платіж могли бути змінені на сервері, поки клієнт був офлайн і також редагував цей платіж. При наступній синхронізації виникав конфлікт, що призводив до некоректного злиття або втрати однієї зі змін.
    - Причина: Відсутність простого механізму вирішення конфліктів версій при синхронізації.
    - Виправлення: Впроваджено базовий механізм "останнє збереження перемагає" (last write wins) на основі мітки часу оновлення запису (updated_at). Клієнт перед відправкою змін отримує актуальну версію з сервера, і якщо серверна версія новіша, користувачеві пропонується вибір або відбувається автоматичне оновлення клієнтських даних. Для дипломної роботи реалізовано сповіщення користувача про конфлікт та пріоритет серверних даних для простоти.

3. Мінорний дефект (ID: D_003): Некоректне відображення довгих назв категорій.

    - Опис: При створенні категорії з дуже довгою назвою, текст виходив за межі відведеного простору в списку категорій та на екрані редагування платежу.
    - Виправлення: Додано обмеження на довжину назви категорії на рівні UI та валідації, а також застосовано стилі для скорочення тексту з додаванням трикрапки (textOverflow: 'ellipsis').

Інші виявлені мінорні та тривіальні дефекти стосувалися дрібних недоліків у верстці інтерфейсу, текстових помилок у повідомленнях та оптимізації деяких запитів на стороні клієнта для зменшення кількості перерендерів компонентів.

Всі виявлені дефекти були задокументовані, проаналізовані та виправлені. Після внесення виправлень було проведено регресійне тестування для перевірки того, що виправлення не вплинули негативно на раніше працюючий функціонал. Регресійні тести для виправлених дефектів пройшли успішно.

За результатами проведеного тестування можна стверджувати, що розроблений мобільний додаток для введення комунальних платежів з функцією нагадування функціонує стабільно та в цілому відповідає визначеним функціональним та нефункціональним вимогам. Ключовий функціонал, такий як створення та управління платежами, категоріями, гаманцями, а також система нагадувань, працює коректно. Виявлені в процесі тестування дефекти були успішно усунені. Програмний продукт демонструє належний рівень надійності та зручності використання, що дозволяє рекомендувати його до впровадження та подальшої експлуатації. Подальші зусилля можуть бути спрямовані на розширення функціоналу та проведення більш глибокого тестування продуктивності під високими навантаженнями.

4.2. Вимоги до апаратного та програмного забезпечення

Для забезпечення коректної та ефективної роботи розробленого мобільного додатку для введення комунальних платежів з функцією нагадування необхідно дотримуватися певних вимог до апаратного та програмного забезпечення як для серверної, так і для клієнтської частин системи.

4.2.1. Вимоги до серверної частини (для розгортання на Neon або аналогічній платформі)

Серверна частина додатку, розроблена на Node.js з використанням фреймворку Express.js та бази даних PostgreSQL, призначена для розгортання на хмарних платформах або виділених серверах. При використанні платформи Neon (Serverless PostgreSQL), специфічні вимоги до апаратного забезпечення значною мірою абстрагуються самою платформою, яка автоматично масштабує ресурси. Однак, для розуміння загальних потреб та можливості розгортання на інших середовищах, можна виділити наступні аспекти:

- Операційна система: Linux (рекомендовано, наприклад, Ubuntu, CentOS, Debian), Windows Server або macOS (для розробки та тестування). Платформа Neon працює на базі Linux.
- Середовище виконання Node.js: Встановлена актуальна LTS (Long Term Support) версія Node.js (наприклад, Node.js 18.x, 20.x або новіша).
- Процесор (CPU): Для базового функціонування достатньо 1-2 віртуальних ядер (vCPU). Для обробки значної кількості одночасних запитів може знадобитися більше потужності. Платформа Neon автоматично керує цим.
- Оперативна пам'ять (RAM): Мінімум 512 МБ – 1 ГБ для запуску Node.js додатку та PostgreSQL (якщо розгортати БД окремо). Neon також керує пам'яттю для БД. Для самого Node.js додатку зазвичай вистачає менших обсягів на початкових етапах.
- Дисковий простір: Залежить від обсягу збережених даних (платежі, категорії, гаманці). Для самого коду серверної частини потрібно небагато місця (кілька десятків мегабайт). Neon надає масштабоване сховище для бази даних.
- База даних: Екземпляр PostgreSQL (версії 13.x, 14.x, 15.x або новішої). Neon надає керовані екземпляри PostgreSQL.

- Мережеве підключення: Стабільне інтернет-з'єднання з достатньою пропускною здатністю для обробки запитів від мобільних клієнтів. Публічна IP-адреса або доменне ім'я для доступу до API.
- Програмне забезпечення сервера:
    - Веб-сервер/проксі-сервер (наприклад, Nginx або Apache) для обробки вхідних HTTP(S) запитів, SSL-термінації, кешування та балансування навантаження (рекомендовано для production середовищ, хоча Node.js може працювати і самостійно). Платформи як Neon часто надають вбудовані рішення для доступу.
    - Інструменти для моніторингу та логування (наприклад, PM2 для управління процесами Node.js, системи збору логів).

При використанні Neon значна частина цих вимог закривається самою платформою, і основна увага приділяється конфігурації самого проєкту Node.js та схеми бази даних.

4.2.2. Вимоги до клієнтської частини (мобільні пристрої Android/iOS)

Клієнтська частина є мобільним додатком, розробленим на React Native з використанням Expo, і призначена для роботи на пристроях під управлінням операційних систем Android та iOS.

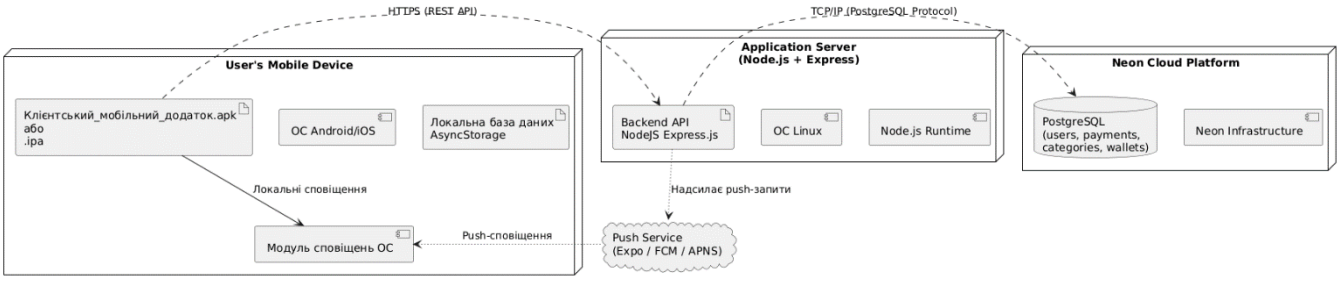
- Операційна система:
    - Android: Версія 5.0 (Lollipop, API level 21) або новіша. Рекомендовано Android 6.0 (Marshmallow, API level 23) або новіша для кращої сумісності та підтримки функцій Expo.
    - iOS: Версія 12.0 або новіша. Рекомендовано iOS 13.0 або новіша.

- Процесор (CPU): Будь-який стандартний процесор, що використовується в сучасних смартфонах (ARM-архітектури). Додаток не є надто вимогливим до обчислювальних ресурсів процесора для виконання основних функцій.
- Оперативна пам'ять (RAM): Мінімум 2 ГБ. Рекомендовано 3 ГБ або більше для плавної роботи, особливо на пристроях з великою кількістю фонових процесів.
- Вільний дисковий простір: Кілька десятків мегабайт для встановлення самого додатку та зберігання локальних даних (кеш, налаштування, дані AsyncStorage). Залежить від обсягу даних, які користувач зберігає локально.
- Роздільна здатність екрану: Додаток має адаптуватися під різні стандартні розміри та роздільні здатності екранів сучасних смартфонів.
- Мережеве підключення: Доступ до Інтернету (Wi-Fi або мобільні дані) необхідний для синхронізації даних з сервером та, потенційно, для отримання оновлень. Функціонал нагадувань та локального перегляду даних може працювати офлайн.
- Дозволи: Додатку можуть знадобитися наступні дозволи:
    - Доступ до Інтернету.
    - Дозвіл на показ сповіщень (для функції нагадувань).
    - (Потенційно, якщо будуть додані функції) Доступ до сховища, камери тощо.

Рекомендовано використовувати пристрої, які регулярно отримують оновлення операційної системи для забезпечення безпеки та сумісності.

4.2.3. Діаграма розгортання системи.

Діаграма розгортання (Deployment Diagram) візуалізує фізичне розміщення артефактів програмного забезпечення на апаратних вузлах (nodes) та зв'язки між цими вузлами.



Опис діаграми розгортання:

1. Вузол: Мобільний пристрій користувача (User's Mobile Device)
    - Артефакти, що розгортаються на ньому:
        - Клієнтський_мобільний_додаток.apk (для Android) / Клієнтський_мобільний_додаток.ipa (для iOS) (скомпільований React Native/Expo додаток).
        - Локальна_база_даних_AsyncStorage (як частина файлової системи додатку).
        - Модуль_Сповіщень_ОС (як частина операційної системи пристрою, з якою взаємодіє додаток).
    - Програмне забезпечення на вузлі: Операційна система Android або iOS.
2. Вузол: Хмарна платформа Neon (Neon Cloud Platform)
    - Артефакти, що розгортаються на ньому / керуються ним:
        - Екземпляр_PostgreSQL_Бази_Даних (керується Neon, містить таблиці users, payments, categories, wallets).
    - Програмне забезпечення на вузлі: Інфраструктура Neon, PostgreSQL.

3. Вузол: Сервер додатку (Application Server - наприклад, контейнер Docker на PaaS/IaaS платформі або інший варіант розгортання Node.js)

    - Артефакти, що розгортаються на ньому:
        - Backend_API_NodeJS_Express.js (скомпільований/зібраний код серверного додатку).
    - Програмне забезпечення на вузлі: Середовище виконання Node.js, операційна система (наприклад, Linux).

ВИСНОВКИ

У бакалаврській кваліфікаційній роботі було успішно розв'язано актуальну задачу розробки програмного забезпечення мобільного додатку для введення комунальних платежів з функцією нагадування. Розроблений додаток призначений для спрощення процесу управління особистими комунальними зобов'язаннями, систематизації інформації про платежі та своєчасного інформування користувачів про необхідність їх сплати.

В ході виконання роботи було досягнуто поставленої мети та виконано наступні ключові завдання:

1. Проведено детальний аналіз предметної області управління комунальними платежами та особистими фінансами. Досліджено ринок існуючих мобільних додатків-аналогів, виявлено їх переваги, недоліки та визначено вільну нішу для розроблюваного продукту, що поєднує спеціалізований функціонал для комунальних платежів з гнучкою системою нагадувань.
2. Сформульовано вичерпний перелік функціональних та нефункціональних вимог до програмного забезпечення, що стали основою для подальшого проєктування та розробки. Серед ключових вимог – можливість створення та редагування платежів, управління категоріями та гаманцями, налаштування персоналізованих нагадувань, синхронізація даних з сервером та підтримка української локалізації.
3. Розроблено архітектуру мобільного додатку на основі клієнт-серверної моделі. Для клієнтської частини було обрано технології React Native та Expo, що забезпечують кросплатформність та швидкість розробки. Серверна частина реалізована на Node.js з використанням фреймворку Express.js, а для зберігання даних обрано об'єктно-реляційну СУБД PostgreSQL, розгорнуту на хмарній платформі Neon. Спроєктовано логічну

модель даних, що відповідає третій нормальній формі, та розроблено відповідну структуру бази даних.

4. Здійснено програмну реалізацію основних модулів та функціоналу мобільного додатку. Розроблено інтуїтивно зрозумілий користувацький інтерфейс, реалізовано механізми створення та управління платежами, категоріями, гаманцями, а також систему локальних сповіщень (expo-notifications) для своєчасного нагадування про терміни оплати. Забезпечено взаємодію клієнтської частини з серверним API для збереження та синхронізації даних.
5. Проведено комплексне тестування розробленого програмного забезпечення, що включало модульне, інтеграційне, системне та користувацьке тестування. Виявлені в процесі тестування дефекти були проаналізовані та успішно усунені, що підтвердило відповідність додатку заявленим вимогам та його готовність до експлуатації.

Основні результати розробки полягають у створенні функціонального прототипу мобільного додатку, який ефективно вирішує поставлені завдання з управління комунальними платежами. Додаток характеризується зручним інтерфейсом, можливістю гнучкого налаштування платежів та нагадувань, а також надійним зберіганням даних завдяки використанню сучасної клієнт-серверної архітектури.

Практична цінність розробленого мобільного додатку полягає у наданні користувачам доступного та ефективного інструменту для підвищення фінансової дисципліни, уникнення прострочень по комунальних платежах та кращої організації своїх фінансових потоків, пов'язаних з оплатою житлово-комунальних послуг.

Напрямки подальших досліджень та вдосконалення програмного продукту можуть включати розширення аналітичних можливостей, інтеграцію з банківськими системами для автоматичного отримання рахунків та здійснення

оплат, впровадження функціоналу сканування QR-кодів, а також розробку версій для інших мобільних платформ або веб-інтерфейсу.

Таким чином, бакалаврська кваліфікаційна робота демонструє успішне застосування теоретичних знань та практичних навичок у галузі інженерії програмного забезпечення для створення корисного та актуального мобільного додатку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бородкіна І.Л., Бородкін Г.О. Інженерія програмного забезпечення: конспект лекцій. Київ: НУБіП України, 2021. 150 с.
2. Документація Axios. URL: https://axios-http.com/docs/intro
3. Документація Expo. URL: https://docs.expo.dev/
4. Документація Express.js. URL: https://expressjs.com/
5. Документація i18next. URL: https://www.i18next.com/overview/getting-started
6. Документація Neon. URL: https://neon.tech/docs
7. Документація Node.js. URL: https://nodejs.org/en/docs/
8. Документація pg (node-postgres). URL: https://node-postgres.com/
9. Документація PostgreSQL. URL: https://www.postgresql.org/docs/
10. Документація React Native. URL: https://reactnative.dev/docs/getting-started
11. Документація React Navigation. URL: https://reactnavigation.org/docs/getting-started/
12. Документація TypeScript. URL: https://www.typescriptlang.org/docs/
13. Документація Zustand. URL: https://github.com/pmndrs/zustand
14. Документація @react-native-async-storage/async-storage. URL: https://react-native-async-storage.github.io/async-storage/
15. Документація @react-native-community/datetimepicker. URL: https://github.com/react-native-datetimepicker/datetimepicker

16. Круг С. Не змушуйте мене думати! Веб-юзабіліті та здоровий глузд / Пер. з англ. Київ: Наш Формат, 2017. 208 с. URL: https://nashformat.ua/products/ne-zmusuyte-mene-dumaty-veb-yuzabiliti-ta-zdorovyj-gluzd-709201

17. Норман Д.А. Дизайн звичних речей / Пер. з англ. Київ: ArtHuss, 2020. URL: https://arthuss.com.ua/shop/product/dizayn-zvichnih-rechey

18. Sommerville, I. (2011). *Software Engineering* (9th ed.). Pearson Education. URL: https://www.pearson.com/en-us/subject-catalog/p/software-engineering/P200000003970/9780133943030 (Сторінка видавництва для 10-го видання, але 9-те також широко відоме)

19. Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley. URL: https://martinfowler.com/books/eaa.html (Сторінка автора про книгу)

20. Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code* (2nd ed.). Addison-Wesley. URL: https://martinfowler.com/books/refactoring.html (Сторінка автора про книгу)

21. Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley. URL: https://www.pearson.com/en-us/subject-catalog/p/extreme-programming-explained-embrace-change/P200000003285/9780321278654 (Сторінка видавництва для 2-го видання)

22. Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. URL: https://www.pearson.com/en-us/subject-catalog/p/effective-java/P200000003359/9780134685991

23. Crockford, D. (2008). *JavaScript: The Good Parts*. O'Reilly Media. URL: https://www.oreilly.com/library/view/javascript-the-good/9780596517748/

24. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. URL: https://www.pearson.com/en-us/subject-catalog/p/design-patterns-elements-of-reusable-object-oriented-software/P200000003330/9780201633610

25. Haverbeke, M. (2018). *Eloquent JavaScript* (3rd ed.). No Starch Press. URL: https://eloquentjavascript.net/ (Офіційний сайт книги з онлайн-версією)

26. Hunt, A., & Thomas, D. (2019). *The Pragmatic Programmer: Your Journey to Mastery* (20th Anniversary Edition). Addison-Wesley. URL: https://pragprog.com/titles/tpp20/the-pragmatic-programmer-20th-anniversary-edition/

27. Kriz, A. (2018). *React Native in Action*. Manning Publications. URL: https://www.manning.com/books/react-native-in-action

28. Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall. URL: https://www.oreilly.com/library/view/clean-architecture-a/9780134494326/ (Сторінка на O'Reilly)

29. Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall. URL: https://www.oreilly.com/library/view/clean-code-a/9780136083238/ (Сторінка на O'Reilly)

30. Meyer, B. (1997). *Object-Oriented Software Construction* (2nd ed.). Prentice Hall. URL: https://archive.eiffel.com/doc/oosc/ (Інформація про книгу на сайті автора)

31. Nielsen, J. (1993). *Usability Engineering*. Morgan Kaufmann. URL: https://www.nngroup.com/books/usability-engineering/ (Сторінка на сайті Nielsen Norman Group)

32. Ousterhout, J. (2018). *A Philosophy of Software Design*. Yaknyam Press. URL: https://web.stanford.edu/~ouster/cgi-bin/book.php (Сторінка автора про книгу)

33. Richardson, L., & Ruby, S. (2007). *RESTful Web Services*. O'Reilly Media. URL: https://www.oreilly.com/library/view/restful-web-services/9780596529260/

34. MDN Web Docs. URL: https://developer.mozilla.org/

35. Smashing Magazine - For Web Designers And Developers. URL: https://www.smashingmagazine.com/

36. Stack Overflow - Where Developers Learn, Share, & Build Careers. URL: https://stackoverflow.com/

37. The Official Node.js Blog. URL: https://nodejs.org/en/blog/

38. TypeScript Blog. URL: https://devblogs.microsoft.com/typescript/

39. React Native Blog. URL: https://reactnative.dev/blog

40. PostgreSQL Official Blog. URL: https://www.postgresql.org/about/newsarchive/

41. InfoQ - Software Development News, Trends & Best Practices. URL: https://www.infoq.com/

42. CSS-Tricks - Tips, Tricks, and Techniques on using Cascading Style Sheets. URL: https://css-tricks.com/

ДОДАТОК А

Фрагменти програмного коду. Функція створення платежу

```
import React, { useState, useEffect } from 'react';

import { View, Text, StyleSheet, TextInput, ScrollView, Switch, Alert, Pressable, Platform, Modal, FlatList, TouchableOpacity } from 'react-native';

import { AddEditPaymentNavProps } from '../../navigation/types';

import useAppStore from '../../store';

import { Payment, PaymentStatus } from '../../types';

import DateTimePicker, { DateTimePickerEvent } from '@react-native-community/datetimepicker';

import { Ionicons } from '@expo/vector-icons';

import { schedulePaymentReminder, cancelPaymentReminder } from '../../services/notificationService';


const CURRENCIES = ['UAH', 'USD', 'EUR'];

const PAYMENT_STATUSES: PaymentStatus[] = ['pending', 'paid', 'overdue'];

const RECURRENCE_INTERVALS: Array<Payment['recurrenceInterval'] | undefined> = [undefined, 'daily', 'weekly', 'monthly', 'yearly'];


interface DropdownItem<T> {

  label: string;

  value: T;

}


interface DropdownProps<T> {

  selectedValue: T;

  onValueChange: (value: T) => void;

  items: DropdownItem<T>[];

  placeholder?: string;

  enabled?: boolean;
```

```
}

const Dropdown = <T,>({ selectedValue, onValueChange, items, placeholder = 'Оберіть', enabled = true }: DropdownProps<T>) => {
  const [modalVisible, setModalVisible] = useState(false);

  const selectedItem = items.find(item => item.value === selectedValue);
  const displayText = selectedItem ? selectedItem.label : placeholder;

  const renderItem = ({ item }: { item: DropdownItem<T> }) => (
    <TouchableOpacity
      style={styles.dropdownItem}
      onPress={() => {
        onValueChange(item.value);
        setModalVisible(false);
      }}
    >
      <Text style={styles.dropdownItemText}>{item.label}</Text>
    </TouchableOpacity>
  );

  return (
    <>
      <Pressable
        style={({ pressed }) => [
          styles.dropdownInput,
          pressed && styles.dropdownInputPressed,
          !enabled && styles.dropdownDisabled,
```

```
    ]}
    onPress={() => enabled && setModalVisible(true)}
    hitSlop={{ top: 20, bottom: 20, left: 20, right: 20 }}
  >
    <Text style={[styles.dropdownText, !selectedItem && styles.dropdownPlaceholder]}>
      {displayText}
    </Text>
    <Ionicons name="chevron-down" size={24} color="#666666" style={styles.dropdownIcon} />
  </Pressable>

  <Modal
    visible={modalVisible}
    transparent
    animationType="fade"
    onRequestClose={() => setModalVisible(false)}
  >
    <View style={styles.modalOverlay}>
      <View style={styles.dropdownModal}>
        <FlatList
          data={items}
          renderItem={renderItem}
          keyExtractor={item => `${item.value}`}
          style={styles.dropdownList}
        />
        <Pressable
          style={styles.dropdownCancelButton}
          onPress={() => setModalVisible(false)}
```

```
                >
                  <Text style={styles.dropdownCancelText}>Скасувати</Text>
                </Pressable>
              </View>
            </View>
          </Modal>
        </>
    );
};

const AddEditPaymentScreen: React.FC<AddEditPaymentNavProps> = ({ navigation, route }) => {
    const existingPayment = route.params?.payment;

    const addPaymentAction = useAppStore(state => state.addPayment);
    const updatePaymentAction = useAppStore(state => state.updatePayment);
    const categories = useAppStore(state => state.categories);
    const wallets = useAppStore(state => state.wallets);
    const defaultDaysBefore = useAppStore(state => state.reminderSettings?.defaultDaysBefore ?? 3);

    // --- State Hooks ---
    const [serviceName, setServiceName] = useState(existingPayment?.serviceName ?? '');
    const [amount, setAmount] = useState(existingPayment?.amount != null ? existingPayment.amount.toString() : '');
    const [currency, setCurrency] = useState(existingPayment?.currency ?? CURRENCIES[0]);
    const [dueDate, setDueDate] = useState(existingPayment?.dueDate ? new Date(existingPayment.dueDate) : new
Date());
    const [paymentDate, setPaymentDate] = useState(existingPayment?.paymentDate ? new
Date(existingPayment.paymentDate) : new Date());
    const [description, setDescription] = useState(existingPayment?.description ?? '');
```

```
  const [status, setStatus] = useState<PaymentStatus>(existingPayment?.status ?? 'pending');
  const [selectedCategoryId, setSelectedCategoryId] = useState<string | undefined>(existingPayment?.categoryId);
  const [selectedWalletId, setSelectedWalletId] = useState<string | undefined>(existingPayment?.walletId);
  const [isRecurring, setIsRecurring] = useState(existingPayment?.isRecurring ?? false);
  const [recurrenceInterval, setRecurrenceInterval] = useState<Payment['recurrenceInterval'] | undefined>(existingPayment?.recurrenceInterval);
  const [reminderEnabled, setReminderEnabled] = useState(existingPayment?.reminderEnabled ?? true);
  const [daysBeforeReminder, setDaysBeforeReminder] = useState(
    existingPayment?.daysBeforeReminder != null ? existingPayment.daysBeforeReminder.toString() : defaultDaysBefore.toString()
  );
  const [showDueDatePicker, setShowDueDatePicker] = useState(false);
  const [showPaymentDatePicker, setShowPaymentDatePicker] = useState(false);

  // --- Effects ---
  useEffect(() => {
    navigation.setOptions({ headerTitle: existingPayment ? 'Редагувати платіж' : 'Новий платіж' });
  }, [navigation, existingPayment]);

  // --- Handlers ---
  const onDueDateChange = (event: DateTimePickerEvent, selectedDate?: Date) => {
    setShowDueDatePicker(Platform.OS === 'ios');
    if (selectedDate) {
      selectedDate.setHours(0, 0, 0, 0); // Normalize to midnight
      setDueDate(selectedDate);
    }
  };
```

```
  const onPaymentDateChange = (event: DateTimePickerEvent, selectedDate?: Date) => {
    setShowPaymentDatePicker(Platform.OS === 'ios');
    if (selectedDate) {
      selectedDate.setHours(0, 0, 0, 0); // Normalize to midnight
      setPaymentDate(selectedDate);
    }
  };

  const formatDateForStorage = (date: Date): string => date.toISOString().split('T')[0];

  const calculateTriggerDateTime = (dueDate: Date, daysBefore: number): Date => {
    const triggerDate = new Date(dueDate.getTime());
    triggerDate.setHours(0, 0, 0, 0); // Normalize to midnight
    triggerDate.setDate(triggerDate.getDate() - daysBefore);
    triggerDate.setHours(9, 0, 0, 0); // Set to 9:00 AM
    console.log(
      `Розраховано triggerDateTime для dueDate=${dueDate.toLocaleString('uk-UA', { timeZone: 'Europe/Kiev' })}, ` +
      `daysBefore=${daysBefore}: ${triggerDate.toLocaleString('uk-UA', { timeZone: 'Europe/Kiev' })}`
    );
    return triggerDate;
  };

  const handleSave = async () => {
    console.log('Starting handleSave with data:', { serviceName, amount, currency, dueDate, status, categoryId: selectedCategoryId, walletId: selectedWalletId });

    if (!serviceName.trim()) {
```

```
            Alert.alert('Помилка', 'Введіть назву послуги.');
            return;
        }
        if (!amount.trim()) {
            Alert.alert('Помилка', 'Введіть суму.');
            return;
        }
        const numericAmount = parseFloat(amount.replace(',', '.'));
        if (isNaN(numericAmount) || numericAmount <= 0) {
            Alert.alert('Помилка', 'Сума повинна бути позитивним числом.');
            return;
        }
        const numericDaysBefore = parseInt(daysBeforeReminder, 10);
        if (isNaN(numericDaysBefore) || numericDaysBefore < 0) {
            Alert.alert('Помилка', 'К-ть днів для нагадування має бути числом >= 0.');
            return;
        }

        // Перевірка, чи dueDate є валідною
        if (isNaN(dueDate.getTime())) {
            Alert.alert('Помилка', 'Некоректна дата терміну оплати.');
            return;
        }

        const paymentDataForAction: Omit<Payment, 'id' | 'createdAt' | 'updatedAt' | 'notificationId'> & { id?: string; notificationId?: string } = {
            id: existingPayment?.id,
            serviceName: serviceName.trim(),
```

```
        amount: numericAmount,
        currency,
        dueDate: formatDateForStorage(dueDate),
        description: description.trim() || undefined,
        status,
        categoryId: selectedCategoryId,
        walletId: selectedWalletId,
        isRecurring,
        recurrenceInterval: isRecurring ? recurrenceInterval : undefined,
        reminderEnabled,
        daysBeforeReminder: numericDaysBefore,
        paymentDate: status === 'paid' ? formatDateForStorage(paymentDate) : undefined,
        notificationId: existingPayment?.notificationId,
      };

      try {
        // Cancel existing notification if it exists
        if (existingPayment?.notificationId) {
          await cancelPaymentReminder(existingPayment.notificationId);
          console.log(`Скасовано попереднє нагадування для "${paymentDataForAction.serviceName}" з ID: ${existingPayment.notificationId}`);
        }

        // Schedule new notification if reminder is enabled and payment is not paid
        let notificationId: string | null = null;
        if (reminderEnabled && status !== 'paid') {
          const triggerDateTime = calculateTriggerDateTime(dueDate, numericDaysBefore);
          // Перевірка, чи triggerDateTime є в майбутньому
```

```
            if (triggerDateTime.getTime() <= Date.now() + 60 * 1000) {
                Alert.alert('Помилка', `Дата нагадування (${triggerDateTime.toLocaleDateString('uk-UA')}) вже минула або надто близько. Виберіть пізнішу дату оплати або зменшіть кількість днів для нагадування.`);
                return;
            }
            notificationId = await schedulePaymentReminder(paymentDataForAction, triggerDateTime);
            paymentDataForAction.notificationId = notificationId || undefined;
        }

        // Save or update payment
        console.log('Saving payment with data:', paymentDataForAction);
        if (existingPayment) {
            await updatePaymentAction({ ...existingPayment, ...paymentDataForAction });
        } else {
            await addPaymentAction(paymentDataForAction);
        }
        console.log('Payment saved successfully, navigating back');
        navigation.goBack();
    } catch (error) {
        console.error('Помилка збереження платежу:', error);
        Alert.alert('Помилка', `Не вдалося зберегти платіж: ${error.message}`);
    }
};

// --- Dropdown Data ---
const currencyItems: DropdownItem<string>[] = CURRENCIES.map(c => ({ label: c, value: c }));
const statusItems: DropdownItem<PaymentStatus>[] = PAYMENT_STATUSES.map(s => ({
    label: s === 'pending' ? 'Очікує' : s === 'paid' ? 'Сплачено' : 'Прострочено',
```

```
    value: s,
  }));
  const categoryItems: DropdownItem<string | undefined>[] = [
    { label: 'Не обрано', value: undefined },
    ...categories.map(cat => ({ label: cat.name, value: cat.id })),
  ];
  const walletItems: DropdownItem<string | undefined>[] = [
    { label: 'Не обрано', value: undefined },
    ...wallets.map(w => ({ label: w.name, value: w.id })),
  ];
  const recurrenceItems: DropdownItem<Payment['recurrenceInterval'] | undefined>[] =
RECURRENCE_INTERVALS.map(i => ({
    label: i ? i.charAt(0).toUpperCase() + i.slice(1) : 'Не вказано',
    value: i,
  }));

  // --- Render ---
  return (
    <ScrollView style={styles.scrollContainer} contentContainerStyle={styles.container}>
      <View style={styles.formGroup}>
        <Text style={styles.label}>Назва послуги <Text style={styles.required}>*</Text></Text>
        <TextInput
          style={styles.input}
          value={serviceName}
          onChangeText={setServiceName}
          placeholder="Наприклад, Інтернет"
          placeholderTextColor="#999999"
        />
```

```
</View>

<View style={styles.formGroup}>
  <Text style={styles.label}>Сума <Text style={styles.required}>*</Text></Text>
  <View style={styles.row}>
    <TextInput
      style={[styles.input, styles.amountInput]}
      value={amount}
      onChangeText={setAmount}
      keyboardType="decimal-pad"
      placeholder="0.00"
      placeholderTextColor="#999999"
    />
    <View style={styles.currencyDropdown}>
      <Dropdown
        selectedValue={currency}
        onValueChange={setCurrency}
        items={currencyItems}
      />
    </View>
  </View>
</View>

<View style={styles.formGroup}>
  <Text style={styles.label}>Термін оплати <Text style={styles.required}>*</Text></Text>
  <Pressable
    onPress={() => setShowDueDatePicker(true)}
```

```
    style={({ pressed }) => [
      styles.dateInput,
      pressed && styles.dateInputPressed,
    ]}
    hitSlop={{ top: 20, bottom: 20, left: 20, right: 20 }}
  >
    <Ionicons name="calendar-outline" size={24} color="#666666" style={styles.dateIcon} />
    <Text style={styles.dateText}>
      {dueDate.toLocaleDateString('uk-UA', {
        year: 'numeric',
        month: 'long',
        day: 'numeric',
      })}
    </Text>
  </Pressable>
  {showDueDatePicker && (
    <DateTimePicker
      testID="dueDatePicker"
      value={dueDate}
      mode="date"
      display={Platform.OS === 'ios' ? 'inline' : 'calendar'}
      onChange={onDueDateChange}
      accentColor="#4caf50"
      themeVariant="light"
      locale="uk-UA"
    />
  )}
```

```
</View>

<View style={styles.formGroup}>
  <Text style={styles.label}>Статус</Text>
  <Dropdown
    selectedValue={status}
    onValueChange={setStatus}
    items={statusItems}
  />
</View>

{status === 'paid' && (
  <View style={styles.formGroup}>
    <Text style={styles.label}>Дата оплати</Text>
    <Pressable
      onPress={() => setShowPaymentDatePicker(true)}
      style={({ pressed }) => [
        styles.dateInput,
        pressed && styles.dateInputPressed,
      ]}
      hitSlop={{ top: 20, bottom: 20, left: 20, right: 20 }}
    >
      <Ionicons name="calendar-outline" size={24} color="#666666" style={styles.dateIcon} />
      <Text style={styles.dateText}>
        {paymentDate.toLocaleDateString('uk-UA', {
          year: 'numeric',
          month: 'long',
```

```
                day: 'numeric',
              })}
            </Text>
          </Pressable>
          {showPaymentDatePicker && (
            <DateTimePicker
              testID="paymentDatePicker"
              value={paymentDate}
              mode="date"
              display={Platform.OS === 'ios' ? 'inline' : 'calendar'}
              onChange={onPaymentDateChange}
              accentColor="#4caf50"
              themeVariant="light"
              locale="uk-UA"
            />
          )}
        </View>
      )}

      <View style={styles.formGroup}>
        <Text style={styles.label}>Категорія</Text>
        <Dropdown
          selectedValue={selectedCategoryId}
          onValueChange={setSelectedCategoryId}
          items={categoryItems}
          enabled={categories.length > 0}
        />
```

```
</View>

<View style={styles.formGroup}>
  <Text style={styles.label}>Гаманець</Text>
  <Dropdown
    selectedValue={selectedWalletId}
    onValueChange={setSelectedWalletId}
    items={walletItems}
    enabled={wallets.length > 0}
  />
</View>

<View style={styles.formGroup}>
  <Text style={styles.label}>Опис</Text>
  <TextInput
    style={[styles.input, styles.textArea]}
    value={description}
    onChangeText={setDescription}
    multiline
    numberOfLines={4}
    placeholder="Додаткова інформація"
    placeholderTextColor="#999999"
  />
</View>

<View style={styles.switchRow}>
  <Text style={styles.labelSwitch}>Регулярний платіж</Text>
```

```
    <Switch
      value={isRecurring}
      onValueChange={setIsRecurring}
      trackColor={{ false: '#d1d1d1', true: '#81c784' }}
      thumbColor={isRecurring ? '#4caf50' : '#ffffff'}
    />
  </View>

  {isRecurring && (
    <View style={styles.formGroup}>
      <Text style={styles.label}>Інтервал повторення</Text>
      <Dropdown
        selectedValue={recurrenceInterval}
        onValueChange={setRecurrenceInterval}
        items={recurrenceItems}
      />
    </View>
  )}

  <View style={styles.switchRow}>
    <Text style={styles.labelSwitch}>Нагадування</Text>
    <Switch
      value={reminderEnabled}
      onValueChange={setReminderEnabled}
      trackColor={{ false: '#d1d1d1', true: '#81c784' }}
      thumbColor={reminderEnabled ? '#4caf50' : '#ffffff'}
    />
```

```
</View>

{reminderEnabled && (
  <View style={styles.formGroup}>
    <Text style={styles.label}>Нагадати за (днів)</Text>
    <TextInput
      style={styles.input}
      value={daysBeforeReminder}
      onChangeText={setDaysBeforeReminder}
      keyboardType="number-pad"
      placeholder={`Стандартно: ${defaultDaysBefore}`}
      placeholderTextColor="#999999"
    />
  </View>
)}

<View style={styles.actionsContainer}>
  <Pressable
    style={({ pressed }) => [
      styles.button,
      styles.saveButton,
      pressed && styles.buttonPressed,
    ]}
    onPress={handleSave}
  >
    <Ionicons name="save-outline" size={24} color="#ffffff" />
    <Text style={styles.buttonText}>Зберегти</Text>
```

```
            </Pressable>
            <Pressable
              style={({ pressed }) => [
                styles.button,
                styles.cancelButton,
                pressed && styles.buttonPressed,
              ]}
              onPress={() => navigation.goBack()}
            >
              <Ionicons name="close-outline" size={24} color="#333333" />
              <Text style={[styles.buttonText, styles.cancelButtonText]}>Скасувати</Text>
            </Pressable>
          </View>
        </ScrollView>
    );
};
export default AddEditPaymentScreen;
```

ДОДАТОК Б

Фрагменти програмного коду. Функція створення категорії

```
import React, { useState, useEffect } from 'react';
import { View, Text, StyleSheet, TextInput, Button, Alert, ScrollView, TouchableOpacity } from 'react-native';
import useAppStore from '../../store';
import { Category } from '../../types';
import { Ionicons } from '@expo/vector-icons';

const ICON_SUGGESTIONS = ['home', 'cart', 'car', 'medkit', 'school', 'airplane', 'game-controller', 'musical-notes', 'paw', 'construct'];
const COLOR_SUGGESTIONS = ['#FF6B6B', '#4ECDC4', '#45B7D1', '#FED766', '#2D3047', '#F2EFEA', '#FFD166', '#06D6A0', '#118AB2', '#EF476F'];

const AddEditCategoryScreen: ({ navigation, route }: { navigation: any; route: any }) => React.JSX.Element = ({ navigation, route }) => {
  const existingCategory = route.params?.category;

  const addCategoryAction = useAppStore(state => state.addCategory);
  const updateCategoryAction = useAppStore(state => state.updateCategory);

  const [name, setName] = useState(existingCategory?.name || '');
  const [iconName, setIconName] = useState(existingCategory?.iconName || '');
  const [color, setColor] = useState(existingCategory?.color || COLOR_SUGGESTIONS[0]);

  useEffect(() => {
    navigation.setOptions({ headerTitle: existingCategory ? 'Редагувати категорію' : 'Нова категорія' });
  }, [navigation, existingCategory]);
```

```
const handleSave = async () => {
    if (!name.trim()) {
        Alert.alert('Помилка', 'Назва категорії є обов\'язковою.');
        return;
    }

    const categoryData: Omit<Category, 'id'> & { id?: string } = {
        id: existingCategory?.id,
        name: name.trim(),
        iconName: iconName.trim() || undefined,
        color: color.trim() || undefined,
    };

    console.log('Saving category:', categoryData);

    try {
        if (existingCategory) {
            await updateCategoryAction(categoryData as Category);
        } else {
            await addCategoryAction(categoryData);
        }
        console.log('Category saved successfully');
        navigation.goBack();
    } catch (error: any) {
        console.error('Error saving category:', error);
        Alert.alert('Помилка', `Не вдалося зберегти категорію: ${error.message || 'Невідома помилка'}`);
    }
```

```
  };

  return (
    <ScrollView style={styles.container} contentContainerStyle={styles.contentContainer} keyboardShouldPersistTaps="handled">
      <View style={styles.formGroup}>
        <Text style={styles.label}>Назва категорії <Text style={styles.required}>*</Text></Text>
        <TextInput
          style={styles.input}
          value={name}
          onChangeText={setName}
          placeholder="Наприклад, Продукти"
        />
      </View>

      <View style={styles.formGroup}>
        <Text style={styles.label}>Назва іконки (з Ionicons, напр., home-outline)</Text>
        <TextInput
          style={styles.input}
          value={iconName}
          onChangeText={setIconName}
          placeholder="Необов’язково"
        />
        <View style={styles.suggestionsContainer}>
          {ICON_SUGGESTIONS.map(icon => (
            <TouchableOpacity key={icon} onPress={() => setIconName(icon + '-outline')} style={styles.suggestionChip}>
              <Ionicons name={icon + '-outline' as any} size={20} color="#555" />
```

```
                </TouchableOpacity>
              ))}
            </View>
          </View>

          <View style={styles.formGroup}>
            <Text style={styles.label}>Колір (HEX, напр., #FF6347)</Text>
            <TextInput
              style={styles.input}
              value={color}
              onChangeText={setColor}
              placeholder="#4CAF50"
            />
            <View style={styles.suggestionsContainer}>
              {COLOR_SUGGESTIONS.map(c => (
                <TouchableOpacity key={c} onPress={() => setColor(c)} style={[styles.colorChip, { backgroundColor:
c }]} />
              ))}
            </View>
          </View>

          <View style={styles.actionsContainer}>
            <TouchableOpacity style={[styles.button, styles.saveButton]} onPress={handleSave}>
              <Ionicons name="save-outline" size={20} color="white" />
              <Text style={styles.buttonTextWhite}>Зберегти</Text>
            </TouchableOpacity>
            <TouchableOpacity style={[styles.button, styles.cancelButton]} onPress={() => navigation.goBack()}>
              <Ionicons name="close-circle-outline" size={20} color="#555" />
```

```
                <Text style={styles.buttonTextDark}>Скасувати</Text>
              </TouchableOpacity>
            </View>
          </ScrollView>
    );
};

export default AddEditCategoryScreen;
```

ДОДАТОК В

Фрагменти програмного коду. Функції опрацювання серверних запитів

```
import { create } from 'zustand';
import { AppState, WalletState, SettingsState, CategoryState, PaymentState } from './types';
import { ReminderSettings, Wallet, Category, Payment, PaymentStatus } from '../types';
import { generateUUID } from '../utils/uuid';
import { schedulePaymentReminder, cancelPaymentReminder } from '../services/notificationService';

const API_URL = 'http://192.168.1.102:3000/api';

const initialCategories: Category[] = [
    { id: generateUUID(), name: 'Комунальні послуги', iconName: 'home-outline', color: '#FF6347' },
    { id: generateUUID(), name: 'Інтернет та ТБ', iconName: 'wifi-outline', color: '#1E90FF' },
    { id: generateUUID(), name: 'Мобільний зв’язок', iconName: 'call-outline', color: '#32CD32' },
    { id: generateUUID(), name: 'Кредити', iconName: 'card-outline', color: '#FFD700' },
    { id: generateUUID(), name: 'Інше', iconName: 'apps-outline', color: '#A9A9A9' },
];

const initialWallets: Wallet[] = [
    { id: generateUUID(), name: 'Готівка', iconName: 'cash-outline', color: '#20B2AA' },
    { id: generateUUID(), name: 'Карта "Monobank"', iconName: 'card-outline', color: '#333333' },
    { id: generateUUID(), name: 'Карта "Privat"', iconName: 'card-outline', color: '#FF8C00' },
];

const defaultReminderSettings: ReminderSettings = {
```

```
    id: 'default-reminder-settings',
    defaultDaysBefore: 3,
    allowNotifications: true,
};

const useAppStore = create<AppState>((set, get) => ({
    // Payment Slice
    payments: [],
    isLoadingPayments: false,
    errorPayments: null,
    loadPayments: async () => {
        set({ isLoadingPayments: true, errorPayments: null });
        try {
            const response = await fetch(`${API_URL}/payments`);
            if (!response.ok) throw new Error(`HTTP error! status: ${response.status}`);
            const data = await response.json();
            const payments = data.map((p: any) => ({
                ...p,
                serviceName: p.service_name,
                dueDate: p.due_date,
                categoryId: p.category_id,
                walletId: p.wallet_id,
                reminderEnabled: p.reminder_enabled,
                daysBeforeReminder: p.days_before_reminder,
                notificationId: p.notification_id,
                paymentDate: p.payment_date,
                isRecurring: p.is_recurring,
```

```
            recurrenceInterval: p.recurrence_interval,
            createdAt: p.created_at,
            updatedAt: p.updated_at,
            amount: typeof p.amount === 'string' ? parseFloat(p.amount) : p.amount,
        }));
        console.log('Loaded payments:', payments);
        set({ payments, isLoadingPayments: false });
    } catch (error) {
        console.error('Error loading payments:', error);
        set({ errorPayments: 'Failed to load payments', isLoadingPayments: false });
    }
},
addPayment: async (paymentData) => {
    const newPayment = {
        ...paymentData,
        id: paymentData.id || generateUUID(),
        serviceName: paymentData.serviceName || 'Без назви',
        status: paymentData.status || 'pending',
        reminderEnabled: paymentData.reminderEnabled || false,
        daysBeforeReminder: paymentData.daysBeforeReminder || 3,
        isRecurring: paymentData.isRecurring || false,
    };
    try {
        console.log('Sending POST /api/payments with data:', newPayment);
        const response = await fetch(`${API_URL}/payments`, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
```

```
    body: JSON.stringify({
        id: newPayment.id,
        service_name: newPayment.serviceName,
        amount: newPayment.amount,
        currency: newPayment.currency,
        due_date: newPayment.dueDate,
        description: newPayment.description,
        status: newPayment.status,
        category_id: newPayment.categoryId,
        wallet_id: newPayment.walletId,
        reminder_enabled: newPayment.reminderEnabled,
        days_before_reminder: newPayment.daysBeforeReminder,
        notification_id: newPayment.notificationId,
        payment_date: newPayment.paymentDate,
        is_recurring: newPayment.isRecurring,
        recurrence_interval: newPayment.recurrenceInterval,
    }),
});
if (!response.ok) {
    const errorText = await response.text();
    throw new Error(`HTTP error! status: ${response.status}, message: ${errorText}`);
}
const savedPayment = await response.json();
const parsedPayment = {
    ...savedPayment,
    serviceName: savedPayment.service_name,
    dueDate: savedPayment.due_date,
```

```
            categoryId: savedPayment.category_id,
            walletId: savedPayment.wallet_id,
            reminderEnabled: savedPayment.reminder_enabled,
            daysBeforeReminder: savedPayment.days_before_reminder,
            notificationId: savedPayment.notification_id,
            paymentDate: savedPayment.payment_date,
            isRecurring: savedPayment.is_recurring,
            recurrenceInterval: savedPayment.recurrence_interval,
            createdAt: savedPayment.created_at,
            updatedAt: savedPayment.updated_at,
            amount: typeof savedPayment.amount === 'string' ? parseFloat(savedPayment.amount) : savedPayment.amount,
        };
        console.log('Payment added successfully:', parsedPayment);
        set((state) => ({
            payments: [...state.payments, parsedPayment],
            errorPayments: null,
        }));
        if (parsedPayment.reminderEnabled && parsedPayment.dueDate) {
            try {
                const notificationId = await schedulePaymentReminder(parsedPayment);
                if (notificationId) {
                    await get().updatePayment({ ...parsedPayment, notificationId });
                }
            } catch (error) {
                console.error('Failed to schedule reminder:', error);
            }
        }
```

```
    } catch (error) {
        console.error('Error adding payment:', error);
        throw error;
    }
},
updatePayment: async (paymentData) => {
    if (!paymentData.id) {
        throw new Error('Payment ID is required');
    }
    const paymentToUpdate = {
        ...paymentData,
        serviceName: paymentData.serviceName || 'Без назви',
        status: paymentData.status || 'pending',
        reminderEnabled: paymentData.reminderEnabled || false,
        daysBeforeReminder: paymentData.daysBeforeReminder || 3,
        isRecurring: paymentData.isRecurring || false,
    };
    try {
        console.log('Sending PUT /api/payments with data:', paymentToUpdate);
        const response = await fetch(`${API_URL}/payments/${paymentData.id}`, {
            method: 'PUT',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({
                service_name: paymentToUpdate.serviceName,
                amount: paymentToUpdate.amount,
                currency: paymentToUpdate.currency,
                due_date: paymentToUpdate.dueDate,
```

```
        description: paymentToUpdate.description,
        status: paymentToUpdate.status,
        category_id: paymentToUpdate.categoryId,
        wallet_id: paymentToUpdate.walletId,
        reminder_enabled: paymentToUpdate.reminderEnabled,
        days_before_reminder: paymentToUpdate.daysBeforeReminder,
        notification_id: paymentToUpdate.notificationId,
        payment_date: paymentToUpdate.paymentDate,
        is_recurring: paymentToUpdate.isRecurring,
        recurrence_interval: paymentToUpdate.recurrenceInterval,
    }),
});
if (!response.ok) {
    const errorText = await response.text();
    throw new Error(`HTTP error! status: ${response.status}, message: ${errorText}`);
}
const updatedPayment = await response.json();
const parsedPayment = {
    ...updatedPayment,
    serviceName: updatedPayment.service_name,
    dueDate: updatedPayment.due_date,
    categoryId: updatedPayment.category_id,
    walletId: updatedPayment.wallet_id,
    reminderEnabled: updatedPayment.reminder_enabled,
    daysBeforeReminder: updatedPayment.days_before_reminder,
    notificationId: updatedPayment.notification_id,
    paymentDate: updatedPayment.payment_date,
```

```
            isRecurring: updatedPayment.is_recurring,
            recurrenceInterval: updatedPayment.recurrence_interval,
            createdAt: updatedPayment.created_at,
            updatedAt: updatedPayment.updated_at,
            amount: typeof updatedPayment.amount === 'string' ? parseFloat(updatedPayment.amount) : updatedPayment.amount,
          };
          console.log('Payment updated successfully:', parsedPayment);
          set((state) => ({
            payments: state.payments.map((p) => (p.id === parsedPayment.id ? parsedPayment : p)),
            errorPayments: null,
          }));
          if (parsedPayment.reminderEnabled && parsedPayment.dueDate) {
            try {
              const notificationId = await schedulePaymentReminder(parsedPayment);
              if (notificationId && notificationId !== parsedPayment.notificationId) {
                await get().updatePayment({ ...parsedPayment, notificationId });
              }
            } catch (error) {
              console.error('Failed to schedule reminder:', error);
            }
          } else if (parsedPayment.notificationId) {
            await cancelPaymentReminder(parsedPayment.notificationId);
            await get().updatePayment({ ...parsedPayment, notificationId: undefined });
          }
        } catch (error) {
          console.error('Error updating payment:', error);
          throw error;
```

```
    }
  },
  deletePayment: async (paymentId) => {
    try {
      const payment = get().payments.find((p) => p.id === paymentId);
      if (payment?.notificationId) {
        await cancelPaymentReminder(payment.notificationId);
      }
      console.log('Sending DELETE /api/payments/', paymentId);
      const response = await fetch(`${API_URL}/payments/${paymentId}`, {
        method: 'DELETE',
      });
      if (!response.ok) {
        const errorText = await response.text();
        throw new Error(`HTTP error! status: ${response.status}, message: ${errorText}`);
      }
      set((state) => ({
        payments: state.payments.filter((p) => p.id !== paymentId),
        errorPayments: null,
      }));
    } catch (error) {
      console.error('Error deleting payment:', error);
      throw error;
    }
  },
  togglePaymentStatus: async (paymentId) => {
    const payment = get().payments.find((p) => p.id === paymentId);
```

```
    if (!payment) {
      throw new Error('Payment not found');
    }
    const newStatus: PaymentStatus = payment.status === 'paid' ? 'pending' : 'paid';
    const paymentDate = newStatus === 'paid' ? new Date().toISOString().split('T')[0] : undefined;
    console.log('Toggling payment status:', { id: paymentId, newStatus, paymentDate });
    await get().updatePayment({
      ...payment,
      serviceName: payment.serviceName || 'Без назви',
      status: newStatus,
      paymentDate,
    });
  },

  // Category Slice
  categories: [],
  isLoadingCategories: false,
  errorCategories: null,
  loadCategories: async () => {
    set({ isLoadingCategories: true, errorCategories: null });
    try {
      console.log('Fetching categories from:', `${API_URL}/categories`);
      const response = await fetch(`${API_URL}/categories`);
      if (!response.ok) {
        throw new Error(`HTTP error! status: ${response.status}`);
      }
      let categories = await response.json();
```

```
            console.log('Loaded categories:', categories);
            if (categories.length === 0) {
                console.log('No categories found, adding initial categories...');
                for (const category of initialCategories) {
                    await fetch(`${API_URL}/categories`, {
                        method: 'POST',
                        headers: { 'Content-Type': 'application/json' },
                        body: JSON.stringify(category),
                    });
                }
                categories = initialCategories;
            }
            set({ categories, isLoadingCategories: false });
        } catch (error) {
            console.error('Error loading categories:', error);
            set({ errorCategories: `Failed to load categories: ${error.message || 'Unknown error'}`, isLoadingCategories: false
});
        }
    },
    addCategory: async (categoryData) => {
        const newCategory: Category = {
            ...(categoryData as Omit<Category, 'id'>),
            id: categoryData.id || generateUUID(),
        };
        try {
            console.log('Sending POST /api/categories with data:', newCategory);
            const response = await fetch(`${API_URL}/categories`, {
                method: 'POST',
```

```
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(newCategory),
        });
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }
        const savedCategory = await response.json();
        console.log('Category added successfully:', savedCategory);
        set((state) => ({
            categories: [...state.categories, savedCategory],
            errorCategories: null,
        }));
    } catch (error) {
        console.error('Error adding category:', error);
        throw error;
    }
},
updateCategory: async (updatedCategoryData) => {
    try {
        console.log('Sending PUT /api/categories with data:', updatedCategoryData);
        const response = await fetch(`${API_URL}/categories/${updatedCategoryData.id}`, {
            method: 'PUT',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(updatedCategoryData),
        });
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
```

```
        }
        const updatedCategory = await response.json();
        console.log('Category updated successfully:', updatedCategory);
        set((state) => ({
          categories: state.categories.map((c) =>
            c.id === updatedCategory.id ? updatedCategory : c
          ),
          errorCategories: null,
        }));
      } catch (error) {
        console.error('Error updating category:', error);
        throw error;
      }
    },
    deleteCategory: async (categoryId) => {
      try {
        console.log('Sending DELETE /api/categories with id:', categoryId);
        const response = await fetch(`${API_URL}/categories/${categoryId}`, {
          method: 'DELETE',
        });
        if (!response.ok) {
          throw new Error(`HTTP error! status: ${response.status}`);
        }
        console.log('Category deleted successfully:', categoryId);
        set((state) => ({
          categories: state.categories.filter((c) => c.id !== categoryId),
          errorCategories: null,
```

```
        }));
    } catch (error) {
        console.error('Error deleting category:', error);
        set({ errorCategories: `Failed to delete category: ${error.message || 'Unknown error'}` });
    }
},

// Wallet Slice
wallets: [],
isLoadingWallets: false,
errorWallets: null,
loadWallets: async () => {
    set({ isLoadingWallets: true, errorWallets: null });
    try {
        const response = await fetch(`${API_URL}/wallets`);
        if (!response.ok) throw new Error(`HTTP error! status: ${response.status}`);
        let wallets = await response.json();
        console.log('Loaded wallets:', wallets);
        if (wallets.length === 0) {
            for (const wallet of initialWallets) {
                await fetch(`${API_URL}/wallets`, {
                    method: 'POST',
                    headers: { 'Content-Type': 'application/json' },
                    body: JSON.stringify(wallet),
                });
            }
            wallets = initialWallets;
```

```
        }
        set({ wallets, isLoadingWallets: false });
    } catch (error) {
        console.error('Error loading wallets:', error);
        set({ errorWallets: 'Failed to load wallets', isLoadingWallets: false });
    }
},
addWallet: async (walletData) => {
    const newWallet: Wallet = {
        ...(walletData as Omit<Wallet, 'id'>),
        id: walletData.id || generateUUID(),
    };
    try {
        console.log('Sending POST /api/wallets with data:', newWallet);
        const response = await fetch(`${API_URL}/wallets`, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(newWallet),
        });
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }
        const savedWallet = await response.json();
        console.log('Wallet added successfully:', savedWallet);
        set((state) => ({
            wallets: [...state.wallets, savedWallet],
            errorWallets: null,
```

```
        }));
    } catch (error) {
        console.error('Error adding wallet:', error);
        throw error;
    }
},
updateWallet: async (walletData) => {
    try {
        console.log('Sending PUT /api/wallets with data:', walletData);
        const response = await fetch(`${API_URL}/wallets/${walletData.id}`, {
            method: 'PUT',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(walletData),
        });
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }
        const updatedWallet = await response.json();
        console.log('Wallet updated successfully:', updatedWallet);
        set((state) => ({
            wallets: state.wallets.map((w) => (w.id === updatedWallet.id ? updatedWallet : w)),
            errorWallets: null,
        }));
    } catch (error) {
        console.error('Error updating wallet:', error);
        throw error;
    }
```

```
},
deleteWallet: async (walletId) => {
  try {
    const response = await fetch(`${API_URL}/wallets/${walletId}`, {
      method: 'DELETE',
    });
    if (!response.ok) throw new Error(`HTTP error! status: ${response.status}`);
    set((state) => ({
      wallets: state.wallets.filter((w) => w.id !== walletId),
      errorWallets: null,
    }));
  } catch (error) {
    console.error('Error deleting wallet:', error);
    set({ errorWallets: 'Failed to delete wallet' });
  }
},

// Settings Slice
reminderSettings: defaultReminderSettings,
isLoadingSettings: false,
loadReminderSettings: async () => {
  set({ isLoadingSettings: true });
  try {
    const response = await fetch(`${API_URL}/reminder-settings`);
    if (!response.ok) throw new Error(`HTTP error! status: ${response.status}`);
    const settings = await response.json();
    console.log('Loaded reminder settings:', settings);
```

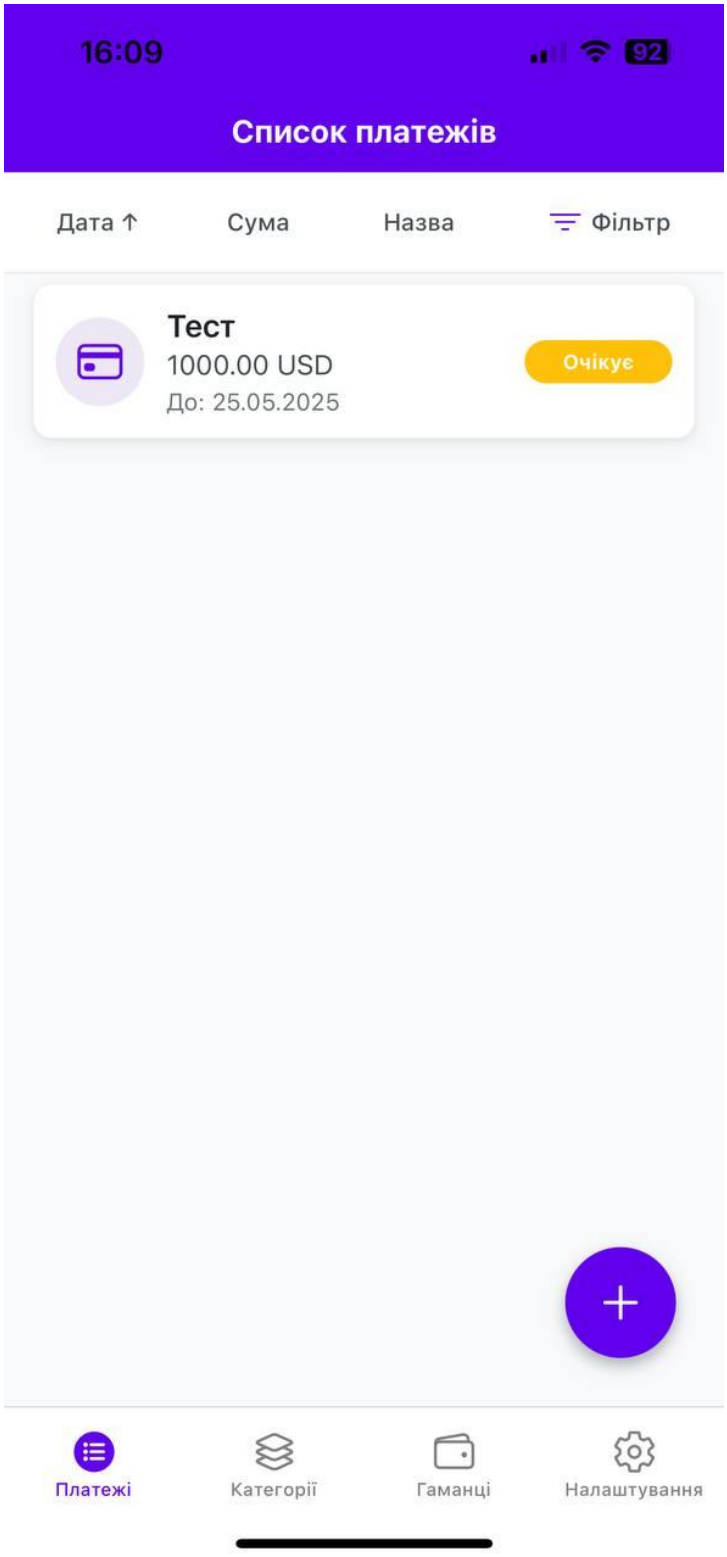
```
            set({ reminderSettings: settings, isLoadingSettings: false });
        } catch (error) {
            console.error('Error loading reminder settings:', error);
            set({ reminderSettings: defaultReminderSettings, isLoadingSettings: false });
        }
    },
    updateReminderSettings: async (settings) => {
        try {
            const updatedSettings = { ...settings, id: settings.id || 'default-reminder-settings' };
            const response = await fetch(`${API_URL}/reminder-settings`, {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify(updatedSettings),
            });
            if (!response.ok) throw new Error(`HTTP error! status: ${response.status}`);
            const savedSettings = await response.json();
            console.log('Updated reminder settings:', savedSettings);
            set({ reminderSettings: savedSettings });
        } catch (error) {
            console.error('Error updating reminder settings:', error);
        }
    },
}));

export default useAppStore;
```

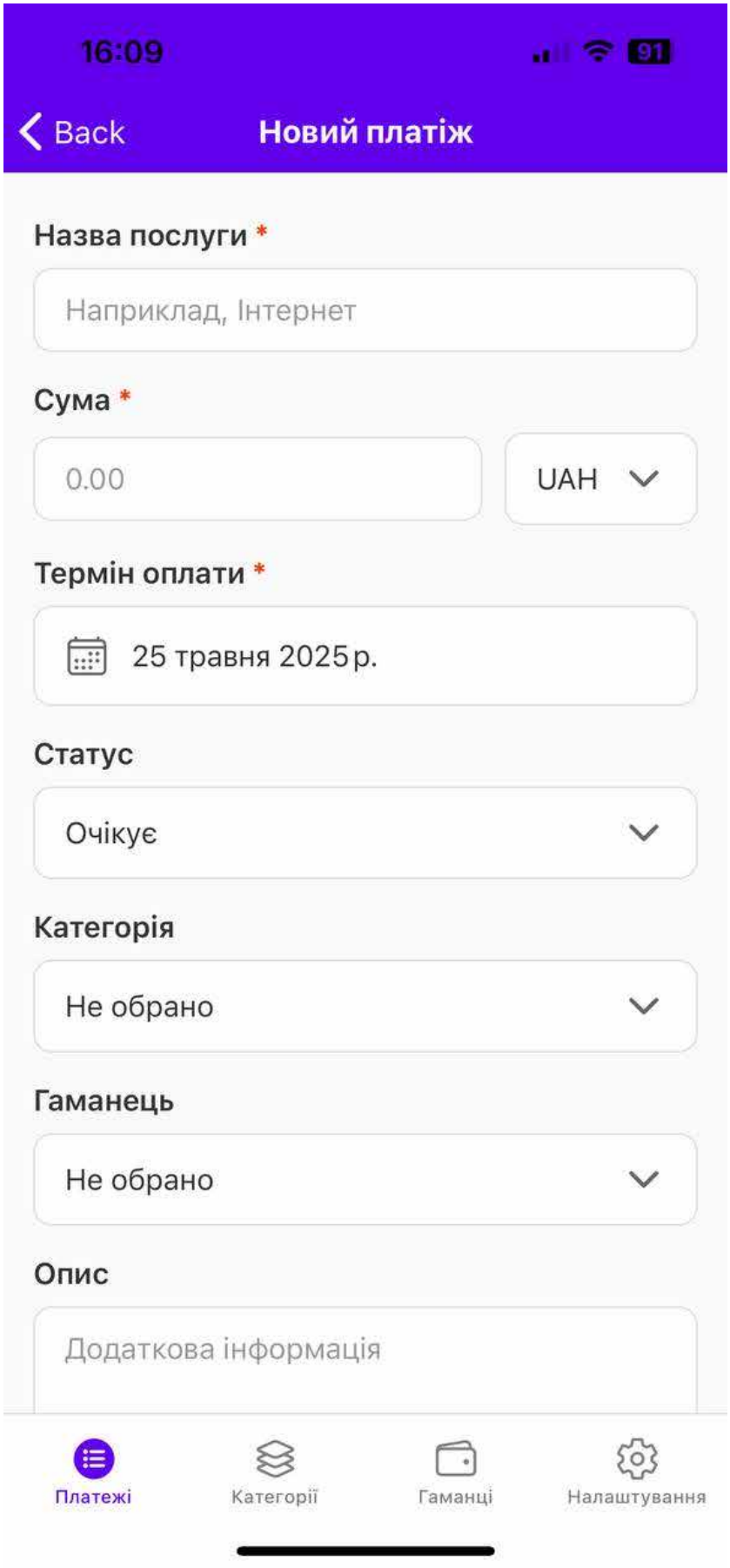
ДОДАТОК Д

Ілюстрації додатку

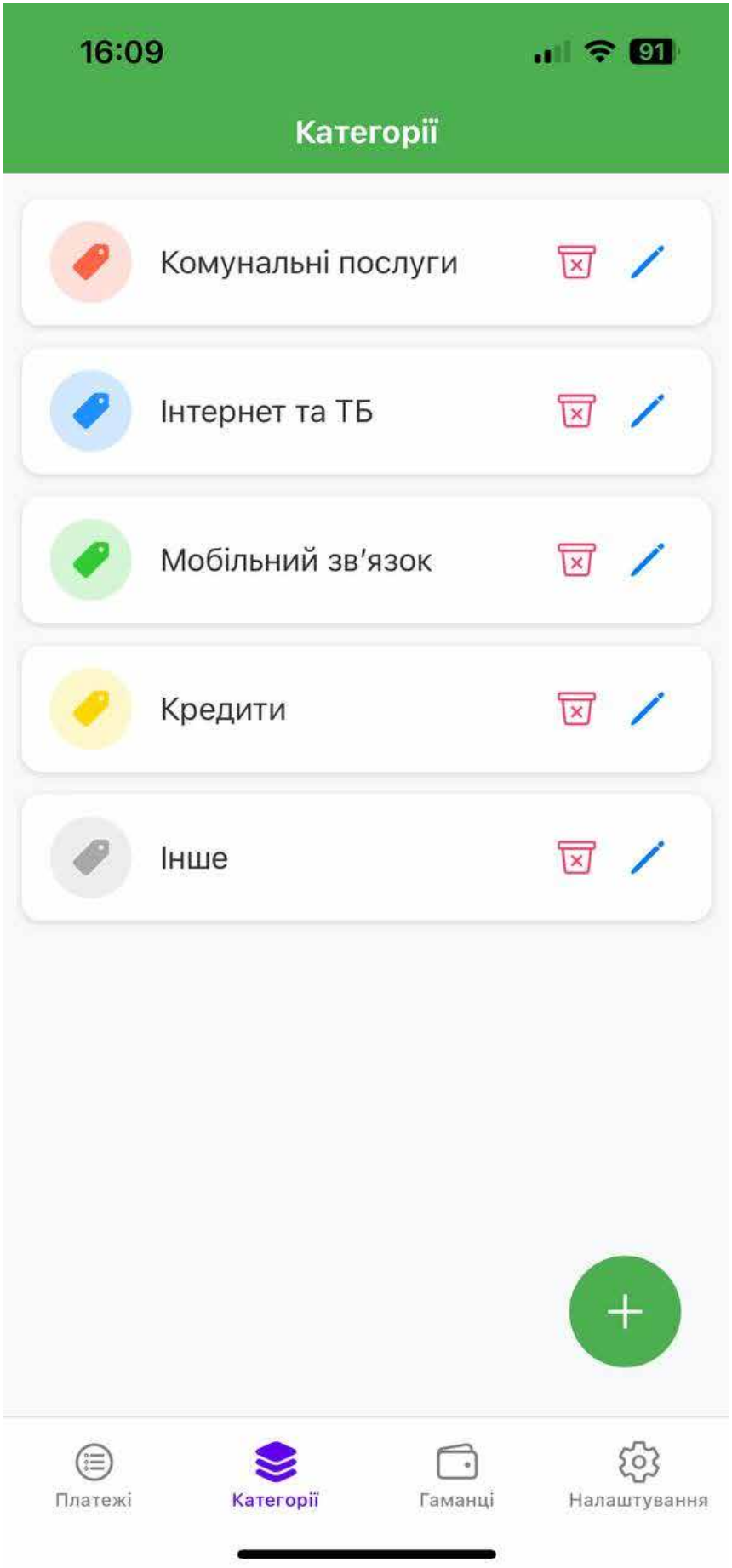
Екран «список платежів»



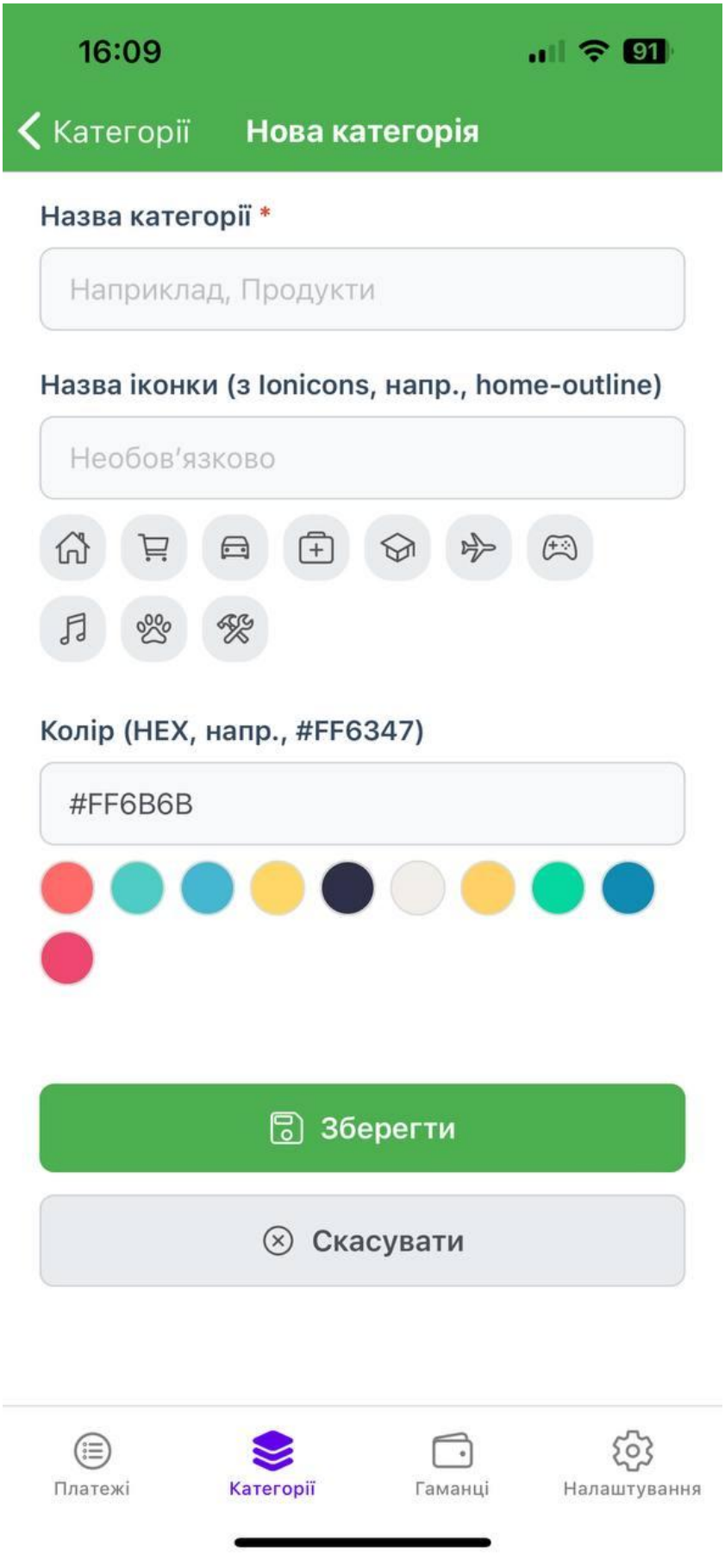
Екран «Новий платіж»



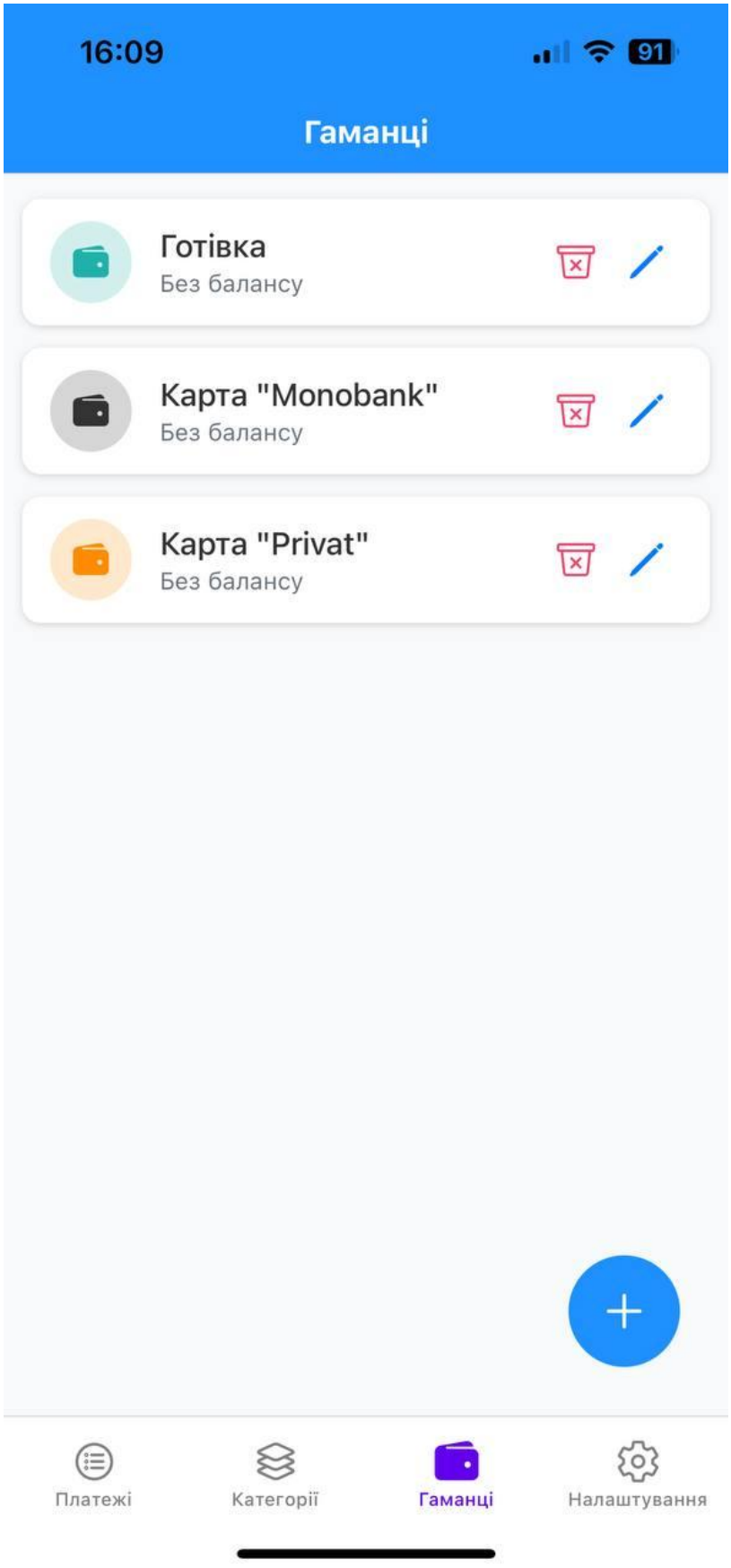
Екран «Категорії»


16:09
91
Категорії
Комунальні послуги
Інтернет та ТБ
Мобільний зв'язок
Кредити
Інше
Платежі
Категорії
Гаманці
Налаштування

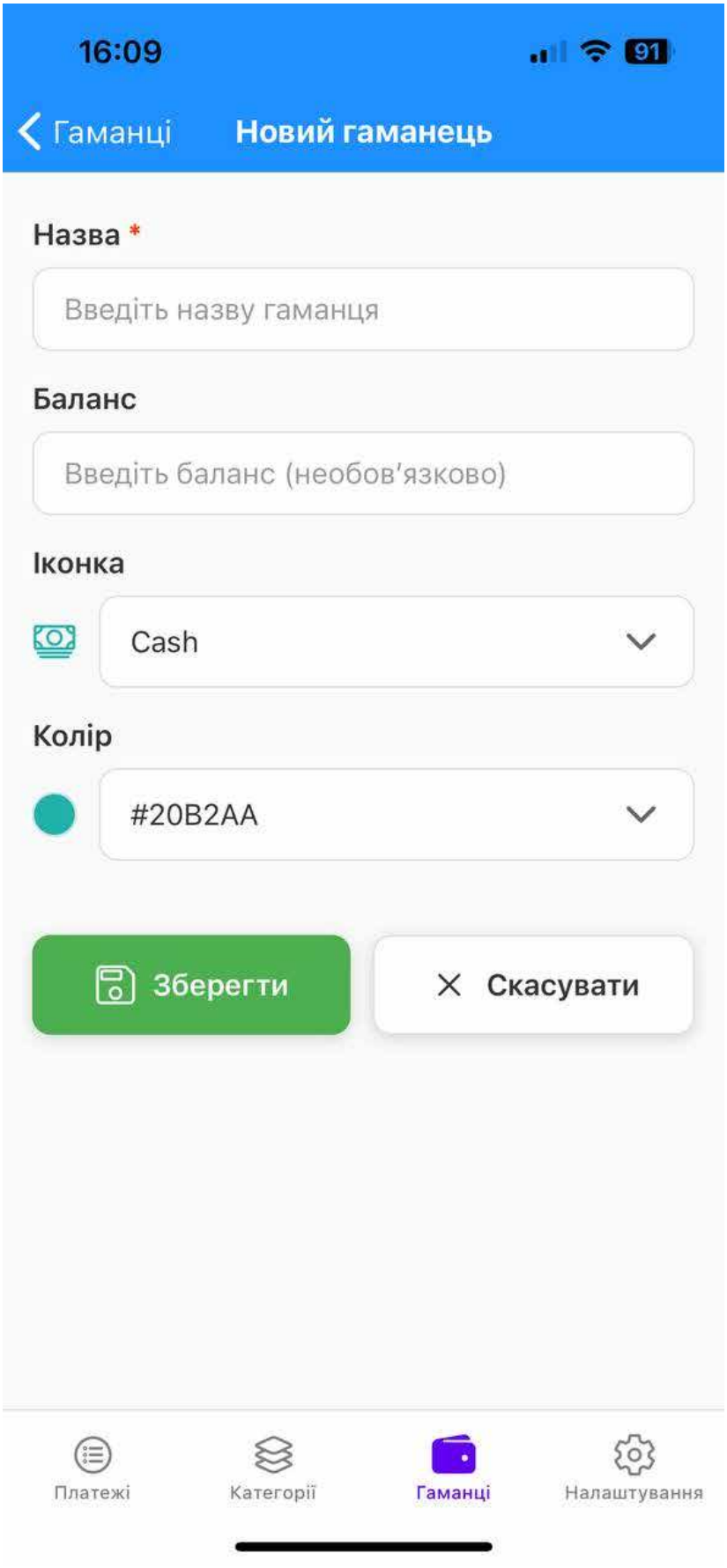
Екран «Нова категорія»



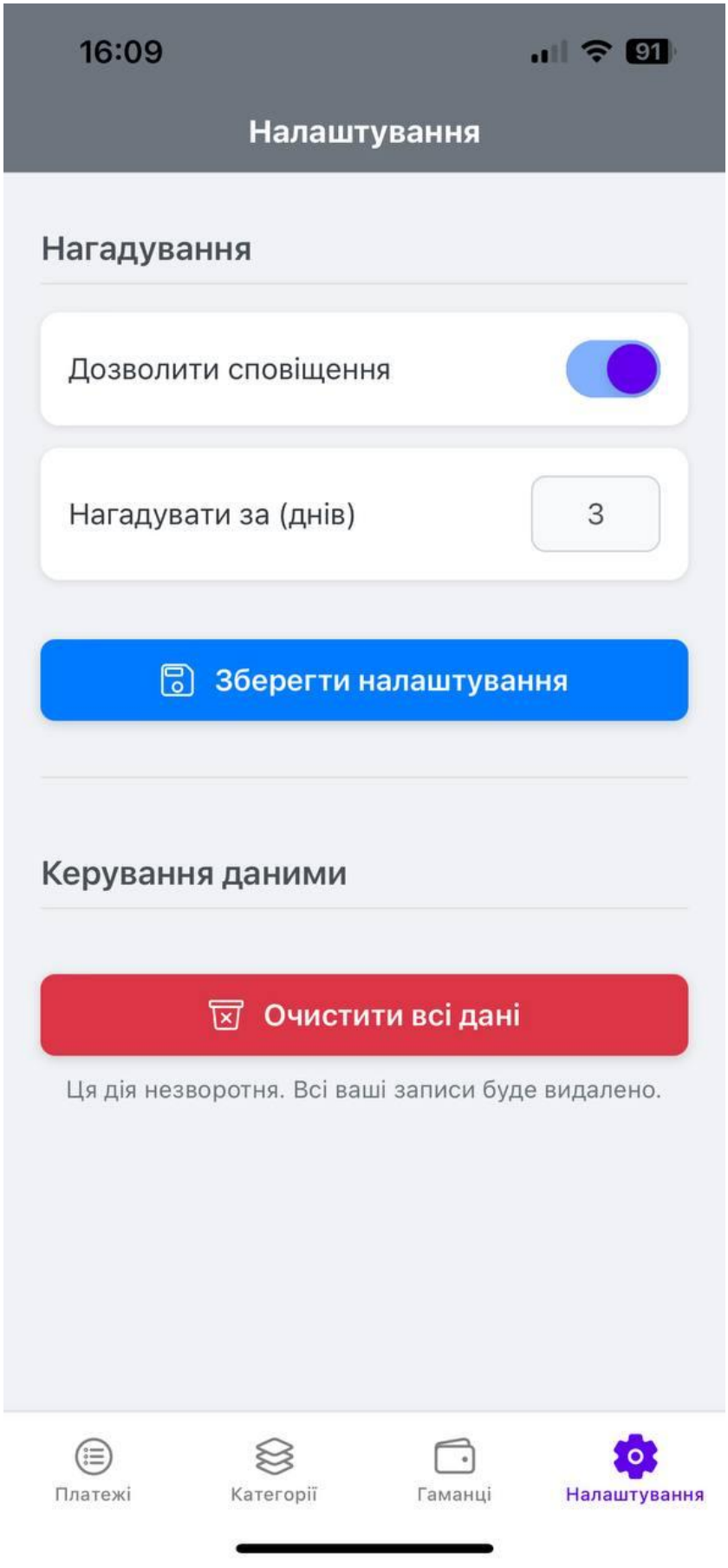
Екран «Гаманці»



Екран «Новий гаманець»


16:09
91
Гаманці
Новий гаманець
Назва *
Введіть назву гаманця
Баланс
Введіть баланс (необов'язково)
Іконка
Cash
Колір
#20B2AA
Зберегти
Скасувати
Платежі
Категорії
Гаманці
Налаштування

Екран «Налаштування»



Екран «Деталі платежу»