МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра комп'ютерної інженерії

КОЦУР Тарас Сергійович

Інформаційна система електронного обліку пасажирів на основі клієнт - серверної архітектури / Information system of electronic registration of passengers on the basis of client - server architecture

спеціальність: 123 – Комп'ютерна інженерія
освітньо-професійна програма – Комп'ютерна інженерія

Кваліфікаційна робота

Виконав: студент групи КІ-42
КОЦУР Тарас Сергійович
_____

Науковий керівник
к.т.н., Піцун О.Й..

Кваліфікаційну роботу
Допущено до захисту
«___» _____ 20 ___ р.

Завідувач кафедри
_____ О.М. Березький

ТЕРНОПІЛЬ - 2021

Навчально-науковий інститут новітніх освітніх технологій
Кафедра комп'ютерної інженерії
Освітній ступінь «бакалавр»
спеціальність: 123 – Комп'ютерна інженерія
освітньо-професійна програма – Комп'ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерної інженерії
д.т.н., проф. О.М. Березький
_____
"____"_____20__ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
Коцуру Тарасу Сергійовичу

1. Тема кваліфікаційної роботи «Клієнтоорієнтована система пасажиропотоку" / Client oriented system of passenger traffic»,

керівник роботи к.т.н., Піцун Олег Йосипович

затверджені наказом по університету від 10 грудня 2020 р. № 167

2. Строк подання студентом закінченої кваліфікаційної роботи 17 травня 2021р.

3. Вихідні дані до кваліфікаційної роботи: Технічне завдання

4. Основні питання, які потрібно розробити
- здійснити порівняльний аналіз існуючих систем;
- розробити додаток на мобільний пристрій;
- розробити серверну частину;
- спроектувати базу даних системи;
- розробити алгоритм оплати;
- розробити механізми захисту системи.

5. Перелік графічного матеріалу у роботі
- Алгоритм оплати проїзду. Схема структурна.
- Алгоритм захисту клієнтських даних. Схема структурна.

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Перевірка унікальності роботи	Мельник Г.М., к.т.н., доцент		
Н. контроль	Мельник Г.М., к.т.н., доцент		
Техніко-економічний розділ	Савка Н.Я, к.т.н., ст. викладач		

7. Дата видачі завдання 26 жовтня 2020 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Аналіз функціонування аналогічних систем	26.10.2020 – 31.12.2021	
2	Проектування системи	10.01.2021 – 15.02.2021	
3	Програмна реалізація мобільного додатку	16.02.2021 – 30.03.2021	
4	Програмна реалізація серверного модуля	1.04.2021 – 30.04.2021	
5	Техніко-економічний розділ	1.05.2021 – 19.05.2021	
6	Нормоконтроль, попередній захист	17.05.2019 – 1.06.2019	
7	Захист	4.06.2021	

Студент _____ Т.С. Коцур
(підпис)

Керівник роботи _____ к.т.н., ст. викладач Піцун О.Й.
(підпис)

РЕЗЮМЕ

Кваліфікаційна робота на тему «Клієнтоорієнтована система пасажиропотоку » зі спеціальності 123 «Комп’ютерна інженерія» освітнього ступеня «бакалавр» містить 50 сторінки пояснюючої записки, 9 рисунків, 8 таблиць, 3 додатки. Обсяг графічного матеріалу 2 аркуші формату А3.

Метою кваліфікаційної роботи є розробка програмного продукту, що зможе покращити досвід роботи з системою перевезень пасажирів в муніципальному транспорті.

Методи дослідження включають методи протоколо-орієнтованого програмування, елементи захисту даних.

У роботі проведено аналіз існуючих систем перевезень пасажирів в муніципального транспорту, що дозволило виділити їх переваги та недоліки.

Розроблено розширювану систему, що складаєтсья з двох програмних модулів. Реалізовано алгоритм безготівкової оплати проїзду в транспорті. Впроваджено систему хешування для захисту даних користувачів.

На мові Swift без використання сторонніх бібліотек розроблено клієнтський програмний модуль для комунікації з серверною частиною.

Використовуючи мову JavaScript реалізовано серверну частину з використанням бази даних MySQL для зберігання користувацьких даних.

Ключові слова: КЛІЄНТ-СЕРВЕР, ХЕШУВАННЯ, ТРАНЗАКЦІЇ, QR-КОД.

RESUME

The qualification work on the topic "customer-oriented passenger traffic system" in specialty 123 "Computer Engineering" with a bachelor's degree contains 50 pages of an explanatory note, 9 figures, 8 Tables, 3 appendices. The volume of graphic material is 2 sheets of A3 format.

The purpose of the qualification work is to develop a software product that can improve the experience of working with the passenger transportation system in municipal transport.

Research methods include protocol-oriented programming methods and data protection elements.

The paper analyzes the existing passenger transportation systems in municipal transport, which made it possible to identify their advantages and disadvantages.

An extensible system consisting of two software modules has been developed. Implemented an algorithm for non-cash payment of travel in transport. A hashing system has been implemented to protect user data.

A client software module for communicating with the server side has been developed in Swift without using third-party libraries.

Using the JavaScript language, the server side is implemented using a MySQL database to store user data.

Keywords: CLIENT-SERVER, HASHING, TRANSACTIONS, QR CODE.

ТЕХНІЧНЕ ЗАВДАННЯ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

1.1 Клієнтоорієнтована система пасажиропотоку.

1.2 Область застосування – Транспортна інфраструктура будь-якого міста.

2. ОСНОВА ДЛЯ РОЗРОБКИ

Завдання на кваліфікаційну роботу, затверджене кафедрою комп’ютерної інженерії факультету комп’ютерних інформаційних технологій Тернопільського національного економічного університету.

3. ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою кваліфікаційної роботи є розробка програмного продукту, що зможе покращити досвід роботи з системою перевезень пасажирів в муніципальному транспорті.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелами даної розробки є матеріали навчальної та реферативної наукової літератури, технічна документація, науково-дослідні роботи.

5. ТЕХНІЧНІ ВИМОГИ

5.1 Вимоги до апаратних засобів.

5.1.1 Вимоги до апаратних засобів

5.1.1.1 Клієнтський застосунок повинен працювати на платформі iOS.

5.1.1.2 Мінімальні вимоги для записку додатку: версія операційної системи iOS молодше 13, включно. Мінімальні вимоги до серверного програмого продукту: 4 ядерний процесор від 3 ГГц, оперативна пам'ять – від 4 ГБ.

5.2 Вимоги до програмної системи

5.2.1 Функціональні вимоги до програмної системи

5.2.1.1 Клієнтский модуль повинен відображати дані пов’язані з користувачем, а серверна частина — надавати за зберігати їх. В сукупності система повинна замінити роль кондукторів, переклавши відповідальність за оплату проїзду на пасажирів.

6. ВИМОГИ ТЕХНІКО-ЕКОНОМІЧНОГО РОЗДІЛУ

В розділі «Техніко-економічний розділ» кваліфікаційної роботи повинен бути даний економічний аналіз витрат на розробку комп’ютерної мережі.

7. ПОРЯДОК КОНТРОЛЮ

7.1 Представлення кваліфікаційної роботи на попередній захист.

7.2 Представлення кваліфікаційної роботи на захист.

Завдання прийняв до виконання _____ _____

(підпис) П.І.П. студента

Керівник кваліфікаційної роботи _____ _____

(підпис) П.І.П. керівника

ЗМІСТ

ВСТУП

В епоху інформатизації всього ми стали більше часу проводити перед екранами мобільних пристроїв. В нормальної сучасної людини не буває і дня без користування мобільним телефоном. Перевірка електронної пошти, соціальних мереж, спілкування через відео звязок, оплата товару за допомогою смарт-пристроїв — це, і не тільки це стало частиною нашого життя. І це не дивно користування телефоном значно спрощує життя людини. Теперішній час — це час, в якому всі процеси стараються автоматизувати до мінімуму, а там де вони не піддаються — робиться усе, щоб користувач залишився задоволений. Для цього зменшуються кількість кроків, які призводять до бажаного результату та й це лиш один з багатьох шляхів, що допомагають покращити користувацький досвід взаємодії.

Взявши за приклад банки і прослідкувавши за розвитком технологій можна чітко замітити, як інформатизація впливала на їх розвиток. Зараз робиться усе, щоб максимальна кількість користувачів банків залишалась задоволена. Раніше мобільні телефони могли дозволити собі лише заможні люди. Тепер же смартфони є у кожного, що і підштовхує на думку перенесення усього «в кишеню». Сучасні банки мають розроблені мобільні додатки для смартфонів, що значно покращують взаємодію з банком. Переказ коштів іншому користувачу здійснюється за пару кліків.

Говорячи про громадський транспорт варто зазначити, що і інформаційний прогрес теж зачепив цю область. Не так вже і давно в Києві розпочала працювати картки оплати в громадському транспорті — «Kyiv Smart Card». Що значно покращило становище жителів міста, адже не завжди в гаманці є паперові гроші. Аналогічна система знедавна запустилась і в Тернополі. «Файна карта», а саме так вона називається, дуже сильно сподобалась молоді, адже в гаманці вже можна не тримати дрібні куп’юри, а розплатитись електронним гаманцем. Оплата здійснюється за допомогою валідаторів — пристроїв, що вмонтовані в кожну одиницю громадського транспорту. Приклавши картку до валідатора —

пристрій обробляє інформацію щодо здійснення оплати і в результаті успішної операції повертає паперовий чек, що засвідчує про оплату проїзду. На чеку розміщено тип картки з якої здійснено оплата, вартість проїзду, дата і час оплати поїздки. Поповнення балансу на електронному гаманці можливе в крамницях з відповідною наліпкою: «Тут можна поповнити Файну Карту», а також через інтернет. Проте і в цьому, здається ідеальному проекті є недоліки.

1 АНАЛІЗ ІСНУЮЧИХ СИСТЕМ ОБЛІКУ ПАСАЖИРОПОТОКУ

1.1. Поняття пасажиропотоку та пасажирообігу

Пасажиропотік — рух пасажирів в рамках одного маршруту. Пасажиропотік вимірюється кількістю пасажирів за заданий період часу, що також називаються пасажирообігом.

Пасажиропотік характеризується навантаженістю пасажирообігу (не враховуючи вид транспорту).

Характерною особливістю пасажиропотоку є непостійність пасажирообігу. Це обумовлюється різними чинниками, що базуються на часі (година, доба, день тижня, пора року).

Пасажирообіг є основним покажчиком ефективності того чи іншого виду транспорту.

Метою заміру є отримання об’єктивних даних, щодо того, скільки пасажирів перевозить той чи інший транспорт. Для реалізації цього по маршруту протягом певного періоду часу фіксується кількість людей, що зайшли та вийшли на кожній зупинці. В результаті таких замірів можна дізнатись:

- Загальну кількість пасажирів, що користуються маршрутом, а також наскільки ця кількість варіюється протягом дня;
- Загальну завантаженість маршрут (щоб визначити перенавантажені або недовантажені маршруту);
- На яких зупинках заходить або виходить більше пасажирів, що допомагає визначити важливі пересадкові вузли;
- В яких місцях між зупинками люди просять зайти/вийти — це дозволяє визначити де саме варто додати офіційні зупинки;
- Швидкість кожного маршруту (загальну швидкість та швидкість проходження окремих ділянок, що може вказувати на проблемні місця маршруту).

1.2. Технології заміру пасажиропотоку

Замір пасажиропотоку здійснюється різними варіантами. Реалізація збору даних про перевезення пасажирів залежить від проектування системи, що буде виконувати збір даних.

Відомі способи підрахунку:

- Автоматичний підрахунок пасажирів за допомогою спеціальних датчиків, що встановлюються над кожною з дверей транспортного засобу. Датчики зчитують відстань до підлоги транспорту. Оскільки відстань від датчика до підлоги буде константою, то при випадках коли датчик визначить, що відстань є меншою — це буде підставою вважати, що пасажир вийшов чи зайшов в транспортний засіб. Таких датчиків встановлюють як мінімум 2 штуки, оскільки неможливо оприділити напрямок руху пасажира за допомогою 1 датчика відстані. Ці пристрої встановлюються під деяким кутом між собою, що дає змогу оприділити рух пасажира.
- Фіксування пасажира за допомогою спеціального предмету для зчитування. Під предметом зчитування можна розуміти предмет, який сумісний з системою фіксування пасажиропотоку. Для прикладу це може бути персоналізована картка з чіпом, QR-код який прив’язаний до певного рахунку користувача в системі чи будь-що інше що можливо зчитати або розпізнати.

QR-код (англ. quick response — швидкий відгук) — матричний код (двовимірний штрих-код). Основна перевага QR-коду — це легке розпізнавання сканувальним обладнанням (в тому числі й фотокамерою мобільного телефона), що дає можливість використання в торгівлі, на виробництві, в логістиці.

Варіанти підрахунку, що описані вище не в змозі скласти повну картину пасажирообігу, оскільки ці дані потрібно передати, зберегти та проаналізувати.

Передача та зберігання даних відбувається за допомогою серверних рішень. Ось приклад реалізації даного механізму: кожен транспортний засіб коли

розпочинає вихід на маршрут — ініціалізує свій засіб в системі, якому в результаті має видатись спеціальний токен, для передачі даних з системи транспорту (клієнта) на сервер. Спілкування з сервером зазвичай здійснюється за допомогою HTTP запитів — такий тип спілкування клієнта з сервером називають REST API або ж архітектурою «клієнт-сервер».

Архітектура клієнт-сервер є одним із архітектурних шаблонів програмного забезпечення та є домінуючою концепцією у створенні розподілених мережних застосунків і передбачає взаємодію та обмін даними між ними. Вона передбачає такі основні компоненти:

- набір серверів, які надають інформацію або інші послуги програмам, які звертаються до них;
- набір клієнтів, які використовують сервіси, що надаються серверами;
- мережа, яка забезпечує взаємодію між клієнтами та серверами.

Сервери є незалежними один від одного. Клієнти також функціонують паралельно і незалежно один від одного. Немає жорсткої прив'язки клієнтів до серверів. Більш ніж типовою є ситуація, коли один сервер одночасно обробляє запити від різних клієнтів; з іншого боку, клієнт може звертатися то до одного сервера, то до іншого. Клієнти мають знати про доступні сервери, але можуть не мати жодного уявлення про існування інших клієнтів.

HTTP — протокол передачі даних, що використовується в комп'ютерних мережах. Назва скорочена від HyperText Transfer Protocol, протокол передачі гіпертекстових документів.

REST (скор. англ. Representational State Transfer, «передача репрезентативного стану») — підхід до архітектури мережевих протоколів, які надають доступ до інформаційних ресурсів. Був описаний і популяризований 2000 року Роєм Філдінгом, одним із творців протоколу HTTP. В основі REST закладено принципи функціонування Всесвітньої павутини і, зокрема, можливості HTTP.

Дані повинні передаватися у вигляді невеликої кількості стандартних форматів (наприклад, XML, JSON). Будь-який REST протокол (HTTP в тому

числі) повинен підтримувати кешування, не повинен залежати від мережевого прошарку, не повинен зберігати інформації про стан між парами «запит-відповідь». Стверджується, що такий підхід забезпечує масштабовність системи і дозволяє їй еволюціонувати з новими вимогами.

XML (скор. англ. Extensible Markup Language) — запропонований консорціумом World Wide Web Consortium (W3C) стандарт побудови мов розмітки ієрархічно структурованих даних для обміну між різними застосунками, зокрема, через Інтернет.

JSON (англ. JavaScript Object Notation, укр. запис об'єктів JavaScript) — це текстовий формат обміну даними між комп'ютерами. JSON базується на тексті, може бути прочитаним людиною. Формат дає змогу описувати об'єкти та інші структури даних. Цей формат використовується переважно для передачі структурованої інформації через мережу (завдяки процесу, що називають серіалізацією).

Аналіз зібраних даних зазвичай здійснюється за допомогою спеціалізованих систем. Проте, варіант з реалізацією таких систем власноруч не можна не врахувати. Це надасть більшу гнучкість, чим використання другорядних сервісів оскільки система буде розроблятись персоналізовано.

1.3. Порівняльний аналіз систем обліку пасажиропотоку

Для порівняння було обрано 3 системи транспортування пасажирів в таких містах як Львів (Приват24), Тернопіль (Файна карта) та Київ (Київ Цифровий).

Спосіб оплати в цих трьох містах дещо відрізняються. Для початку Львів. Система стартувала на початку лютого в 2017 році. Державний банк ПриватБанк з мерією міста Львів розробили систему безготівкової оплати проїзду в трамваях та тролейбусах. Будь-який інший транспорт не передбачений в системі. Оплата здійснюється з-за допомогою QR-коду, що роздрукований на наклейці навпроти кожного входу в транспортний засіб. Далі користувачу необхідно увійти в банківський додаток «Приват24», обрати відповідне меню та просканувати QR-

код, після, обрати тип оплати. Серед типів оплати доступні «Звичайний» «Студентський» або «Пересадковий». Білет компостується прямо в додатку.

Таблиця 1.1 — Тарифи на проїзд в Львівській системі

Тип оплати	Вартість
Звичайний	6 гривень
Студентський	3 гривні
Пересадковий	8 гривень

Поповнення рахунку здійснюється напрямку на банківську картку користувача, оскільки списання коштів відбувається з неї.

Із явних мінусів такої системи можна виділити наступні пункти:

- Немає повного спектру послуг. Оплата проїзду здійснюється тільки в трамваях;
- Банківська сфера поєднується з транспортною. Великий мінус, що банківський додаток виконує дві і більше ролі. Порушується цілісність продукту, так званим терміном Single Responsibility (укр. Принцип єдиної відповідальності).

Перегляд балансу та деталі оплати за проїзд доступні зразу ж в додатку Приват24, що неодмінно є плюсом. Оплата здійснюється за декілька секунд і не створює незручностей.

Файна карта у місті Тернопіль. В жовтні 2017 року стартував проект соціальних карток. Система є дещо складнішою ніж у Львові. Для сплати проїзду користувачу потрібно придбати персональну картку, з якої буде відбуватись списання коштів. Всього є 4 типи карток: «Пільгова», «Загальна», «Учнівська», «Звичайна». Картка «Звичайна» не є персональною, тому може передаватись іншим людям для користування. Усі інші картки є містять дані власника, які користувач вказує при реєстрації.

Для реєстрації необхідно відкрити веб-сайт системи, та обрати пункт «придбати карту», де після заповнення всіх даних користувача попросять почекати до робочих 20 днів. Після тривалого очікування карту необхідно забрати в пункті видачі.

На веб-ресурсі «Файної карти» також доступна функція перегляду балансу, де потрібно ввести порядковий номер карти.

Для поповнення балансу карти існують спеціальні місця поповнення, які дають змогу зарахувати готівку на рахунок картки.

В кожному транспортному засобі який під'єднаний до системи «Файна карта» розміщені валідатори, що і здійснюють оплату.

Валідатор (в контексті «Файної карти») — це пристрій, що зчитує ідентифікатор картки, яку приклали до валідатора. Отримує дені про рахунок та вирішує чи можливо сплатити проїзд. Результатом роботи валідатора є чек, на якому вказано дані про сплату проїзду.

Окрім карток валідатор підтримує оплату за допомогою банківських карток, що безумовно є великим плюсом.

Таблиця 1.2 — Тарифи на проїзд в Тернопільскій системі

Тип оплати	Автобус	Тролейбус
Пільговий	Безкоштовно	Безкоштовно
Студентський	6 гривень	Безкоштовно
Загальний	6 гривень	5 гривень
Звичайний	6 гривень	5 гривень

Із очевидних плюсів можна виділити підтримку усього муніципального транспорту в системі «Файна картка» та зручну сплату проїзду.

Також було виявлено наступні недоліки:

- Немає зручного способу перевірити баланс та історію транзакцій;
- Функція перевірки балансу працює не належним способом. Списання грошей в системі спрацьовує не синхронно, і може займати від години до дня.
- Оффлайн — єдиний метод поповнення картки. Немає змоги перерахувати кошти на картку швидко, зручно та онлайн;
- Довгий період очікування при реєстрації.

Взявши до уваги систему міста Київ «Київ Цифровий» — було виявлено, що запуск відбувся одразу після закриття проєкту «Kyiv Smart City». Останній — був підконтрольний громадській організації «Смарт Сіті Хаб», а не місту. В наслідок цього Київ відмовився від пролонгації контракту після 31 грудня 2020 року. В кінцевому результаті місто запустило власну систему «Київ Цифровий», що мала схожості в інтерфейсі користувача з попереднім проектом.

В якості об’єкту дослідження взято новий проект «Київ Цифровий».

Для реєстрації в системі користувачу необхідно завантажити одно іменний додаток в якому необхідно зареєструватись ввівши лише номер телефону. Далі користувачу необхідно ввести код підтвердження який був надісланий СМС-повідомленням для підтвердження особи.

Після успішного входу користувачу буде доступний інтерфейс що дозволить здійснювати купівлю білетів, оплату паркінгу, дасть змогу написати в службу підтримки і також буде доступна історія всіх операцій, що здійснювались.

Білет котрий був придбаний в додатку дає змогу їздити в усіх видах муніципального транспорту міста Київ, а саме в автобусах, трамваях, тролейбусах, метро та фунікулері, а також в міських електричках.

Ціна білету є єдиною для усіх видів транспорту, що дозволяє зручно курувати своїми коштами та опиратись на вибір зручного транспорту, а не дешевого.

В якості білету на проїзд є QR-код, що зчитуєтсья при вході в транспорт. Такий QR-квиток можна придбати за допомогою банківської карти, яку можна приєднати до облікового запису для купівлі білетів в один клік.

Варто зазначити, що купівля квитків на довгий період часу є неможливою, адже квиток діє лише 15 днів з дня покупки.

Окрім додатку в системі існує можливість замовити картку для оплати проїзду з її допомогою. Поповнення картки відбувається за допомогою оффлайн точок поповнення та онлайн в різних інтернет-терміналах.

В залишку варто зазначити, що цифровий додаток є кросплатформним додатком на базі React Native, що дає змогу розробляти програму використовуючи лише один інструмент, і може бути експортованим на дві різні мобільні платформи Android та iOS.

Багатоплатформність (кросплатформність) — властивість програмного забезпечення працювати більш ніж на одній апаратній платформі. Кросплатформність дозволяє суттєво скоротити витрати на розробку нового або адаптацію існуючого програмного забезпечення. Проте може нести за собою обмеження в використанні специфічних функцій деяких платформ.

React Native — це фреймворк мобільних додатків з відкритим кодом, створений Facebook, Inc. Він використовується для розробки програм для Android, Android TV, iOS, macOS, tvOS, Web, Windows та UWP, дозволяючи розробникам використовувати React фреймворк разом із можливостями власної платформи.

Фреймворк (англ. Framework, каркас, платформа, структура, інфраструктура) — інфраструктура програмних рішень, що полегшує розробку складних систем. Спрощено дану інфраструктуру можна вважати своєрідною комплексною бібліотекою, але при цьому вона має ряд обмежень, що задають правила створення структури проєкту та написання коду.

Android — це мобільна операційна система, заснована на модифікованій версії ядра Linux та іншого програмного забезпечення з відкритим кодом, призначена в основному для сенсорних мобільних пристроїв, таких як смартфони та планшети.

iOS — мобільна операційна система, створена та розроблена компанією Apple Inc. виключно для свого обладнання. Саме операційна система працює на багатьох мобільних пристроях компанії. Це друга за популярністю мобільна операційна система у світі після Android. Це запатентоване програмне забезпечення, хоча деякі його частини є відкритими за ліцензією Apple Public Source License та іншими ліцензіями.

Повертаючись до характеристики продукту можна зазначити, що відсутність покупки квитків на довгий строк є непорозумінням і значним мінусом з сторони системи «Київ Цифровий».

Зібравши відгуки про додаток можна зрозуміти, що система є не досконалою і часто створює незручності при поповненні рахунків, додавання карток, тощо.

Таблиця 1.3 — Порівняльна характеристика аналогічних систем

Система	Приват Банк	Файна Картка	Київ Цифровий
Місто	Львів	Тернопіль	Київ
Спосіб оплати	QR-код	Картка з чіпом	QR-код
Мобільний додаток	+	-	+
Підтримка транспорту	Трамваї	Повна	Повна
Збереження коштів	+	+	-
Перегляд коштів	+	-	+
Перегляд історії операцій	+	-	+

1.4. Висновки

В першому розділі було проаналізовано технічні рішення для фіксування пасажирообігу. Обрано ряд систем, які виконують облік пасажиропотоку та надають послуги для зручного користування транспорту.

Аналіз предметної області передбачав розбір інформаційних систем для обліку пасажирообігу на окремі складові, що мали слугувати підґрунтям для створення продукту. Обравши три найбільш популярні системи для транспортування пасажирів в містах України, таких як Львів, Тернопіль та Київ — було знайдено ряд недоліків, що викликають дискомфорт для користувачів та жителів цих міст.

При проектуванні системи будуть враховані недоліки існуючих рішень, що дасть змогу потенційним користувачам почуття задоволення.

Метою роботи є створення продукту з використанням Web-технологій, а саме сервера з доступом до даних через API.

Для досягнення поставленої мети потрібно виконати наступні задачі:

- виявити недоліки аналогічних рішень та уникнути їх в реалізації;
- створити зручний користувацький додаток;
- розробити розширювану серверну частину;
- забезпечити захист системи.

2 АЛГОРИТМ ОБЛІКУ ПАСАЖИРОПОТОКУ

2.1 Обладнання та технології

На основі систем, що були проаналізовані в минулому розділі було прийнято рішення про створення власної системи з використанням клієнт-серверної архітектури або API. Дуже важливо ясно уявляти, хто або що розглядається як «клієнт». Загальноприйнятим є положення, що клієнти та сервери — це перш за все програмні модулі. В даному випадку програмну складову сервера буде побудовано на технології Node.js.

Node.js — платформа з відкритим кодом для виконання високопродуктивних мережевих застосунків, написана мовою JavaScript.

JavaScript (JS) — динамічна, об’єктно-орієнтована прототипна мова програмування. Найчастіше використовується для створення сценаріїв вебсторінок, що надає можливість на боці клієнта (пристрої кінцевого користувача) взаємодіяти з користувачем, керувати браузером, асинхронно обмінюватися даними з сервером, змінювати структуру та зовнішній вигляд веб-сторінки.

Node.js надасть можливість використовувати JavaScript-скрипти на серверах та надсилати відповідь на запити з мережі. Платформа Node.js перетворила JavaScript на мову загального використання із великою спільною розробниками.

Як було вище сказано Node.js надасть змогу запускати серверний застосунок, окрім цього буде використано розширення Express.js.

Express.js, або просто Express — це внутрішня веб-програма для Node.js, випущена як безкоштовне програмне забезпечення з відкритим кодом. Він призначений для створення веб-додатків та API. Раніше його називали стандартною серверною структурою для Node.js.

Аби сервер міг оперувати даними йому необхідний доступ до бази даних. В якості сховища даних буде використовуватись MySQL. Це система управління

реляційними базами даних із відкритим кодом (СУБД). Реляційна база даних організовує дані в одну або кілька таблиць даних, в яких типи даних можуть бути пов’язані між собою; ці зв’язки допомагають структурувати дані. SQL — це мова, яку програмісти використовують для створення, модифікації та вилучення даних з реляційної бази даних.

Маючи досвід роботи з SQL запитами до бази даних MySQL було обрано фреймворк Knex.js для скорочення запитів до бази даних; створення моделей для таблиць та міграцій бази даних.

Knex.js — це конструктор запитів SQL для Postgres, MSSQL, MySQL, MariaDB, SQLite3, Oracle та Amazon Redshift, розроблений для гнучкості, портативності та простоти у використанні. Він має інтерфейс для більш чистого управління асинхронним потоком, повнофункціональні конструктори запитів та моделей, підтримку транзакцій (з точками збереження).

Варто підмітити, що лише авторизовані юзери зможу здійснювати оплату за перевезення. Це являється базовим функціоналом, щоб уникнути різних незручностей при захисті прав користувачів.

Авторизація буде здійснюватись сервером, за це буде відповідати бібліотека Passport.js, що якраз призначається для захисту контенту.

Кожен день мільйони користувачів розміщають конфіденційну інформацію в Інтернеті, і забезпечення безпеки цієї інформації є однією з найбільших проблем для розробників. На щастя, це не нова проблема, і в розпорядженні спеціалістів є кілька інструментів для забезпечення безпеки даних користувачів. Одним з таких інструментів, який особливо практичний і простий в реалізації, є веб-токен JSON.

Також буде реалізовано облікові записи користувачів за допомогою хешування паролів. Детальніше про це буде описано нижче.

Загалом, цього буде достатньо щоб розгорнути простий веб-сервер. Далі для роботи з ним необхідний клієнт або клієнти, що будуть звертатись до даного веб-ресурсу. Перейнявши досвід існуючих систем — було вирішено, що

мобільний додаток буде єдиним клієнтом для керування обліковим записом користувача.

Однією з найпопулярніших мобільних операційних систем станом на 2021 рік — це iOS. Тому застосунок буде виконуватись саме на цій платформі.

Для побудови застосунку необхідне програмне середовище Xcode та мова програмування Swift.

Swift — це потужна та інтуїтивно зрозуміла мова програмування для macOS, iOS, watchOS, tvOS та інших. Написання коду Swift є інтерактивним та цікавим, синтаксис стислий, але виразний. Це результат останніх досліджень мов програмування у поєднанні з багаторічним досвідом створення платформ Apple. Керування пам'яттю здійснюється автоматично за допомогою жорсткого, детермінованого підрахунку посилань, це зводить використання пам’яті до мінімуму без накладних витрат на збір залишків даних.

Аби додаток (клієнт) міг спілкуватись з сервером — йому потрібно надати таких можливостей. Зв’язок між клієнтом і сервером буде відбуватись за допомогою інтернету через API який буде розгорнуто сервером.

API (англ. Application Programming Interface, API) — це набір методів для взаємодії різних компонентів. API надає розробнику засоби для швидкої розробки програмного забезпечення. API може бути для веб-базованих систем, операційних систем, баз даних, апаратного забезпечення, програмних бібліотек.

Як стало відомо застосунок буде лише запитувати інформацію в сервера та відображати користувачу. Аби додаток міг не тільки відображати інформацію з сховища буде реалізовано функціонал розпізнавання QR-кодів, які будуть ідентифікуватись додатком та розділити різні сценарії при користуванні застосунком.

В сухому залишку додаток буде виконувати роль помічника. Відсканувавши QR-код — користувачу буде доступне меню оплати за проїзд.

2.2 Алгоритм підрахунку пасажиропотоку

Підрахування пасажиропотоку здійснюється для аналізу маршрутів, направлень та іншого. Серед основних параметрів для оцінки міського руху існує відповідність між зонами їх руху. Вони змінюються залежно від часу доби, дня тижня та пори року. Транспортні перевезення є змінними, оскільки на них впливає багато факторів. З метою вдосконалення міського пасажирського транспорту необхідно попередньо зрозуміти ступінь використання рухомого складу та задоволеності пасажирів.

Цю інформацію можна отримати на основі опитувань пасажирів. Такі дослідження можуть бути безперервними та вибірковими. Перший передбачає набір організаційних та технічних заходів для виявлення всіх типів пасажиропотоку у місті. Вибірково досліджуйте пасажирський потік лише одного типу, одного напрямку або, як правило, на одному маршруті.

Такі опитування можна здійснювати в додатку системи, проте, це залишається не бажаним варіантом. Відомо, що цілі сервісів — покращувати та робити життя людей простішим. Даний тип дослідження слід використовувати в крайніх випадках. Першочергово потрібно “холодним” методом аналізувати популярність маршрутів або напрямків за допомогою вбудованих в додаток інструментів, без приваблювання користувача прийняти участь в опитуванні.

Вираховування пасажиропотоку здійснюється за допомогою кількості оплачених білетів. Взявши об’єм проданих білетів за окремими маршрутом можна оприділити пасажипопотік даного маршруту. Аби це працювало — необхідно визначити механізм купівлі білетів в міський транспорт.

При запуску додатку користувач зможе просканувати QR-код, який призначений для ідентифікації транспортого засобу та сформувати запит до сервера. Сервер сформує запит в базу даних для створення нового запису. Після успішної обробки сервер дасть відповідь по типу «Операція успішна», яку необхідно перехопити та обробити вже на стороні клієнта (мобільного додатку). При невдалому сценарії сервер відправить відповідь в стилі «Сталась помилка,

повторіть пізніше» і в той час сповістить розробників про неполадку, якщо це збій. Таку відповідь також необхідно перехопити та відобразити користувачу.

Для наглядності створено схему за номером 3.1. Вона відображає поведінку системи при спробі оплатити квиток.

Рисунок 2.1 — Блок-схема поведінки при покупці квитка

Після того як користувач успішно оплатить проїзд в транспорті — в базі даних буде створено новий запис про купівлю білету разом з часом його купівлі. Це дасть змогу оприділити пасажиропотік для конкретного маршруту.

2.3 Структура бази даних

Для сервера було обрано базу даних MySQL. Це реляційна база даних, сутності якої подаються в вигляді таблиць, а властивості в вигляді стовпців. Кожен стовпець таблиці бази даних повинен мати ім'я та тип даних.

Розробник SQL повинен вирішити, який тип даних буде зберігатися всередині кожного стовпця під час створення таблиці. Тип даних — це вказівка для SQL, щоб оприділити, який тип даних очікується всередині кожного стовбця. Кожна властивість має мати визначений тип даних.

Програмування в MySQL, однак, не дуже відрізняється від функцій програмування на інших мовах. Єдина фундаментальна відмінність полягає в тому, що замість того, щоб приймати надані користувачем вхідні дані, MySQL має справу виключно з вмістом бази даних.

Основним об’єктом в даній системі є користувач. Адже саме для нього розробляється продукт.

Таблиця 2.1 — Таблиця властивостей сутності Users

Властивість	Тип даних	Опис
id	Integer	Ідентифікатор користувача
picturePath	String	Шлях до картинки користувача
isActivated	Boolean	Значення чи користувач активований
name	String	Ім’я користувача
email	String	Пошта користувача
password	String	Пароль користувача
cardId	Integer	Ідентифікатор картки користувача
createdAt	Date	Дата створення запису
updatedAt	Date	Дата редагування запису

Для покращення користувацього досвіду з додатком який буде розроблятись — окрім стандартних полів для користувача було додано поле

picturePath, що визначає шлях до аватара. Такий шлях є умовним, оскільки зберігання картинки здійснюєтсья на сервері і при запиті на її отримання — серверу потрібно буде пройти за допомгою цього параметру по внутрішнім картотекам та знайти відповідний файл та відправити його в відповідь.

На рахунок поля password є декілька поміток. Зберігання паролю в базі даних є небезпечним. Тому було вирішено використати функції хешування, щоб уникнути пограбування облікових записів. Для хешування буде використовуватись хеш-функція SHA256. SHA-256 розшифровується як Secure Hash Algorithm 256-bit і використовується для криптографічної безпеки.

Алгоритми хешування — є односторонніми функціями. Вони перетворюють будь-яку кількість даних фіксованої довжини, який неможливо змінити. Вони також мають властивість, що якщо введення змінюється хоча б на крихітний біт, отриманий хеш зовсім інший. Це чудово підходить для захисту паролів, оскільки потрібно зберігати паролі у формі, яка захищає їх, навіть якщо сам пароль скомпрометований, але в той же час потрібно мати можливість перевірити правильність пароля користувача.

Загальний робочий процес реєстрації та автентифікації облікового запису в системі облікових записів на основі хешу такий:

1. Користувач створює обліковий запис.
2. Його пароль хешується і зберігається в базі даних.
3. Коли користувач намагається увійти, хеш пароля, який він ввів, перевіряється на відповідність хешу його реального пароля (отримується з бази даних).
4. Якщо хеші збігаються, користувачеві надається доступ. Якщо ні, користувачеві повідомляють, що він ввів недійсні облікові дані для входу.
5. Крок 3 і 4 повторюйте кожного разу, коли хтось намагається увійти до свого облікового запису.

Стосовно хешованих паролів — є спосіб обійти таку автентифікацію користуача. Оскільки хешування є незворотнім процесом — хакери викрадають

хешовані паролі та встановлюють злочинні програми для моніторингу даних на сервери, звідки паролі і були вкрадені. Таким чином вони встановлюють зв’явок між хешованим паролем та оригінальним. І при наступній атаці на іншу організацію чи компанію — вони будуть мати більший вплив на систему. Так як викравши хешовані паролі вони можуть співставити результати хешування і оприділити початковий пароль, який ввів користувач. Для уникнення таких ситуацій було придумане хешування з сіллю.

Хешування з сіллю — генерація випадкових байтів (сіль) та поєднання їх із паролем перед хешуванням створює унікальні хеші для кожного пароля користувача. Якщо два користувачі мають однаковий пароль, вони не матимуть однаковий хеш пароля. Це робиться для запобігання атакам райдужної таблиці, які можуть змінити хешовані паролі, використовуючи загальні функції хешування, які не використовують сіль.

Продовжуючи тему користувача. Як було вказано раніше — в сутності користувача є поле під назвою isActivated. Дане поле відповідає за стан користувача. Даний стан є змінний і за замовчуванням “false”, що означає обліковий запис не активований. При реєстрації користувачу необхідно буде пройти верифікацію за допомогою надісланого коду на його електронну пошту. В хорошому випадку значення поля isActivated зміниться на “true”, після чого він зможе увійти в систему.

Щодо активації — перед надсиланням активаційного листа створюється хеш, який присвоюється певній події. Що б це не було, реєстрація, переказ коштів, будь-що. Це допомагає запобігти створення роботів, що стараютьсья поламати систему — так званих фродів. В подальшому планується розширення функціоналу системи з використанням промо-кодів, тощо. Тому, цей механізм, так званий антифрод, буде не зайвим.

Зазвичай антифрод (від англійського anti-fraud) асоціюється тільки з банківським сектором, коли фінансові транзакції оцінюються на предмет шахрайства, наприклад, коли платіжна карта використовується зловмисником, а не її власником. Однак, антифрод потрібен не тільки кредитним установам та

інтернет-магазинам. Цікавий приклад з великою мережею автозаправок, де була запущена вигідна програма лояльності з частковим поверненням коштів за покупки, т. зв. кешбек. На одній з бензоколонок оператор проводив всі платежі через свою особисту карту з таким кешбек, приймаючи у клієнтів готівку і проганяючи їх через свій рахунок. Хоча цей випадок номінально і не порушує законів, проте, за фактом він є інсайдерською поведінкою персоналу в своїх інтересах. Обманна схема була розкрита вручну: проаналізувавши всі операції за день, виявилося, що на одну і ту ж банківську карту було куплено майже цистерна бензину. Антифрод - система ідентифікувала б це і заморозила подібні транзакції автоматично.

2.4Висновки

В третьому розіділі було описано обрано технології та положення, що будуть використовуватись для побудови системи пасажирообігу. На пару було охарактеризовано та ретельно описано основні алгоритми роботи додатку та його взаємодія з системою.

Також було описано основні елементи системи, що стосуються бази даних та підкреслено ключові засоби захисту при проектування структури бази даних.

3 ПРОГРАМНО-АПАРАТНА РЕАЛІЗАЦІЯ

3.1 Програмна реалізація користувацького інтерфейсу

Розробка зручного користувацького інтерейсу, як правило, займає багато часу, оскільки враховується весь набутий досвід за роки проектування мобільних застосунків. Для правильного проектування компанія Apple розробила ряд правил, які повинні виконуватись кожним розробником, назва правил — Human Interface Guidelines (скорочено HIG) (укр. Правила Інтерфейсу для Людей).

Корпорація Apple виділила 3 принципи, які виділяють розробку програмних продуктів для iOS від інших платформ:

- Виразність. У всій системі текст розбірливий в будь-якому розмірі, значки точні і зрозумілі, прикраси тонкі і доречні, а загострений акцент на функціональності мотивує дизайн. Негативний простір, колір, шрифти, графіка і елементи інтерфейсу тонко виділяють важливий контент і передають інтерактивність.
- Повага. Плавний рух і чіткий, красивий інтерфейс допомагають людям розуміти і взаємодіяти з контентом, ніколи не конкуруючи з ним. Вміст зазвичай заповнює весь екран, в той час як прозорість і розмитість часто натякають на більше. Мінімальне використання обрамлень, градієнтів і тіней зберігає інтерфейс легким і повітряним, забезпечуючи при цьому першорядне значення контенту.
- Глибина. Чіткі візуальні шари і реалістичний рух передають ієрархію, надають життєву силу і полегшують розуміння. Дотик і відкритість підсилюють захоплення і забезпечують доступ до функціональності і додаткового контенту без втрати контексту. Переходи забезпечують відчуття глибини при навігації по контенту.

Звучить досить абстрактно, проте до кожного з пунктів знаходиться пояснення на сторінці Apple. Для прикладу — вміст повинен бути центрований і симетрично вставлений, щоб відмінно виглядати в будь-якій орієнтації. Для досягнення найкращих результатів слід використовувати стандартні системні елементи інтерфейсу і автоматичне компонування для створення інтерфейсу і дотримуватись посібників з компонування в безпечій області (англ. Safe Area). Коли пристрій знаходиться в альбомній орієнтації, для деяких додатків, таких як Ігри, може бути доцільно розмістити елементи управління з можливістю натискання в нижній частині екрана (нижче безпечної області), щоб звільнити більше місця для контенту. Потрібно використовувати відповідні місця для елементів управління у верхній і нижній частині екрана. Слід залишати достатньо місця навколо індикатора “додому” (англ. Home), щоб люди випадково не націлилися на нього при спробі взаємодіяти з елементом управління. Оскільки Індикатор “додому” залишається центрованим на екрані, його розташування щодо інтерфейсу вашої програми може змінитися.

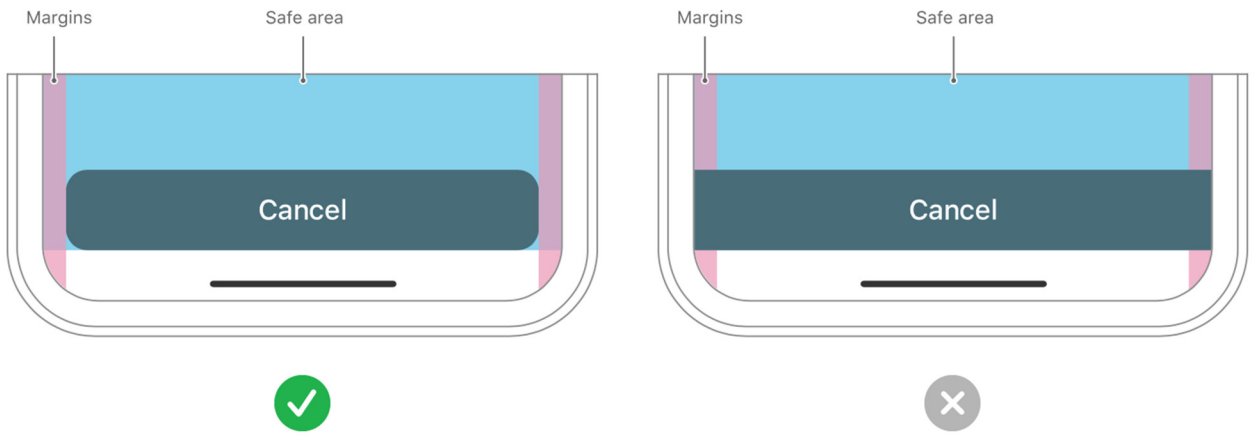


Рисунок 3.1 — Одне з правил по проектування інтерфейсу в iOS

За основу було взято програму Apple Wallet. Інтерфейс додатку представлено на рисунку 3.2.

Apple Wallet — це додаток для iPhone та Apple Watch, який надійно та зручно організовує кредитні та дебетові картки, транзитні пропуски, посадкові

талони, квитки, студентський квиток, ключі від автомобіля, заохочувальні картки та багато іншого. Все в одному місці.

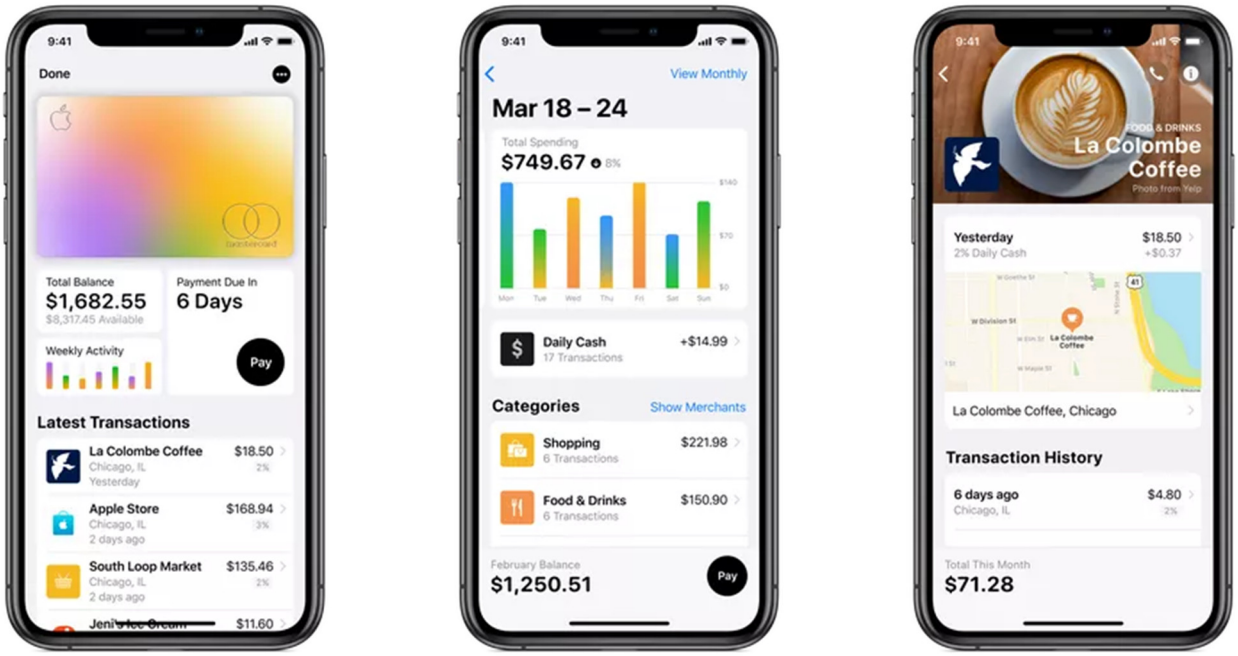


Рисунок 3.2 — Програмний додаток Apple Wallet

Для розробки програмного продукту на платформі iOS необхідне відповідне програмне середовище та мову програмування. Xcode — це інтегроване середовище розробки Apple для macOS, що використовується для розробки програмного забезпечення для macOS, iOS, iPadOS, watchOS і tvOS.

Саме Xcode буде використаний в якості середовища для розробки проекту, та Swift — в якості мови програмування.

Розробляти проект слід з вибору архітектури. Для простого вдосконалення в подальшому було обрано архітектуру додатку — Service Oriented Architecture (скорочено SOA), що походить від Clean Architecture (англ. Чиста Архітектура).

Чиста Архітектура розбиває додаток на шари, кожен шар має певну роль. Цей підхід додає більше складності на початку, але він окупається в міру зростання кодової бази. Це полегшує додавання / зміну функцій і усунення помилок. І це підвищує тестованість і можливість повторного використання.

Дуже часто чиста архітектура має 3 рівні: представлення (англ. presentation), домен (англ. domain) і дані (англ. data).

Рівень представлення: ми розміщуємо наш користувальницький інтерфейс тут (подання та моделі подання)

Доменний рівень: містить бізнес-логіку

Рівень даних: отримує дані з віддалених або локальних джерел даних.

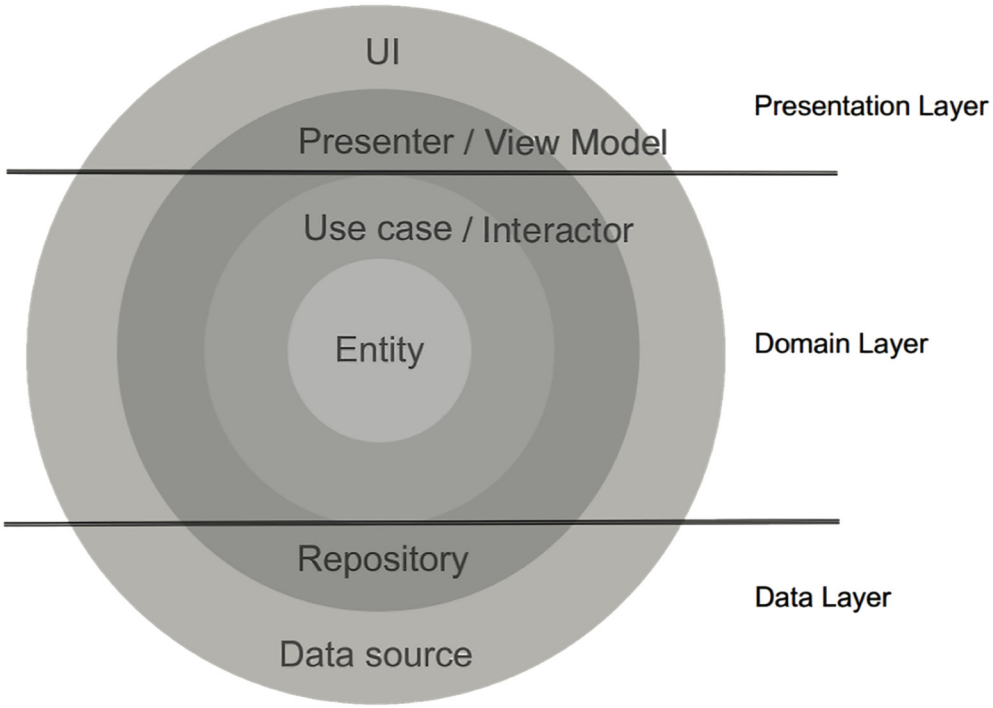


Рисунок 3.3 — Структура чистої архітектури

Дуже часто доводиться впроваджувати мережеві менеджери, клієнти API, джерела даних, контейнери збереження і так далі. Сервіс-орієнтована архітектура спрощує розробку такого програмного продукту, структуруючи взаємодію між реалізаціями високого і нижчого рівня.

На рисунку 3.4 представлено як можна реалізувати сервіс-орієнтовану архітектуру. Важливо зрозуміти, що це однонаправлена архітектура. Тобто, вищі рівні можуть мати доступ до нижнього рівня, проте не навпаки. Сервіси повинні виконувати конкретну роботу і не залежати від верхнього рівня.

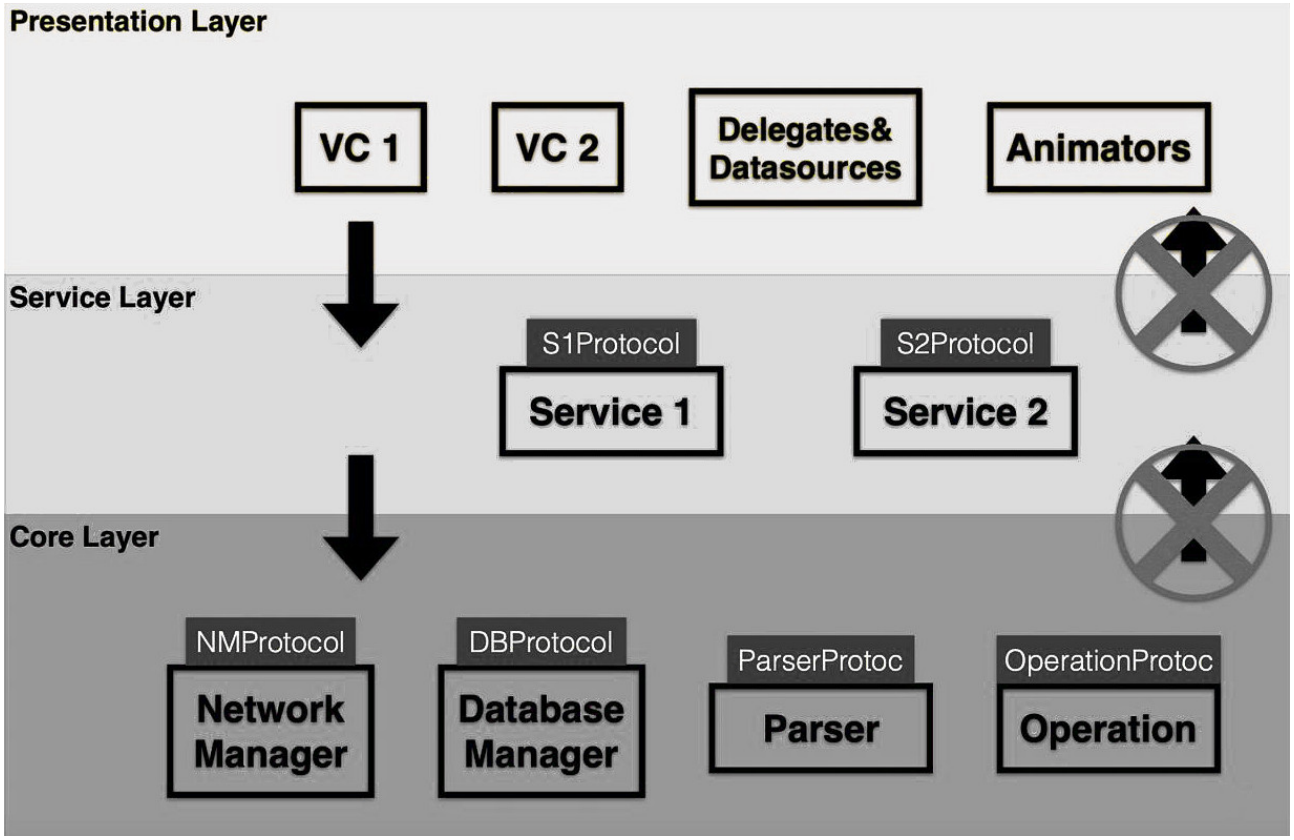


Рисунок 3.4 — Сервіс-орієнтована архітектура

Отож, підсумовуючи SOA — це чудова архітектура, яка дозволяє легко розширювати функціонал додатку. За допомогою неї зберігається принцип єдиної відповідальності — один з принципів SOLID.

SOLID — це абревіатура для п'яти принципів проектування, призначених для того, щоб зробити розробку програмного забезпечення більш зрозумілою, гнучкою і підтримуваною.

Принцип єдиної відповідальності говорить, що кожен модуль, клас або функція повинні нести відповідальність за одну частину функціональності, наданої програмним забезпеченням, і ця відповідальність повинна бути повністю інкапсульована класом. Всі його послуги повинні бути тісно пов'язані з цією відповідальністю.

Окрім архітектури самого додатку — слід обрати структуру презентаційного шару. Під визначення розширюваного структурного патерна підпадає MVP, оскільки вона легко може перетворитись в VIPER, проте не потребує створення дрібних сутностей (стосується VIPER).

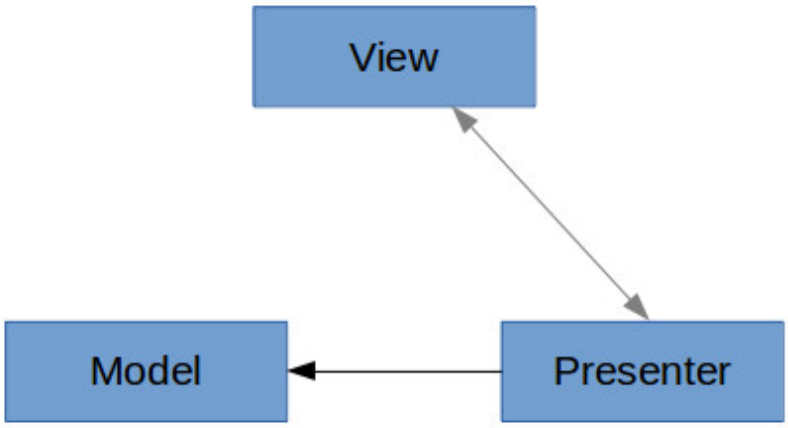


Рисунок 3.5 — Структурний патерн MVP

Вь’ю (англ. View) — Сутність, яка відповідає за візуальне представлення екрану. Саме з цим працює користувач. Спілкується з презентером посилаючи дії користувача для обробки.

Модель (англ. Model) — Відповідає за дані, з якими буде працювати користувач. Спілкується тільки з презентером, оскільки тільки він повинен обробляти всі дані.

Presenter — Виконує усю бізнес-логіку, спілкується з моделлю для обробки даних, та вь’ю для обробки подій від користувача. Саме цей об’єкт має право звертатись до сервісі, надсилати дані та отримувати їх.

Особливості використання MVP:

- Розподіл — велика частина обов'язків розділена між презентером і моделлю.
- Тестованість — відмінна, за рахунок вь’ю, яка передає всі події на обробку в презентер, можна протестувати більшу частину бізнес-логіки.
- Простота використання — обсяг коду подвоюється в порівнянні з MVC, але в той же час ідея MVP дуже ясна.

Стосовно реалізації додатку. Розроблено головну панель, на якій представлено набір віджетів для керування рахунком користувача. Візуальне представлення на рисунку 3.6.

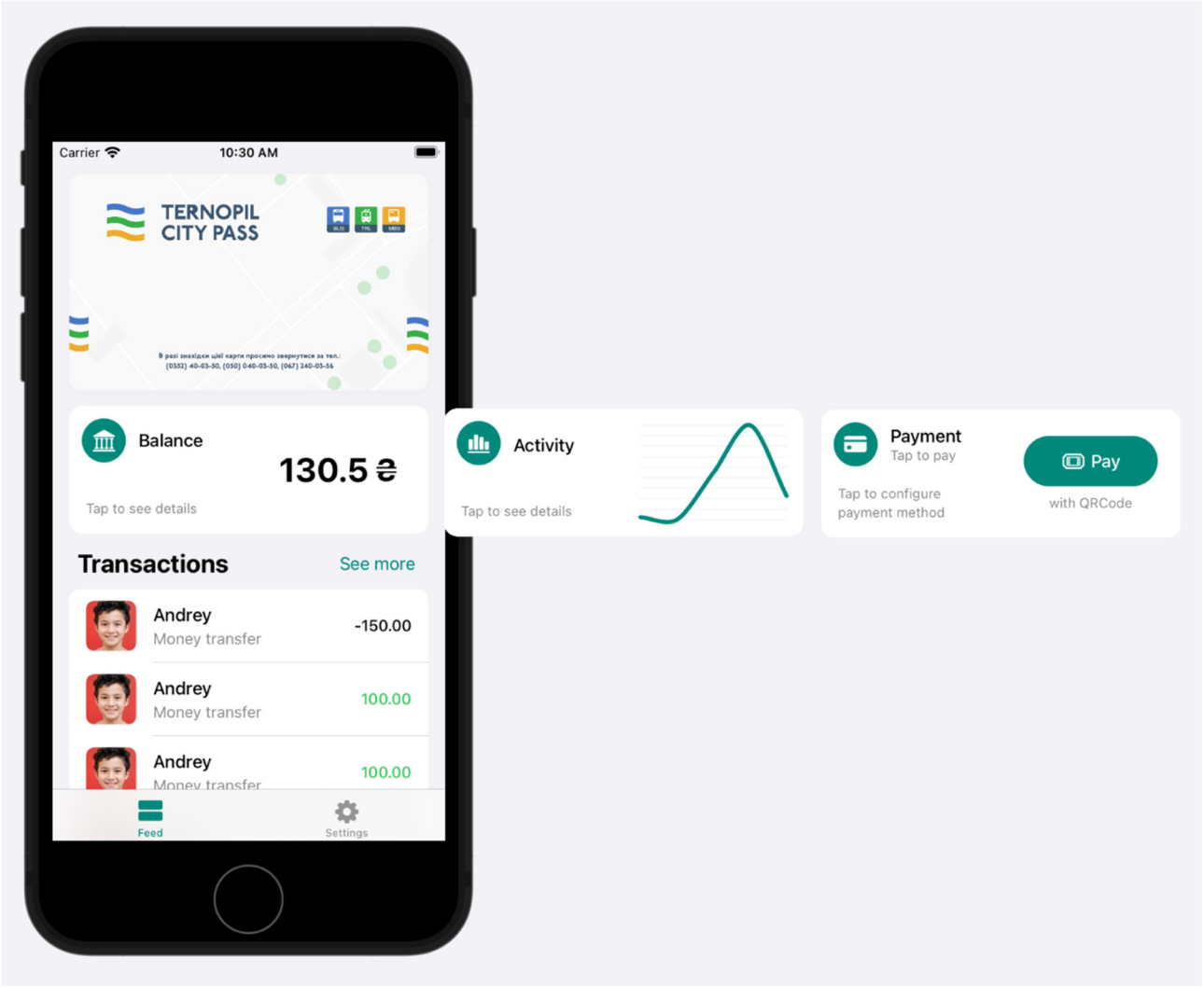


Рисунок 3.6 — Головна сторінка

В самому верху відображено карту користувача — елемент інтерфейсу, який розміщений для візульної складової, ролі ніякої не виконує.

Далі в ряд розміщені набір віджетів. Кожен віджет виконує свою роль. Наразі розроблено 3 таких: віджет балансу — відображає баланс рахунку; віджет активності — відображає в графіку наявний баланс з проміжком в 5 останніх операцій; віджет виконання оплати — щоб здійснити оплату слід перемістити на даний віджет та натиснути на кнопку “Оплатити” (англ. Pay).

Дані віджети розроблені за власними правилами. Ліва частина віджета представлена для додаткової інформації, для прикладу: ідентифікація (як от Баланс, Активність, Оплата, що знаходяться у верхній частині віджета), та додаткова інформація (у нижній частині). Справа віджет повинен містити

корисну інформацію, як от графік, кнопки активації чи звичайний текст. Нижче представлено ієрархія елементів, які сладають віджет балансу (рис. 3.7). Аналогічна ієрархія і у інший віджетів

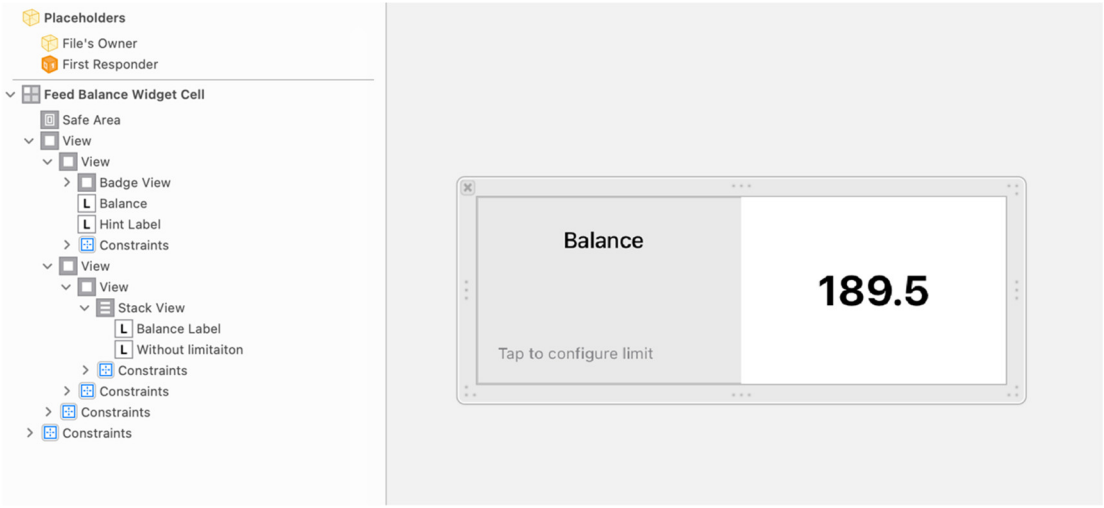


Рисунок 3.7 — Ієрархія елементів віджета балансу

Усі віджети мають візуальне представленя, як на рисунку вище, так і з контроллера, який відображає візуальну частину. Нижче представлено код, який відображає дані в вью.

```
final class FeedBalanceWidgetCell: FeedBaseWidgetCell {
    // MARK: - IBOutlet

    @IBOutlet private var balanceLabel: UILabel!

    // MARK: - Methods

    func configure(withBalance balance: Double) {
        setup(badgeImage: UIImage(systemName: "building.columns.fill"))

        balanceLabel?.text = String(format: "%.1f ₴", balance)
        hintLabel?.text = "Tap to see details"
    }
}
```

В даному клаптику коду представлено метод, який являється єдиним місцем для ініціалізації даного віджету. Ззовні передається лише число, що стосується балансу рахунку. Оскільки даний віджет поміщається в колекцію елементів, то ініціалізатор для таких елементів закритий завдяки механізму перевикористання

елементів колекції, тому єдиний спосіб ініціалізації таких елементів — створення спецальної функції-ініціалізатора.

Нижче віджетів представлено список транзакцій, які відокремлені в групу та характеризують заголовком “Транзакції” (англ. Transactions).

Формування такого елементу відбувається за аналогічним алгоритмом, що і в віджети.

```
final class TransactionCell: UICollectionViewListCell {
    // MARK: - Properties

    var imageProxyService: ImageProxyServiceProtocol?

    // MARK: - Configuration methods

    func setup(with transaction: Transaction, object: Transactable) {
        contentConfiguration = defaultContentConfiguration()

        let value = transaction.value.wrappedValue * (transaction.target
== .sender ? 1 : -1)
        setup(title: object.name)
        setup(subtitle: transaction.kind.description)
        setup(value: value)
        setup(image: object.alternativeImage.imageData ?? Data())

        guard let imagePath = object.imagePath else { return }
        imageProxyService?.fetchImage(
            with: imagePath,
            completion: { [weak self] data in
                guard let data = data else { return }

                DispatchQueue.main.async {
                    self?.setup(image: data)
                }
            }
        )
     }
}
```

По аналогії в функцію-ініціалізатор передаються параметри, які потрібні для відображення контенту. В параметрах вказано 2 елементи: модель транзакції та об’єкт, який передбачає собою людину з якою взаємодіяв користувач або транспорт, тощо. Тип Transactable — це складений тип з набором полів, який повинен містити об’єкт.

```
typealias Transactable = Identifiable & Namable & Imagable
```

Даний складений тип являється сукупністю протоколів Identifiable (відповідає за ідентифікацію), Namable (відповідає за зручне отримання найменування екземпляру класу) та Imagable (віповідає за зручне отримання картинки, які стосується конкретного екземпляру класу).

Прикладом такого класу, який реалізує такий набір протоколів є струптури User, та Vehicle. Нижче представлено реалізацію структури User.

```
struct User: Codable, Transactable {
    enum CodingKeys: String, CodingKey {
        case id, picturePath, name, accountId, account
    }

    // MARK: - Properties

    let id: Int                         // Identifiable protocol
    let picturePath: String?
    let name: String                    // Namable protocol
    let accountId: Int
    let account: Account?

    var imagePath: String? {            // Imagable protocol
        picturePath
    }

    var alternativeImage: Image = {     // Imagable protocol
        .bundleName("avatar")
    }()
}
```

Обєкт User отримується в результаті запиту до сервера. Відповідь з сервера приходить в JSON форматі, тому для парсингу даних потрібно підписатись на протокол Codable, який бере роботу парсингу на себе.

Стосовно елементу транзакцій — важливо підмітити використання змінної imageProxyService з типом ImageProxyServiceProtocol. Дана містить екземпляр класу, що відповідає за надання картинки по шляху. Усі картинки збережені на сервері, тому для отримання їх потрібен мережевий запит. Проте, реалізацію такого класу буде розглянуто пізніше.

Усі ці елементи, як от група з транзакціями та група з віджетами відображаються в колекції, яка в свою чергу відобраєаться в класі FeedViewController. Елементи колекції поступають через окремий клас —

FeedItemsService. Даний клас являється яскравим прикладом розділення логіки на сервіси, що важливо при Service Oriented Architecture, а також принципу Single Responsibility з SOLID, що був описаний вище.

FeedItemsService — створює елементи колекції, які будуть відображатись користувачу. Відповідно має функції для створення таких елементів, параметрами яких є дані отримані з інтернету. Та функції для отримання елементів.

Даний сервіс використовується в зв’язці з FeedPresenter, що являється презентером головного екрану додатку. FeedPresenter по суті являється медіатором (посередником) між FeedItemsService та FeedViewController.

Посередник — це поведінковий шаблон проектування, який зменшує зв'язок між компонентами програми, змушуючи їх спілкуватися побічно, через спеціальний об’єкт посередника.

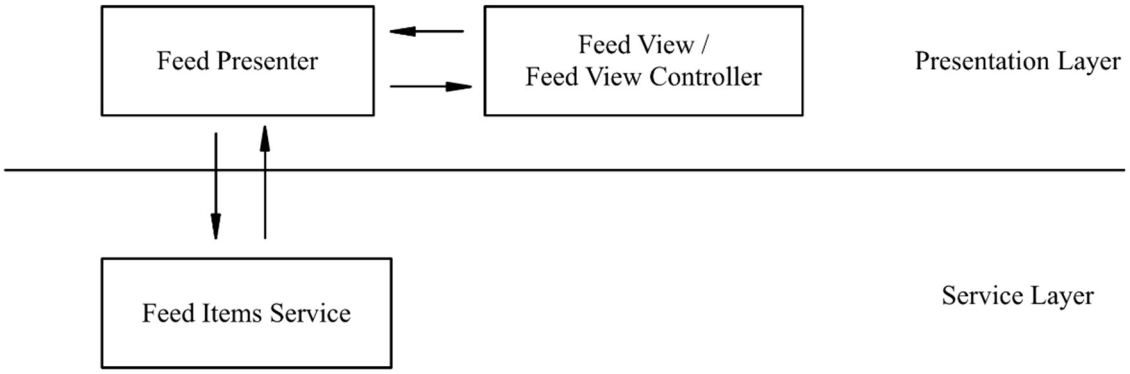


Рисунок 3.8 — Структура екрану Feed

Насправді дана структура є не повною, оскільки до презентера Feed під’єднані і інші сервіси: SocketService, ActivityParseService, NetworkService та ItemsService (див. рис. 3.9). Окрім сервісів, презентер також поєднаний з координатором.

Координатор — це структурний шаблон проектування для організації логіки переміщення між екранами.

Цей шаблон використовується, щоб відокремити екрани один від одного. Єдиний компонент, який безпосередньо знає про екрани — це координатор. Це дозволяє перевикористовувати екрани.

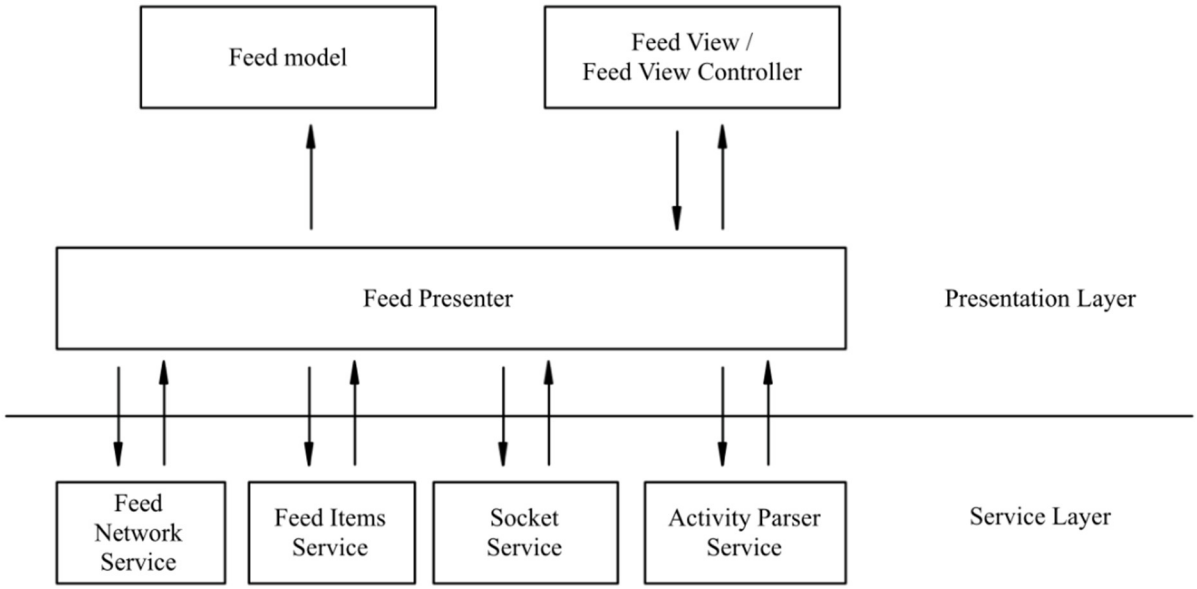


Рисунок 3.9 — Повна структура екрану Feed

На даній схемі представлено зв’язки між компонентами, одного екрану. Додатково реалізацію презентера розміщено в додатку А.

Дуже багато розробників на початку кар’єри не розуміють як реалізовані зв’язки між сутностями як зображено виже на рисунку. Насправді, це звичайні поля (константи та зміні), до яких можна звертатись за домопомогою полів чи методів.

Варто описати як функціонує екран Feed. Презентер даного екрану підписаний на протокол FeedPresenterProtocol.

```
protocol FeedPresenterProtocol {
    func setup()
    func fetchData()

    func numberOfSections() -> Int
    func numberOfItems(in section: Int) -> Int
    func item(by indexPath: IndexPath) -> HolderItemProtocol?
    func header(by section: Int) -> HolderHeaderProtocol?
    func footer(by section: Int) -> HolderFooterProtocol?
}
```

Це означає, що клас який буде наслідуваний цим протоколом повинен реалізувати усі методи даного протоколу та містити усі зміні вказані в протоколі.

Для прикладу метод setup — призначений для початкового налаштування презентеру. В даному методі зручно отримувати дані з мережі, виконувати початкові налаштування та інше.

```
func setup() {
     socketService.subscribe(self, on: [
          .receivedTransaction,
          .sentTransaction
     ])
     fetchData()
}
```

Наразі, він використовується для налшатування наглядача для SocketService'а. Даний сервіс отримує події від сервера через протокол WebSocket. Обробляє, та передає слухачам (наглядачам) цю подію. Дана реалізація зручна тим, що будь-який презентер може стати слухачем для SocketService'а та отримувати події і реагувати на них. Для повної реалізаці слухача потрібно розширити клас презентера FeedPresenter протоколом SocketServiceSubscriberProtocol, для реалізації метода, що буде спрацьовувати при надісланій події.

```
extension FeedPresenter: SocketServiceSubscriberProtocol {
    func socketService(_ socketService: SocketServiceProtocol, didReceive
data: Data?, with event: SocketEvent) {
        switch event {
        case .receivedTransaction:
            fetchData()

        case .sentTransaction:
            fetchData()

        default: break
        }
    }
}
```

Варто підмітити, що встановлення наглядача потребує вибору події, на які буде реагувати презентер. Для того, щоб уникнути непотрібних спрацювань.

Після налаштувань презентера SocketService’ом йде виклик метода fetchData. Даний метод також викликається при спрацюванні метода слухача сокетів.

Реалізація fetchData виконана наступним чином. Створюється операція, яка контролює вкладені операції. Під вкладеними операціями мається на увазі звичайні асинхронні задачі, які зв’язані між собою буфером — дана схема зв’язування операцій через буфер була названа “Сполучна Операція” (англ. Compound Operation), а вкладені операції — “Ланцюгова операція”(англ. Chainable Operations). На рисунку 3.10 представлено схематичну реалізацію.

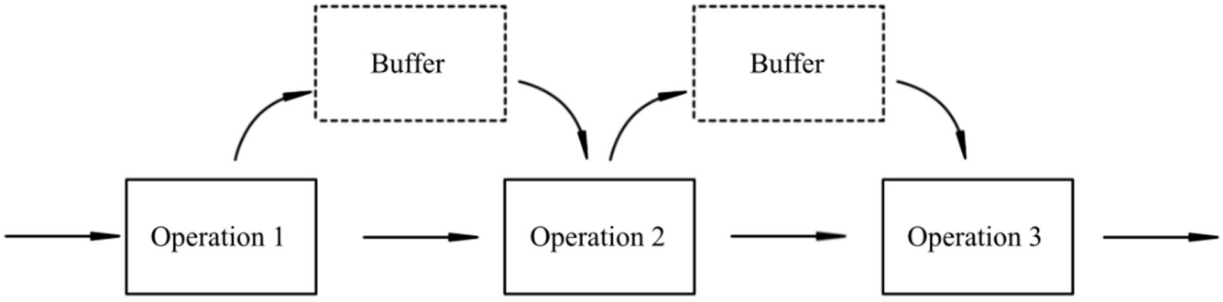


Рисунок 3.10 — Структура Chainable Operations

Механізм доволі простий. Операції виконуються одна за одною. Умова використання такого алгоритму — задачі, які потребують на вхід результат попередньої задачі.

Після виконання першої операції — результат поміщається в буфер, починається виконання наступної операції. Ця операція бере дані з буферу, які були записані попередньою задачою, та приступає до виконання свого алгоритму. Такий порядок дій виконується до тих пір, поки операції не закінчаться. Важливо знати, що буфер створюється окремим об’єктом між двома суміжними операціями, а не один буфер на усі операції.

Наприклад: для здійснення операції по надсиланню даних на сервер попередньо потрібно вийняти з сховища токен, який ідентифікує користувача на сервері. Тому, в результаті у нас появиться 2 операції: операція, яка асинхронно

витягує токен автентифікаці з зашифрованого сховища, та операція яка використовуючи цей токен надсилає дані на сервер. Саме такий сценарій використовується Chainable Operations виконується найчастіше.

Нижче представлено реалізацію таких операцій для екрану Feed. Дані операції відповідають за отримання даних про транзакції від сервера.

```
let operations: [ChainableOperation] = [
    TokenFetchOperation(tokenStorageService: TokenStorage.shared),
    TransactionsFetchOperation(transactionNetworkService: networkService)
]

let operation = CompoundOperation<Any, TransactionsData>()
operation.configure(operations: operations, completion: ...)
operation.start()
```

Операції асинхронні, тому необхідно їх результат обробляти. Це здійснюється за допомогою замикання (completion параметр).

Замикання — це автономні функціональні блоки, які можуть захоплювати і зберігати посилання на будь-які константи і змінні з контексту, в якому вони визначені. Це називається закриттям цих констант і змінних.

Дані, які були отримані в результаті зверення до сервера передаються в замикання, де вони записуються в модель, та передаються в FeedItemsService для їх відображення, після чого презентер просить FeedViewController перемалювати інтерфейс, оскільки отримались дані для відображення.

На цьому етапі нас вже доступні транзакції

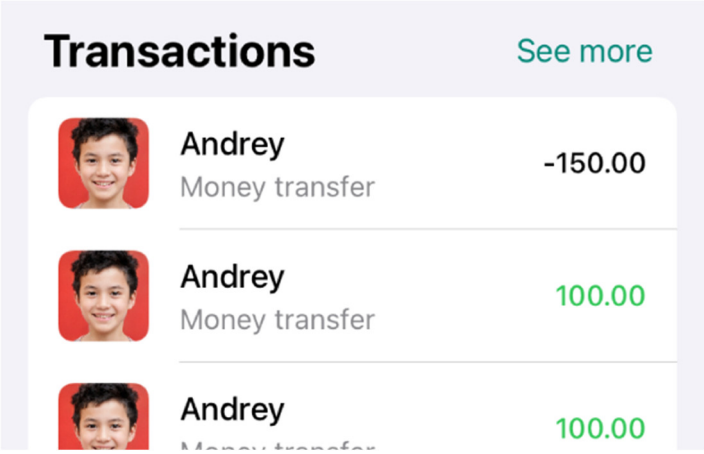


Рисунок 3.11 — Список транзакцій

FeedViewController запитує у презентера наявність елементів колекції. Презентер в свою чергу запитує FeedItemsService для отримання цих елеменів, далі тепер все відбувається в зворотньому напрямку і користувачу відображаються транзакції.

Стосовно елементів списку транзакцій (клас TransactionCell). Як можна замітити на рисунку вище (рис. 3.11) явно видно аватари користувача на імя Андрій (анг. Andrey). Дана картинка зберігається на сервері, а клієнтский додаток їх відображає.

Важливо зазначити, що в даному мобільному додатку реалізовано механізм кешування зображень. За допомогою ImageProxyService здійснюється запит на сервер або локальне сховище. Даний клас використовується в різних обєктах системи, проте було розглянуто його використання в класі TransactionCell, що відповідає за відображення транзакцій.

Дана сутність, ImageProxyService, реалізовує одразу 2 шаблони проектування: одинак (англ. Singletone) та проксі (англ. Proxy).

Шаблони проектування — це типові рішення поширених проблем при розробці програмного забезпечення. Кожен шаблон подібний до креслення, який можна налаштувати для вирішення конкретної проблеми проектування в коді.

Сінглтон — це шаблон проектування створення, який дозволяє гарантувати, що клас має тільки один екземпляр, забезпечуючи при цьому глобальну точку доступу до цього екземпляру.

Проксі — це структурний шаблон проектування, який дозволяє надати заміну для іншого об'єкта.

За допомогою шаблона “сінглтон” ми реалізовуєм доступ до класу ImageProxyService через эдиний екземпляр. Наразі кешуваня відбується в локальне сховище, тому, якщо при подальшій розробці продукту прийдеться відмовитись від локального збереження картинок в сторону зберігання їх поки додаток запущений, то це зекономить багато часу.

Далі про розробку самого сервісу. Конкретну реілзацію сервіса представлено в додатку Б.

Сервіс задовільняє вимоги протоколу ImageServiceProxyProtocol реалізовуючи всього лиш один метод, який являється єдиним джерелом даних про запрошувану картинку.

```
func fetchImage(with path: String, completion: @escaping (Data?) -> ())
```

Для початку — ініціалізація. Об’єкт створюється за допомогою 3 допоміжних об’єктів: DispatchQueue, NetworkingProtocol, ImageFileStorageServiceProtocol. Два останніх параметра зав’язані на тип протокола, щоб забезпечити інверсію залежностей.

Інверсія залежностей — це спосіб роз’єднання програмних модулів шляхом інвертування зв’язків між ними і використання проміжних абстракцій для приховування деталей реалізації. Це допомагає створювати незалежні модулі, змінювати або замінювати один модуль, не торкаючись залежних модулів. Ось лише деякі переваги:

- Модулі можуть бути повторно використані повторно;
- Код може бути протестований незалежно за допомогою mock залежностей;
- Фреймворки і бібліотеки можуть бути легко оновлені і замінені.
- Інверсія залежностей є частиною SOLID принципів, що використовуються в якості керівництва при розробці сучасного програмного забезпечення.

NetworkProtocol дозволяє використовувати об’єкт який його реалізовує — це сам об’єкт Networking, що відповідає за надсилання мережевих запитів.

Оскільки Proxy — це патерн підміни реалізації, то замість мережевих запитів буде використовуватись сервіс локального сховища, який реалізовує протокол ImageFileStorageServiceProtocol. Даний протокл має лише два методи: отримати дані файлу за назвою та записати дані в файл використовуючи назву.

Об’єкт DispatchQueue відповідає за виконання коду синхроно чи асинхроно та синхронізацію даних між потоками. Його використання в даному проксі є вирішенням проблем багатопоточності.

На жаль, незважаючи на всі переваги, що надаються DispatchQueue, вони не є панацеєю від усіх проблем з продуктивністю. Є три добре відомі проблеми, з якими розробник можете зіткнутися при реалізації паралелізму в додатку, якщо не бути обережні:

- Стан гонитви
- Глухий кут
- Інверсія пріоритетів

Наразі, в цьому класі застосовано механізм проти стану гонитви. Потоки, які спільно використовують один і той же процес, включаючи сам додаток, використовують один і той же адресний простір. Це означає, що кожен потік намагається читати і записувати на один і той же загальний ресурс. Якщо не бути обережним, то можна зіткнутися з проблемою гонки станів, в яких кілька потоків одночасно намагаються записати одну і ту ж змінну.

Ближче до реалізації. Протокол вимагає реалізації метода fetchImage. В його параметрах 2 об’єкти: шлях до картинки, який був повернутий з сервера та замикання, що спрацює коли картинка буде отримана.

```
queue.async {
    switch fileStorageService?.get(with: path) {
    case let .some(data):
        completion(data)

    case .none:
        networkService?.perform(
            request: NetworkingRequest.get(route: "/shared/\(path)"),
            completion: { response in

            }
        )
    }
}
```

З самого початку переміщаємо виконання коду в блок асинхронного виклику. Це допоможе уникнути затримок в інтерфейсі, якщо даний код буде запущено в основному потоці.

Далі виконується код по запиту даних з локального сховища. При умові, якщо дані будуть наявні — то вони одразу передадуться в замикання для обробки і функція закінчить своє виконання.

Якщо, картинки в локальному сховищі за вказаним шляхом не буде знайдено — запуститься механізм завантажування картинки з інтернету, а саме з сховища на сервері.

```
networkService?.perform(
    request: NetworkingRequest.get(route: "/shared/\(path)"),
    completion: { response in
        switch response {
        case let .data(data):
            queue?.async(qos: .default, flags: .barrier) {
                fileStorageService?.save(data, with: path)
            }
            completion(data)

        case .error:
            completion(nil)
        }
    }
)
```

Запит на сервер являється асинхроним, тому результат може прийти через деякий час, а дані передадуться з замикання функції perform, яка надсилає запит в мережу. Далі, результат отриманий з сервера перевіряється на наявність картинки. Якщо картинки немає — в замикання самого проксі передається літерал nil, що вказує на відсутність результату.

У випадку коли зображення отрималось, то виконується ще один асинхронний код — запис картинки в локальне сховище. Оскільки запис даних в локальне сховище займає досить часу, щоб заблокувати потік. Після чого дані картинки передаються в замикання і функція завершує роботу. При наступних запитах картинка буде міститись в локальному сховищі, і запрошувати її в сервера не знадобиться. Це дозволяє економити трафік користувача та в подальшому розширити програму для повноцінної підтримки internet-less користування (користування без інтернету).

3.2 Реалізація серверної частини

Основою серверу є набір інструментів бібліотеки Express.js, що дозволяють розгорнути додаток з API для звертання до нього по мережі. Express.js працюэ як надбудова над інструментом під назвою Node.js.

Варто підмітити, що Node дозволяє працювати з інструментами незалежно від апаратної складової, тому малопотужні комп'ютери підійдуть для розгортання проекту як даний.

Специфікація комп'ютера, який виконує код сервера це:

- Процесор — Intel Core 2 Duo / 2 core / 3.7 Ghz
- Оперативна пам'ять — 4 GB
- Операційна система — Linux Ubuntu
- Об'єм накопичувача — 250 GB

Тепер стосовно реалізації. Оскільки JavaScript дозволяє імлементувати код не використовуючи парадигму ООП — було прийнято рішення реалізувати серверну частину використовуючи функціональних підхід.

Функціональне програмування стало дійсно гарячою темою в світі JavaScript. Лише кілька років тому деякі програмісти JavaScript знали, що таке функціональне програмування. Проте, з часом спільнота розробників, активно використовує ідеї функціонального програмування.

Функціональне програмування — це процес створення програмного забезпечення шляхом створення чистих функцій (англ. pure functions), уникаючи загального стану (англ. shared state), змінюваних даних (англ. mutable data) і побічних ефектів (англ. side effects). Функціональне програмування є більш декларативним і стан програми протікає через чисті функції. На відміну від об'єктно-орієнтованого програмування, де стан програми зазвичай спільно використовується і спільно розташоване з методами в об'єктах.

Загальний стан — це будь-яка змінна, об'єкт, що існує в загальній області або як властивість об'єкта, переданого між областями. Загальна область може

включати глобальну область або області закриття. Часто в об’єктно-орієнтованому програмуванні об'єкти спільно використовуються між областями, додаючи властивості до інших об'єктів.

Даний клаптик коду являється стартовою точкою для запуску сервера, який виконується єдиний раз налаштовуючи систему. Команда server.listern() присвоює порт виконання програми для приймання запитів через мережу.

```
const app = express()
const server = http.createServer(app)
const socketApp = socket.use(server)

passport.use('jwt', passportStrategy)

app.use(express.json())
app.use(express.urlencoded({ extended: true }))
app.use(passport.initialize())
app.use(socketMiddleware.initialize(socketApp))
app.use('/', routes)

server.listen(config.urls.port)
```

Оскільки система пов’язана з користувачами, а особливо з рахунками користувачів — було прийнято рішення використовувати алгоритм доступу до системи з використанням токенів. Це забезпечує використання такої бібліотеки як Passport.js, функція якого відображена в коді вище — passport.use('jwt', passportStrategy).

Автентифікація за токенами — це найпопулярніший спосіб автентифікації користувачів у веб-додатках в даний час. Існує великий інтерес до автентифікації за токенами, оскільки в деяких сценаріях вона може бути швидше, ніж традиційна автентифікація на основі сесії, а також забезпечує деяку додаткову гнучкість.

Автентифікація — це процес перевірки особистості користувача.

Автентифікація користувачів у додатку відбувається за допомогою токена (зазвичай це JSON Web Token, JWT) замість фактичних облікових даних.

Це працює наступним чином:

- Користувач хоче увійти на через додаток
- Користувач надає свою адресу електронної пошти та пароль на веб-сайт (свої облікові дані)
- Сервер генерує токен для користувача
- Клієнтська частина з яких надіслані дані — зберігає токен
- Коли користувач робить наступні запити на сервер, його токен буде відправлений разом з його запитом
- Веб-сайт перевірить токен і використовує його, щоб з'ясувати, хто є користувачем

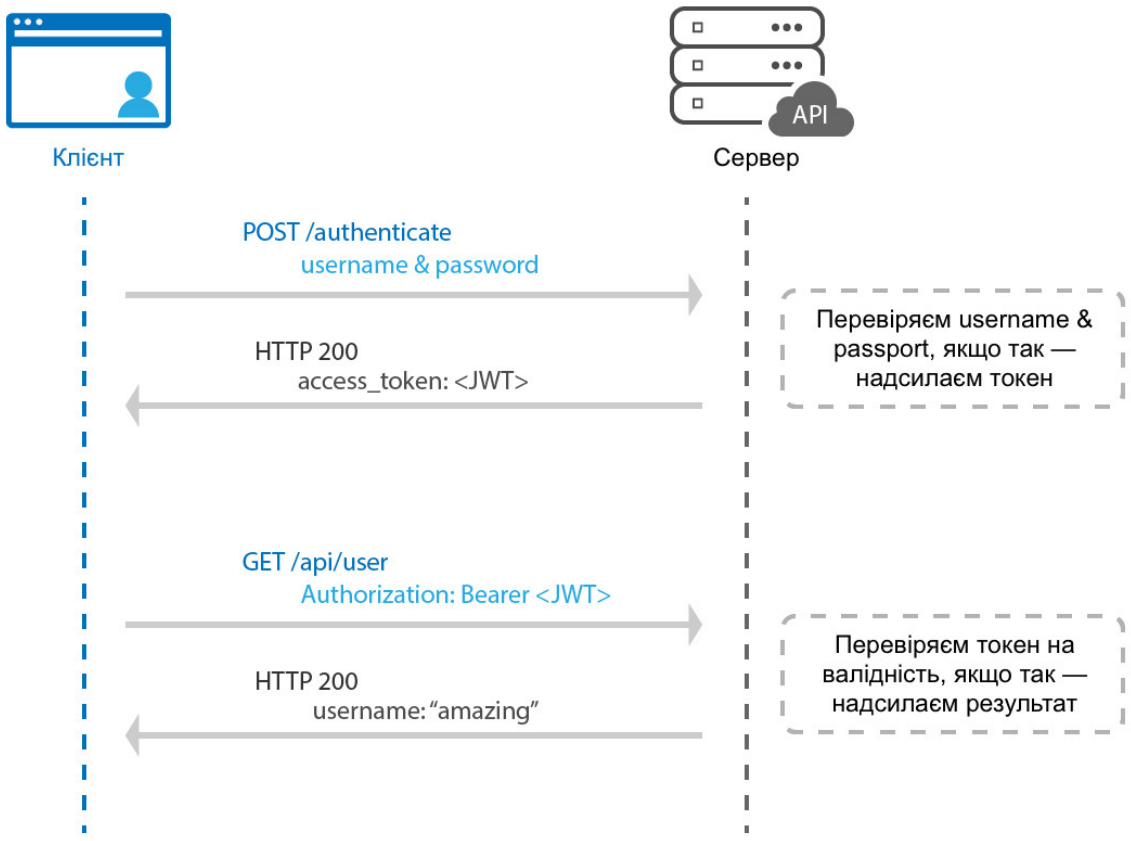


Рисунок 3.12 — Схематичне зображення роботи токена

Перевага цього підходу полягає в тому, що токени містять вбудовану інформацію про користувача, тому клієнт може отримати токен і дізнатися, хто

цей користувач і які у нього дозволи. Це означає, що не потрібно підтримувати сховище сеансів.

Даний механізм додатково використовується до окремих запитів. Приклад такого коду представлено нижче.

```
router.get(
    '/info',
    passport.authenticate('jwt', { session: false }),
    (req, res) => {
        const { id: userId } = req.user

        Accounts.where({ ownerId: userId })
            .fetch({ withRelated: ['card', 'card.type'] })
            .then((account) => {
                res.json({
                    message: 'Account data successfully fetched',
                    data: account,
                })
            })
            .catch(Accounts.NotFoundError, () => {
                res.status(404).json({ error: 'Account not found' })
            })
            .catch((error) => {
                console.log(error)
                res.status(500).json({ error: 'Internal error' })
            })
    }
)
```

Функція passport.authenticate(), що розташована як параметр в батьківській функції router.get() та позначає, що даний метод потребує JWT токена (так як результат буде виданий на основі токена користувача або просто вказує, що метод доступний для користувачів).

В тілі функції router.get() виконується код отримання інформації щодо рахунку користвача. Отримується ідентифікатор через зміну req.user. Ця зміна створюється в результаті виконання декодуваня токена. Алгоритм декодування буде розглянуто пізніше.

Далі за допомогою ідентифікатора користувача виконується звернення до таблиці Account в бази даних через одноіменну модель, в результаті корисувач отримає результат, позитивний чи негативний. Такий тип звернення до бази данх

стає можливим в результаті застосування бібліотеки Bookshelf.js, що являється прошарком між базою даних та сервером. З нею легко будувати запити до бази даних використовуючи зручні методи, що в результаті конвертуються в SQL.

Стосовно декодування токену та токену в загальному. Токен — це об’єкт, який можна використовувати для аутентифікації користувача на сервері. Токени містять вбудовані дані користувача, які використовуються для ідентифікації та аутентифікації користувача.

Веб-токени JSON (JWT) — це відкритий стандарт, який визначає безпечний спосіб передачі інформації між сторонами з використанням об’єкта JSON. JWT завжди мають криптографічний підпис і можуть бути підписані за допомогою секретного ключа (симетричного) або пари відкритого/закритого ключа (асиметричного).

JWT — це найпопулярніший тип токенів, і часто люди мають на увазі, коли вони посилаються на "аутентифікацію токенів" в цілому.

Ось як може виглядати Типовий JWT в стислій, безпечній для URL формі:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MywiZW1haWwiOiJ0YXJhc
y5rb3RzdXIudG5ldUBnbWFpbC5jb20iLCJpYXQiOjE2MjA0OTcyNjB9.bjnZHl3xPr
GX6xGp0qWSYDvynZMwSOnrdGALCV3FRh8
```

Хоча на перший погляд це може здатися складним і нечитабельним, насправді це не так вже й складно. JWT складаються з трьох частин, розділених крапками (.): xxxxxx.yyyyyy.zzzzzz. Ці розділи представляють заголовок JWT, корисне навантаження та підпис відповідно. Далі, детаільніше про кожен пункт.

Заголовок JWT являє собою об'єкт JSON в кодуванні Base64URL. Він містить інформацію, що описує тип токена і використовуваний алгоритм підпису, наприклад HMAC, SHA256 або RSA.

Наприклад:

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

Специфікація JWT є гнучкою і дозволяє використовувати різні типи алгоритмів, тому це поле заголовка завжди буде присутнім.

Корисне навантаження JWT містить дані про якусь сутність (зазвичай про користувача) і додаткові дані.

У наведеному вище прикладі токена корисне навантаження при десеріалізації виглядає наступним чином:

```
{
  "id": 3,
  "email": "taras.kotsur.tneu@gmail.com",
  "iat": 1620497260
}
```

Поле підпису JWT створюється шляхом взяття закодованого заголовка, закодованого корисного навантаження, секретного ключа і використання алгоритму, зазначеного в заголовку, для криптографічного підпису цих значень.

Наприклад, якщо використовується стандартний симетричний алгоритм HMAC SHA256, підпис буде створений шляхом обчислення:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
)
```

Це поле підпису використовується сервером для перевірки цілісності токена і забезпечення того, щоб він не був змінений або відредагований третьою стороною.

При запуску сервера ініціалізуються його сутності. Поміж них також ініціалізується сервіс для ідентифікаїї користувачів, який був згаданий раніше — це Passport.js. Нижче представлено алгоритм по декодуванні токена.

```
const { Strategy, ExtractJwt } = require('passport-jwt')
module.exports = new Strategy({
        jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
        secretOrKey: SECRET,
    }, (payload, done) => {
        User.where({ id: payload.id }).fetch()
            .then((user) => done(null, user.toJSON()))
            .catch((error) => done(error, false))
    }
)
```

За допомоги створення стратегії (механізм декодування) відбувається декодування токену, в параметрах вказується тип та секретний ключ, який використовувався при генерації токена.

Стратегія допомагає ідентифікувати користувача через параметр id записаний у токені. Після чого даний параметр використовується для зчитування інформації в базі даних. В результаті вони будуть поміщені в зміну req.user, що була згадана раніше.

Наступним методом захисту в даній системі — хешовані дані. Даний спосіб дозволяє використовувати системні дані не переймаючись за їх витік.

Такий спосіб використовується в багатьох кейсах даної системи. Наприклад: при переказі коштів між користувачами чи оплати транспорту.

Далі буде описано реалізований механізм на прикладі переказу коштів.

Переказ коштів відбувається за допомогою сканування QR коду відправником, проте щоб попасти на екран — дані проходять великий шлях.

Для початку отримувач переходить в меню налаштувань та обирає пунтк “Ідентифікатор” (англ. Identifier). В цей момент додаток надсилає запит на сервер за шляхом …/personal/identifier з токеном. Обробник, який спрацьовує — отримує всі дані та декодує токен. Після декодування дані про користувача стають доступними серверу і він може ними оперувати. Виконується створення хеш запису в базі даних в таблиці hashes.

Дана таблиця має в собі наступні поля:

Таблиця 3.1 — Поля таблиці Hashes

Назва поля	Тип даних поля	Опис
id	Integer	Ідентифікатор запису
hash	String	Хеш; зовнішній ідентифікатор
type	Integer	Тип події для хеша
entityId	Integer	Ідентифікатор прив’язаної сутності
entity	String	Назва таблиці сутності
expiratingAt	Timestamp	Термін дії хешу
createdAt	Timestamp	Дата створення хешу
updatedAt	Timestamp	Дата оновлення хешу

В поле hash записується корисна нагрузка у вигляді набору наступних значень: тип події, ідентифіактор сутності до якої прив’язаний хеш, назва таблиці сутності та термін дії хешу. Це дає змогу перевірити достовірність хеша при його використанні.

Під типом події мається наувазі те, як даний хеш буде використовуватись. Для прикладу типи хешів, які стосуються таблиці User:

- activation — використовується для першої активації облікового запису з терміном дії в 3 дні;
- changePassword — використовується для зміни паролю облікового запису з терміном дії в 2 години;
- forgotPassword — використовується для відновлення паролю облікового запису з терміном дії в 30 хвилин;
- identifier — використовується для ідентифікації облікового запису з терміном дії в 2 хвилини.

Якщо хеш користувача з таким типом вже існує, тоді він перезаписується на новий. Після цього згенерований хеш попередньо записаний в базу даних передається користувачу для генерації QR коду.

Нижче представлено приклад того як це працює.

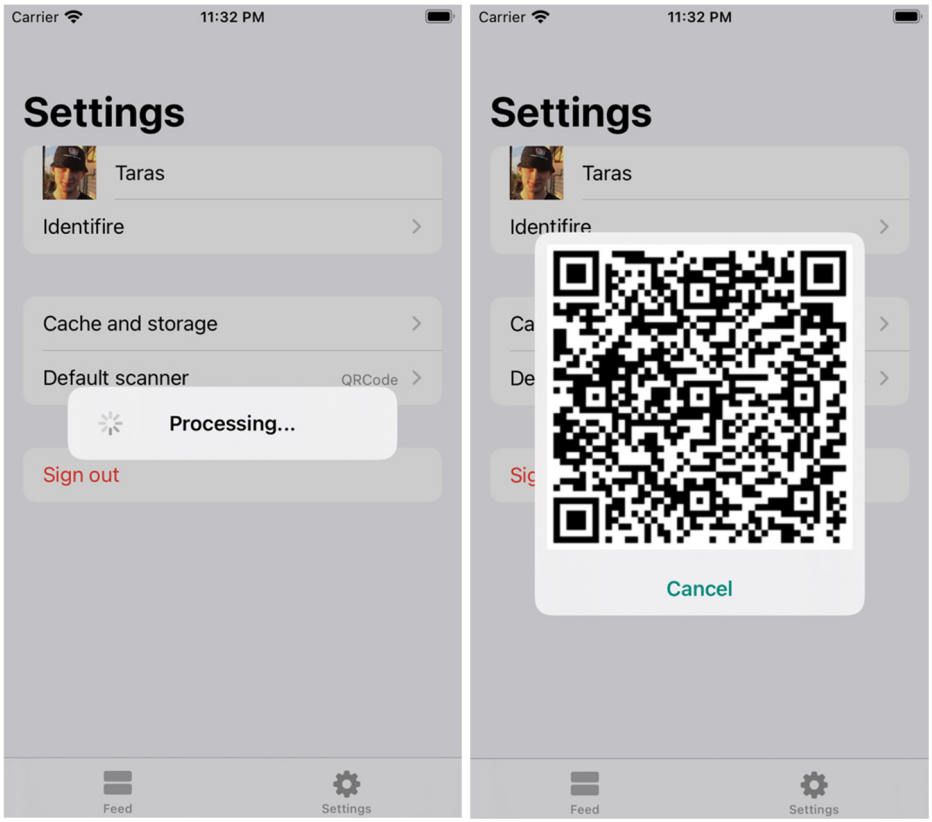


Рисунок 3.13 — Результат генерування QR коду з створеного хеша

Далі цей хеш сканується платником через згенерований в отримувача QR код. Даний хеш від сторони платника передається на сервер для встановлення об’єкта. Очікується, що в результаті він отримає мінімальний набір даних користувача — ім’я та шлях до картинки.

В результаті платнику буде доступний інтерфейс для зручного переказу коштів (рис. 3.14). На даному екрані користувач зможе встановити суму для переказу та переказати.

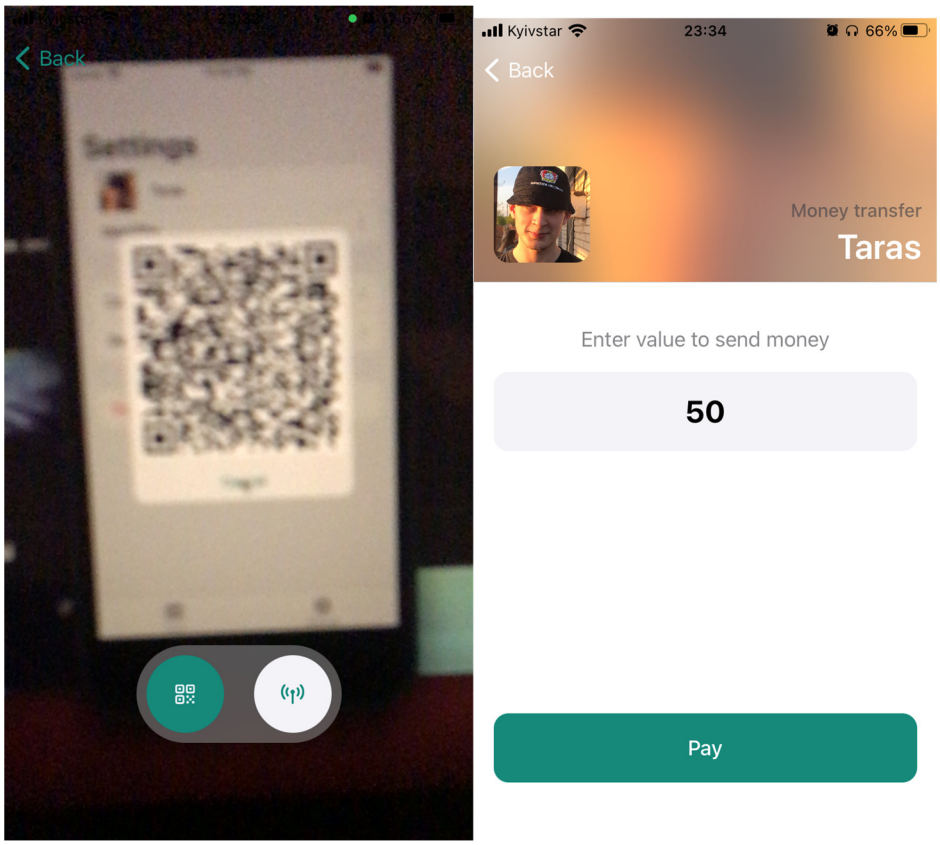


Рисунок 3.14 — Інтерфейс переказу коштів

Після натиску на кнопку оплати користувач надсилає запит за шляхом "…/transactions/transfer" з параметрами hash та value, де hash — хеш потенційного отримувача, а value — сума переказу.

На стороні сервера даний запит обробляється наступним чином:

- Отриманий хеш перевіряється на валідність та відповідність події;
- Перевіряється баланс відправника на відповідність сумі переказу;
- Створюється транзакційний ланцюжок, де відправнику зміншується баланс, а отримувачу збільшується.
- Після успішного внесення змін перевіряється минула транзакція на внесення змін з третьої сторони.
- Створюється запис про транзакцію в таблиці бази даних
- Транзакція завершується та результат відправляється користувачу

Транзакція в контексті бази даних — це набір задач, які виконуються для вилучення або оновлення даних.

У традиційному дизайні реляційних баз даних транзакції завершуються операторами COMMIT або ROLLBACK SQL, які вказують початок або кінець транзакції.

На будь-якому етапі транзакції може статись помилка після чого весь ланцюжок змін буде відмяняти свої зміни повертаючи дані до вихідних.

Стосовно перевірки минулої транзакції перед створенням нової. Даних механізм впроваджений для запобігання вмішування третіх осіб в роботу ситеми. В системі де гроші вважаються важливими даними такий механізм дозволяє запобігти несанкціонованих змін в рахунках користувачів.

Механізм транзакцій розроблений по принципу Blockchain.

Термін “блокчейн” відноситься до того факту, що це “ланцюжок “ “блоків”. “Ланцюжок”, тому що все записано в хронологічному порядку. І “блоки”, тому що транзакції додаються в ланцюжок групами, а не окремо.

Кожен блок записує кілька транзакцій, подібно сторінці в книзі. Кількість транзакцій в блоці варіюється від блокчейна до блокчейну.

Ці транзакції записуються у вигляді хешів-рядків цифр і букв. Хеш кожної транзакції генерується для включення інформації з поточних і минулих транзакцій. Це створює ефект ланцюжка, при якому порядок хешів не може бути змінений. В результаті транзакції стають незмінними після їх додавання.

Всі транзакції повинні бути перевірені, перш ніж вони будуть додані в блокчейн.

В даному випадку перевірка відбувається до внесення в базу даних та після (коли наступна транзакція попадає в чергу на запис).

Таблиця з транзакціями має наступну структуру.

Таблиця 3.2 — Поля таблиці Transactions

Назва поля	Тип даних поля	Опис
id	Integer	Ідентифікатор запису
senderId	Integer	Ідентифікатор сутності відправника
senderEntity	String	Назва таблиці відправника
senderPrevBalance	Double	Попередній баланс відправника
receiverId	Integer	Ідентифікатор сутності отримувача
receiverEntity	String	Назва таблиці отримувача
receiverPrevBalance	Double	Попередній баланс отримувача
value	Double	Сума відправлення
method	Integer	Тип створення транзакції
status	Integer	Статус транзакції
hash	String	Хеш транзакції
createdAt	Timestamp	Дата створення транзакції
updatedAt	Timestamp	Дата оновлення транзакції

Хеш транзакції будується на основі усіх параметрів записаних в конкретний запис, тому при зміні будь-якого значення поведе за собою різницю хеш-сум, що в результаті не дасть змогу іншим користувачам здійснювати перекази.

3.3 Тестування розробленого додатку та порівняння з аналогами

Розробка програмного забезпечення — складний процес. Окрім оволодіння різними інструментами та мовами програмування, існує також розуміння різних ролей у розробці програмного забезпечення. Як відомо, великі проекти пов’язані не тільки з програмуванням. Вони також включають інші ресурси для збору вимог, створення прототипів та тестування.

Процес перевірки коду часто називають забезпеченням якості. Однак ця фраза дещо вводить в оману. Замість того, щоб гарантувати якість коду, реальна мета полягає в тому, щоб виміряти його — часто в міру розвитку проекту. Також відома як розробка на основі тестування (англ. Test Driven Developing, TDD), ця тонка різниця може бути помічена при вивченні ролей функцій якості.

Вимірювання якості часто підрозділяється на автоматизовані та ручні дії. Додатковий код, відомий як модульні тести (англ. Unit Tests), працює для тестування основного програмного проекту. Модульні тести зазвичай пишуться розробниками і знаходяться в центрі уваги в середовищі TDD. Використовуючи фреймворк iOS XCTest, далі буде описано процес тестування авторизації.

Стосовно правил написання тестів. Хоча, це і не потрібно, але рекомендується, щоб файл модульного тесту повністю відповідав іменам файлів реалізації з приміткою “Test”. У даному випадку буде використовуватись файл LoginNetworkServiceTests.swift, що буде тестувати сервіс LoginNetworkService. Даний сервіс призначений для авторизації користувача.

```
import XCTest
@testable import OneTransport

final class LoginNetworkServiceTests: XCTestCase {
    // MARK: - Properties

    var networkService: LoginNetworkServiceProtocol!

    // MARK: - Setup methods

    override func setUp() {
        super.setUp()

        ...
    }

    // MARK: - Test cases

    func testFetchingDataWithMocketURLRequest() {
        ...
    }
}
```

Як і інші платформи модульного тестування, XCTest працює шляхом інтеграції функцій мови Swift з конкретними функціями, пов’язаними з тестуванням. Основні методи згруповані разом у вигляді тверджень. При

створенні тестів компілятор розпізнає модульні тести з функціями з префіксом ключового слова test.

На відміну від звичайних функцій Swift, модульні тести навмисно розробляються як автономні логічні одиниці. В результаті методи тестування не приймають Аргументи і не повертають значення. Крім того, тестовими даними можна управляти за допомогою допоміжних методів і послідовностей ініціалізації / руйнування.

Основним завданням сервісу авторизації, LoginNetworkService — є отримання токена авторизації, з за допомоги якого користувач ідентифікується.

Алгоритм тестування наступний: для початку необхідно написати тест-функцію, яка буде виконувати код по відправці запиту на сервер; зробити mock-об'єкт (заглушку) запиту, щоб підміняти виклик запиту в мережу.

Тесту насправді все одно, викликається функція чи ні на заглушці, якщо тестовий об'єкт (або тестована система) отримує необхідні дані з заглушки і робить все правильно.

Це робиться для того, щоб уникнути звертання до сервера напряму. Так як виконується unit-тестування, а не інтеграційне.

Інтеграційне тестування — це тестування інтеграції різних частин системи разом. Спочатку інтегруються дві різні частини або модуля системи, а потім проводиться інтеграційне тестування.

Дана функція являється тест-кейсом, що використовує екземпляр класу LoginNetworkService:

```
func testFetchingDataWithMocketURLRequest() throws {
      let loginData = LoginData(email: "some@mail.com", password:
"password")
      networkService.performLogin(with: loginData, completion: { response in
            guard case let .success(data) = response else {
                XCTFail("Expected successfull response")
                return
            }
            XCTAssert(data.token.isEmpty == false, "Expected not empty
token")
      })
}
```

В цьому кейсі створюєм модель даних, яка буде передана в сервіс на обробку. Поля моделі — пошта та пароль. Після виклику функції performLogin очікуєм результату, що викличе запикання completion. В даному замикані обробляєм результат, якщо він позитивний — перевіряєм токен (що відповіждає успішному результату). У випадку, якщо результат негативний — вказуєм, що тест провалився.

Як було згадано вище, потрібно створити mock-об’єкт запиту, що буде імітувати запит в мережу, щоб не залежати від сервера. Даний об’єкт буде реалізовувати зчитування дані з файлу, в який записано приклад даних, що повертає сервер. Таких файлів буде багато, один на унікальний запит (рис.). Вони будуть статичними.

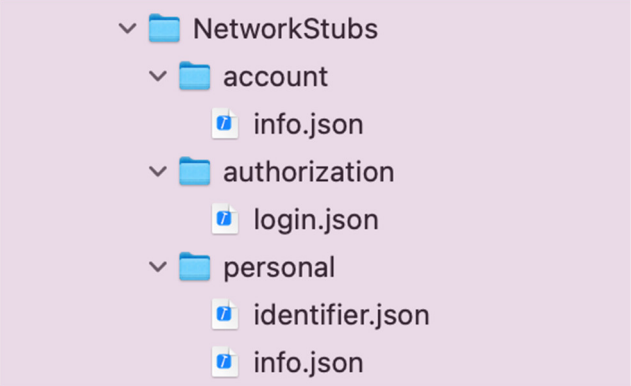
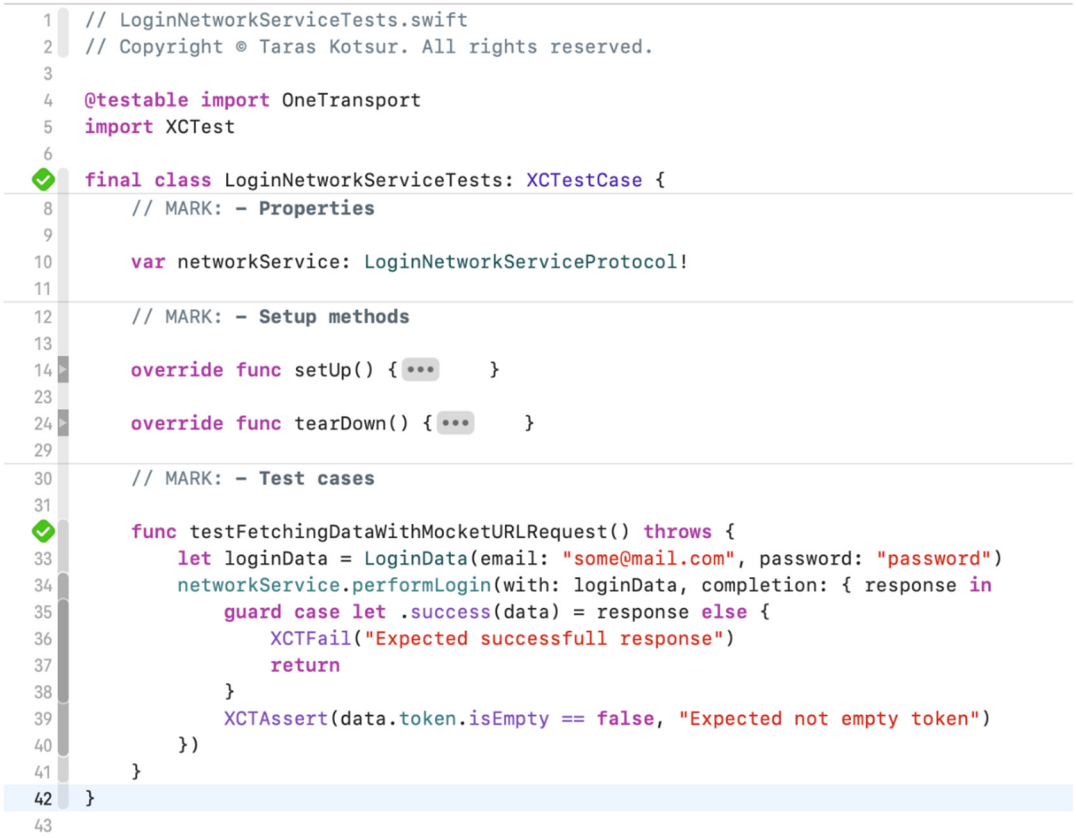


Рисунок 3.15 — Структура фейкових результатів запитів

Реалізація об’єкту, який реалізовує підміну реального запиту на той, що зчитує дані з файлів розташовано в додатку В.

Запуск даного тесту здійснюється через контекстне меню середовища або комбінацією клавіш Command+U. Результат тестування вказано на рисунку 3.16.

По ліву сторону тест-кейси, що пройшли тестування помічається зеленою галочкою.



```swift
// LoginNetworkServiceTests.swift
// Copyright © Taras Kotsur. All rights reserved.

@testable import OneTransport
import XCTest

final class LoginNetworkServiceTests: XCTestCase {
    // MARK: - Properties

    var networkService: LoginNetworkServiceProtocol!

    // MARK: - Setup methods

    override func setUp() { ••• }

    override func tearDown() { ••• }

    // MARK: - Test cases

    func testFetchingDataWithMocketURLRequest() throws {
        let loginData = LoginData(email: "some@mail.com", password: "password")
        networkService.performLogin(with: loginData, completion: { response in
            guard case let .success(data) = response else {
                XCTFail("Expected successfull response")
                return
            }
            XCTAssert(data.token.isEmpty == false, "Expected not empty token")
        })
    }
}
```

Рисунок 3.16 — Результат виконання тесту

Для наступного етапу тестування нам знадобиться даний сервіс з підробленими запитами. Далі буде виконуватись тестування презентера для екрана авторизації — LoginPresenter. Код тест-кейсів розміщено в додатку Г.

Презентер авторизації має всього 3 методи: встановити пошту (setEmail), встановити пароль (setPassword) та здійснити авторизацію (selectLogin). Дані методи потрібно протестувати.

Два перших методи: setEmail та setPassword буде протестовано впершому тест-кейсі, так як вони подібні і виконують схожий функціонал. Для цього необхідно визначити вхідні параметри. Виконати функції використовуючи їх та порівняти результат.

```
func testModelDataSaving() throws {
    let credentials = (email: "some@mail.com", password: "password")
    presenter.set(email: credentials.email)
    presenter.set(password: credentials.password)

    XCTAssert(presenter.email == credentials.email)
    XCTAssert(presenter.password == credentials.password)
}
```

Наступний тест буде покривати функцію selectLogin — вона виконується при натиску на кнопку “Авторизуватись”.

За допомогою інверсії залежностей — їх ін’єкція не створює проблем. Презентер потребує для ініціалізації 3 залежності та 1 модель, яка була протестована в попередньому кейсі. Залежності такі: LoginNetworkService (який був протестований раніше), TokenStorageService та AuthFlowCoordinator. Останні два відповідають за збереження токена користувача та навігацію в додатку відповідно.

Завдання функції selectLogin отримати токен з мережі використовуючи параметри користувача, які він ввів, зберегти токен та передати його координатору, щоб здійснити перехід на основний екран.

Для реалізації цього кейсу було створено stub-об’єкти TokenStorageService та AuthFlowCoordinator, щоб уникнути запису токена в сховище та використання робочого сервісу переходів між екранами відповідно.

```
func testTokenServiceOnSavingToken() throws {
    let expectation = XCTestExpectation(description: "Expecting async
operation while doing login action")

    coordinator.didFinishLogin = { token in
        expectation?.fulfill()

        XCTAssert(tokenService?.token?.isEmpty == false, "Expected not
empty token")
        XCTAssert(token.isEmpty == false, "Expected not empty token")
    }

    coordinator.didStartErrorAlert = { error in
        expectation?.fulfill()

        XCTFail("Expected token instead of error, \(error)")
    }

    tokenService.token = nil

    let credentials = (email: "some@mail.com", password: "password")
    presenter.set(email: credentials.email)
    presenter.set(password: credentials.password)
    presenter.selectLogin()

    wait(for: [expectation], timeout: 5)
}
```

Оскільки в середині метода selectLogin виконується асинхронний код — було використано об’єкт типу XCTestExpectation, що допомагає протестувани виконання асинхронного коду через допоміжну функцію wait. В даній функції потрібно вказати максимальний час очікування.

Як було сказано вище, презентер звернеться до координатора, коли результат буде готовий. Тому, stub-об’єкт коондинатора спроектований таким чином, що буде виконуватись його замикання при виконанні функцій з презентера. В даних замикання перевіряється результат та наявність токена.

Результат описаних тестів вказано на рисунку 3.17.

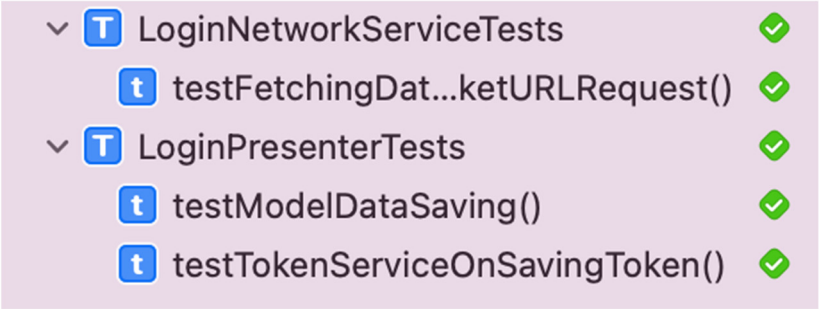


Рисунок 3.17 — Результати виконаних тестів

В даному пункті було на прикладах показано як легко покривати код тестами використовуючи структурний патерн MVP та дотримуватись SOLID принципів. Розробляючи систему схожим шляхом розробник прийде до висновку, що будь-який елемент можливо вирізати та вмонтувати в будь-який інший проект розроблений по похожим принципах.

Стосовно аналогічних рішень. Дана система забезпечує зручну оплату за проїзд в транспорті та багато інших можливостей, які в порівнянні з схожими системами чітко виходить на перші місця.

Нижче сформована таблиця (таблиця 3.3) переваг та недоліків аналогічних систем і порівняння з розробленим проектом. Серед схожих рішень було обрано 2 системи, які більш здатні до конкуренції. Власна система виділена темнішим кольором для зручності.

Таблиця 3.3 — Переваги та недоліки систем

Можливості	Файна карта	Київ Цифровий	One Transport
Переказ коштів між рахунками	Ні	Ні	Так
Безготівкова оплата транспорту	Так	Так	Так
Walletless	Ні	Так	Так
Автоматичне очищення балансу	Ні	Так	Ні
Перегляд витрат	Ні	Так	Так
Висока швидкість синхронізації	Ні	Так	Так
Велика функціональна гнучкість	Ні	Так	Так

Виходячи з результатів порівняння систем можна зробити висновок щодо поточної системи, яка діє в місті Тернопіль. Даний проект проектувався даного міста як аналогічне рішення, та потенційний аналог. Мета даної роботи виконана успішно.

4 Техніко-економічний розділ

У даному розділі кваліфікаційної роботи проводиться економічне обґрунтування доцільності розробки програмного забезпечення мережевого обладнання. Зокрема, здійснюється розрахунок витрат на розробку даного програмного продукту, експлуатаційних витрат, ціни на споживання проектного рішення, визначаються показники економічної ефективності нового програмного продукту, обґрунтовуються відповідні висновки.

Розроблене програмне забезпечення мережевого обладнання призначено для правильного налаштування обладнання.

4.1 Розрахунок витрат на розробку програмного забезпечення

Витрати на розробку і впровадження програмних засобів ($К$) включають [30]:

$$К = К_1 + К_2, \tag{4.1}$$

де $К_1$ – витрати на розробку програмних засобів, грн.;

$К_2$ – витрати на відлагодження і дослідну експлуатацію програми рішення задачі на комп'ютері, грн.

Витрати на розробку програмних засобів включають:

- витрати на оплату праці розробників ($В_{ОП}$);
- витрати на відрахування у спеціальні державні фонди ($Вф$);
- витрати на покупні вироби ($Пв$);
- витрати на придбання спецобладнання для проведення експериментальних робіт ($Об$);
- накладні витрати ($Н$);
- інші витрати ($Ів$).

Витрати на оплату праці включають заробітну плату (ЗП) всіх категорій працівників, безпосередньо зайнятих на всіх етапах проектування. Розмір ЗП обчислюється на основі трудомісткості відповідних робіт у людино-днях та середньої ЗП відповідних категорій працівників.

У розробці проектного рішення задіяні наступні спеціалісти-розробники, а саме: керівник проекту; студент-дипломник; консультант техніко-економічного розділу (таблиця 4.1).

Таблиця 4.1 – Вихідні дані для розрахунку витрат на оплату праці

Посада виконавців	Місячний оклад, грн.
Керівник БР, ст. викладач	7449
Консультант техніко-економічного розділу, доцент	7449
Студент	1400

Витрати на оплату праці розробників проекту визначаються за формулою:

$$B_{ОП} = \sum_{i=1}^{N} \sum_{j=1}^{M} n_{ij} \cdot t_{ij} \cdot C_{ij} , \qquad (4.2)$$

де n_{ij} – чисельність розробників i-ої спеціальності j-го тарифного розряду, осіб;

t_{ij} – затрачений час на розробку проекту співробітником i-ої спеціальності j-го тарифного розряду, год;

C_{ij} – годинна ставка працівника i-ої спеціальності j-го тарифного розряду, грн.

Середньогодинну ставку працівника розраховуємо за формулою:

$$C_{ij} = \frac{C_{ij}^{0}(1+h)}{РЧ_i}, \quad (4.3)$$

де C_{ij}^{0} – основна місячна заробітна плата розробника *i*-ої спеціальності *j*-го тарифного розряду, грн.;

h – коефіцієнт, що визначає розмір додаткової заробітної плати (при умові наявності доплат);

$РЧ_i$ – місячний фонд робочого часу працівника *i*-ої спеціальності *j*-го тарифного розряду, год. (приймаємо 168 год.).

Результати розрахунку записуємо у таблицю 4.2, зважаючи на те, що коефіцієнт *h* для керівника і для консультанта з техніко-економічного розділу – 1,47.

Таблиця 4.2 – Розрахунок витрат на оплату праці

Посада виконавців	Час розробки, год.	Погодинна заробітна плата, грн/год.	Витрати на розробку, грн
Керівник ДП, ст. викладач	16	64,3	1028
Консультант техніко-економічного розділу, ст. викладач	2	59,9	119,8
Студент	144	8	1199
Разом			2346

Величину відрахувань у соціальні фонди визначаємо згідно діючого законодавства у розмірі 20,5% від суми заробітної плати:

$$B_{Ф} = \frac{20,5}{100} \cdot 2346 = 480,93 \text{ грн.}$$

У таблиці 4.3 наведений перелік матеріалів та комплектуючих і розраховані витрати на них.

Таблиця 4.3 – Розрахунок витрат на матеріали та комплектуючі

№ п/п	Найменування купованих виробів	Одиниця виміру	Ціна, грн	Кількість куповани х виробів	Сума, грн	Транспортні витрати (10% від суми)	Загальна сума, грн
1	Папір (формат А4)	уп.	100,0	2	200,00	20,0	220,0
2	Ручка кулькова	шт.	10,0	1	10,00	1,0	11,0
3	Олівець простий	шт.	8	2	16,00	1,6	17,6
4	Диски CD-R	шт.	9	2	18,00	1,8	19,8
5	Зошит, 96 арк.	шт.	20	1	20	2	22
6	Тонер для принтера	уп.	80	1	80	8,0	88
Разом							378,4

Витрати на використання комп’ютерної техніки включають витрати на амортизацію комп’ютерної техніки, витрати на користування програмним забезпеченням, витрати на електроенергію, що споживається комп’ютером. За даними обчислювального центру ТНЕУ для комп’ютера типу IBM PC/ATX вартість години роботи становить 6 грн. Середній щоденний час роботи на комп’ютері – 2 години. Розрахунок витрат на використання комп’ютерної техніки приведений в таблиці 4.4.

Накладні витрати включають три групи видатків: витрати на управління, загальногосподарські витрати, невиробничі витрати. Вони розраховуються за встановленими відсотками від витрат на оплату праці.

Таблиця 4.4 – Розрахунок витрат на використання комп'ютерної техніки

№ п/п	Назва етапів робіт, при виконанні яких використовується комп'ютер	Час використання комп'ютера, год.	Витрати на використання комп'ютера, грн.
1	Проведення досліджень та оформлення їх результатів	90	405
2	Оформлення техніко-економічного розділу	10	45
4	Оформлення ДП	24	108
Разом		124	558

Середньостатистичний відсоток накладних витрат приймемо 150% від заробітної плати:

$$H = 1{,}5 \cdot 2346 = 3519 \text{ (грн)}.$$

Інші витрати є витратами, які не враховані в попередніх статтях. Вони становлять 10% від заробітної плати:

$$I_B = 2346 \cdot 0{,}1 = 234{,}6 \text{ (грн)}.$$

Витрати на розробку програмного забезпечення складають:

$$K_1 = B_{ОП} + B_Ф + B_{ПВ} + H + I,$$

$$K_1 = 2346+481{,}36+1606{,}00+3522{,}18+234{,}6+960{,}00=9150{,}14 \text{ (грн)}.$$

Витрати на відлагодження і дослідну експлуатацію програмного продукту визначаємо за формулою:

$$K_2 = S_{м.г.} \cdot t_{від}, \quad (4.4)$$

де $S_{м.г.}$ – вартість однієї машино-години роботи ПК, грн./год;

$t_{від}$ – комп'ютерний час, витрачений на відлагодження і дослідну експлуатацію створеного програмного продукту, год.

Загальна кількість днів роботи на комп'ютері дорівнює 25 днів. Середній щоденний час роботи на комп'ютері – 2 години. Вартість години роботи комп'ютера дорівнює 6 грн. Тому

$$K_2 = 6 \cdot 25 \cdot 2 = 300 \text{ грн.}$$

На основі отриманих даних складаємо кошторис витрат на розробку програмного забезпечення (таблиця 4.5).

Таблиця 4.5 – Кошторис витрат на розробку програмного забезпечення

№ п/п	Найменування витрат	Сума витрат, грн.
1	Витрати на оплату праці	2064
2	Відрахування у спеціальні державні фонди	423,12
3	Витрати на куповані вироби	378,4
4	Накладні витрати	3096
5	Інші витрати	206,4
6	Витрати на відлагодження і дослідну експлуатацію програмного продукту	300
Разом		6467,92

4.2Розрахунок експлуатаційних витрат і ціни споживання

Для оцінки економічної ефективності розробленого програмного продукту слід порівняти його з аналогом, тобто існуючим програмним забезпеченням ідентичного функціонального призначення. Для цього визначимо експлуатаційні витрати на розробку проекту.

Експлуатаційні одноразові витрати по програмному забезпеченню і аналогу включають вартість підготовки даних і вартість роботи комп'ютера (за час дії програми):

$$E_{П} = E_{1П} + E_{2П}, \tag{4.5}$$

де $E_{П}$ – одноразові експлуатаційні витрати на ПЗ (аналог), грн.;

$E_{1П}$ – вартість підготовки даних для експлуатації ПЗ (аналогу), грн.;

$E_{2П}$ – вартість роботи комп'ютера для розробки програмного продукту (аналогу), грн.

Річні експлуатаційні витрати $B_{ЕП}$ визначаються за формулою:

$$B_{ЕП} = E_{П} \cdot N_{П}, \tag{4.6}$$

де $N_{П}$ – періодичність експлуатації ПЗ (аналогу), раз/рік.

Вартість підготовки даних для роботи на комп'ютері визначається за формулою:

$$E_{1П} = \sum_{l=1}^{n} n_i t_i c_i, \tag{4.7}$$

де i – категорії працівників, які приймають участь у підготовці даних (i=1,2,...n);

n_i – кількість працівників i-ої категорії, осіб;

t_i – трудомісткість роботи співробітників i-ої категорії по підготовці даних, год.;

c_i – середньогодинна ставка працівника *i*-ої категорії з врахуванням додаткової заробітної плати, що знаходиться із співвідношення (4.3):

$$c_i = \frac{1400(1+0)}{21 \cdot 8} = 8 \text{ грн/год.}$$

Трудомісткість підготовки даних для даного проектного рішення складає 2 год., відповідно для аналога – 3 год. Результати представлені у таблиці 4.6.

Таблиця 4.6 – Розрахунок витрат на підготовку даних та реалізацію проектного рішення на комп'ютері

Час роботи співробітників, год.	Середньогодинна заробітна плата, грн./год.	Витрати , грн.
Проектне рішення		
2	8	16
Аналог		
3	8	24

Витрати на експлуатацію комп'ютера визначаються за формулою:

$$E_{2П} = t * S_{МГ}, \quad (4.8)$$

де t – витрати машинного часу для реалізації проектного рішення (аналогу), год.;

$S_{МГ}$ – вартість однієї години роботи комп'ютера, грн./год.

Отже,

$$E_{2П} = 2 \cdot 2{,}5 = 5 \text{ грн.}, \; E_{2П_a} = 3 \cdot 2{,}5 = 7{,}5 \text{ грн.};$$

$$E_{П} = 16 + 5 = 21 \text{ грн.}, \; E_{П_a} = 24 + 7{,}5 = 31{,}5 \text{ грн.};$$

$$B_{ЕП} = 21 \cdot 252 = 5292 \text{ грн.}, \; B_{ЕП_a} = 31{,}5 \cdot 252 = 7938 \text{ грн.}$$

Ціна споживання програмного продукту – це витрати на придбання і експлуатацію програмного засобу за весь період його служби:

$$Ц_{С(П)} = Ц_{П} + B_{(E)NPV}, \tag{4.9}$$

де $Ц_{П}$ – ціна придбання програмного засобу, грн.

$$Ц_{П} = K(1 + \frac{П_{Р}}{100}) + K_{0} + K_{к},$$

де K – кошторисна вартість;

$П_{р}$ – рентабельність;

K_{0} – витрати на прив'язку та освоєння програмного засобу на конкретному об'єкті, грн.;

$K_{к}$ – витрати на доукомплектування технічних засобів на об'єкті, грн.

Зважаючи на вищеописане, розрахуємо ціну програмного засобу:

$$Ц_{П} = 6467,92 \cdot (1 + 0,3) = 8408,30 \text{ грн.}$$

Вартість витрат на експлуатацію проектного продукту (за весь час його експлуатації), в грн. обчислюється так:

$$B_{enpv} = \sum_{t=0}^{T} \frac{B_{ЕП}}{(1+R)^{t}}, \tag{4.10}$$

де $B_{ЕП}$ – річні експлуатаційні витрати, грн.;

T – термін служби програмного засобу, років;

R – відсоткова річна ставка банку.

Розрахуємо витрати на експлуатацію для розробленого програмного продукту та його аналогу:

$$B_{enpv} = \sum_{t=1}^{5} \frac{5292}{(1+0{,}18)^t} = 16549 \text{ грн.},$$

$$B_{enpv} = \sum_{t=1}^{5} \frac{7938}{(1+0{,}18)^t} = 24823 \text{ грн.}$$

Тоді ціна споживання для розробленого програмного продукту та його аналогу становитиме:

$$Ц_{С(П)} = 8408{,}30 + 16549 = 24957{,}30 \text{ грн.},$$

$$Ц_{С(П)_a} = 5480 + 24823 = 30303 \text{ грн.}$$

4.3 Визначення показників економічної ефективності

Для того, щоб побудувати таблицю показників економічної ефективності розробки програмного продукту, проведемо розрахунки необхідних показників. Розрахуємо на початку економічний ефект в сфері проектування рішення за формулою:

$$E_{ПР} = Ц_{П} - Ц_{А}, \quad (4.11)$$

$$E_{ПР} = 8408{,}30 - 5480 = 2928{,}30 \text{ грн.}$$

Річний економічний ефект в сфері експлуатації програмного продукту одержимо із співвідношення:

$$E_{КС} = B_{ЕА} - B_{ЕП}, \quad (4.12)$$

$$E_{КС} = 7938 - 5292 = 2646 \text{ грн.}$$

Додатковий економічний ефект у сфері експлуатації програмного продукту розраховуємо таким чином:

$$\Delta E_{ekc} = \sum_{t=1}^{T} E_{ekc}(1+R)^{T-t} \qquad (4.13)$$

$$\Delta E_{ekc} = \sum_{t=1}^{5} 2646(1+0{,}18)^{5-t} = 18930 \text{ грн.}$$

Зважаючи на проведені розрахунки ефективності розробки програмного забезпечення, обчислимо сумарний ефект від розробки програмного продукту:

$$E = E_{ПР} + \Delta E_{ekc} = 2928{,}30+18930=21858{,}30 \text{ грн.}$$

Результати усіх здійснених розрахунків представлені в таблиці 4.7.

Таблиця 4.7 – Показники економічної ефективності проектного рішення

№	Найменування	Значення показників	
		Аналог	Новий варіант
1	Капітальні вкладення	-	6467,92
2	Ціна придбання	5480	8408,30
3	Річні експлуатаційні витрати	7938	5292
4	Ціна споживання	30303	24957,30
5	Економічний ефект в сфері проектування	-	2928,30
6	Економічний ефект в сфері експлуатації	-	2646
7	Додатковий ефект в сфері експлуатації	-	18930
8	Сумарний ефект	21858,30	

Отже, у цьому розділі проведено розрахунок витрат на розробку програмного забезпечення. Показники, що характеризують витрати на розробку програмного продукту порівняно із показниками, як характеризують програмний продукт із аналогічним функціональним призначенням.

Із результатів порівняння видно, що розроблене програмне забезпечення має суттєві переваги у порівнянні із аналогами, зокрема простота використання, зручність.

Згідно із проведеними розрахунками, що обґрунтовують економічну ефективність, можна зробити висновок, що розроблене програмне забезпечення є конкурентоздатним. Крім того, отримано економічний ефект у розмірі 21858,30 грн., що свідчить про економічну доцільність розробки і впровадження програмного забезпечення мережевого обладнання.

ВИСНОВКИ

В першому розділі було проаналізовано технічні рішення для фіксування пасажирообігу. Обрано ряд систем, які виконують облік пасажиропотоку та надають послуги для зручного користування транспорту. Аналіз предметної області передбачав розбір інформаційних систем для обліку пасажирообігу на окремі складові, що мали слугувати підґрунтям для створення продукту. Обравши три найбільш популярні системи для транспортування пасажирів в містах України, таких як Львів, Тернопіль та Київ — було знайдено ряд недоліків, що викликають дискомфорт для користувачів та жителів цих міст. При проектуванні системи будуть враховані недоліки існуючих рішень, що дасть змогу потенційним користувачам почуття задоволення.

В другому розділі було описано основні елементи системи, що стосуються бази даних та підкреслено ключові засоби захисту при її проектуванні. Чітко визначено засоби захисту інформації та можливі наслідки в разі їх невиконання. Також визначено основні технології та положення, що будуть використовуватись для побудови системи пасажирообігу. Охарактеризовано та ретельно описано основні алгоритми роботи додатку та його взаємодія з системою.

Реалізацію програмного продукту описано в третьому розділі. В даній секції розписано архітектурні рішення проектування, як для мобільного додатку, так і для серверної частини. Мобільний застосунок реалізовано за допомогою сервісно-орієнтованої архітектури з використанням MVP в якості презентаційного шару. Окрім цього рішення було імплементовано ряд інших патернів, які дозволили реалізувати проект start-up рівня. Серед цих патернів приділено увагу реалізації кешуваня зображення, мережевому шару, трішки socket з’єднанню та кординаторам. Велику увагу приділено реалізації користувацькому інтерфейсу та правилам, які диктує компанія Apple для розробки якісного мобільного додатку.

Визначено парадигми та методи тестування. Описано основні поняття розробки на основі тестування. Покрито код, який відповідає за надсилання запиту до сервера. Імплементовано механізм підміни мережевого запиту, що в подальшому може призвести до розвитку даної функції. Результатом розвитку може стати незалежність клієнтської частини від сервера на час розробки другого. У випадку, коли розробка серверної частини буде затягуватись — мобільний додаток зможе майже повноцінно працювати на час надолужування бекендом.

Протестовано бізнес-логіку екрану авторизації. Використано mock-об’єкти для ін’єкції залежностей в презентер екрану авторизації — тим самим відокремивши об’єкт тестування від решти залежностей.

В четвертому розділі розраховується вартість розробки програмного забезпечення. Визначено показники, що характеризують витрати на її розробку, порівнюються з показниками, що характеризують програмні продукти зі схожими функціональними цілями. Результат порівняння показує, що розроблене програмне забезпечення має очевидні переваги порівняно з подібним програмним забезпеченням, особливо простота використання та зручність.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Fundamental concepts of OOP in Swift and how to implement them [Електронний ресурс] — Режим доступу до ресурсу: https://dmytro-anokhin.medium.com/fundamental-concepts-of-oop-in-swift-and-how-to-implement-them-in-c-6581d405f23f

2. Object-oriented programming in Swift [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/macoclock/object-oriented-programming-in-swift-8e0a379b111a

3. OOPs in SWIFT [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/swift-india/oops-in-swift-998738407423

4. Swift Protocol Oriented Programming (POP) [Електронний ресурс] — Режим доступу до ресурсу: https://lawrey.medium.com/swift-protocol-oriented-programming-pop-ceef8e2b7cca

5. How Swift Developers Should Be Using Protocol Oriented Programming [Електронний ресурс] — https://medium.com/swift-development/how-swift-developers-should-be-using-protocol-oriented-programming-982891fe34cc

6. How To Write A Simple Node.js/MongoDB Web Service for an iOS App [Електронний ресурс] — Режим доступу до ресурсу: https://www.raywenderlich.com/2663-how-to-write-a-simple-node-js-mongodb-web-service-for-an-ios-app

7. How to Create a Simple RESTful API in Node.js [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/swlh/how-to-create-a-simple-restful-api-in-node-js-ae4bfddea158

8. Documentation Bookshelf.js — Режим доступу до ресурсу: https://bookshelfjs.org/api.html

9. Develop and Deploy a Scalable RESTful API using Node.js & Mongo. [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/swlh/how-to-create-a-simple-restful-api-in-node-js-ae4bfddea158

10. Understanding Socket Connections in Computer Networking [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/swlh/understanding-socket-connections-in-computer-networking-bac304812b5c

11. What are Web Sockets? [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/@td0m/what-are-web-sockets-what-about-rest-apis-b9c15fd72aac

12. Integrate Socket.IO with Node.Js + Express [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/@raj_36650/integrate-socket-io-with-node-js-express-2292ca13d891

13. A SQL query builder that is flexible, portable, and fun to use! [Електронний ресурс] — Режим доступу до ресурсу: Режим доступу до ресурсу: http://knexjs.org/

14. A brief introduction to the MySQL database and the query language, SQL. [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/@ashiqgiga07/working-with-mysql-dae8f149aa57

15. Session vs Token Based Authentication [Електронний ресурс] — Режим доступу до ресурсу: https://sherryhsu.medium.com/session-vs-token-based-authentication-11a6c5ac45e4

16. Using JSON Web Tokens for User Authentication in Your Web Application [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/geekculture/using-json-web-tokens-for-user-authentication-in-your-web-application-855addb00858

17. Learn using JWT with Passport authentication [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/front-end-weekly/learn-using-jwt-with-passport-authentication-9761539c4314

18. Documentation passport-jwt [Електронний ресурс] — Режим доступу до ресурсу: https://www.npmjs.com/package/passport-jwt

19. SHA256 Algorithm [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/@shahharsh961/sha-265-algorithm-f49184e782d2

20. Hashing in Action: Understanding bcrypt [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/@auth0/hashing-in-action-understanding-bcrypt-7893c2924e52

21. Cryptography for Absolute Beginners [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/@hashelse/cryptography-for-absolute-beginners-3e274f9d6d66

22. iOS Architecture Patterns [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52

23. Design Patterns on iOS using Swift – Part 1 [Електронний ресурс] — Режим доступу до ресурсу: https://www.raywenderlich.com/477-design-patterns-on-ios-using-swift-part-1-2

24. Design Patterns on iOS using Swift – Part 2 [Електронний ресурс] — Режим доступу до ресурсу: https://www.raywenderlich.com/476-design-patterns-on-ios-using-swift-part-2-2

25. Swift World: Design Patterns — Bridge [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/swiftworld/swift-world-design-patterns-bridge-a20bbe999059

26. Refactoring Guru. Design Patterns [Електронний ресурс] — Режим доступу до ресурсу: https://refactoring.guru/design-patterns

27. 8 Patterns to Help You Destroy Massive View Controller [Електронний ресурс] — Режим доступу до ресурсу: https://khanlou.com/2014/09/8-patterns-to-help-you-destroy-massive-view-controller/

28. Ray Wanderlich. Coordinator Pattern [Електронний ресурс] — Режим доступу до ресурсу: https://www.raywenderlich.com/books/design-patterns-by-tutorials/v3.0/chapters/23-coordinator-pattern

29. Back Buttons and Coordinators [Електронний ресурс] — Режим доступу до ресурсу: https://khanlou.com/2017/05/back-buttons-and-coordinators/

30. The Coordiantor [Електронний ресурс] — Режим доступу до ресурсу: https://khanlou.com/2017/05/back-buttons-and-coordinators/

31. Model Mutation In Coordinators [Електронний ресурс] — Режим доступу до ресурсу: https://khanlou.com/2017/05/model-mutation-in-coordinators/

32. Migrating To Coordinators [Електронний ресурс] — Режим доступу до ресурсу: https://khanlou.com/2017/04/migrating-to-coordinators/

33. Refactoring iOS app with Coordinator Pattern for Navigation [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/swift2go/refactoring-ios-app-with-coordinator-pattern-for-navigation-alfian-losari-50081bfa7a4a

34. iOS Swift: MVP Architecture [Електронний ресурс] — Режим доступу до ресурсу: https://saad-eloulladi.medium.com/ios-swift-mvp-architecture-pattern-a2b0c2d310a3

35. Clean Architecture for MassiveToBe Mobile Apps [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/swift2go/clean-architecture-for-massivetobe-mobile-apps-bf8e44a98b37

36. Архитектурные паттерны в iOS [Електронний ресурс] — Режим доступу до ресурсу: https://habr.com/ru/company/badoo/blog/281162/

37. The perfect iOS app architecture [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/flawless-app-stories/the-perfect-ios-app-architecture-4e3d1ab96fa4

38. Passing data to another controller using MVP pattern iOS [Електронний ресурс] — Режим доступу до ресурсу: https://stackoverflow.com/questions/54444627/passing-data-to-another-controller-using-mvp-pattern-ios

39. Sockets+ MVVM in Swift [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/mindful-engineering/sockets-mvvm-in-swift-8f32b1401aa5

40. Real time client-server communication with Socket.IO [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/cocoaacademymag/real-time-client-server-communication-with-socket-io-4311a79b0553

41. How to create a QR code [Електронний ресурс] — Режим доступу до ресурсу: https://www.hackingwithswift.com/example-code/media/how-to-create-a-qr-code

42. How to scan a QR code [Електронний ресурс] — Режим доступу до ресурсу: https://www.hackingwithswift.com/example-code/media/how-to-scan-a-qr-code

43. iOS Networking with Swift [Електронний ресурс] — Режим доступу до ресурсу: https://vuralkaan.medium.com/ios-networking-with-swift-374e0993d937

44. Writing a Network Layer in Swift: Protocol-Oriented Approach [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/flawless-app-stories/writing-network-layer-in-swift-protocol-oriented-approach-4fa40ef1f908

45. Generic Network Layer in iOS Development with Swift [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/flawless-app-stories/generic-network-layer-in-ios-development-2bffff780832

46. Mocking in Swift [Електронний ресурс] — Режим доступу до ресурсу: https://www.swiftbysundell.com/articles/mocking-in-swift/

47. Test Driven Development Tutorial for iOS: Getting Started [Електронний ресурс] — Режим доступу до ресурсу: https://www.raywenderlich.com/5522-test-driven-development-tutorial-for-ios-getting-started

48. How to Build iOS Apps with Swift, TDD & Clean Architecture [Електронний ресурс] — Режим доступу до ресурсу: https://www.essentialdeveloper.com/professional-ios-engineering-series

49. Применение MVP+TDD в разработке iOS приложений [Електронний ресурс] — Режим доступу до ресурсу: https://habr.com/ru/post/335908/

ДОДАТОК А

Реалізація сутності презентера (англ. Presenter) екрану Feed

```
protocol FeedPresenterProtocol {
    func setup()
    func fetchData()

    func numberOfSections() -> Int
    func numberOfItems(in section: Int) -> Int
    func item(by indexPath: IndexPath) -> HolderItemProtocol?
    func header(by section: Int) -> HolderHeaderProtocol?
    func footer(by section: Int) -> HolderFooterProtocol?
}

final class FeedPresenter: FeedPresenterProtocol {

    // MARK: - Properties

    private let socketService: SocketServiceProtocol
    private let activityParseService: ActivityParseServiceProtocol
    private let networkService: FeedNetworkServiceProtocol
    private let itemsService: FeedItemsServiceProtocol

    private var model: FeedModel
    weak var view: FeedViewProtocol?
    private weak var coordinator: FeedFlow?

    // MARK: - Initializer

    init(
        model: FeedModel,
        socketService: SocketServiceProtocol,
        activityParseService: ActivityParseServiceProtocol,
        networkService: FeedNetworkServiceProtocol,
        itemsService: FeedItemsServiceProtocol,
        coordinator: FeedFlow
    ) {
        self.model = model
        self.socketService = socketService
        self.activityParseService = activityParseService
        self.networkService = networkService
        self.itemsService = itemsService
        self.coordinator = coordinator
    }

    // MARK: - Protocol Methods

    func setup() {
        socketService.subscribe(self, on: [.receivedTransaction,
.sentTransaction])
        fetchData()
    }

    func fetchData() {
        fetchTransactions(completion: { [weak self] response in
            guard let self = self else { return }
```

```
            switch response {
            case let .success(data):
                self.model.transactions = data.transactions
                self.model.profiles = data.users
                self.model.transportSessions = data.transportSession

                self.itemsService.prepareTransactionItems(
                    with: data.transactions
                )

                let reversedTransactions: [Transaction] =
data.transactions
                self.model.activities =
self.activityParseService.parse(from: reversedTransactions, delay: 5)

                let data = FeedItemsData(
                    balance:
self.model.user?.account?.balance.wrappedValue ?? 0.0,
                    activities: self.model.activities,
                    basePaymentMethod: self.model.basePaymentType
                )
                self.itemsService.prepareWidgets(with: data)

                DispatchQueue.main.async {
                    self.view?.reloadContentView()
                }

            case let .failure(error):
                DispatchQueue.main.async {
                    self.coordinator?.startErrorAlert(error:
error.toOneError())
                }
            }
        })

        fetchSystemUser(completion: { [weak self] userResponse in
            guard let self = self else { return }
            switch userResponse {
            case let .success(user):
                self.model.user = user

                let data = FeedItemsData(
                    balance: user.account?.balance.wrappedValue ?? 0.0,
                    activities: self.model.activities,
                    basePaymentMethod: self.model.basePaymentType
                )
                self.itemsService.prepareWidgets(with: data)

                DispatchQueue.main.async {
                    let section =
FeedItemsService.Section.widgets.rawValue
                    self.view?.reloadContentView(sections: [section])
                }

            case let .failure(error):
                DispatchQueue.main.async {
                    self.coordinator?.startErrorAlert(error:
error.toOneError())
```

```
                }
            }
        })
    }

    func fetchTransactions(completion: @escaping
(Result<TransactionsData, Error>) -> ()) {
        let operations: [ChainableOperation] = [
            TokenFetchOperation(tokenStorageService:
TokenStorage.shared),
            TransactionsFetchOperation(transactionNetworkService:
networkService)
        ]

        let operation = CompoundOperation<Any, TransactionsData>()
        operation.configure(operations: operations, completion:
completion)
        operation.start()
    }

    func fetchSystemUser(completion: @escaping (Result<User, Error>) ->
()) {
        let operations: [ChainableOperation] = [
            TokenFetchOperation(tokenStorageService:
TokenStorage.shared),
            SystemUserInfoOperation(systemUserProxy:
SystemUserProxyService.shared, forced: true)
        ]

        let operation = CompoundOperation<Any, User>()
        operation.configure(operations: operations, completion:
completion)
        operation.start()
    }

    // MARK: - Items Controller methods

    func numberOfSections() -> Int {
        itemsService.numberOfSections()
    }

    func numberOfItems(in section: Int) -> Int {
        itemsService.numberOfItems(in: section)
    }

    func item(by indexPath: IndexPath) -> HolderItemProtocol? {
        itemsService.item(by: indexPath)
    }

    func header(by section: Int) -> HolderHeaderProtocol? {
        itemsService.header(by: section)
    }

    func footer(by section: Int) -> HolderFooterProtocol? {
        itemsService.footer(by: section)
    }
}
```

```
extension FeedPresenter: FeedItemsServiceTransactionsDataSource {
    func feedItemsService(
        _ feedItemsService: FeedItemsServiceProtocol,
        entitiyWith transaction: Transaction
    ) -> Entity {
        transaction.receiverEntity
    }

    func feedItemsService(_ feedItemsService: FeedItemsServiceProtocol,
profileWith transaction: Transaction) -> User {
        switch transaction.target {
        case .receiver:
            guard let target = model.profiles.first(where: { $0.accountId
== transaction.receiverId })
            else { fatalError() }
            return target

        case .sender:
            guard let target = model.profiles.first(where: { $0.accountId
== transaction.senderId })
            else { fatalError() }
            return target
        }
    }

    func feedItemsService(
        _ feedItemsService: FeedItemsServiceProtocol,
        vehicleWith transaction: Transaction
    ) -> Vehicle {
        switch transaction.target {
        case .receiver:
            guard let target = model.transportSessions.first(where: {
$0.id == transaction.receiverId })
            else { fatalError() }
            return target.transport
        case .sender:
            guard let target = model.transportSessions.first(where: {
$0.id == transaction.senderId })
            else { fatalError() }
            return target.transport
        }
    }
}

// MARK: - Items Controller delegate

extension FeedPresenter: FeedItemsServiceDelegate {
    func didTapBalanceWidget() {
        coordinator?.startBalanceWidgetConfigurator()
    }

    func didTapTicketsWidget() {}

    func didTapActivityWidget() {
        coordinator?.startActivityWidgetConfigurator()
    }

    func didTapPaymentWidget() {
```

```
        coordinator?.startActivityWidgetConfigurator()
    }

    func didTapPaymentButton(with paymentType: PaymentType) {
        coordinator?.startPayScreen(with: paymentType)
    }

    func didTapTransactions() {
        coordinator?.startTransactions()
    }

    func didTapTransaction(with transaction: Transaction, target:
Transactable) {
        coordinator?.startTransaction(with: transaction, target: target)
    }
}

extension FeedPresenter: SocketServiceSubscriberProtocol {
    func socketService(_ socketService: SocketServiceProtocol, didReceive
data: Data?, with event: SocketEvent) {
        switch event {
        case .receivedTransaction:
            fetchData()

        case .sentTransaction:
            fetchData()

        default: break
        }
    }
}
```

ДОДАТОК Б

Реалізація сервісу Imageproxyservice для кешування зображень

```
protocol ImageProxyServiceProtocol: AnyObject {
    func fetchImage(with path: String, completion: @escaping (Data?) ->
())
}

final class ImageProxyService: ImageProxyServiceProtocol {
    // MARK: - Properties

    let queue: DispatchQueue
    let networkService: NetworkingProtocol
    let fileStorageService: ImageFileStorageServiceProtocol

    // MARK: - Initializer

    init(
        queue: DispatchQueue,
        networkService: NetworkingProtocol,
        fileStorageService: ImageFileStorageServiceProtocol
    ) {
        self.queue = queue
        self.networkService = networkService
        self.fileStorageService = fileStorageService
    }

    convenience init() {
        let queue = DispatchQueue(
            label: "image-proxy-service",
            attributes: .concurrent,
            autoreleaseFrequency: .workItem
        )
        let networkService = Networking.shared
        let fileStorageService = ImageFileStorageService()
        self.init(queue: queue, networkService: networkService,
fileStorageService: fileStorageService)
    }

    // MARK: - Methods

    func fetchImage(with path: String, completion: @escaping (Data?) ->
()) {
        queue.async { [weak fileStorageService, weak networkService, weak
queue] in
            switch fileStorageService?.get(with: path) {
            case let .some(data):
                completion(data)

            case .none:
                networkService?.perform(
                    request: NetworkingRequest.get(route:
"/shared/\(path)"),
                    completion: { [weak fileStorageService, weak queue]
response in
```

```
                            switch response {
                            case let .data(data):
                                queue?.async(qos: .default, flags: .barrier)
{
                                    fileStorageService?.save(data, with:
path)
                                }
                                completion(data)

                            case .error:
                                completion(nil)
                            }
                        }
                    )
                }
            }
        }
}

extension ImageProxyService: Sharable {
    static let shared: ImageProxyServiceProtocol = ImageProxyService()
}
```

ДОДАТОК В

Реалізація MOCK-запиту

```
enum URLProtocolMockError: Error {
    case failedToGetPath, failedToGetData
}

final class URLProtocolMock: URLProtocol {
    override class func canInit(with request: URLRequest) -> Bool {
        true
    }

    override class func canonicalRequest(for request: URLRequest) ->
URLRequest {
        request
    }

    override func startLoading() {
        guard let url = request.url,
              let mockURL = URL(string:
"/NetworkStubs")?.appendingPathComponent(url.path)
        else {
            client?.urlProtocol(self, didFailWithError:
URLProtocolMockError.failedToGetPath)
            return
        }

        let relativePath =
mockURL.pathComponents.dropLast().joined(separator: "/")
        let file = (name: mockURL.lastPathComponent, type: "json")

        guard let bundleURL = Bundle.main.path(
            forResource: file.name,
            ofType: file.type,
            inDirectory: relativePath
        ) else {
            client?.urlProtocol(self, didFailWithError:
URLProtocolMockError.failedToGetPath)
            return
        }

        let fileURL = URL(fileURLWithPath: bundleURL)
        guard let data = try? Data(contentsOf: fileURL) else {
            client?.urlProtocol(self, didFailWithError:
URLProtocolMockError.failedToGetData)
            return
        }

        let response = URLResponse()

        client?.urlProtocol(self, didReceive: response, cacheStoragePolicy:
.notAllowed)
        client?.urlProtocol(self, didLoad: data)
        client?.urlProtocolDidFinishLoading(self)
    }

    override func stopLoading() {}
```

```
}
```

ДОДАТОК Г

Реалізація тест-кейсів для презентера Loginpresenter

```
// LoginPresenterTests.swift
// Copyright © Taras Kotsur. All rights reserved.

@testable import OneTransport
import XCTest

final class LoginPresenterTests: XCTestCase {
    // MARK: - Properties

    var model: LoginModel!
    var networkService: LoginNetworkServiceProtocol!
    var tokenService: TokenStorageMock!
    var coordinator: AuthenticationCoordinatorMock!
    var presenter: LoginPresenterProtocol!

    // MARK: - Setup methods

    override func setUp() {
        super.setUp()

        model = LoginModel(email: "", password: "")

        let configuration = URLSessionConfiguration.default
        configuration.protocolClasses = [URLProtocolMock.self]

        let networking = Networking(sessionConfiguration: configuration,
environment: .mock)
        networkService = LoginNetworkService(networking: networking)
        tokenService = TokenStorageMock()
        coordinator = AuthenticationCoordinatorMock()

        presenter = LoginPresenter(
            model: model,
            networkService: networkService,
            tokenService: tokenService,
            coordinator: coordinator
        )
    }

    override func tearDown() {
        networkService = nil
        tokenService = nil
        coordinator = nil
        presenter = nil
        model = nil

        super.tearDown()
    }

    // MARK: - Test cases

    func testModelDataSaving() throws {
```

```
        let credentials = (email: "some@mail.com", password: "password")
        presenter.set(email: credentials.email)
        presenter.set(password: credentials.password)

        XCTAssert(presenter.email == credentials.email)
        XCTAssert(presenter.password == credentials.password)
    }

    func testTokenServiceOnSavingToken() throws {
        let expectation = XCTestExpectation(description: "Expecting async
operation while doing login action")

        coordinator.didFinishLogin = { [weak tokenService, weak
expectation] token in
            expectation?.fulfill()

            XCTAssert(tokenService?.token?.isEmpty == false, "Expected
not empty token")
            XCTAssert(token.isEmpty == false, "Expected not empty token")
        }

        coordinator.didStartErrorAlert = { [weak expectation] error in
            expectation?.fulfill()

            XCTFail("Expected token instead of error, \(error)")
        }

        tokenService.token = nil

        let credentials = (email: "some@mail.com", password: "password")
        presenter.set(email: credentials.email)
        presenter.set(password: credentials.password)
        presenter.selectLogin()

        wait(for: [expectation], timeout: 5)
    }
}
```