Міністерство освіти і науки України
Чорноморський національний університет імені Петра Могили

Методична серія. Випуск 463

Заснована в 2016 році

І. С. Бурлаченко, В. Ю. Савінов

СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

Методичні рекомендації до виконання
практичних та самостійних робіт

Миколаїв – 2025

УДК 004.45
Б90

Рекомендовано до друку Вченою радою факультету комп'ютерних наук Чорноморського національного університету імені Петра Могили (протокол № 9 від 10 квітня 2025 року).

Рецензенти:
І. М. ЖУРАВСЬКА, д-р техн. наук, проф., завідувачка кафедри комп'ютерної інженерії ЧНУ імені Петра Могили.
О. Ф. ПРИЩЕПОВ, канд. техн. наук, доцент, доцент кафедри автоматизації і комп'ютерно-інтегрованих технологій ЧНУ імені Петра Могили.

Бурлаченко І. С.
Б90 Системне програмне забезпечення : метод. рек. до виконання практичних та самостійних робіт / І. С. Бурлаченко, В. Ю. Савінов. – Миколаїв : Вид-во ЧНУ ім. Петра Могили, 2025. – 108 с. – (Методична серія ; вип. 463).

Методичні рекомендації призначені для студентів спеціальності 123 Комп'ютерна інженерія, що вивчають дисципліну «Системне програмне забезпечення» на факультеті комп'ютерних наук Чорноморського національного університету ім. Петра Могили.

УДК 004.45

ISSN 1811-492X

ЗМІСТ

1. ФОРМУВАННЯ ТЕХНІЧНОГО ЗАВДАННЯ

Специфікація вимог до програмного забезпечення (англ. Software Requirements Specification, SRS) містить перелік вимог, очікувань, дизайну та стандартів для майбутнього проєкту. До них належать бізнес-вимоги високого рівня, що визначають мету проєкту, вимоги та потреби кінцевого користувача, а також технічні можливості продукту. Простіше кажучи, SRS надає детальний опис того, як має працювати програмний продукт і як ваша команда розробників повинна його забезпечувати. SRS складається з 11 пунктів, які будуть розглянуті в цьому розділі.

1. Вступ

1.1 Вступ

Метою цього документа технічного завдання є визначення й опис вимог проєкту, а також роз'яснення функціональних можливостей системи та її обмежень.

1.2 Сфера застосування цього документу

Замовником і користувачем системи є співробітники CEBSNU, включаючи пані Шейлу Руп, а розробниками системи є Shock Force Software Team. Наші обмеження для цього розділу включають кінцевий термін для документа, який має бути 29.

1.3 Огляд

Продукт являє собою базу даних PostgreSQL із можливостями імпорту/експорту з Excel, яка містить інформацію про аукціони, пропозиції та їх учасників, зокрема дохід від аукціону, витрати на аукціоні, суму ставки, збільшення ставки, ідентифікатор ставки, донора предмета, початкову вартість предмета, оплату тип, ім'я учасника та електронна адреса учасника.

1.4 Бізнес-контекст

Що стосується цього проєкту, Агентство з інтелектуальними вадами долини Нью-Рівер є некомерційною організацією, яка підтримує людей, які борються з інтелектуальними вадами, проводячи збори коштів і тихі аукціони для збору допомоги, а також підвищення обізнаності.

2. Загальна характеристика

2.1 Функції продукту

Продукт має спростити введення даних і весь процес оформлення замовлення для користувачів (співробітників), а також заощадити час для учасників торгів.

2.2 Інформація про подібні системи

Продукт розробляється з PostgreSQL, тому існує велика кількість подібних баз даних або систем, і вони використовуються для широкого спектру різних цілей. Можлива перевага нашої системи над більшістю — це функція імпорту/експорту програми Excel.

2.3 Характеристики користувача

Серед користувачів – співробітники CEBSNU, оскільки вони вводять дані про пропозиції та їх учасників. Для цієї системи від користувача вимагається знання базової зручності використання Excel, а також розуміння базового рівня доступу, якому, сподіваюся, допоможе команда програмного забезпечення шляхом навчання.

2.4 Постановка проблеми користувача

Система користувача наразі повільна та неефективна, що стосується процесу оформлення замовлення. Учасники торгів повинні чекати годинами, щоб перевірити товар, який вони виграли. Надто багато людиногодин знадобилося, щоб увійти в багатство зібраної інформації.

2.5 Цілі користувача

Користувач хоче базу даних, яка буде зберігати інформацію про аукціони. Програмне забезпечення повинне забезпечувати швидкість і легкість введення даних. ПЗ також повинно зберігати елементи у JSON та XML-форматах.

2.6 Загальні обмеження

Обмеження включають простий у використанні інтерфейс програми через форми, платформу Windows або, як мінімум, Mac із встановленим PostgreSQL і Excel для Mac. Крім того, він має бути створений у PostgreSQL, Excel або іншій програмі, яку легко освоїти.

3. Функціональні вимоги

3.1 Елементи, надані JSON та XML, зберігаються в базі даних PostgreSQL

3.1.1 Елементи повинні зберігатися на ноутбуці та мати заповнені поля.

3.1.2 Дуже висока критичність

3.1.3 Обмежена доступність мережі/Wi-Fi може становити технічну проблему

3.1.4 Вищезазначений фактор є ризиком, з яким ми зіткнулися. Усуньте це, зменшивши залежність нашої програми від цих речей.

3.1.5 Ця вимога є основою проєкту; всі інші аспекти залежать від цього.

3.2 Елементи мають бути доступні через запити та звіти

3.2.1 Користувачі бази даних повинні мати можливість створювати звіти про дані, які були внесені в базу даних. Вони також повинні мати можливість виконувати запити.

3.2.2 Дуже висока критичність

3.2.3 Ми не передбачаємо жодних технічних проблем, які перешкоджатимуть реалізації цього.

3.2.4 Враховуючи можливості PostgreSQL, ця вимога може бути задоволена.

3.2.5 Ця вимога залежить від вимоги номер один.

3.3 Дані, що зберігаються, повинні мати можливість маніпулювати за допомогою форм

3.3.1 Позиції та інші дані повинні мати можливість додавати та оновлювати за допомогою форм.

3.3.2 Дуже висока критичність.

3.3.3 Ми не передбачаємо жодних технічних ризиків, пов’язаних із цією вимогою.

3.3.4 Єдиним фактором, з яким ми можемо зіткнутися тут, є те, що користувач системи не може використовувати її належним чином. Ми подолаємо це шляхом навчання тих, хто ним користуватиметься.

3.3.5 Ця вимога залежить від першої вимоги.

4. Вимоги до інтерфейсу

4.1 Інтерфейси користувача

4.1.1 GUI

Інтерфейс користувача для цієї програми — це інтерфейс, наданий Microsoft PostgreSQL 2007. PostgreSQL містить форми та звіти, за допомогою яких користувачі можуть запитувати та впорядковувати дані відповідно до своїх потреб. І форми, і звіти мають конструктори, які дозволяють користувачеві вказувати, які поля він хоче використовувати та які обмеження він хоче визначити.

4.1.2 CLI

Немає інтерфейсу командного рядка

4.1.3 API

Немає API для продукту

4.1.4 Діагностика або ПЗУ

Існує розділ усунення несправностей і довідка, наданий Microsoft

4.2 Апаратні інтерфейси

Програма (PostgreSQL) використовує жорсткий диск. Доступом до жорсткого диска та іншого обладнання керує операційна система та PostgreSQL.

4.3 Комунікаційний інтерфейс

Якщо ми вирішимо реалізувати спеціальну мережу для спільної бази даних, операційна система оброблятиме ці з'єднання.

4.4 Програмні інтерфейси

Систему PostgreSQL можна використовувати для імпорту та експорту даних за допомогою Google Spreadsheets. Ця функція вбудована в інтерфейс користувача.

5. Вимоги до виконання

База даних призначена для роботи через PostgreSQL, тому не існує жодних додаткових системних вимог, окрім вимог, необхідних для запуску Docker Container, за винятком незначної кількості місця на жорсткому диску для зберігання бази даних.

Корпорація Майкрософт перераховує вимоги до PostgreSQL у 2023 таким чином:

- процесор 500 МГц або вище;
- 4 Гбайт оперативної пам'яті або більше;
- 2,5 Гбайт вільного місця на жорсткому диску;
- операційна система Windows 12 або новіша;

– Docker (PostgreSQL)

Існує також PostgreSQL Available для Mac OS X, клієнти поки що не заявляли про потребу.

6. Інші нефункціональні атрибути

6.1 Безпека

Система повинна бути розроблена з рівнем безпеки, відповідним конфіденційності інформації, що міститься в базі даних. Необхідно більше взаємодії з клієнтом щодо мінливості інформації. Оскільки немає очевидної інформації з високим рівнем безпеки, такої як інформація кредитної картки, єдині вимоги, які можна реалізувати, це шифрування бази даних та/або захист бази даних паролем за запитом користувача.

6.2 Двійкова сумісність

Ця система буде сумісна з будь-яким комп'ютером, на якому встановлено Docker або новішої версії (ПК або Mac), і вона буде розроблена для кількох комп'ютерів.

6.3 Надійність

Надійність – одна з ключових характеристик системи. Резервне копіювання буде створено регулярно, щоб відновлення з мінімальною втратою даних було можливим у разі непередбачених подій. Система також буде ретельно перевірена всіма членами команди для забезпечення надійності.

6.4 Ремонтопридатність

Систему обслуговує Джон Буліченко з CEBSNU або доручає її іншому співробітнику.

6.5 Портативність

Система має бути розроблена таким чином, щоб вона могла працювати на кількох комп'ютерах із інстальованим ПЗ Docker версії X або пізнішою.

6.6 Розширюваність

Система має бути розроблена та задокументована таким чином, щоб будь-хто, хто має розуміння PostgreSQL, міг розширити систему відповідно до своїх потреб за допомогою основних інструкцій команди.

6.7 Повторне використання

Система повинна бути розроблена таким чином, щоб дозволити регулярне повторне використання бази даних для різноманітних тихих аукціонів, які проводить організація.

6.8 Спорідненість/сумісність програми

Для цієї системи потрібен Docker, оскільки вона працює переважно через PostgreSQL у поєднанні з Node.js.

6.9 Використання ресурсів

Ресурси, використані для створення цієї системи, включають: доктора Льюїса, клієнта (Шейла Руп), комп'ютери в Девіс Холі та Інтернет.

6.10 Справність

Обслуговування системи має бути в змозі виконувати будь-яка особа, яка має базове розуміння PostgreSQL.

7. Операційні сценарії

Сценарій А: Початкові визначення елементів

Користувач повинен ввести інформацію про елементи в базу даних для її початкової побудови та розвитку. Поля будуть заповнені за допомогою форми, яка маніпулюватиме даними.

Сценарій В: Виписка клієнта

Користувач повинен мати можливість вводити інформацію про клієнта, який купує певний товар, і записувати його ставку та іншу інформацію. Вони також візьмуть участь у переможній ставці.

Сценарій С: Обслуговування бази даних

Користувач може захотіти змінити/видалити інформацію після завершення аукціону. У цьому випадку йому потрібно буде мати можливість видалити введені дані.

8. Попередні моделі варіантів використання та діаграми послідовності

У цьому розділі наведено список основних діаграм послідовності та випадків використання, які задовольняють вимогам системи. Мета полягає в тому, щоб забезпечити альтернативний, «структурний» погляд на вимоги, зазначені вище, і те, як вони можуть бути задоволені в системі.

8.1 Модель випадку використання.

8.2 Діаграми послідовності.

9. Оновлений розклад

Оновлена діаграма PERT/GANTT додається в кінці документа.

10. Оновлений бюджет

Оновлений бюджет додається в кінці цього документа.

11. Додатки

11.1 Визначення, акроніми, скорочення

CEBSNU – Агентство з інтелектуальними правами у долині Південного Бугу

Приклад формування вступу до технічного завдання для проєкту «Планування завдання для бізнесу». Одна з найбільших проблем довготривалої розробки програмного забезпечення у великій команді – ефективне керування ресурсами. Для вирішення цієї проблеми існує велика кількість застосунків, кожний з яких має свої особливості та переваги. Більша частина розроблених продуктів спрямована задовольняти потреби великих команд, які працюють над складними проєктами. У зв'язку з цим, використання цих застосунків для особистого користування чи у команді з декількох людей є складним та не зручним. Було вирішено виконати аналіз існуючих продуктів та створити особисту реалізацію інструмента, який дозволить відслідковувати і ефективно керувати ресурсами під час роботи над проєктом.

2. ПРИКЛАДИ ТЕХНІЧНИХ ЗАВДАНЬ

Приклади вступу у технічних завданнях.

Розглянемо програмну систему для планування проєктних завдань для бізнесу. Одна з найбільших проблем довготривалої розробки програмного забезпечення у великій команді – ефективне керування ресурсами. Для вирішення цієї проблеми існує велика кількість застосунків, кожний з яких має свої особливості та переваги. Більша частина розроблених продуктів спрямована задовольняти потреби великих команд, які працюють над складними проєктами. У зв'язку з цим, використання цих застосунків для особистого користування чи у команді з декількох людей є складним та незручним. Було вирішено виконати аналіз існуючих продуктів та створити особисту реалізацію інструмента, який дозволить відслідковувати і ефективно керувати ресурсами під час роботи над проєктом.

Розглянемо багатофункціональну інформаційну систему організації діяльності з можливістю соціальної взаємодії. Сучасний світ розвивається досить швидко. Мало хто помічає, але ми вже живемо у майбутньому. Ще майже вчора люди читали газети, слухали радіо, а зараз кожен з нас має у кишені міні-комп'ютер підключений до всесвітньої мережі з налаштованими «user frendly» програмами, які допомагають оперувати інформацією у два кліки. Так, технології вже пронизують світ та покращують життя людей. Можна із впевненістю сказати, що сучасні люди – це нові люди, спектр можливостей яких великий як ніколи завдяки новим технологіям. За допомогою застосунків кожен із нас зараз може кожну хвилину дізнаватися щось нове, швидко і легко планувати свій час та оптимізувати його, спілкуватися зі своїми друзями або колегами, знаходячись далеко від них.

Але, як було сказано раніше, світ постійно розвивається, технології дуже швидко застарівають, ті, що колись ефективно вирішували проблеми, вже не можуть забезпечити людські потреби або забезпечують не повною мірою. І люди також не стоять на місці, інформаційних потоків стає дедалі більше, інформації стає більше, у людей накопичуються справи або з'являються нові хобі, на які у більшості з нас зовсім не вистачає часу. Люди розвиваються як «online», так і «offline». «Offline» рішення більш різноманітні та індивідуально спрямовані, ніж здається на перший погляд. Вони полягають у самоконтролі, психологічному розвитку, фізичних вправах, навчальних методиках і так далі. Але все це не має значення без «online» компоненти.

На сьогоднішній день існує багато рішень для оптимізування часу для сучасної людини. Їх умовно можна поділити на три категорії. Перші – це ті, які дозволяють спілкуватися та обмінюватись інформацією між

людьми на відстані, так звані месенджери. Вони в свою чергу поділяються на корпоративні та загального призначення. Корпоративні спрямовані на захист даних та обговорення ділових питань у робочому колективі. Месенджери загального призначення спрямовані на використання широким загалом людей для спілкування на різноманітні теми. Другий тип програм – це органайзери, застосунки, які допомагають фіксувати всі задачі й пов'язану з ними інформацію у зручній формі та планувати вирішення цих задач у часі. Вони також бувають корпоративними та загального призначення. Також існує третій тип застосунків, який поєднує в собі перші два – інтегровані середовища для командної роботи, або органайзери з можливістю соціальної взаємодії. Але вони розповсюджені лише у корпоративному секторі. І головною проблемою програм корпоративного сектору на сьогоднішній день залишається їх висока вартість. Більшість функцій в цих програмах спрямовані на бізнес та дорого коштують.

Отже, спираючись на вищесказане, було вирішено за кінцеву мету дослідження реалізувати клієнт-серверну багатофункціональну інформаційну систему організації діяльності з можливістю соціальної взаємодії, яка би реалізувала більшість функціоналу, який реалізують схожі програми в корпоративному секторі, але спрямовані на загальне призначення для пересічних людей. Головною метою якого б було покращення якості спілкування та планування повсякденних задач між людьми. А саме було заплановано реалізацію таких функцій як:

– обмін повідомленнями та інформацією як у месенджерах;
– створення та редагування чатів;
– налаштування зовнішнього вигляду власної сторінки;
– додавання та видалення контактів;
– створення чатів для однієї особи або для групи;
– прив'язка повідомлень до часу;
– підтримка тегування чатів та повідомлень для зручного пошуку;
– реалізація фільтрових директорій для організації чатів за тегами;
– підтримка чек-листів у повідомленнях для зручної роботи з задачами.

Об'єктом дослідження під час аналізу були системи передачі даних та організації робочого процесу у цифровому оточенні. Даний функціонал був обраний орієнтуючись на сучасний ринок тенденцій. Цей органайзер допомагає спланувати та організувати час, роботу, повсякденне життя для самого себе або для групи осіб. Щоб реалізувати даний комплекс було поставлено такі задачі:

– спроєктувати зовнішній вигляд(UI) та взаємодію(UX) клієнтського застосунку за допомогою wireframes;
– спроєктувати архітектуру клієнтського застосунку;

– впровадити патерни проєктування для зручної підтримки коду у майбутньому;
– розробити екосистему серверу;
– спроєктувати модель бази даних;
– впровадити патерни в архітектуру серверної частини проєкту;
– відобразити принципи взаємодії з клієнтом серверної частини за допомогою UML sequence diagrams;
– сконфігурувати Docker для зручного розгортання у кластері;
– описати процес розгортання серверної частини за допомогою Docker Swarm у AWS.

За предмет дослідження для реалізації поставлених задач було обрано сучасні фреймворки для Android застосунків та фреймворки для організації серверної частини на базі кластерного комплексу Nodejs.

Приклади аналізу конкурентів

Розглянемо програмну систему для планування проєктних завдань для бізнесу. Був виконаний аналіз існуючих конкурентних продуктів: *YouTrack* [1].

1. *Youtrack*

Цей продукт від компанії *Jetbrains* дає змогу використовувати свій функціонал командам від 1 до 10 користувачів безкоштовно (рис. 1).

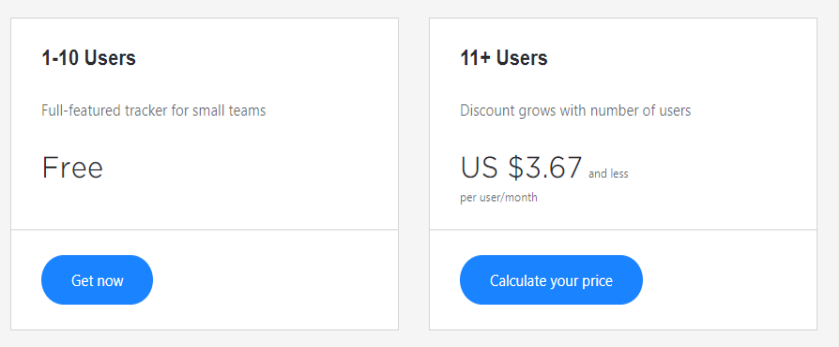


Рисунок 1 – Вартість використання *Youtrack*

У цьому застосунку є можливість створювати завдання та назначати їх виконавцями окремих користувачів, виставляти пріоритети завданням. Для опису завдання використовується *Markdown* синтаксис. На вкладці *Dashboards* у користувача є можливість створювати власні віджети. За їх допомогою можна зберегти вибірку задач чи залишити собі нотатки у форматі *Markdown* (рис. 2).

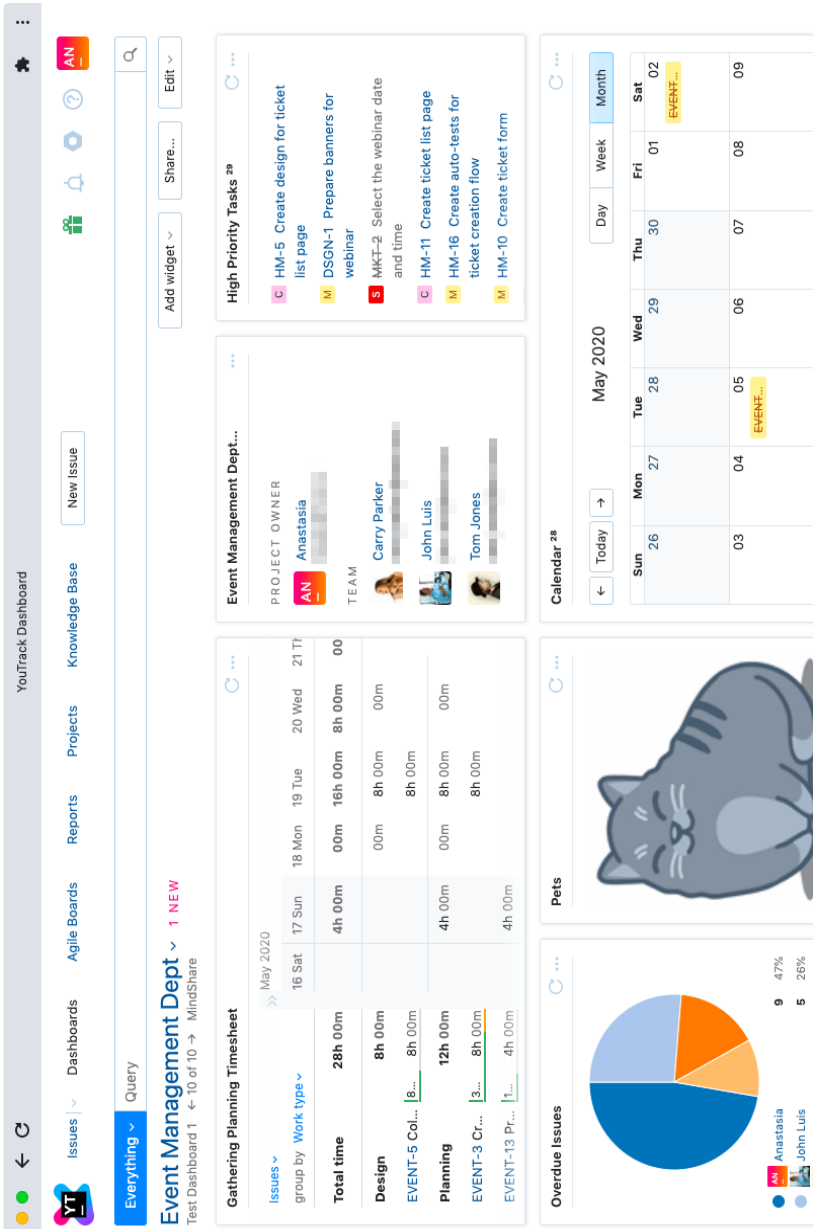


Рисунок 2 – *Youtrack*-сторінка *dashboards*

Застосунок спрямований на використання командою методології *Agile*. Для кожного спринту існує окрема дошка, на якій знаходяться завдання, згруповані за своїм станом (рис. 3).

Одним з недоліків *Youtrack* є відсутність зручного календаря для відслідковування поточних подій.

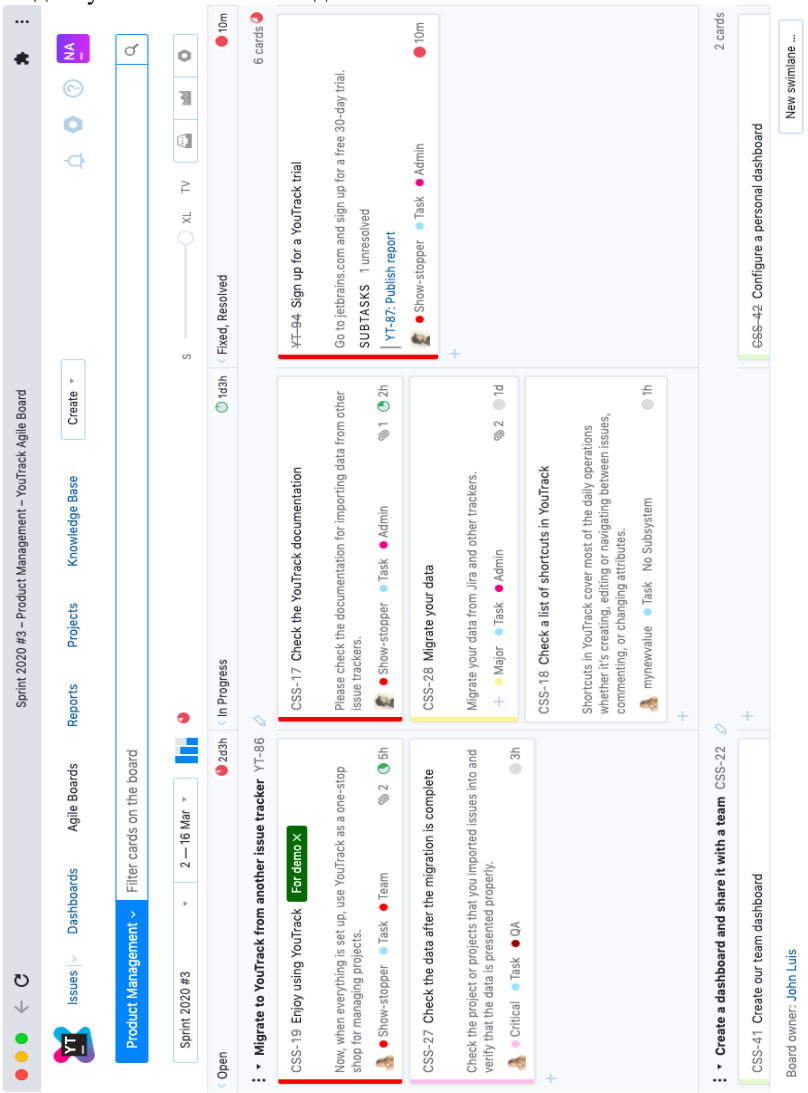


Рисунок 3 – *Youtrack*-сторінка дошок

3. ВИЗНАЧЕННЯ СИСТЕМНИХ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. USE CASE ДІАГРАМИ

Розглянемо програмну систему для планування проєктних завдань для бізнесу. Аналіз існуючих продуктів показав, що застосунки для ефективного керування ресурсами у проєкті мають базові функції (створення задач, їх перегляд, розподіл задач поміж членами команди). Були виділені завдання, які необхідно виконати під час реалізації застосунку планування проєктних завдань.

1. Як адміністратору

Мені необхідна можливість створювати нових користувачів

Таким чином я зможу надавати доступ до застосунку новим учасникам команди

2. Як адміністратору

Мені необхідна можливість редагувати дані користувачів

Таким чином я зможу коригувати дані користувачів за необхідністю

3. Як адміністратору

Мені необхідна можливість надавати ролі для клієнтських користувачів

Таким чином я зможу регулювати операції, які може виконувати користувач

4. Як адміністратору

Мені необхідна можливість переглядати список існуючих користувачів

Таким чином я буду проінформований про користувачів застосунку

5. Як адміністратору

Мені необхідна можливість видаляти профілі користувачів

Таким чином я зможу забирати доступ до застосунку у користувачів

6. Як користувачу

Мені необхідна можливість створювати завдання

Таким чином я зможу документувати завдання, які потрібно виконати

7. Як користувачу

Мені необхідна можливість визначати пріоритет завданню

Таким чином я зможу виділяти більш нагальні і важливі завдання

8. Як користувачу

Мені необхідна можливість визначати виконавця завдання

Таким чином інші члени команди будуть інформовані про поточні завдання

9. Як користувач

Я хочу мати можливість редагувати завдання

Таким чином завдання може бути відкориговане після створення

10. Як користувач

Я хочу мати можливість визначати виконані завдання

Таким чином, я зможу відокремлювати виконані завдання від поточних

11. Як користувачу

Мені необхідна можливість створювати документацію

Таким чином я зможу зберігати задачі та проєктну документацію в одному застосунку

12. Як користувачу

Мені необхідна можливість редагувати документацію

Таким чином я зможу тримати документацію в актуальному стані

13. Як користувачу

Мені необхідна можливість видаляти документацію

Таким чином я зможу видаляти не потрібні статті

14. Як користувачу

Користувачу необхідна можливість пошуку за документами. Таким чином користувач зможе швидко знайти потрібну інформацію.

Сценарії використання

За користувацькими сценаріями, наведеними раніше, була створена діаграма сценаріїв використання (рис. 4).

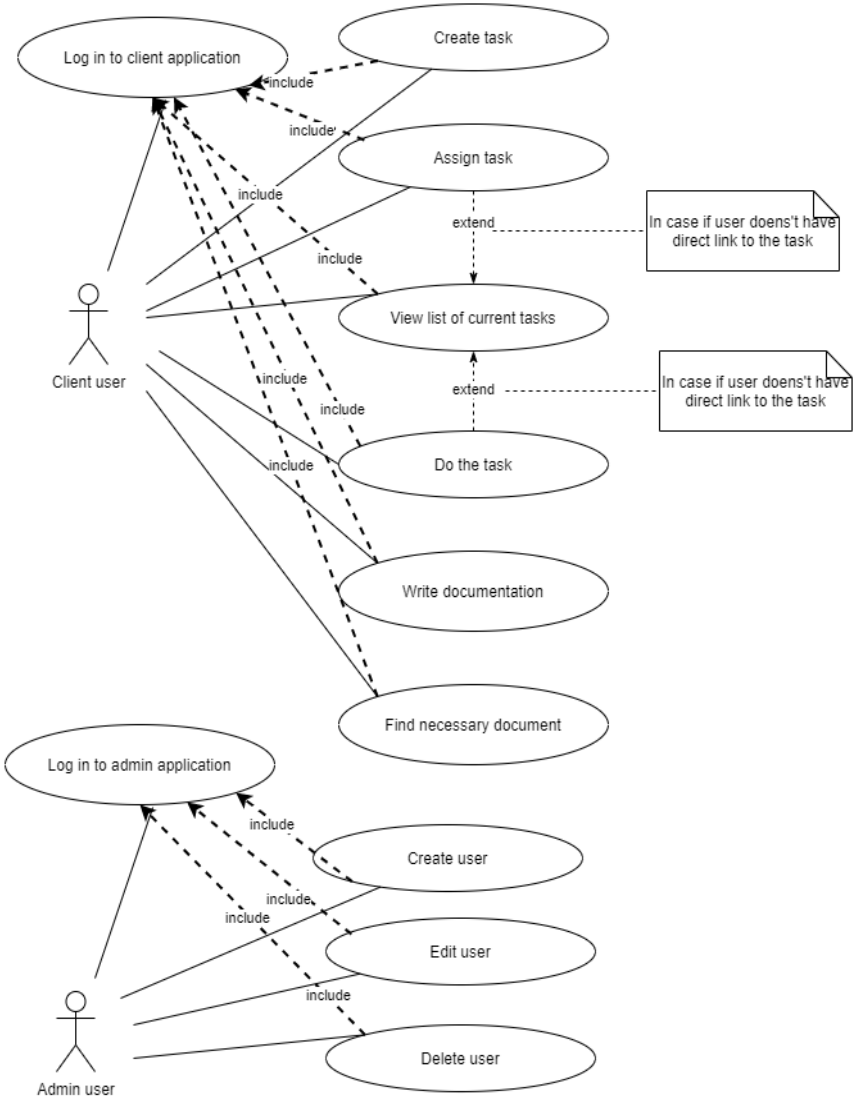


Рисунок 4 – Діаграма сценаріїв використання

Окрім діаграми був опрацьований текстовий опис функції, які зображені на рис. 4.

Варіант використання: створення клієнтського користувача.

Діюча особа: адміністратор.

Ціль: надати доступ новому користувачу

Передумова: адміністратор зайшов у систему

Головна послідовність:

1. Адміністратор натискає кнопку «Додати користувача»
2. Адміністратор заповнює форму створення користувача: записує логін, пароль та ім'я користувача і вказує роль нового користувача.
3. Адміністратор натискає кнопку «Зберегти»
4. Сервер створює нового користувача.
5. Адміністратор перенаправляється на сторінку списку користувачів і бачить нового користувача.

Альтернативна послідовність:

1. Адміністратор натискає кнопку «Додати користувача»
2. Адміністратор заповнює форму створення користувача: записує логін, пароль та ім'я користувача і вказує роль нового користувача.
3. Адміністратор повертається назад
4. Зміни не були збережені

Альтернативна послідовність:

1. Адміністратор натискає кнопку «Додати користувача»
2. Адміністратор заповнює форму створення користувача: записує логін, пароль та ім'я користувача і вказує роль нового користувача.
3. Адміністратор натискає кнопку «Зберегти».
4. Сервер не зміг обробити дані.
5. Адміністратор бачить помилки на формі створення користувача
6. Адміністратор виправляє помилки
7. Адміністратор натискає кнопку «Зберегти»
8. Сервер створює нового користувача.
9. Адміністратор перенаправляється на сторінку списку користувачів і бачить нового користувача.

Альтернативна послідовність:

1. Адміністратор натискає кнопку «Додати користувача»
2. Адміністратор заповнює форму створення користувача: записує логін, пароль та ім'я користувача і вказує роль нового користувача.
3. Адміністратор натискає кнопку «Зберегти».
4. Сервер відповідає помилкою.
5. Адміністратор потрапляє на сторінку з помилкою.

Варіант використання: редагування користувача.

Ціль: змінити дані користувача

Передумова: адміністратор зайшов у систему

Головна послідовність:

1. Адміністратор натискає кнопку «Редагувати» на записі користувача
2. Адміністратор змінює необхідні дані.
3. Адміністратор натискає кнопку «Зберегти»
4. Сервер оброблює зміни та зберігає нові дані користувача.
5. Адміністратор перенаправляється на сторінку списку користувачів і бачить змінені дані.

Альтернативна послідовність:

1. Адміністратор натискає кнопку «Редагувати» на записі користувача
2. Адміністратор змінює необхідні дані.
3. Адміністратор повертається назад.
4. Зміни не були збережені.

Альтернативна послідовність:

1. Адміністратор натискає кнопку «Редагувати» на записі користувача
2. Адміністратор змінює необхідні дані.
3. Адміністратор натискає кнопку «Зберегти».
4. Сервер не зміг обробити дані.
5. Адміністратор бачить помилки на формі редагування.
6. Адміністратор виправляє помилки
7. Адміністратор натискає кнопку «Зберегти»
8. Сервер оброблює зміни та зберігає нові дані користувача.
9. Адміністратор перенаправляється на сторінку списку користувачів і бачить змінені дані.

Альтернативна послідовність:

1. Адміністратор натискає кнопку «Редагувати» на записі користувача
2. Адміністратор змінює необхідні дані.
3. Адміністратор натискає кнопку «Зберегти».
4. Сервер відповідає помилкою.
5. Адміністратор потрапляє на сторінку з помилкою.

Варіант використання: створення завдання.

Ціль: задокументувати нове завдання.

Передумова: користувач зайшов у систему та відкрив сторінку завдань.

Головна послідовність:

1. Користувач натискає кнопку «Додати завдання»

2. Користувач вносить необхідні дані: назву та опис завдання. За бажанням можна встановити пріоритет, тип та виконавця завдання.
3. Користувач натискає «Зберегти».
4. Сервер створює нове завдання.
5. Користувач перенаправляється на сторінку зі завданням.

Альтернативна послідовність:

1. Користувач натискає кнопку «Додати завдання»
2. Користувач вносить необхідні дані: назву та опис завдання. За бажанням можна встановити пріоритет, тип та виконавця завдання.
3. Користувач переходить на іншу сторінку.
4. Дані не були збережені.

Альтернативна послідовність:

1. Користувач натискає кнопку «Додати завдання»
2. Користувач вносить необхідні дані: назву та опис завдання. За бажанням можна встановити пріоритет, тип та виконавця завдання.
3. Форма була заповнена не вірно.
4. Користувач бачить помилки на формі редагування.
5. Користувач виправляє помилкові дані.
6. Користувач натискає «Зберегти».
7. Сервер створює нове завдання.
8. Користувач перенаправляється на сторінку зі завданням.

Альтернативна послідовність:

1. Користувач натискає кнопку «Додати завдання»
2. Користувач вносить необхідні дані: назву та опис завдання. За бажанням можна встановити пріоритет, тип та виконавця завдання.
3. Користувач натискає зберегти.
4. Сервер відповідає помилкою.
5. Користувач бачить повідомлення про помилку.

Варіант використання: редагування завдання.

Ціль: тримати у актуальному стані завдання: його опис, пріоритет та виконавця.

Передумова: користувач зайшов у систему та відкрив сторінку деталей завдання.

Головна послідовність:

1. Користувач натискає кнопку «Редагувати».
2. Користувач вносить необхідні зміни.
3. Користувач натискає «Зберегти».
4. Сервер оброблює зміни та зберігає їх.
5. Користувач бачить змінені деталі завдання.

Альтернативна послідовність:

1. Користувач натискає кнопку «Редагувати».
2. Користувач вносить необхідні зміни.
3. Користувач переходить на іншу сторінку.
4. Дані не були збережені.

Альтернативна послідовність:

1. Користувач натискає кнопку «Редагувати».
2. Користувач вносить необхідні зміни.
3. Форма була заповнена не вірно.
4. Користувач бачить помилки на формі редагування.
5. Користувач виправляє помилкові дані.
6. Користувач натискає «Зберегти».
7. Сервер оброблює зміни та зберігає їх.
8. Користувач бачить змінені деталі завдання.

Альтернативна послідовність:

1. Користувач натискає кнопку «Редагувати».
2. Користувач вносить необхідні зміни.
3. Користувач натискає «Зберегти».
4. Сервер відповідає помилкою.
5. Користувач бачить повідомлення про помилку

Варіант використання: виконати завдання.

Ціль: тримати у актуальному стані завдання: його опис, пріоритет та виконавця.

Передумова: користувач зайшов у систему та відкрив сторінку зі списком завдань

Головна послідовність:

1. Користувач натискає кнопку «Виконати».
2. Сервер оброблює зміну та зберігає її.

Альтернативна послідовність:

1. Користувач натискає кнопку «Виконати».
2. Сервер відповідає помилкою.
3. Користувач бачить повідомлення про помилку.

Варіант використання: створити новий документ.

Ціль: мати проєктну документацію разом з завданнями у одному місці.

Передумова: користувач зайшов у систему та відкрив сторінку зі списком документів

Головна послідовність:

1. Користувач натискає кнопку «Додати документ»
2. Користувач вносить необхідні дані: назву та опис документу.

3. Користувач натискає «Зберегти».
4. Сервер створює новий документ.
5. Користувач перенаправляється на сторінку з деталями документу.

Альтернативна послідовність:
1. Користувач натискає кнопку «Додати документ»
2. Користувач вносить необхідні дані: назву та опис документу.
3. Користувач переходить на іншу сторінку.
4. Дані не були збережені.

Альтернативна послідовність:
1. Користувач натискає кнопку «Додати документ»
2. Користувач вносить необхідні дані: назву та опис документу.
3. Форма була заповнена не вірно.
4. Користувач бачить помилки на формі створення документу.
5. Користувач виправляє помилкові дані.
6. Користувач натискає кнопку «Зберегти».
7. Сервер створює новий документ.
8. Користувач перенаправляється на сторінку з деталями документу.

Альтернативна послідовність:
1. Користувач натискає кнопку «Додати документ»
2. Користувач вносить необхідні дані: назву та опис документу.
3. Користувач натискає кнопку «Зберегти».
4. Сервер відповідає помилкою.
5. Користувач бачить повідомлення про помилку.

Варіант використання: редагування документу.

Ціль: тримати у актуальному стані проєктну документацію.

Передумова: користувач зайшов у систему та відкрив сторінку деталей документу.

Головна послідовність:
1. Користувач натискає кнопку «Редагувати».
2. Користувач вносить необхідні зміни.
3. Користувач натискає кнопку «Зберегти».
4. Сервер оброблює зміни та зберігає їх.
5. Користувач бачить змінені деталі документу.

Альтернативна послідовність:
1. Користувач натискає кнопку «Редагувати».
2. Користувач вносить необхідні зміни.
3. Користувач переходить на іншу сторінку.
4. Дані не були збережені.

Альтернативна послідовність:

1. Користувач натискає кнопку «Редагувати».
2. Користувач вносить необхідні зміни.
3. Форма була заповнена не вірно.
4. Користувач бачить помилки на формі редагування.
5. Користувач виправляє помилкові дані.
6. Користувач натискає кнопку «Зберегти».
7. Сервер обробляє зміни та зберігає їх.
8. Користувач бачить змінені деталі документу.

Альтернативна послідовність:

1. Користувач натискає кнопку «Редагувати».
2. Користувач вносить необхідні зміни.
3. Користувач натискає кнопку «Зберегти».
4. Сервер відповідає помилкою.
5. Користувач бачить повідомлення про помилку

Варіант використання: пошук документів.

Ціль: мати можливість швидко отримати необхідну інформацію за проєктом.

Передумова: користувач зайшов у систему та відкрив сторінку списку документів.

Головна послідовність:

1. Користувач вводить запит у поле пошуку.
2. Сервер обробляє запит та повертає відфільтрований список документів.
3. Користувач отримує необхідний документ.

Альтернативна послідовність:

1. Користувач вводить запит у поле пошуку.
2. Сервер відповідає помилкою.
3. Сторінка документів скидається до початкового виду.

4. ПРОТОТИПУВАННЯ UX ТА UI ЗА ДОПОМОГОЮ WIREFRAMES

Розглянемо багатофункціональну інформаційну систему організації діяльності з можливістю соціальної взаємодії. Даний розділ повністю присвячено проєктуванню та розробці клієнтської частини застосунку на платформі Android. Для клієнтської частини було обрано саме платформу Android (native розробка), тому що більшість сучасних кросплатформних рішень мають обмеження у продуктивності при роботі з великим навантаженням. Мовою програмування було обрано Kotlin, оскільки це сучасна мова програмування, яка поєднує в собі принципи об'єктно-орієнтованої та функціональної парадигм та має сувору типізацію. Весь конвеєр розробки побудований на принципах TDD та таким чином, щоб комфортно реалізувати всі етапи життя програмного продукту, і складається з таких пунктів:

1) прототипування UX та UI за допомогою wireframes;
2) розробка архітектури Android-застосунку;
3) впровадження патернів у архітектуру та рефакторінг;
4) написання тестів до застосунку;
5) написання коду та реалізація архітектури застосунку;
6) перевірка тестів.

У даному розділі буде розглянуто перші 3 пункти, оскільки вони є головними у проєктуванні застосунку, в той час як інші три є не спеціалізованими та представляють собою звичайну реалізацію принципу TDD.

Кожен проєкт мобільного застосунку (в даному випадку Android) починається з прототипування інтерфейсу застосунку та проєктування взаємодії користувача з ним. Для першого прототипу було вирішено використовувати звичайний блокнот та ручку, адже більшість програмного забезпечення так чи інакше сковують творчий підхід та потребують деяких навички роботи з ними.

Спочатку було створено перші 11 основних wireframes, які відображали б головний функціонал. Потім було додано 4 допоміжні wireframes для реалізації меню, спливаючі повідомлення, пошукові видачі, варіації головних wireframes. Також було спроєктовано приблизне знаходження таких функціональних блоків як теги, кнопки взаємодії, фотографії та інше. Після цього було додано користувацький досвід тобто, переходи

між wireframes з мінімізацією користувацької взаємодії задля досягнення мети. Wireframes та переходи були створені з використанням власної нотації. Після розробки базових wireframes було прийнято рішення деталізувати інтерфейс. Прототипування повноцінного UI відбувалося у програмному забезпечені Figma, адже воно є повністю безкоштовним та дозволяє легко і швидко прототипувати інтерфейс відразу у браузері, а також потребує від програміста мінімальних навичок роботи з графікою та лише базові знання роботи з векторною графікою.

Проєктування інтерфейсу відбувалось за принципом від головного до незначного. Спираючись на сучасні тенденції було реалізовано дві теми: спочатку світлу, а потім темну.

Повну картину інтерфейсу можна побачити на рис. 5 та рис. 6.

Рисунок 5 – Детальні wireframes застосунку у програмному забезпеченні Figma (світла тема)

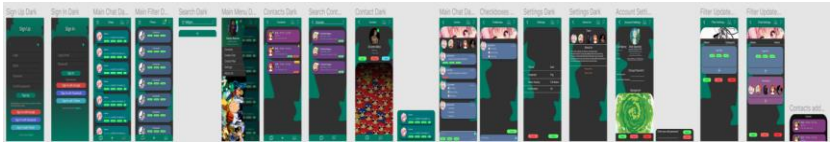


Рисунок 6 – Детальні wireframes застосунку у програмному забезпеченні Figma (темна тема)

Також були спроби спроєктувати інтерфейс у новітньому стилі neomorphism, але від цієї ідеї довелось відмовитись, оскільки цей стиль є дуже «згладженим», що не підпадає під філософію даного застосунку. Порівняння інтерфейсів можна спостерігати на рис. 7.

Рисунок 7 – Дизайн головної сторінки застосунку
(зліва – material design, справа – neomorphism)

Розробку даного прототипу можна розділити на декілька етапів. Спочатку було спроєктовано wireframes, які відповідають за реєстрацію та аутентифікацію, для того щоб визначитися з основним стилем. Приклад wireframes реєстрації та авторизації можна побачити на рис. 8.

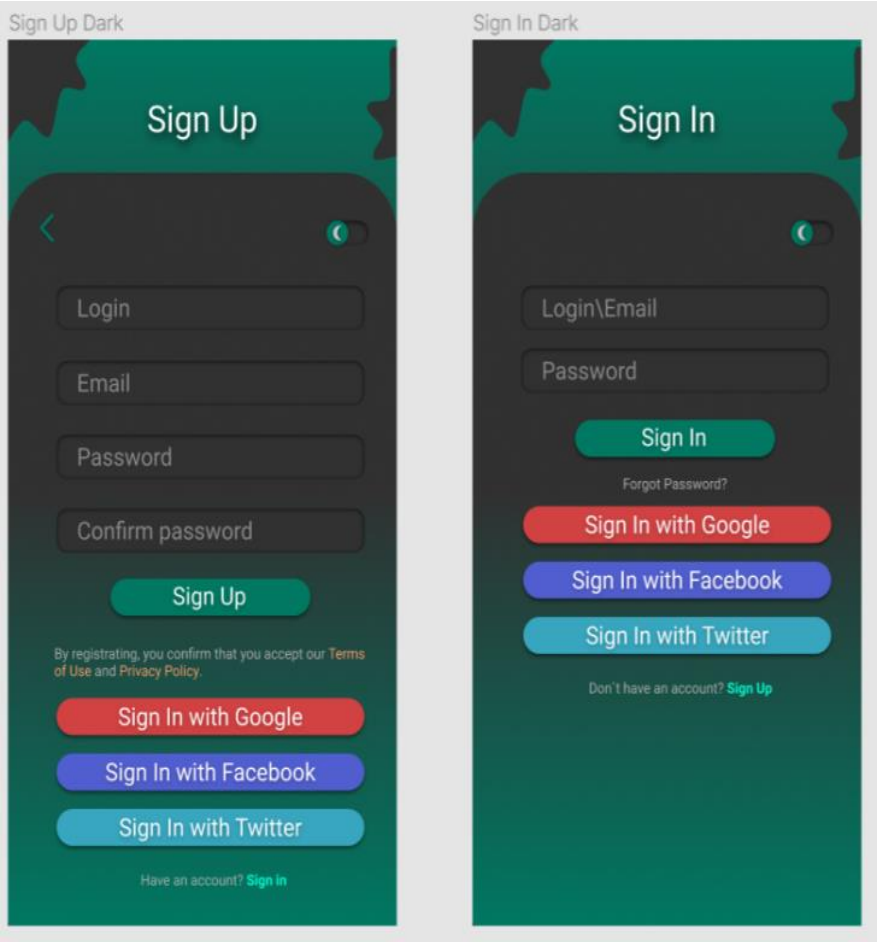
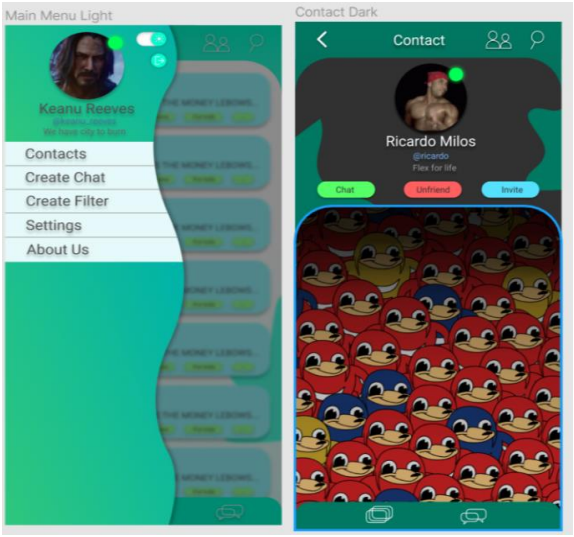


Рисунок 9 – Дизайн сторінок реєстрації та аутентифікації у програмному забезпеченні Figma (темна тема)

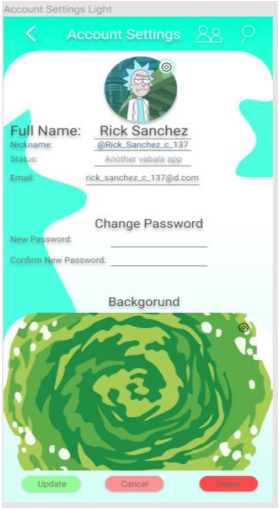
Потім створено вікна по відеозображенню так званих items, тобто вікон чатів, фільтрів та контактів, які можна побачити на рис. 10.

Рисунок 10 – Дизайн Recyclerview сторінок
у програмному забезпеченні Figma

Далі було реалізовано вікна стосовно користувацьких сторінок, таких як редагування власної сторінки та сторінки контактів, які можна спостерігати на рис. 11.



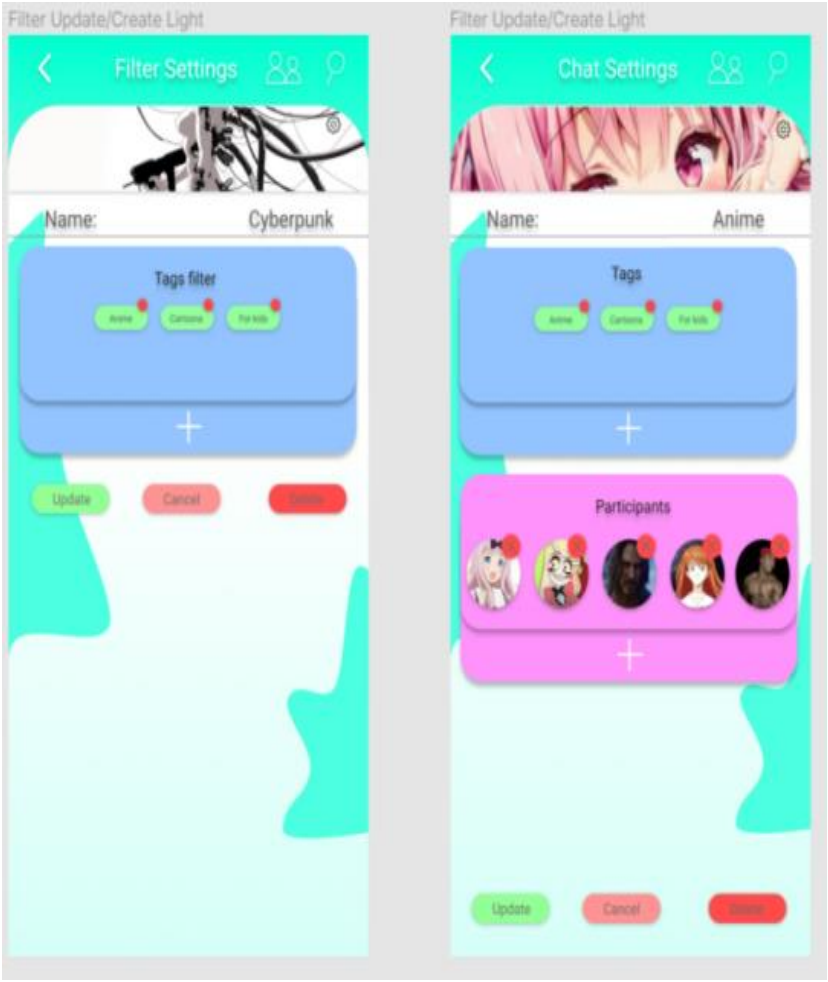
а) б)



в)

Рисунок 12 – Дизайн користувацьких сторінок
у програмному забезпеченні Figma

Наприкінці були реалізовані сторінки чату, сторінки створення та редагування фільтрів та чатів, їх можна побачити на рис. 13.



а)

б)

Рисунок 13 – Дизайн ключових сторінок у програмному забезпеченні Figma: а) сторінки створення та редагування фільтрів та чатів, б) сторінки чату

Після завершення визначення дизайну кольорів та тем, закінчення моделювання всіх сторінок застосунку було спроєктовано логотип проєкту. Було розроблено декілька варіацій логотипу, їх всі, а також останню версію, можна побачити на рис. 14.

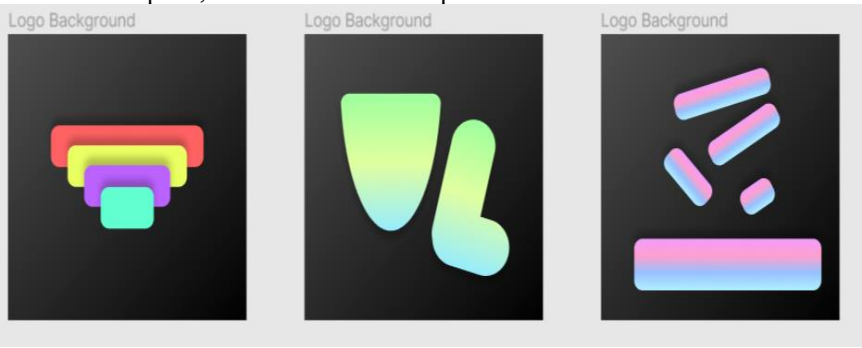
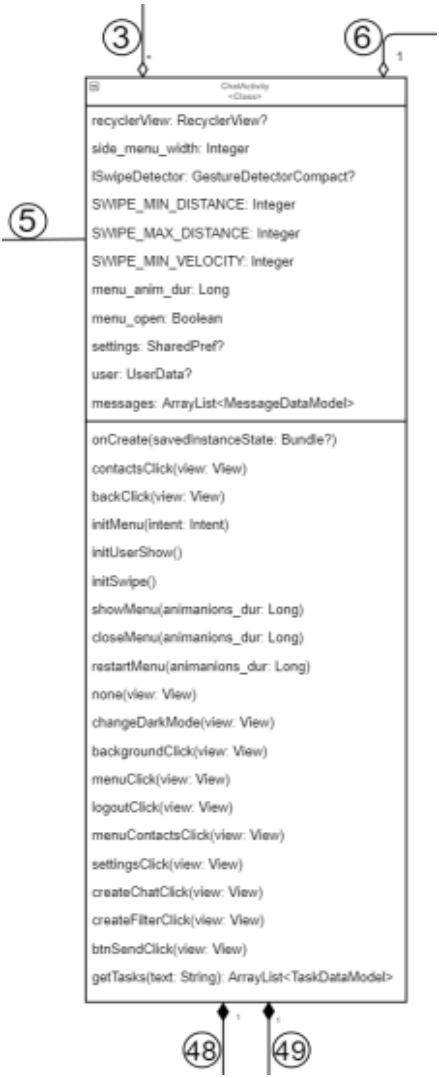
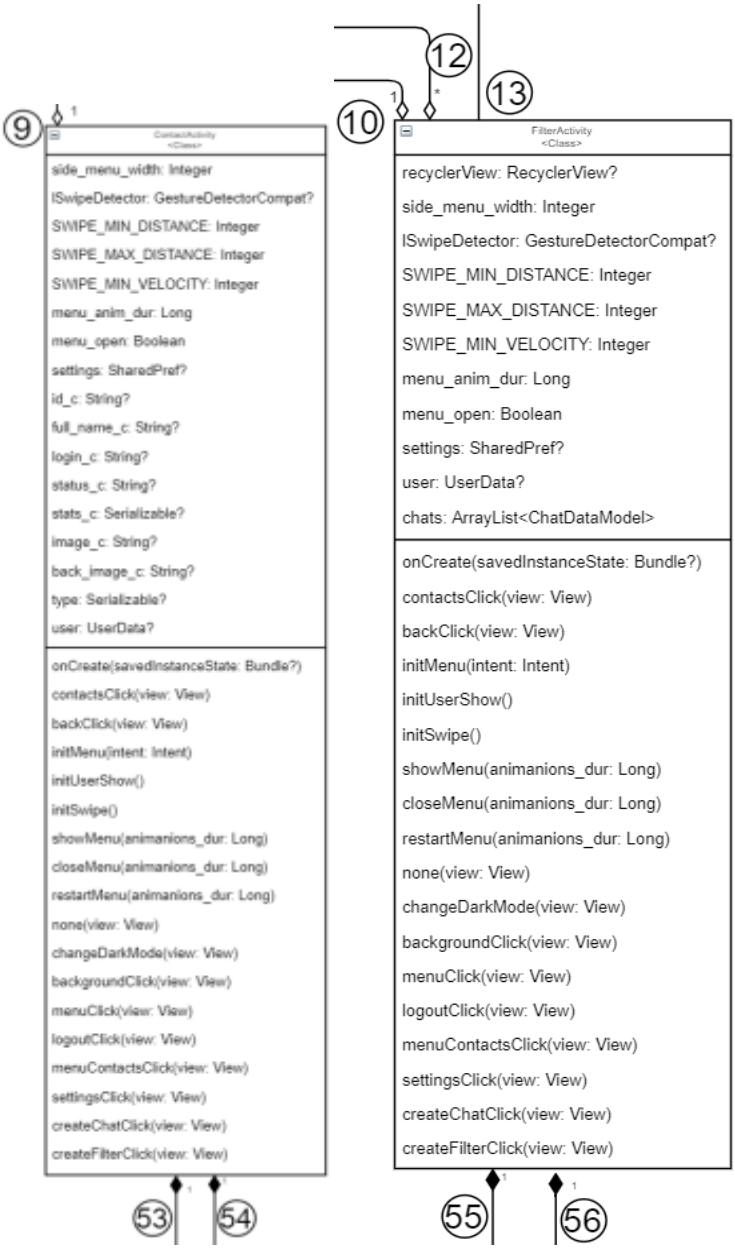


Рисунок 14 – Дизайн логотипів у програмному забезпеченні Figma (у центрі кінцевий варіант)

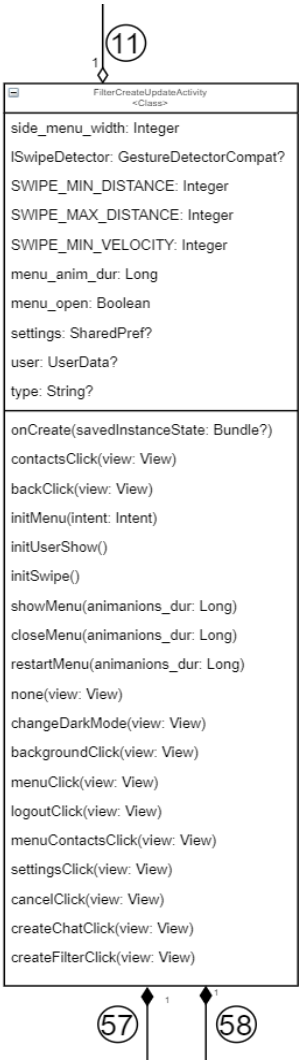
Проєктування архітектури мобільного застосунку є одним із найважливіших етапів розробки програмного продукту. Важливим його робить те, що під час цього процесу йде детальний аналіз предметної області, тобто аналіз потреб користувача, відображення предметної області за допомогою принципів ООП, а також декомпозиція та застосування патернів для підтримки чистоти коду та простоти масштабування у майбутньому. Але при проєктуванні архітектури також враховувалось мінливість ринку та було відкинуто проєктування за принципом «моноліту», тож архітектура була розроблена з можливістю подальшого розширення та змінення у майбутньому. Представлена архітектура, яку можна побачити на рис. 15, була розроблена за допомогою об’єктно-орієнтованої декомпозиції wireframes з попереднього розділу. Головними принципами при побудуванні архітектури були принципи SOLID. Дані принципи дозволили побудувати архітектуру відповідно сучасним тенденціям та вимогам enterprise-проєктів та легко внести патерни в дану архітектуру.


3
6
1
ChatActivity
<Class>
recyclerView: RecyclerView?
side_menu_width: Integer
lSwipeDetector: GestureDetectorCompact?
SWIPE_MIN_DISTANCE: Integer
SWIPE_MAX_DISTANCE: Integer
SWIPE_MIN_VELOCITY: Integer
menu_anim_dur: Long
menu_open: Boolean
settings: SharedPref?
user: UserData?
messages: ArrayList<MessageDataModel>
onCreate(savedInstanceState: Bundle?)
contactsClick(view: View)
backClick(view: View)
initMenu(intent: Intent)
initUserShow()
initSwipe()
showMenu(animanions_dur: Long)
closeMenu(animanions_dur: Long)
restartMenu(animanions_dur: Long)
none(view: View)
changeDarkMode(view: View)
backgroundClick(view: View)
menuClick(view: View)
logoutClick(view: View)
menuContactsClick(view: View)
settingsClick(view: View)
createChatClick(view: View)
createFilterClick(view: View)
btnSendClick(view: View)
getTasks(text: String): ArrayList<TaskDataModel>
5
48
49
7
8
1
ChatCreateUpdateActivity
<Class>
side_menu_width: Integer
SWIPE_MIN_DISTANCE: Integer
SWIPE_MAX_DISTANCE: Integer
SWIPE_MIN_VELOCITY: Integer
menu_anim_dur: Long
menu_open: Boolean
settings: SharedPref?
user: UserData?
type: String?
contactChips: ArrayList<ContactChip>?
onCreate(savedInstanceState: Bundle?)
getColorsFromAttrs(context: Context, @AttrRes vararg attrFields: Int): IntArray
contactsClick(view: View)
backClick(view: View)
initMenu(intent: Intent)
initUserShow()
initSwipe()
showMenu(animanions_dur: Long)
closeMenu(animanions_dur: Long)
restartMenu(animanions_dur: Long)
none(view: View)
changeDarkMode(view: View)
getContactChip(): ArrayList<ContactChip>
backgroundClick(view: View)
menuClick(view: View)
logoutClick(view: View)
menuContactsClick(view: View)
settingsClick(view: View)
cancelClick(view: View)
createChatClick(view: View)
createFilterClick(view: View)
50
51
52

а) б)


9
1
ContactActivity
<Class>
side_menu_width: Integer
ISwipeDetector: GestureDetectorCompat?
SWIPE_MIN_DISTANCE: Integer
SWIPE_MAX_DISTANCE: Integer
SWIPE_MIN_VELOCITY: Integer
menu_anim_dur: Long
menu_open: Boolean
settings: SharedPref?
id_c: String?
full_name_c: String?
login_c: String?
status_c: String?
stats_c: Serializable?
image_c: String?
back_image_c: String?
type: Serializable?
user: UserData?
onCreate(savedInstanceState: Bundle?)
contactsClick(view: View)
backClick(view: View)
initMenu(intent: Intent)
initUserShow()
initSwipe()
showMenu(animanions_dur: Long)
closeMenu(animanions_dur: Long)
restartMenu(animanions_dur: Long)
none(view: View)
changeDarkMode(view: View)
backgroundClick(view: View)
menuClick(view: View)
logoutClick(view: View)
menuContactsClick(view: View)
settingsClick(view: View)
createChatClick(view: View)
createFilterClick(view: View)
53
54
12
13
10
1
*
FilterActivity
<Class>
recyclerView: RecyclerView?
side_menu_width: Integer
ISwipeDetector: GestureDetectorCompat?
SWIPE_MIN_DISTANCE: Integer
SWIPE_MAX_DISTANCE: Integer
SWIPE_MIN_VELOCITY: Integer
menu_anim_dur: Long
menu_open: Boolean
settings: SharedPref?
user: UserData?
chats: ArrayList<ChatDataModel>
onCreate(savedInstanceState: Bundle?)
contactsClick(view: View)
backClick(view: View)
initMenu(intent: Intent)
initUserShow()
initSwipe()
showMenu(animanions_dur: Long)
closeMenu(animanions_dur: Long)
restartMenu(animanions_dur: Long)
none(view: View)
changeDarkMode(view: View)
backgroundClick(view: View)
menuClick(view: View)
logoutClick(view: View)
menuContactsClick(view: View)
settingsClick(view: View)
createChatClick(view: View)
createFilterClick(view: View)
55
56

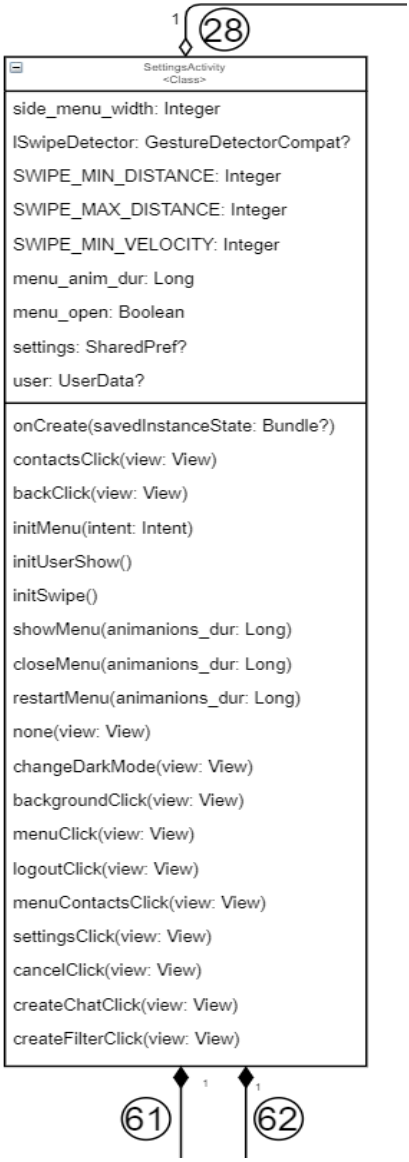
в) г)


11
1
FilterCreateUpdateActivity
<Class>
side_menu_width: Integer
lSwipeDetector: GestureDetectorCompat?
SWIPE_MIN_DISTANCE: Integer
SWIPE_MAX_DISTANCE: Integer
SWIPE_MIN_VELOCITY: Integer
menu_anim_dur: Long
menu_open: Boolean
settings: SharedPref?
user: UserData?
type: String?
onCreate(savedInstanceState: Bundle?)
contactsClick(view: View)
backClick(view: View)
initMenu(intent: Intent)
initUserShow()
initSwipe()
showMenu(animanions_dur: Long)
closeMenu(animanions_dur: Long)
restartMenu(animanions_dur: Long)
none(view: View)
changeDarkMode(view: View)
backgroundClick(view: View)
menuClick(view: View)
logoutClick(view: View)
menuContactsClick(view: View)
settingsClick(view: View)
cancelClick(view: View)
createChatClick(view: View)
createFilterClick(view: View)
57
58

г)


16
18
22
21
23
24
25
17
recyclerView: RecyclerView?
side_menu_width: Integer
lSwipeDetector: GestureDetectorCompat?
SWIPE_MIN_DISTANCE: Integer
SWIPE_MAX_DISTANCE: Integer
SWIPE_MIN_VELOCITY: Integer
chats: ArrayList<ChatDataModel>
chats_counts_notify: Integer
contacts: ArrayList<ContactDataModel>
contacts_counts_notify: Integer
filters: ArrayList<FilterDataModel>
filters_counts_notify: Integer
_mode: String
menu_open: Boolean
menu_anim_dur: Long
change_mode_anim_dur: Long
settings: SharedPref?
user: UserData?
onCreate(savedInstanceState: Bundle?)
initMenu(intent: Intent)
initMode(intent: Intent)
initUserShow()
initSwipe()
addChat()
addFilter()
addContact()
chatsClick(view: View)
addClick(view: View)
filtersClick(view: View)
contactsClick(view: View)
cleanAccepted()
changeModeVisual(mode: String, anim_dur: Long)
changeModeData(mode: String)
changeMode(mode: String, anim_dur: Long)
changeMainText(text: String, anim_dur: Long)
setCountsNotifyContacts(count: Int)
setCountsNotifyChats(count: Int)
setCountsNotifyFilters(count: Int)
showMenu(animanions_dur: Long)
closeMenu(animanions_dur: Long)
restartMenu(animanions_dur: Long)
none(view: View)
menuContactsClick(view: View)
changeDarkMode(view: View)
backgroundClick(view: View)
menuClick(view: View)
logoutClick(view: View)
settingsClick(view: View)
createChatClick(view: View)
createFilterClick(view: View)
59
60

д)



е)

Рисунок 15 – Програмна архітектура Android-застосунку

Проєктування архітектури застосунку відбувалось за таким алгоритмом:

1) обрання мови проєктування та програмного забезпечення для побудови діаграм, які б відображали архітектуру проєкту;

2) об'єктно-орієнтована декомпозиція інтерфейсу Android-застосунку, яка базується на wireframes;

3) об'єктно-орієнтована декомпозиція бізнес-логіки та сутностей даних Android-застосунку, яка базується на wireframes;

4) компонування інтерфейсного набору класів та набору класів бізнес логіки;

5) впровадження патернів проєктування та рефакторніг отриманої архітектури.

У першу чергу перед створенням архітектури було обрано уніфіковану мову проєктування UML редакції 2.5.1. Ця мова вже стала стандартом за минулий час, вона увібрала в себе основні переваги попередніх мов проєктування та витіснила їх. На сьогоднішній день це єдина мова проєктування, яка продовжує свій розвиток, а також підтримує найбільшу кількість діаграм.

Середовищем для побудови діаграм класів було обрано draw.io, адже воно підтримує останні специфікації UML та більшу частину зв'язків та сутностей. Має зручний та зрозумілий інтерфейс та дві реалізації: web та desktop, що дозволяє зручно редагувати діаграми з будь-якого пристрою.

Особливістю проєктування Android-застосунків є побудова архітектури класів відображення так званих Activity. Класи Activity є одними з найважливіших класів у Android-застосунку, що приймають участь у його життєвому циклі. Тому другим пунктом проєктування системи класів було перенесення інтерфейсу застосунку, описаного в wireframes, у Android-реалізацію за допомогою класів Activity та побудову переходів між ними. Головні класи Activity можна побачити на рис. 16.

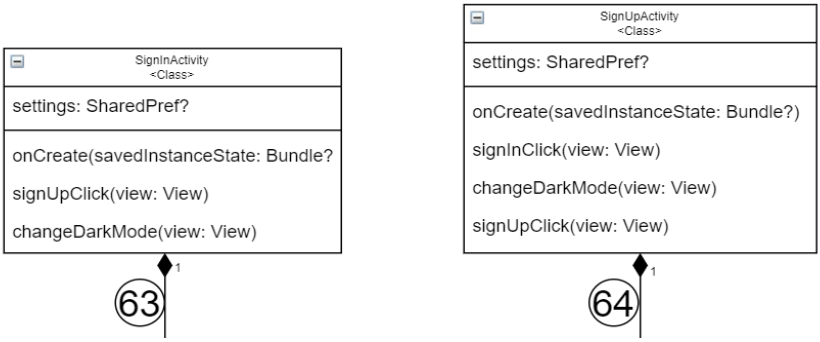


Рисунок 16 – Головні класи Activity

Проміжним етапом було додавання специфічних класів, необхідних для відображення та повного функціонування застосунку: реалізації проміжної логіки, рендерингу, налаштувань, прототипу файлової системи тощо. Їх можна побачити на рис. 17.

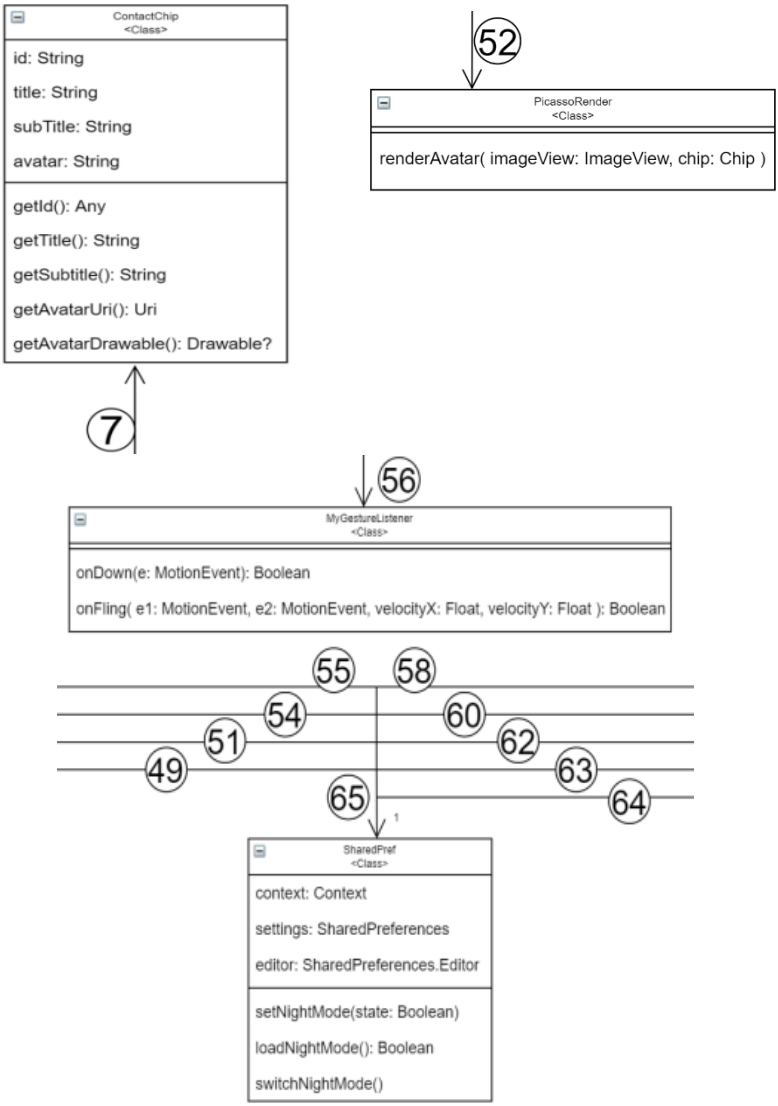
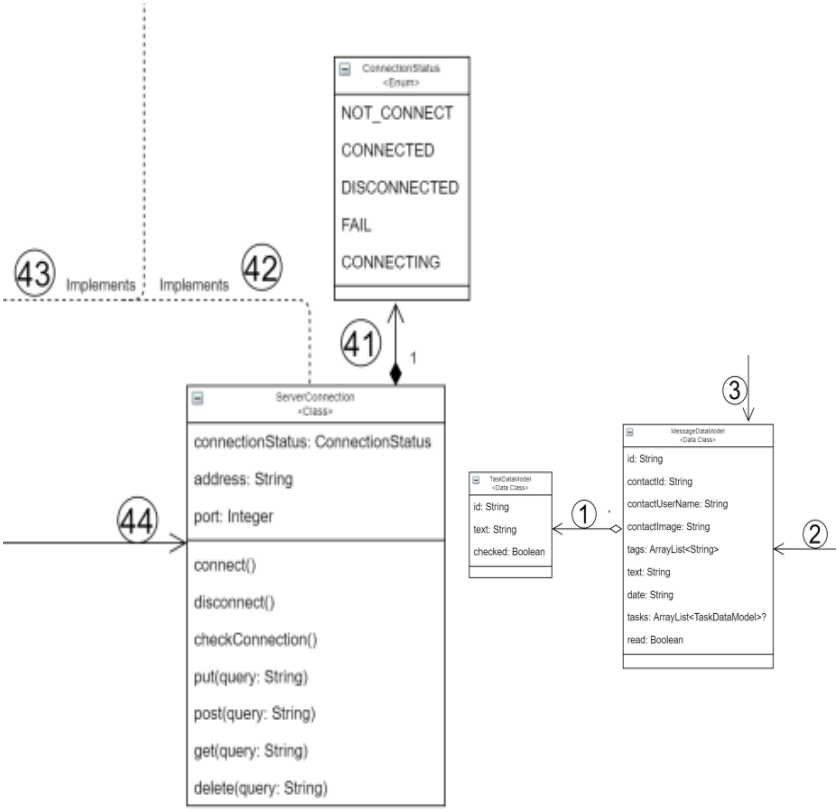
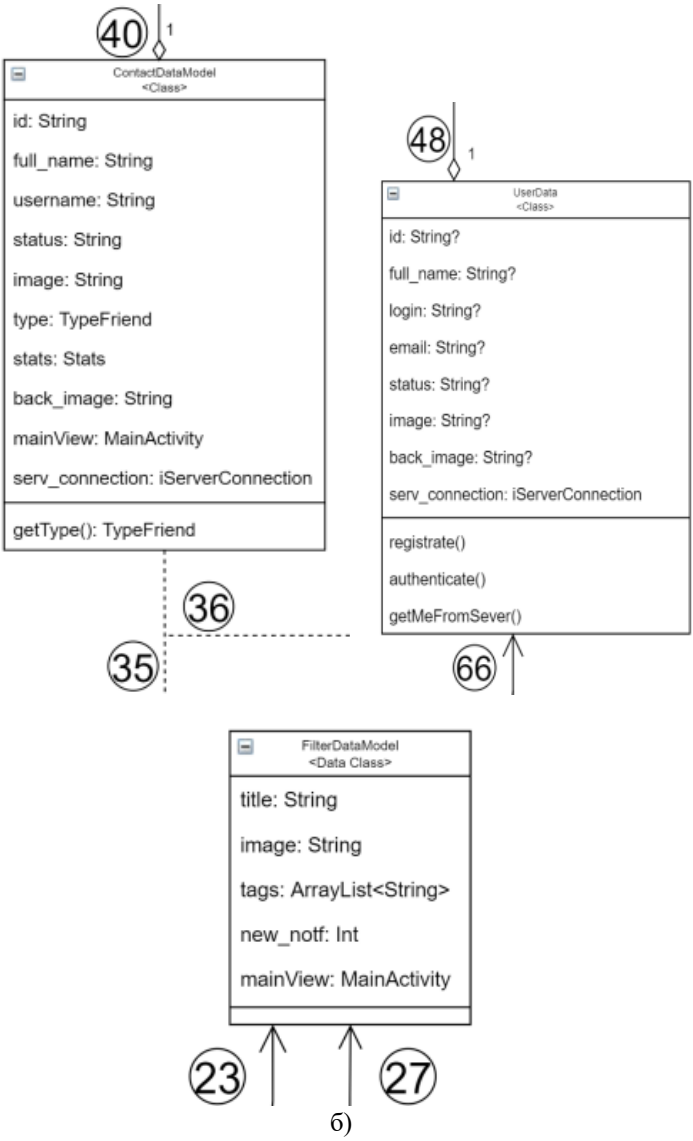


Рисунок 17 – Допоміжні класи

Іншим важливим етапом створення архітектури застосунку було моделювання класів бізнес-логіки та класів сутностей даних. Цей етап здебільшого спільний як для серверної частини, так і для будь-яких реалізацій інтерфейсу: desktop, web, mobile, інше. На даному етапі відбувається основний аналіз предметної області, абстрагування та виділення ключових логічних елементів для повної реалізації логіки роботи застосунку. Класи бізнес-логіки можна побачити на рис. 18.



а)



б)

Рисунок 18 – Класи бізнес-логіки

Передостаннім етапом було компонування бізнес-логіки та класів відображення між собою. Деякі класи, що не передбачали прямого відображення у інтерфейсі, вбудовувались у класи Activity за допомогою звичайної агрегації, інші ж, у свою чергу, тобто ті, які потребували відображення, впроваджувались за допомогою патерна ViewHolder поєднаного з Adapter та відображеного у RecyclerView. Adapter у даному випадку не є реалізацією класичного патерну, це клас, який поєднує клас даних та клас відображення у RecyclerView і є так званим адаптером даних для нього. Приклад реалізації поєднання класів бізнес-логіки та класів відображення можна побачити на рис. 19.

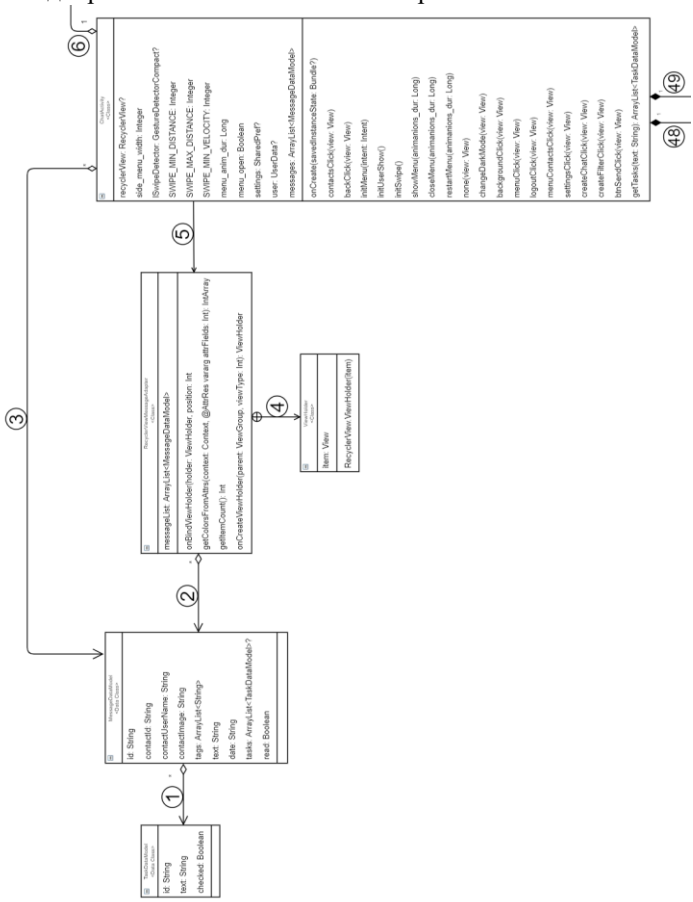


Рисунок 19 – Приклад поєднання класів бізнес логіки та відображення за допомогою RecyclerView та ViewHolder

5. ПАТЕРНИ В ANDROID-ЗАСТОСУНКУ

Розглянемо багатофункціональну інформаційну систему організації діяльності з можливістю соціальної взаємодії. Останнім головним етапом проєктування архітектури будь-якого застосунку є остаточний рефакторінг та впровадження патернів проєктування. Патерни проєктування є головною складовою будь-якого програмного продукту. Патерни – це типові рішення, які допомагають вирішувати деякі проблеми оптимальнішим шляхом, та роблять код програмного продукту більш гнучким для замін у майбутньому. В даному проєкті були застосовані класичні GoF патерни, а також деякі патерни EAA.

Патерни, задіяні при проєктуванні даного застосунку можна поділити на дві частини: невеликі – прості патерни, які або дуже розповсюджені, або не змінюють основну класову структуру проєкту, та архітектурні або складні – ті, які вносять важливі зміни у проєкт. Із невеликих патернів, які використовувались в архітектурі можна виділити Singleton та Lazy Load.

Патерн Singleton використовувався при проєктуванні класу UserDataModel і допомагав вирішувати проблему повторного його використання у класах Activity та у інших класах, які так чи інакше взаємодіяли з цим класом, оскільки даний патерн дозволяє створити лише один екземпляр класу.

Патерн Lazy Load використовувався під час завантаження із серверу даних щодо контактів, чатів та фільтрів. Основна його роль – це знизити навантаження на сервер та на частину відображення. Даний патерн допомагає підвантажувати дані з серверу порціями. Тобто користувач може швидко спостерігати дані, які йому саме зараз потрібні, а якщо знадобиться інша порція, вона буде підвантажуватися, поки користувач спостерігає та працює зі вже підвантаженою, і так далі.

Із архітектурних патернів в проєкті було використано патерн Proxy, для впровадження деяких проміжних дій та зв’язку з патернами Factory Method та Decorator для простого створення комбінацій класів.

Патерн Proxy використовувався під час зв’язування класів даних (UserDataModel та ContactDataModel) та відображення (класи Activity) з класом підключення до серверу (ServerConnection). Оскільки перед підключенням до серверу та відправкою до нього певних запитів, було потрібно виконати певні дії, такі як шифрування та валідація даних та запиту, логування, було створено проміжний клас ServerConnectionProxy,

який слугував посередником та обгорткою для класу ServerConnection та перед підключенням та запитом до серверу виконував вищенаведені дії за рахунок класів Validator, Logger та Encryptor. Впровадження даного патерну дозволило розмежувати обов'язки класу щодо взаємодії з сервером та інших дій по класам, а також додало гнучкості щодо впровадження інших реалізацій проміжних класів та дій. Приклад повної реалізації патерну за допомогою UML-діаграми класів можна спостерігати на рис. 20.

Патерни Factory Method та Decorator використовувалися під час проєктування класу ContactDataModel. Оскільки клас ContactDataModel міг мати декілька реалізацій, які б відображали степінь відношення користувача до контакту та перелік можливих дій щодо взаємодії у предметній області, то було вирішено, задля більшої гнучкості щодо розширення переліку взаємодій у майбутньому та легшої параметризації класу цими можливими функціями, а також розділення базової частини класу від його варіацій, замінити наслідування на патерн Decorator. Далі під час подальшого аналізу архітектури було виявлено проблему, щодо породжування декорованих класів контактів у класах відображення, тому було прийнято рішення реалізувати патерн Factory Method, який би делегував алгоритм створення та декорування класу ContactDataModel відповідно до типів контактів у предметній області, та зручно дозволяв би створювати ці декоровані класи у класах Activity. Наприкінці було отримано три декоратори: ContactNonDecorator, ContactReqDecorator, ContactResDecorator. А також було отримано три фабрики: ContactNonDataModelFactory, ContactReqDataModelFactory, ContactRes DataModelFactory. Дані класи відображають відношення «не друзі», «запит у друзі», «прохання про дружбу» у предметній області. Реалізацію цих наведених вище патернів зображено на рис. 21.

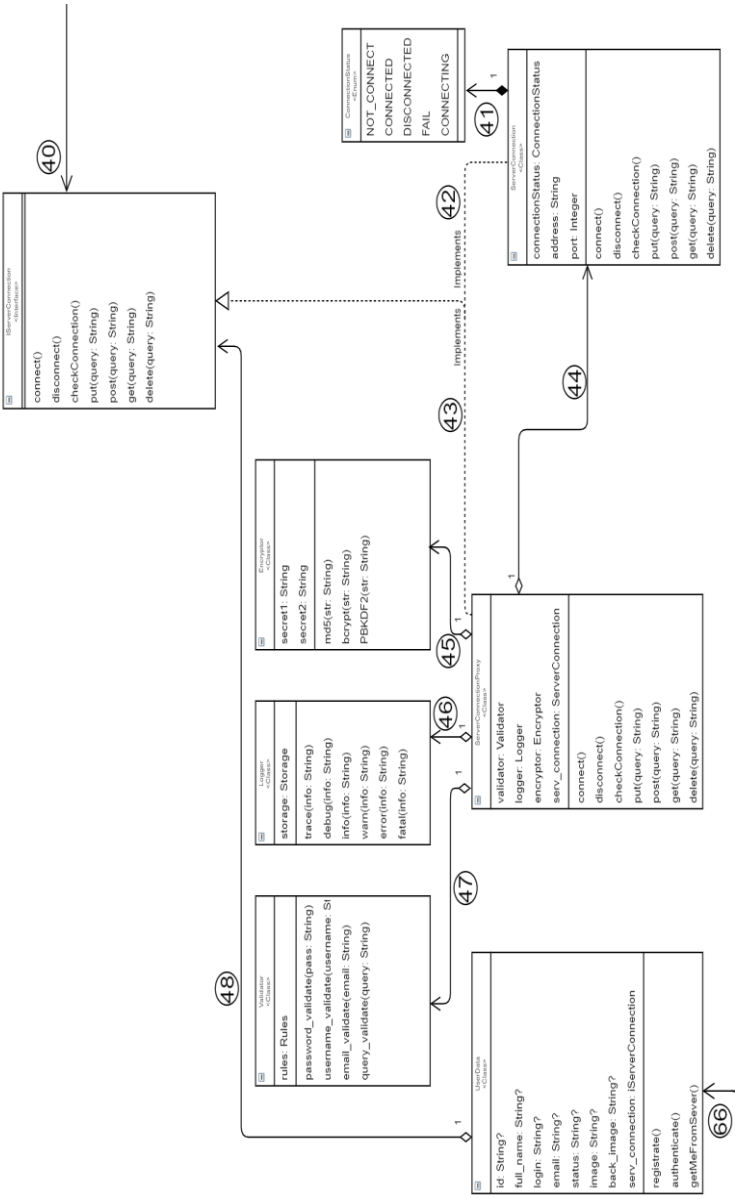


Рисунок 20 – Приклад реалізації патерну Proxy

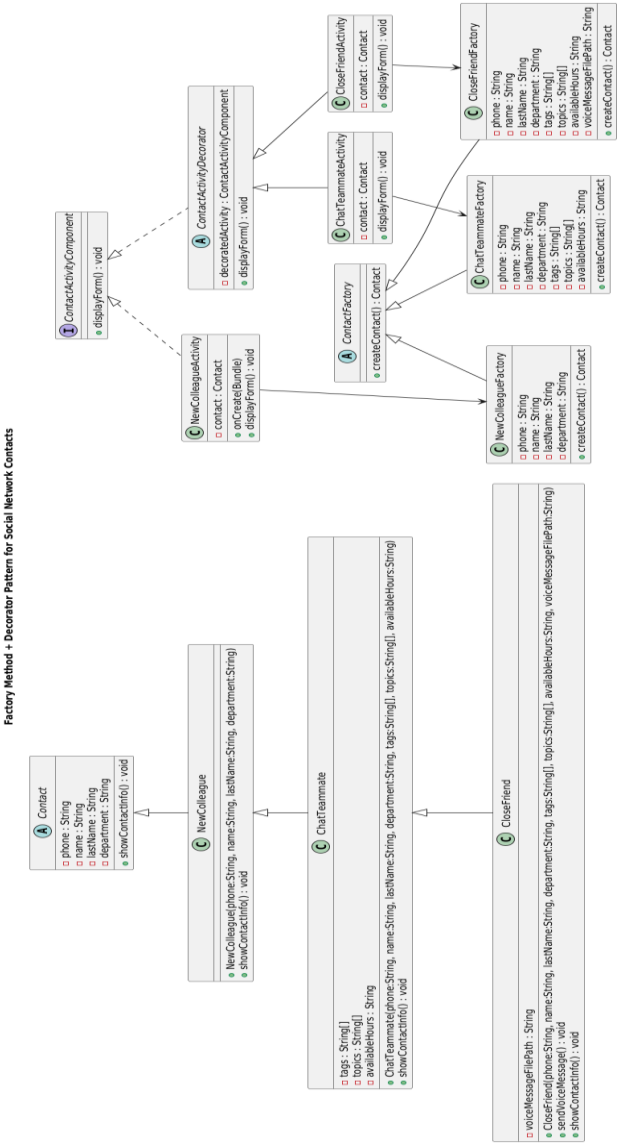


Рисунок 21 – Приклад реалізації патерна Factory Method та Decorator

6. РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ ПРОЄКТУ

Розглянемо багатофункціональну інформаційну систему організації діяльності з можливістю соціальної взаємодії. В цьому розділі буде розглянуто проєктування серверної частини проєкту, основних його компонентів, таких як сервіси, бази даних, балансери та інше. Сервер є головною частиною проєкту мобільного застосунку, оскільки він делегує та реалізує майже всю бізнес-логіку, в той час як клієнтська частина відповідає здебільшого за відображення та з'єднання із сервером, а також інколи кешування даних для offline перегляду. Для написання серверної частини проєкту було обрано платформу Node.js, оскільки ця платформа має модульну структуру та підтримує мікросервісну архітектуру. За мову програмування було обрано javascript, оскільки ця мова поєднує функціональну та об'єктно-орієнтовану парадигми, має великий користувацький репозиторій бібліотек npm та є основною мовою написання для платформи Node.js. Для балансування навантаження було обрано сервер Nginx, оскільки даний сервер підтримує більшу кількість алгоритмів балансування, а також має функції кешування та реверсивного проксі. Для кешування сесій та маршрутів підключень було обрано key-value базу даних Redis, оскільки на даний момент вона має більше можливостей порівняно з аналогами представленими на ринку, такими як Memcached.

Під час проєктування архітектури серверу було виконано такий алгоритм дій:

1) побудова технічного завдання та вимог до серверної частини проєкту;
2) вибір технологій;
3) створення схеми бази даних;
4) аналіз взаємодії клієнту з серверною частиною;
5) створення архітектури проєкту;
6) написання тестів до серверної частини;
7) написання коду серверної частини;
8) перевірка тестів;
9) конфігурування допоміжних сервісів Nginx, HAProxy, Redis;
10) створення контейнерів сервісів за допомогою Dockerfile у середовищі Docker;
11) конфігурування серверної частини у середовищі docker-compose;
12) розгортання серверної частини на кластері EC2 у AWS за допомогою Docker Swarm.

2.1 Опис серверної частини та перелік виконаних вимог

Серверна частина проєкту побудована на мікросервісах та відповідає заданому переліку вимог:

– масштабування;
– відмовостійкість;
– гнучкість розширення;
– захищеність від проникнень.

Також сервер мав виконувати такі задачі:

– реєструвати, аутентифікувати та авторизувати користувачів;
– дати можливість користувачам завантажувати та вивантажувати файли з серверу;
– дати можливість користувачам створювати, редагувати, видаляти, та отримувати чати, фільтри, повідомлення та інші сутності на сервері;
– підтримувати real-time оповіщення.

2.1.1 Виконання вимоги масштабування

Вимога щодо масштабування була реалізована за допомогою балансерів та кластерів. Для вертикального масштабування для Node.js сервісів у кожному з них було увімкнено cluster-mode, для того щоб кожен екземпляр сервісу міг розпаралелити обрахунки запитів по ядрах комп'ютеру. Для горизонтального масштабування було створено декілька екземплярів для кожного з сервісів Node.js та збалансовано потік запитів за допомогою балансеру Nginx між ними.

Щоб можна було горизонтально масштабувати Redis-сервер було також створено декілька екземплярів Redis. Всі вони були запущені у режимі Redis cluster та збалансовані за допомогою HAProxy. Для балансування було обрано саме HAProxy, адже він вміє працювати з TCP-трафіком. Саме цей трафік використовується при роботі з Redis, оскільки він не працює за звичайним HTTP-протоколом. У Nginx функція балансування TCP-трафіку є платною.

Для горизонтального масштабування бази даних Dgraph було використано у якості балансувальника сервіс dgraph-zero, які балансує трафік між екземплярами dgraph-alpha.

2.1.2 Виконання вимоги відмовостійкості

Кожен сервіс Redis, Node.js та Dgraph запущений у режимі кластеру, це означає, що, якщо виникне помилка у деякому екземплярі, на його заміну завжди стане інший запасний екземпляр і трафік від користувачів буде перенаправлений на нього. Кількість екземплярів для кожного з сервісів залежить від кількості трафіку, направленого на нього. Кожен «впавший» екземпляр перезапускається знову, це прописано у конфігурації.

База даних Dgraph завдяки технології shard rebalancing перебалансовує навіть дані зі «впавшого» екземпляру на «живий». Що до сесій користувача, то вони зберігаються у кластеру Redis за функцією реплікації, що ніби відзеркалює дані та дає більшу відмовостійкість ніж при звичайному кластерному режимі.

У випадку, якщо всі кластери всіх сервісів упадуть, то вони просто перезапустяться, а користувач не помітить змін, оскільки для балансування в усіх кластерах використовуються алгоритми, які запам'ятовують шлях, як, наприклад, алгоритм ip hash, який створює хеш-таблицю відповідності ip-адреси до конкретного екземпляру сервісу. А також усі дані про кеш, шляхи, данні користувача та сесії зберігаються на накопичувачах.

2.1.3 Виконання вимоги гнучкості розширення

Оскільки в основі серверної частини лежить мікросервісна архітектура, а в основі сервісів лежить технологія Node.js, це передбачає розширення як за допомогою сервісів, так і за допомогою модулів. А використання різного роду балансувальників допомагає додавати різні технології: кешування, проксування, балансування, брокерів, пошукових систем тощо.

2.1.4 Виконання вимоги захищеності від проникнень

Усі сервіси проєкту запускаються у контейнерах та у закритих мережах, – тобто, це дозволяє закрити доступ до сервісів, до яких користувач не повинен мати прямого доступу, як, наприклад, база даних. Контейнери були поділені та зв'язані у декілька закритих мереж, тобто навіть у деяких сервісів немає доступу до інших, якщо це було передбачено наперед. У кожного сервісу лише єдина точка входу.

Також сервіси, які відповідають за дані, такі як кеш, сесії, бази даних, шифруються, а закриті API мають по декілька «секретів»\паролів. Для кожного сервісу прописана своя авторизація. Під час завантаження файлів через спеціальний middleware, відбувається перевірка, а під час створення, отримання, редагування чи видалення даних відбувається декілько-етапна валідація даних через middleware.

2.1.5 Виконання вимог щодо обов'язків серверу

Усі вимоги були розподілені по сервісам, які були реалізовані у Node.js. Звернення до більшості сервісів відбувається через RESTful API, реалізованого за допомогою бібліотеки express, окрім серверу оповіщення. За реєстрацію, аутентифікацію та авторизацію відповідає сервіс auth, а в основі механізмів лежать бібліотеки passport.js та crypto-js. За завантаження та вивантаження файлів відповідають сервіси upload та download, реалізовано все за допомогою бібліотеки formidable. Для змінення та отримання даних було створено відповідні сервіси mutate та

get, в яких для підключення до бази даних використовувалась бібліотека-клієнт dgraph-js. А для оповіщення та функції real-time було створено сервіс notify, в основі якого лежить бібліотека socket-io, яка являє собою реалізацію протоколу web-socket.

7. ПРОЄКТУВАННЯ БАЗИ ДАНИХ

При проєктуванні архітектури як базу даних було обрано графову базу даних Dgraph. У даний час існує багато типів баз даних, таких як реляційні, документно-орієнтовані, стовпчико-орієнтовані, об'єктні, key-value, графові, гібридні та інші. Під час проєктування спочатку було відібрано лише реляційні, документно-орієнтовані та графові, тому що інші типи зазвичай не використовуються для зберігання саме постійних даних, або є менш поширеними. Потім, після проєктування сутностей даних за допомогою entity relationship diagram, яку можна побачити на рис. 22, та детального аналізу, було виявлено велику кількість зв'язків між сутностями. З цього було зроблено висновок, що реляційні та документо-орієнтовані типи баз даних не підходили для реалізації схеми даних, оскільки документо-орієнтований тип підходить до мало-зв'язних, а реляційний до середньо-зв'язних даних.

Аналізуючи предметну область та бізнес-логіку було виділено 6 таких сутностей як User, Chat, Filter, Message, Tag, Task, які всі мають деякі атрибути, а також тим чи іншим чином пов'язані між собою. User являє собою відображення користувацьких даних у застосунку, має атрибути паролю, ім'я, повного ім'я, аватару та інших персоналізованих даних. Кожен User може бути учасником Chat, який представляється чатом у предметній області та має атрибути назви та фонової картинки. User також має власні Filters, тобто фільтри, які зберігають відфільтровані за тегами чати. В Chats, можуть зберігатися Messages, тобто повідомлення від користувачів, які в свою чергу можуть містити Tasks, тобто чек-бокси, з певним завданням.

Після обрання саме графових баз даних подальший аналіз ринку привів до трьох фаворитів: ArangoDB, Neo4j та Dgraph. І врешті решт було вирішено використовувати базу даних Dgraph. Основною причиною цього рішення було те, що порівняно з Neo4j Dgraph має більше функцій, які у Neo4j або відсутні взагалі, або є платними. ArangoDB було відкинуто через їх мультипарадигмальну структуру зберігання даних та не дуже зручну мову запитів AQL

Після того, як було спроєктовано схему сутностей та обрано графову базу даних, наступним етапом стало проєктування схеми даних у графовому представлені. Повну схему даних у графовому представлені можна спостерігати на рис. 23.

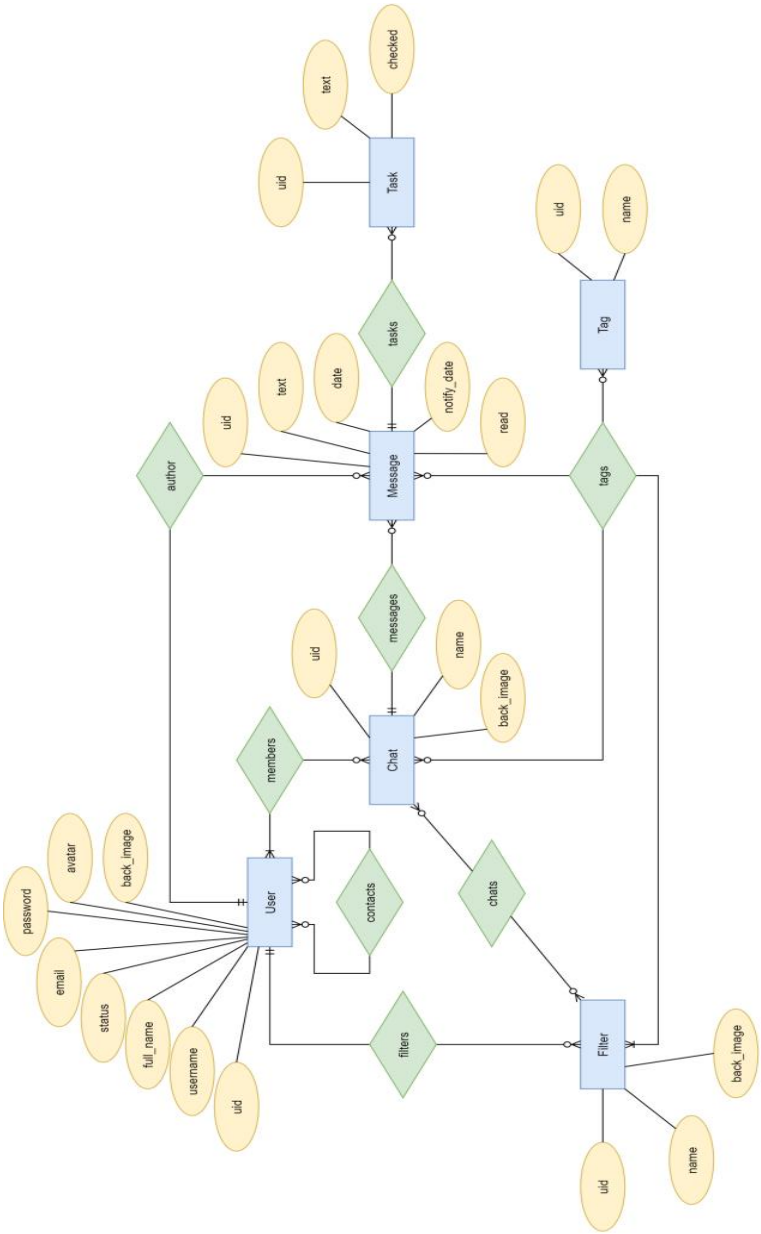


Рисунок 22 – Entity relationship diagram даних застосунку

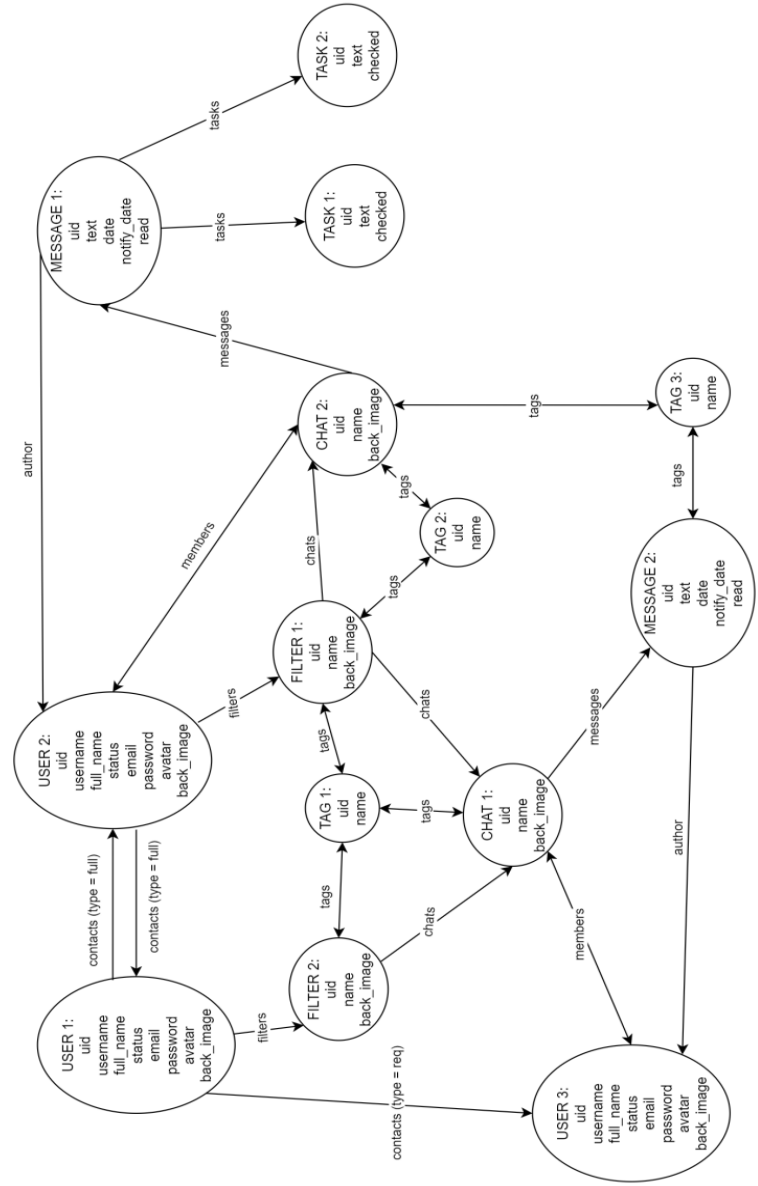


Рисунок 23 – Представлення схеми даних у вигляді графу

Основною метою побудови графового представлення було перетворення схеми даних в схему зберігання та відображення принципу зберігання даних в цілому. Головною проблемою графового представлення бази даних Dgraph є двонаправлений зв'язок: його там у звичайному виді немає. Двонаправлений зв'язок у Dgraph можна реалізувати за допомогою або двох протилежних зв'язків, або одного реверсивного зв'язку. У випадку протилежних при мутації потрібно змінювати дві сутності, а у випадку реверсивного зв'язку – одну, але потрібно використовувати два різних запити по отриманню сутностей в різних напрямках. Тому у випадку, де зв'язок був двонаправлений, але два різних напрямки являли собою дві різні потреби користувача, використовувалося реверсивний зв'язок (tags, members). Якщо зв'язок був двонаправлений як фактично, так і семантично, то використовувалося два протилежних зв'язки (contacts). В усіх інших випадках, де двонапрямленість була зайвою, використовувались однонапрямлені зв'язки. Також потрібно додати, що дані у Dgraph представляються у вигляді триплетів, тобто <uid> <name> <some_name> або <uid> <contact> <uid>, де <uid> – це суб'єкт або сутність; <name>, <contact> – предикат або атрибут, <some_name>, <uid> (у другому випадку) – об'єкт або фактичне значення. Тобто на основі цих триплетів будуються сутності та зв'язки між ними, що було враховано при розробці графового представлення.

Наступний етап – побудова схеми даних та підбір необхідних типів даних по кожному зі зв'язків. Повну схему даних Dgraph можна спостерігати на рис. 24.

```
"<author>: uid .",
"<chats>: [uid] .",
"<checked>: bool .",
"<read>: bool .",
"<contacts>: [uid] @count @reverse .",
"<date>: datetime @index(hour) .",
"<email>: string @index(term) .",
"<filters>: [uid] .",
"<full_name>: string @index(term) .",
"<members>: [uid] @count @reverse .",
"<messages>: [uid] .",
"<name>: string @index(fulltext) .",
"<notify_date>: datetime @index(day) .",
"<password>: password .",
"<status>: string .",
"<tags>: [uid] @count @reverse .",
"<tasks>: [uid] @count .",
"<text>: string @index(fulltext) .",
"<username>: string @index(term) .",
"<avatar>: string .",
"<back_image>: string ."
```

Рисунок 24 – Представлення схеми даних у Dgraph

Дані у Dgraph представляються у такому вигляді: <status>: string ., де <status> – ім'я поля, string – тип даних, . – кінець представлення. Також у випадках, коли даних декілька, як, наприклад, повідомлень, то назва даного поля писалась у множині, та тип даних брався у квадратні дужки, що означало список даних. У деяких випадках зв'язки були прореверсовані за допомогою модифікатора @reverse, а також де потрібно було рахувати точну кількість даних у списку було встановлено модифікатор @count. Для індексації та оптимізації швидкого пошуку використовувався модифікатор @index(), в дужках якого прописувався тип індексації, для кожного типу даних набір допустимих типів унікальний. Для індексації типу даних datetime було використано типи day та hour, що дозволяло, у свою чергу, сортувати дані за днями та годинами. А для індексації типу даних string використовувалися типи fulltext – для повнотекстового пошуку, та term – для пошуку за ключовим терміном. Також слід зазначити, що у Dgraph є підтримка локалізації полів, але у даному випадку вона не використовувалась. Типи даних, які використовувались: string (строковий тип), uid(тип даних сутності, за допомогою якого сутності поєднуються), bool (булевий тип), datetime (тип часу) та password (тип паролю, який шифрується).

8. ПАТЕРНИ СЕРВЕРНОЇ ЧАСТИНИ

Серверна частина будувалася за принципами мікросервісної архітектури, спираючись на принципи DDD (предметно-орієнтоване проєктування). Першим задіяним принципом був принцип «єдиної мови», тобто для більшості сервісів було обрано одну технологію та мову – Node.js та Javascript, а також було виділено деяку специфікацію при написанні коду, як, наприклад, імена змінних. Також всі важливі функції були представленні в моделях, а моделі в сервісах.

Головним архітектурним патерном DDD, який було використано на початку був патерн багаторівневої архітектури. Реалізацію цього патерну у вигляді діаграми можна спостерігати на рис. 25.

Усього в серверній частині 5 рівнів. Перший рівень – рівень представлення – відповідає реалізації Android-застосунку або інших front-end представлень. Рівень балансувальників сервісів є другим і являє собою набір Nginx-сервісів, які виступають у якості балансувальників трафіків для кожного із сервісів. Третій рівень – рівень сервісів, представляє собою набір горизонтально масштабованих по фізичним комп'ютерам сервісів, написаних на Node.js. Четвертим рівнем є рівень балансувальників сервісів даних, який складається з двох HAProxy та одного Dgraph-zero, які у свою чергу балансують трафік по сервісам, які відповідають за збереження та доступ до даних. Останнім п'ятим рівнем є рівень сервісів даних, який складається з горизонтально масштабованих екземплярів баз даних, таких як Dgraph-alpha та Redis. На останньому рівні екземпляри Dgraph-alpha відповідають за збереження даних предметної області, а екземпляри Redis – за збереження сесій та маршрутів з'єднань, тобто стану системи та користувачів. Даний підхід дозволив розмежувати рівні серверної частини, що дозволило налаштовувати рівні незалежно один від одного.

Також під час проєктування архітектури серверної частини та реалізації патерну багаторівневого представлення було впроваджено патерн CQRS (command-query responsibility segregation). Даний патерн дозволяє розділити навантаження та обов'язки щодо запису та зчитування по різним моделям (command, query). Графічне представлення реалізації цього патерну можна спостерігати на рис. 26.

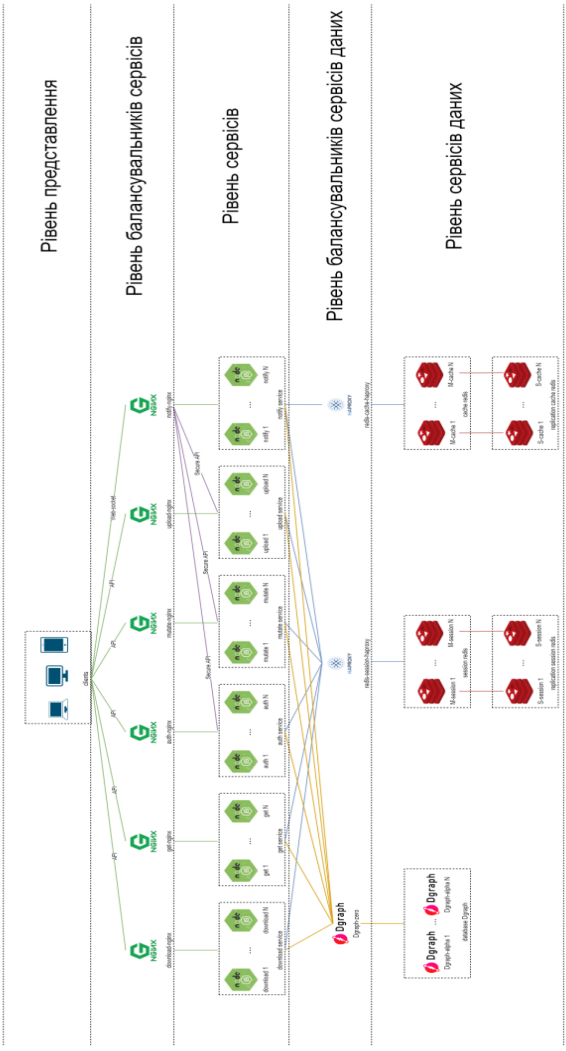


Рисунок 25 – Патерн багаторівневої архітектури
у графічному представленні

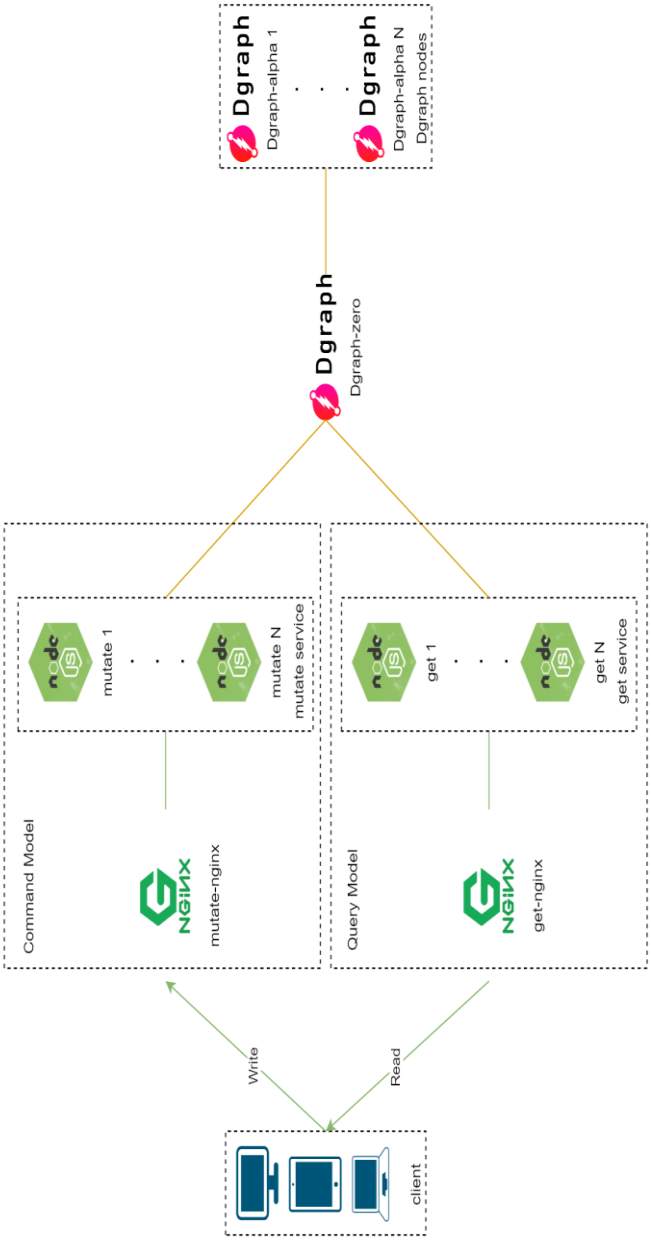


Рисунок 26 – Патерн CQRS у графічному представленні

У даній реалізації за модель Command відповідає сервіс mutate, який обробляє користувацькі запити на зміну даних у базі, а за модель Query відповідає сервіс get, в обов'язки якого входить віддавати дані по запитам. Також цей патерн був впроваджений для сервісів роботи з файлами, тобто для завантаження файлів було створено сервіс upload (Command Model), а для вивантаження download (Query Model). Впровадження цього патерну допомогло розділити навантаження на сервер та спростило розробку. А завдяки горизонтальному масштабуванню можна балансувати кількість сервісів залежно від потреб людей на зчитування чи запис.

Останнім патерном було впроваджено Event Bus. Застосування цього патерну було викликано необхідністю оповіщати користувачів у реальному часі у випадку зміни стану серверної частини. Реалізацію та впровадження цього патерну у графічному представленні можна побачити на рис. 27.

У цій реалізації патерну всі клієнти підключаються до сервісу notify, щоб слідкувати за змінами у реальному часі. Сервіс notify представляє собою Event Bus, а також у свою чергу підключається до сервісів, які відповідають за зміну даних, таких як mutate, auth, upload. Тобто коли один з користувачів щось записує на сервер, сервіс для запису оповіщає про це сервіс notify за допомогою API, а потім сервіс notify перевіряє, хто підписаний саме на ці зміни, і відправляє повідомлення про зміну клієнтам через websocket.

Діаграма послідовності серверної частини

Задля розуміння роботи однієї або декількох функцій, так званих прецедентів програми використовують діаграми послідовностей. Тому в даному випадку перед написанням коду було вирішено побудувати діаграму послідовностей, щоб проаналізувати базовий хід роботи серверної частини у часі та виділити основні сутності та взаємодію між ними. Під час моделювання було виділено 3 сутності клієнтів (користувачів, Android-застосунків), у фактичному рішенні їх може бути безліч, але для прикладу було обрано саме 3, так як їх вистачає для відображення основних взаємодій. Також було виділено сутності сервісів та баз даних. Діаграму послідовності можна спостерігати на рис. 28-а та 28-б.

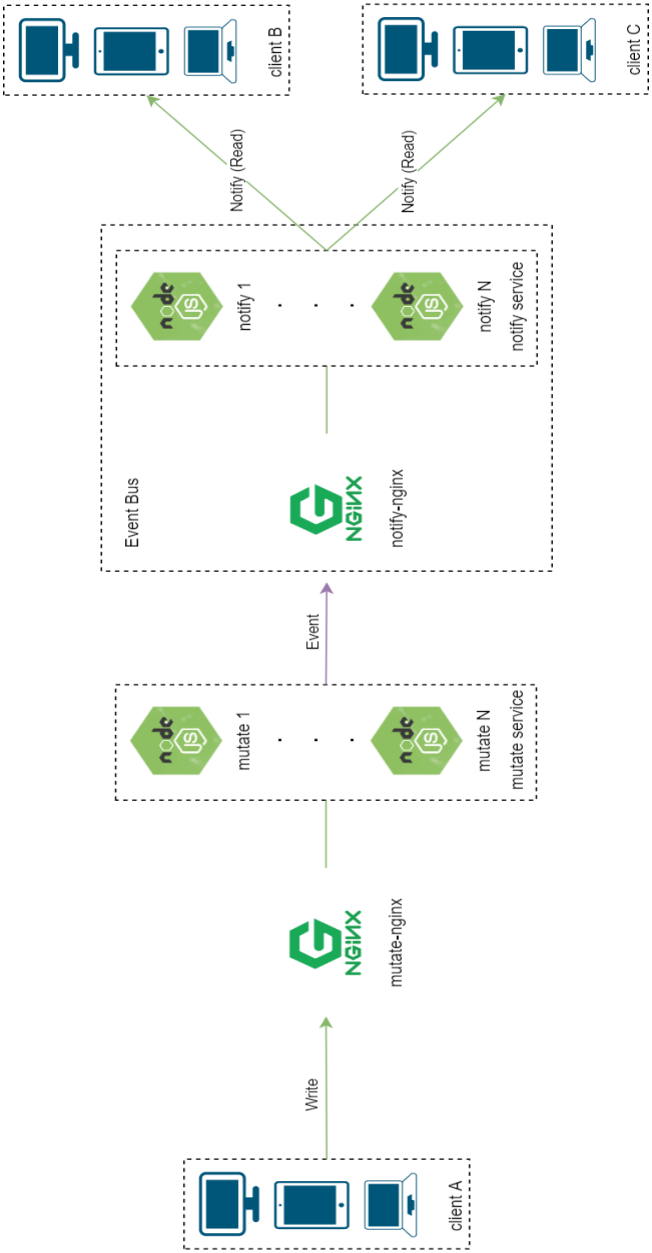


Рисунок 27 – Патерн Event Bus у графічному представленні

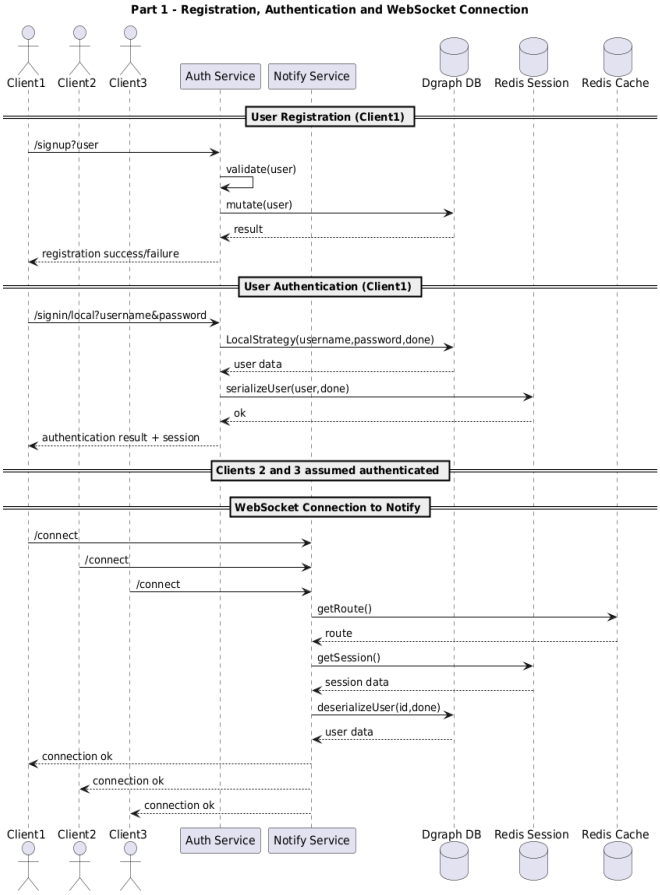


Рисунок 28 – а) Діаграма послідовності серверної частини (Part 1)

Клієнти позначені на діаграмі як актори або «чоловічки», а сутності баз даних та сервісів як прямокутники у верхній частині діаграми. Спочатку на діаграмі демонструється процес реєстрації користувача за такою послідовністю:

1) заповнення форми в Android-застосунку для реєстрації;
2) клієнтський запит на реєстрацію на сервіс auth (/signup?user);
3) валідація запиту (validate(user)) на сервері (auth);
4) запит сервісу auth до бази даних Dgraph (mutate(user));
5) повернення результатів дій сервісу, а сервіс – користувачу.

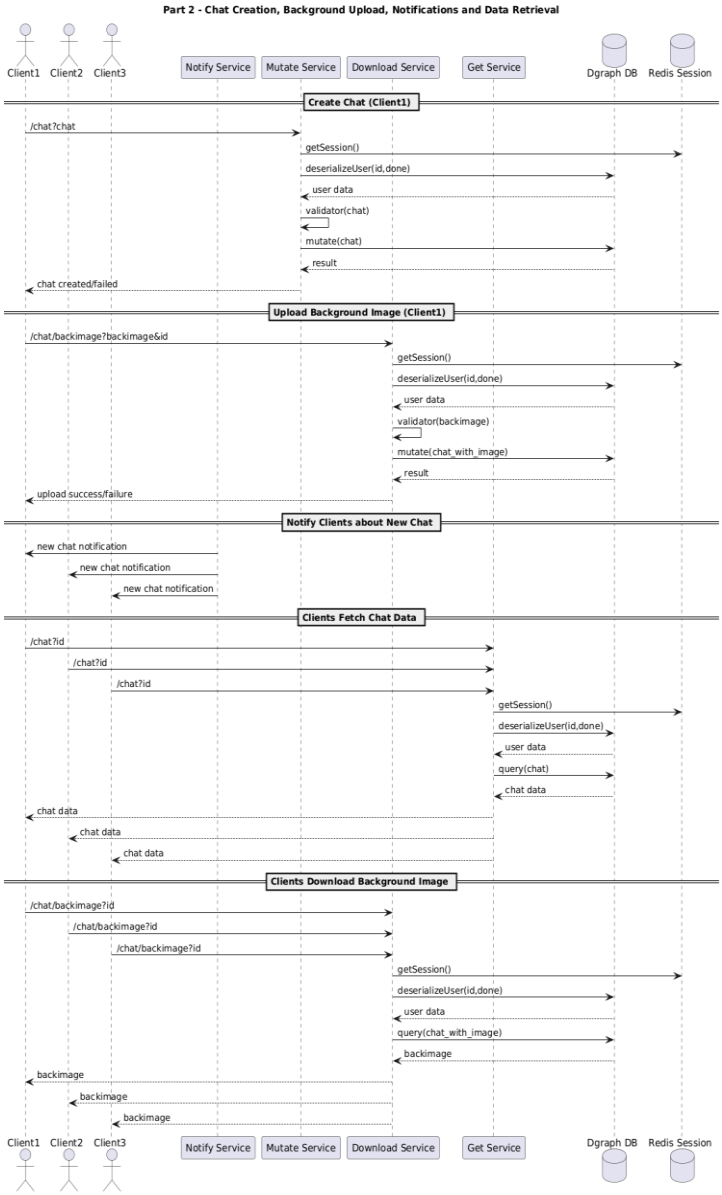


Рисунок 28 – б) Діаграма послідовності серверної частини (Part 2)

Наступним етапом відображається аутентифікації за таким алгоритмом:

1) заповнення форми в android застосунку для аутентифікації;
2) запит до серверу auth (http://signin/local?username&password);
3) сервіс auth за допомогою бібліотеки passport.js повертає з бази даних Dgraph дані користувача за вказаним ім'ям та паролем (LocalStrategy(username, password, done));
4) на основі даних створюється сесія в сервісі auth та зберігається у базі даних Redis session (serealizeUser(user, done));
5) останнім кроком сервісу повертається клієнтські дані, а він їх відправляє клієнту.

Далі діаграма розглядає випадок, коли інші два клієнти вже зареєструвалися та аутентифікувалися, тому ці кроки пропускаються щоб зекономити місце на діаграмі, тому що алгоритм повторюється, і суттєво не відображає нові взаємодії. Наступним кроком всі три клієнти підключаються до сервісу notify по транспортному каналу websockets за наступним алгоритмом:

1) клієнтський запит до серверу notify (/connect);
2) сервіс notify отримує шлях підключення до нього в базі даних Redis cache (getRoute());
3) сервіс notify отримує сесію для авторизації з бази даних Redis session (getSession());
4) сервіс notify десеріалізує дані за сесією за допомогою бази даних Dgraph (deserealizeUser(id, done));
5) сервіс notify відправляє результат клієнту.

Далі розглядається ситуація, коли один клієнт створює конференцію (чат) на декількох користувачів, відбувається це у два етапи, які користувач сприймає як один, адже це автоматично та послідовно виконується та реалізується у Android-застосунку за допомогою двох запитів. Перший етап відбувається наступним чином:

1) клієнт заповнює форму в Android-застосунку для створення чату необхідними даними;
2) клієнт відправляє дані за запитом (/chat?chat) до сервісу mutate;
3) два кроки по аналогії до сервісу notify(3, 4) для авторизування;
4) сервіс mutate валідує дані отримані від клієнту (validator(chat));
5) сервіс mutate зберігає дані у базі даних Dgraph (mutate(chat));
6) сервіс mutate повертає дані про успішність або некоректність операції.

Другим кроком відбувається завантаження фонової картинки до чату за такими кроками:

1) клієнтський запит до сервісу download (/chat/backimage? backimage&id);

2) два кроки по аналогії до сервісу notify(3, 4) для авторизування;

3) сервіс download валідує дані отримані від клієнту (validator(backimage));

4) сервіс download зберігає дані у базі даних Dgraph (mutate(chat_with_image));

5) сервіс download повертає клієнту інформацію про успішність або помилку операції.

Далі всі клієнти, які були додані до конференції, оповіщаються про створення чату через сервіс notify та транспортний протокол websocket. Після цього клієнти отримують дані із сервісу get, попередньо авторизувавшись (за аналогічними кроками 3, 4, як у сервісі notify) за запитом /chat?id, get у свою чергу отримує їх з бази даних Dgraph. Останнім кроком клієнти завантажують фонову картинку конференції з сервісу download, авторизувавшись перед цим.

Даний підхід дозволив продивитись взаємодію між сутностями, виробити алгоритм проєктування серверної частини, а саме запитів та API, а також оптимізувати їх у часі.

9. КОНФІГУРАЦІЯ DOCKER

Для простого розгортання серверів, налаштування середовища, відлагоджування, простої розробки на комп'ютері використовується контейнерні технології. Найпоширенішою є технологія контейнеризації Docker. При проєктуванні систем у даному середовищі використовуються Dockerfile – файл конфігурації контейнеру, docker-compose.yml – файл конфігурації декількох контейнерів та docker-machine – система віртуалізації для симуляції багатокомп'ютерної кластерної системи. Остання технологія не використовувалася в даному проєкті, оскільки розгортання відбувалося або на локальному комп'ютері, або відразу на кластері AWS EC2.

Першим кроком по контейнеризації проєкту було сконфігурувати Dockerfile для сервісів. Для всіх сервісів Dockerfile виглядає приблизно однаково, його можна побачити на рис. 29.

```
1   FROM node:lts-alpine3.9
2
3   # Create app directory
4   RUN mkdir -p /usr/src/app
5   WORKDIR /usr/src/app
6
7   # Install app dependencies
8   COPY package.json /usr/src/app
9
10  # Bundle app source
11  COPY . /usr/src/app
12
13  RUN apk add bash
```

Рисунок 29 – Dockerfile для сервісу auth

За основу для контейнерів сервісів було взято образ node:lts-alpine3.9, який містить в собі простий образ linux системи alpine версії 3.9, та встановлений Node.js версії LTS. Також у конфігурації прописані команди копіювання проєкту до контейнеру та запуск bash для подальших дій чи команд у середовищі docker-compose.

Наступним кроком було конфігурування системи у docker-compose.yml. Спочатку було проініціалізовано volumes, тобто томи для зберігання постійних даних. Конфігурацію можна спостерігати на рис. 29.

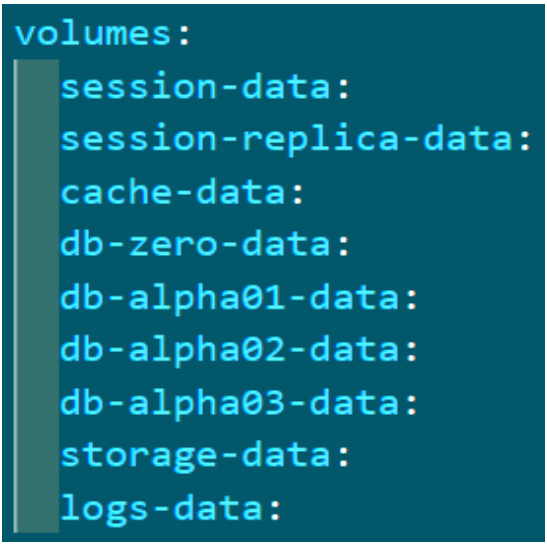


Рисунок 30 – Docker-compose конфігурація для volumes

У конфігурацію було виділено такі томи: session-data – том для зберігання сесій, session-replica-data – том для зберігання копій сесій, cache-data – том для зберігання кешу, db-zero-data – том для зберігання даних про балансування бази даних, db-alpha[номер]-data – томи для зберігання бази даних, storage-data – том для зберігання файлів, logs-data – том для зберігання логів.

Наступним кроком було сконфігуровано простір мереж, що відображено на рис. 31.

```
networks:
  auth-network:
  upload-network:
  download-network:
  mutate-network:
  get-network:
  notify-network:
  notify-api-network:
  seesion-network:
    ipam:
      config:
        - subnet: 170.20.1.0/24
  cache-network:
    ipam:
      config:
        - subnet: 170.20.2.0/24
  db-network:
```

Рисунок 31 – Docker-compose конфігурація для networks

У конфігурації було створено такі ізольовані мережі як: db-network – мережа, яка об'єднує кластер та балансувальник Dgraph, а також поєднує їх із сервісами; cache-network – мережа, яка має певний адресний простір, який потрібно вказувати для налаштування Redis-cluster, поєднує кластер Redis та сервіс notify для збереження маршрутів з'єднань; session-network – мережа по аналогії до cache-network, але створена для поєднання Redis-cluster сесій між сервісами; notify-api-network – мережа, яка була створена для зв'язку сервісів запису (mutate, upload, auth) та notify; інші мережі були створені для поєднання екземплярів сервісів та їх балансувальників, як наприклад auth-network поєднує тільки екземпляри сервісу auth між собою.

Наступним кроком було конфігурування сервісів. Спочатку було створено конфігурацію для розгортання бази даних Dgraph, яку можна побачити рис. 32.

```yaml
alpha01:
  image: dgraph/dgraph:latest
  hostname: alpha01
  networks:
    - db-network
  command: dgraph alpha --my alpha01:7080 --lru_mb 2048 --zero zero:5080
  volumes:
    - db-alpha01-data:/dgraph
  depends_on:
    - zero
  ports:
    - 8080:8080
    - 9080:9080

alpha02:
  image: dgraph/dgraph:latest
  hostname: alpha02
  networks:
    - db-network
  command: dgraph alpha --my alpha02:7081 --lru_mb 2048 --zero zero:5080 -o 1
  volumes:
    - db-alpha02-data:/dgraph
  depends_on:
    - zero
  ports:
    - 8081:8081
    - 9081:9081

alpha03:
  image: dgraph/dgraph:latest
  hostname: alpha03
  networks:
    - db-network
  command: dgraph alpha --my alpha03:7082 --lru_mb 2048 --zero zero:5080 -o 2
  volumes:
    - db-alpha03-data:/dgraph
  depends_on:
    - zero
  ports:
    - 8082:8082
    - 9082:9082

zero:
  image: dgraph/dgraph:latest
  networks:
    - db-network
  command: dgraph zero --my zero:5080 --replicas 3 --idx 1
  volumes:
    - db-zero-data:/dgraph
  ports:
    - 5080:5080
    - 6080:6080
```

Рисунок 32 – Docker-compose конфігурація для кластеру Dgraph

У даній конфігурації присутні три dgraph-alpha для кожного з них, було відкрито порти, присвоєно томи та виконано команди запуску. За основу було взято образ dgraph:latest. Також було запущено для ребалансування dgraph-zero. В основній конфігурації dgraph-zero запускається раніше за екземпляри dgraph-alpha, тому було встановлено налаштування depends_on, що дозволяє налаштувати послідовність запуску.

Далі було створено конфігурації Redis-cluster сесій та Redis-cluster кешу. Конфігурації для обох кластерів здебільшого однакові за винятком різних томів сховищ, назв, мережевого простору. Також для підсилення відмовостійкості для Redis-cluster сесій було додано екземпляри реплікацій, та прописано команду для їх запуску. Приклад конфігурації Redis-cluster кешу можна спостерігати на рис. 33.

```
redis-cache01:
  image: redis:latest
  hostname: redis-cache01
  networks:
    cache-network:
      ipv4_address: 170.20.2.2
  command: redis-server /data/config/redis.conf --cluster-config-file nodes01.conf
  volumes:
    - cache-data:/data
    - ./redis/redis-cache.conf:/data/config/redis.conf

redis-cache02:
  image: redis:latest
  hostname: redis-cache02
  networks:
    cache-network:
      ipv4_address: 170.20.2.3
  command: redis-server /data/config/redis.conf --cluster-config-file nodes02.conf
  volumes:
    - cache-data:/data
    - ./redis/redis-cache.conf:/data/config/redis.conf

redis-cache03:
  image: redis:latest
  hostname: redis-cache03
  networks:
    cache-network:
      ipv4_address: 170.20.2.4
  command: redis-server /data/config/redis.conf --cluster-config-file nodes03.conf
  volumes:
    - cache-data:/data
    - ./redis/redis-cache.conf:/data/config/redis.conf

redis-cache-cluster:
  tty: true
  image: redis:latest
  hostname: redis-cache-cluster
  networks:
    - cache-network
  command: redis-cli --cluster create 170.20.2.2:6380 170.20.2.3:6380 170.20.2.4:6380 --cluster-yes
  volumes:
    - cache-data:/data
  depends_on:
    - redis-cache01
    - redis-cache02
    - redis-cache03

redis-cache-haproxy:
  image: haproxy:latest
  hostname: redis-cache-haproxy
  networks:
    - cache-network
  volumes:
    - ./haproxy/redis-cache-haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg
  depends_on:
    - redis-cache-cluster
```

Рисунок 33 – Docker-compose конфігурація для кластеру Redis для кешу

У конфігурації можна побачити redis-cache[номер] сервіси – це і є екземпляри Redis, для них також було скопійовано конфігурації та приєднано том для зберігання даних. Для кожного екземпляру було прописано адрес у мережевому просторі, це необхідно для ініціалізації кластеру. Сервіс redis-cache-cluster ініціалізує кластер за допомогою вхідної команди. Командою depends_on як і у попередньому випадку задається пріоритетність запуску контейнерів, тобто спочатку екземпляри Redis, а потім ініціалізація кластеру. Також як балансувальник було сконфігуровано сервіс redis-cache-haproxy, який приймає конфігурацію та базується на образі haproxy:latest.

Останнім кроком було сконфігуровано сервіси на Node.js. Всі конфігурації майже однакові, окрім notify, який під'єднується до мережі кешування, також слід додати ще одну відмінність, що сервіси mutate, upload та auth під'єднуються до мережі notify-api. Приклад конфігурації сервісу можна побачити на рис. 34.

У цьому прикладі розглядається сервіс get, як можна побачити у конфігурації ініціалізуються три екземпляри сервісу, основою для яких є раніше сконфігурований Dockerfile. Також на рис. 33 можна спостерігати, що до контейнеру копіюються деякі конфігурації та загальні модулі. Для запуску контейнеру використовуються команди, які оновлюють та інсталюють потрібні артефакти, а також запускають сервіс Node.js. Для балансування використовується образ nginx:latest, до якого передається конфігурація. Також можна побачити, що у балансувальника відкритий порт, для кожного сервісу він свій, також кожний сервіс замкнений у власному мережевому просторі. Кожен балансувальник запускається після того, як запустяться екземпляри сервісу, що встановлюється командою depends_on у конфігурації балансувальника.

```
get01:
  build: ./services/get
  hostname: get01
  environment:
    - NAME=get01
  command: [ "bash", "-c", "npm update && npm install && npm start" ]
  networks:
    - get-network
    - db-network
    - seesion-network
  volumes:
    - logs-data:/usr/src/app/logs
    - ./common-modules:/usr/src/app/common-modules
    - ./config:/usr/src/app/config
  depends_on:
    - redis-session-haproxy

get02:
  build: ./services/get
  hostname: get02
  environment:
    - NAME=get02
  command: [ "bash", "-c", "npm update && npm install && npm start" ]
  networks:
    - get-network
    - db-network
    - seesion-network
  volumes:
    - logs-data:/usr/src/app/logs
    - ./common-modules:/usr/src/app/common-modules
    - ./config:/usr/src/app/config
  depends_on:
    - redis-session-haproxy

get03:
  build: ./services/get
  hostname: get03
  environment:
    - NAME=get03
  command: [ "bash", "-c", "npm update && npm install && npm start" ]
  networks:
    - get-network
    - db-network
    - seesion-network
  volumes:
    - logs-data:/usr/src/app/logs
    - ./common-modules:/usr/src/app/common-modules
    - ./config:/usr/src/app/config
  depends_on:
    - redis-session-haproxy

get-nginx:
  image: nginx:latest
  hostname: get-nginx
  networks:
    - get-network
  volumes:
    - ./nginx/get-nginx.conf:/etc/nginx/nginx.conf
  depends_on:
    - get01
    - get02
    - get03
  ports:
    - 89:89
```

Рисунок 34 – Docker-compose конфігурація для кластеру сервісу get

Процес розгортання

Оскільки для розгортання на локальній машині використовувався docker-compose, то було вирішено для розгортання у хмарному середовищі використовувати docker-swarm та swarmpit. Docker-swarm – це технологія, яка дозволяє розгортати docker контейнери на декількох фізичних або віртуальних машинах, балансувати контейнери, а також слідкувати за їх активністю. Swarmpit – це технологія, яка дозволяє візуалізувати процес розгортання контейнерів у docker-swarm, а також надає контрольну панель для керування та всі необхідні метрики для аналізу за станом та ресурсами кластеру.

Для зручності адміністрування застосовано централізоване керування секретами та змінними середовища, що значно спрощує оновлення конфігурацій без потреби перезапуску всього кластера. Важливою складовою є моніторинг стану інфраструктури, що реалізується завдяки інтеграції з інструментами логування та збору метрик. Це дозволяє швидко виявляти потенційні проблеми і оперативно реагувати на них. Крім того, було впроваджено механізм масштабування сервісів у відповідь на зміну навантаження, що дає змогу оптимально використовувати обчислювальні ресурси. Для підвищення безпеки застосовано обмеження доступу до окремих сервісів за допомогою ролей та прав доступу. Також проведено оптимізацію ресурсів контейнерів, шляхом встановлення лімітів пам’яті та процесорного часу. У комплексі ці технології формують гнучку, надійну та керовану інфраструктуру для ефективного розгортання додатків у хмарному середовищі.

Як хмарну платформу було обрано AWS, так як дана платформа має вигідні ціни, а також безліч компонентів для масштабування хмарної інфраструктури. Усього було закуплено 1 AWS ECR для приватного збереження образів контейнерів та 29 екземплярів AWS EC2, це віртуальні комп’ютери, на які можна комфортно встановлювати необхідні сервіси. Повну хмарну інфраструктуру можна побачити на рис. 35; даний рисунок був побудований у середовищі Cloudcraft. Цей сервіс допомагає легко будувати хмарну інфраструктуру у тривимірному просторі, робити на основі цього розрахунки, а також купувати сервери на обраній платформі за шаблоном побудованої інфраструктури.

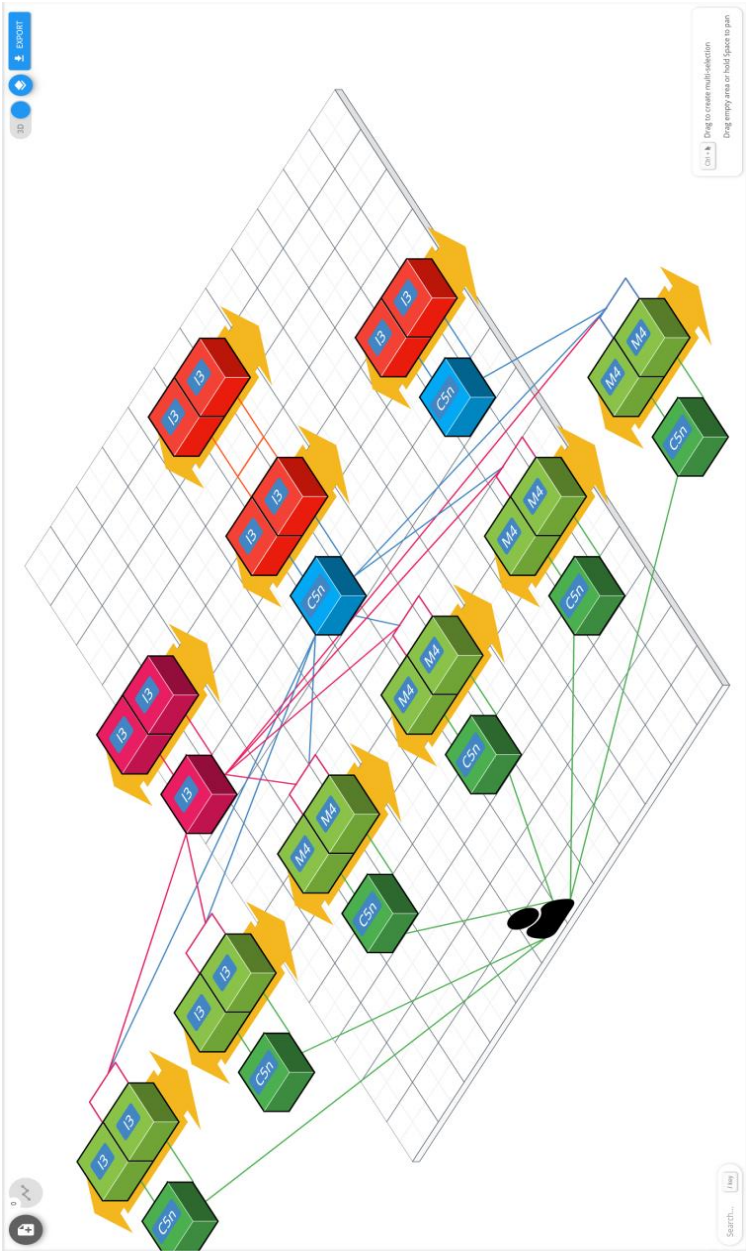


Рисунок 35 – Хмарна інфраструктура побудована на AWS EC2 у середовищі Cloudcraft

Для кожного типу сервісу було обрано відповідний екземпляр AWS EC2. Для балансувальників (Nginx, HAProxy) було обрано екземпляр C5n – так як вони оптимізовані для мережевих обчислень та мають високу пропускну спроможність. Для баз даних та сервісів завантаження та вивантаження файлів було обрано екземпляр I3, який оптимізований для збереження даних. Для всіх інших сервісів було обрано екземпляр M4 – це екземпляр загального призначення, який збалансований по ресурсам. Також по кожному з екземплярів було обрано підтип: для сервісів завантаження та вивантаження – 2xlarge, для сервісів баз даних Dgraph – xlarge, для сервісів баз даних Redis – large, для Nginx балансувальників – xlarge, для HAProxy балансувальників – large, для всіх інших сервісів – large. Всі ці підтипи відрізняються кількістю ресурсів: large має найменшу оперативну пам'ять, найслабший процесор, найменшу кількість фізичної пам'яті і так далі, у свою чергу 16xlarge має найбільшу кількість ресурсів.

Наступним етапом після придбання серверів була ініціалізація docker-swarm та swarmpit на кластері. Спочатку було встановлено безпосередньо сам Docker на всі машини. В останній версії docker-swarm ініціалізація відбувається у дві команди: спочатку було обрано декількох менеджерів, якими виступали потужні екземпляри AWS EC2 та запущено команду "docker swarm init" на них. Після цього було скопійовано токен та за допомогою команди "docker swarm join --token [token] [manager ip]:[manager port]" було приєднано робітників до менеджерів, тобто всі інші сервери. Далі наступним кроком було проініціалізовано swarmpit за допомогою команди "docker run -it --rm --name swarmpit-installer --volume /var/run/docker.sock:/var/run/docker.sock \ swarmpit/install:1.9". Далі у форматі REPL було налаштовано середовище swarmpit.

Щоб завантажити на docker-swarm контейнери за допомогою docker-compose.yml, потрібно було docker-compose.yml відредагувати, оскільки звичайне налаштування для запуску на локальній машині не підходило.

По-перше, потрібно було замінити позиції build на image, попередньо побудувавши та завантаживши образи сервісів на AWS ECR. По-друге, було потрібно прив'язати за допомогою команди constraint сервіси до реальних машин. Після виконання всіх дій було отримано повністю доступний, функціональний мікросервісний кластер у хмарній інфраструктурі AWS. Повну кількість робочих контейнерів, томів, образів та мереж можна спостерігати на рис. 36. Реалізовано використання механізмів автоматичного перезапуску контейнерів та реплікації сервісів. Це забезпечує безперервність роботи у випадку відмови контейнера.

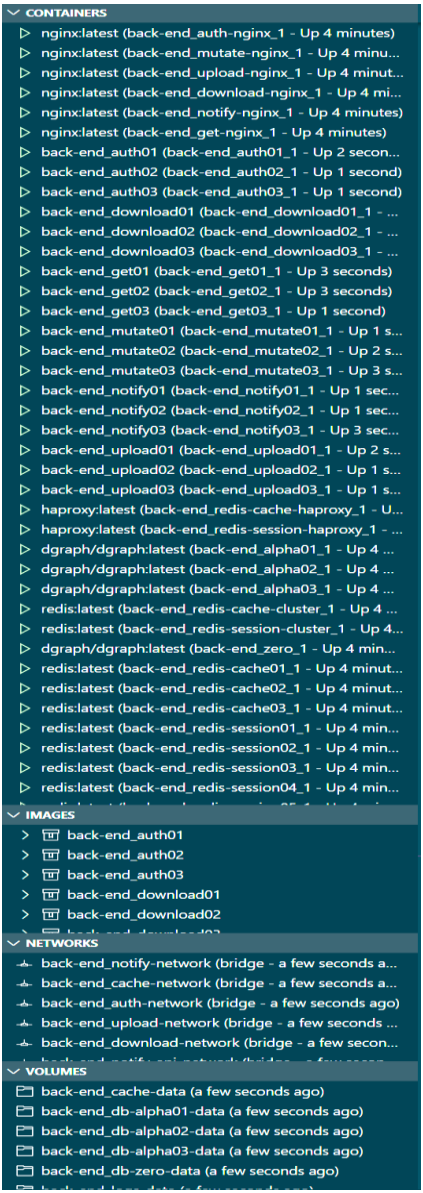


Рисунок 36 – Часткова статистика по контейнерам, томам, мережам та образам Docker

10. АНАЛІЗ СТРУКТУРИ БАЗИ ДАНИХ

Розглянемо програмну систему для планування проєктних завдань для бізнесу. Як база даних була обрана PostgreSQL, оскільки, згідно з [5], ця реляційна база даних має ряд зручних функцій для працювання з текстовими даними. Були виокремлені наступні сутності: DiaryUser (AdminUser та ClientUser), ClientUserRole, Task, Document, TaskType. Роль клієнтського користувача була виокремлена у окрему таблицю, що надає змогу додавати власні ролі для користувачів у подальшому. Так само і з типами задач (рис. 37).

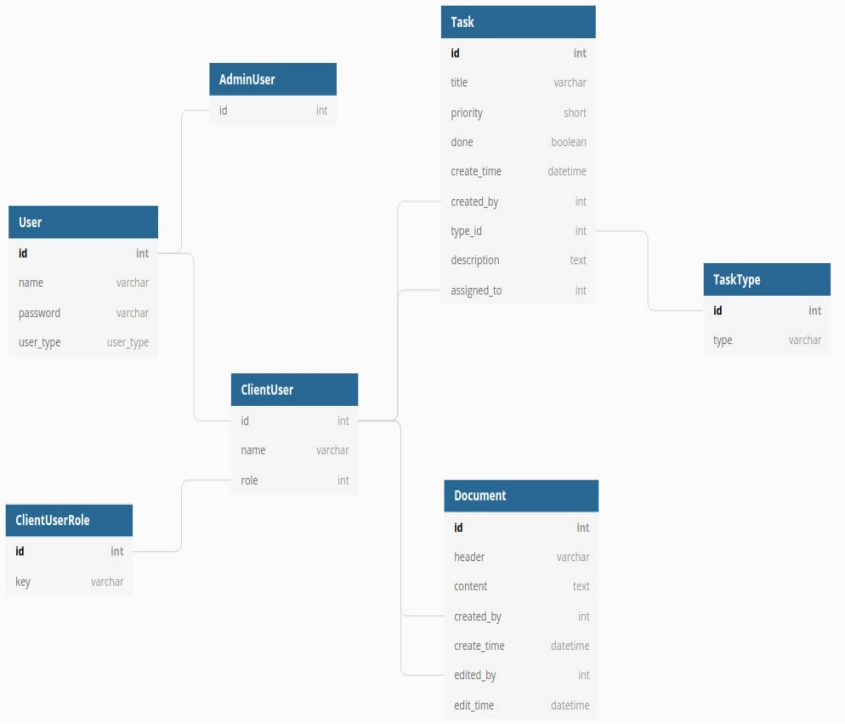


Рисунок 37 – Схема відношення таблиць у базі даних

Дана схема була описана у коді за допомогою ORM-фреймворку Hibernate. Для створення зв'язку між таблицями користувачів був використаний принцип наслідування. Згідно з [6], Hibernate дає на вибір 4 різні способи реалізації наслідування таблиць: Mapped Superclass; Single table; Table per class та Joined. У даному випадку реалізація базувалась на Single table стратегії (рис. 38).

```java
@Entity
@Table(name = "diary_user")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorFormula("user_type")
public abstract class DiaryUser {
    @Id
    @Column(name = "id")
    @SequenceGenerator(name = "diary_user_id_gen", sequenceName = "diary_user_id_seq", allocationSize = 1)
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "diary_user_id_gen")
    protected Long id;

    @Column(name = "login", unique = true)
    protected String login;

    @Column(name = "password")
    protected String password;

    @Enumerated(EnumType.STRING)
    @Type(type = "edu.baiev.com.core.db.types.CustomEnumType")
    @Column(name = "user_type")
    protected UserType userType;
```

Рисунок 38 – Опис сутності DiaryUser у коді

Для цього на батьківському класі була вказана анотація @Inheritance із стратегією SINGLE_TABLE та анотацією @DiscriminatorFormula визначена колонка, яка буде відрізняти тип користувача: user_type (рис. 39).

```java
import ...

@Entity
@DiscriminatorValue("CLIENT")
public class ClientUser extends DiaryUser {
    @Column(name = "name")
    private String name;

    @ManyToOne
    @JoinColumn(name = "role_id")
    public ClientUserRole role;
```

AdminUser.java

```java
package edu.baiev.com.core.db.entities.user;

import ...

@Entity
@DiscriminatorValue("ADMIN")
public class AdminUser extends DiaryUser {

    public AdminUser() { this.userType = UserType.ADMIN; }
}
```

Рисунок 39 – Наслідники класу DiaryUser

У класах-наслідниках була вказана анотація @DiscriminatorValue(“Value”), яка визначає значення колонки-дискримінатору для цього класу. Таким чином під час вибірки записів за класом наслідником, Hibernate буде відокремлювати записи за цим значенням. Тип перелічення user_type був реалізований за допомогою розширення класу EnumType (рис. 40) [7].

```
package edu.baiev.com.core.db.types;

import ...

public class CustomEnumType<T extends Enum> extends EnumType<T> {

    @Override
    public void nullSafeSet(PreparedStatement st, Object value, int index, SharedSessionContractImplementor session)
            throws HibernateException, SQLException {
        if (value == null) {
            st.setNull(index, Types.OTHER);
        } else {
            st.setObject(index, value.toString(), Types.OTHER);
        }
    }
}
```

Рисунок 40 – Клас-розширення EnumType

На рис. 38 можна побачити, що поле userType анотоване анотацією @Type. Ця анотація вказує як фреймворк буде обробляти поле під час генерації об'єкту з рядка бази даних. Для визначення зв'язків між декількома таблицями була використана анотація @JoinTable. Приклад її використання наведено на рис. 41.

Для позначення автоматично генерованої колонки ідентифікатору були використані анотації @GeneratedValue та @SequenceGenerator. Анотація @GeneratedValue використовується разом з анотацією @Id і вказує на те, що дане поле автоматично генерується під час запиту. Існує декілька різних стратегій генерації ідентифікатору, у даному випадку для сутностей була використана стратегія SEQUENCE; дана стратегія використовує послідовність бази даних. Для конфігурації послідовності використовується анотація @SequenceGenerator. У цій анотації може бути вказана конфігурація послідовності: ім'я послідовності, схема, початкове значення, значення інкременту.

```java
@Entity
@NoArgsConstructor
@Data
public class Document {

    @Id
    @SequenceGenerator(name = "document_id_gen", sequenceName = "document_id_seq", allocationSize = 1)
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "document_id_gen")
    @Column(name = "id")
    private Long id;

    @Column(name = "header")
    private String header;

    @Column(name = "content")
    private String content;

    @ManyToOne
    @JoinColumn(name = "created_by")
    private ClientUser userCreate;

    @ManyToOne
    @JoinColumn(name = "edited_by")
    private ClientUser userEdit;

    @Column(name = "create_time")
    private LocalDateTime createTime;

    @Column(name = "edit_time")
    private LocalDateTime editTime;
}
```

Рисунок 41 – Опис сутності Document у коді

11. АРХІТЕКТУРА СЕРВЕРНОЇ ЧАСТИНИ ЗАСТОСУНКУ

Серверна частина застосунку була реалізована за допомогою мови Java та ряду бібліотек, наданих разом зі SpringBoot, окрім цього були застосовані бібліотеки lombok, apache-utils та io.jsonwebtoken.jjwt. Для збірки проєктів був застосований інструмент Maven. Серверна частина була розділена на декілька різних модулів. Це дозволяє відокремлювати різні частини застосунку та виконувати зміни в різних модулях незалежно один від одного. Під час реалізації класів була використана анемічна доменна модель [8]. Це означає, що класи сутностей бази даних мають лише поля. Уся логіка роботи з цими класами знаходиться у класах сервісах.

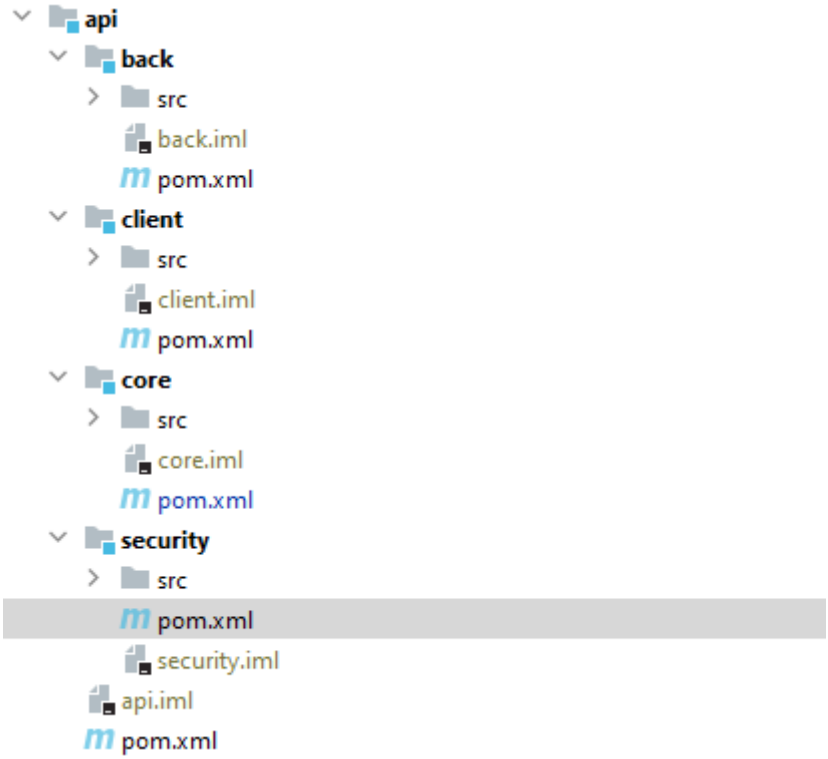


Рисунок 42 – Модулі серверної частини застосунку

Модуль Core зберігає головну частину програми, без якої жодний модуль не може існувати. У цьому модулі були описані сутності бази даних та Spring Data JPA репозиторії для них. Кожний із модулів повинен у себе включати модуль core

Модуль Security зберігає у собі класи-реалізації Spring Security. Таким чином, цей модуль надає класи для захисту серверу від неавторизованого доступу.

Модуль Client має класи-контролери та сервіси для роботи із клієнтським застосунком. Контролери надають можливість за допомогою Rest API використовувати сервіси застосунку.

Модуль Back має у собі класи-контролери та сервіси для роботи із адміністраторським застосунком. Цей модуль написаний із використанням шаблону MVC. Контролери цього модулю відображають сторінки адміністраторського застосунку та дають змогу використовувати різні сервіси.

Модуль Core

Реалізація сутностей бази даних була описана у розділі 3. Для надання доступу до бази даних була використана бібліотека Spring Data Jpa. За допомогою наслідування відповідних інтерфейсів цієї бібліотеки можна отримати CRUD-операції.

```
@Repository
public interface ClientUserRepository extends PagingAndSortingRepository<ClientUser, Long> {

    @Query("select u from ClientUser u where u.login = :login")
    Optional<ClientUser> findByLogin(@Param("login") String login);
}
```

Рисунок 43 – Репозиторій сутності ClientUser

Власні запити можна написати декількома шляхами. Запит до бази даних може бути описаний за допомогою ключових слів, але такий спосіб не є безпечним. Запит до бази даних може бути вказаний у анотації @Query за допомогою мови JPA. Третій шлях написання запиту – вказати рідний запит у анотації @Query. Таким чином можна використовувати специфічні для бази даних оператори. Приклад такого запиту наведено на рис. 44.

```java
@Repository
public interface DocumentRepository extends PagingAndSortingRepository<Document, Long> {

    @Query(value = "select * from document where (document.header || document.content) ~* :part", nativeQuery = true)
    Page<Document> findAllByPart(Pageable pageable, @Param("part") String part);
}
```

Рисунок 44 – Репозиторій до сутності Document

Модуль Security

Модуль безпеки був реалізований на основі Spring Security та із застосуванням стандарту JWT. Була створена реалізація інтерфейсу UserDetails. Реалізація у своїх методах приховує екземпляр сутності DiaryUser. Така реалізація демонструє шаблон проєктування, відомий як Декоратор.

```java
public class SecuredUserDetails implements UserDetails {
    private final DiaryUser diaryUser;

    public SecuredUserDetails(DiaryUser diaryUser) { this.diaryUser = diaryUser; }

    public DiaryUser getUser() { return diaryUser; }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return Collections.singletonList(new SimpleGrantedAuthority( role: "ROLE_" + diaryUser.getUserType().getName()));
    }

    @Override
    public String getPassword() { return diaryUser.getPassword(); }

    @Override
    public String getUsername() { return diaryUser.getLogin(); }

    @Override
    public boolean isAccountNonExpired() { return true; }

    @Override
    public boolean isAccountNonLocked() { return true; }

    @Override
    public boolean isCredentialsNonExpired() { return true; }

    @Override
    public boolean isEnabled() { return true; }
}
```

Рисунок 45 – Реалізація UserDetails

Була написана реалізація інтерфесу UserDetailsService. Цей інтерфейс надає метод завантаження користувача за його логіном. Був створений клас конфігурації, який визначає екземпляри класів безпеки. У

подальшому ці об’єкти будуть використані у клієнтських модулях під час конфігурації безпеки (рис. 46).

```
@Configuration
public class SecurityProvider {

    @Bean
    AuthenticationProvider authenticationProvider(PasswordEncoder encoder,
                                                  UserDetailsService service) {
        DaoAuthenticationProvider authenticationProvider = new DaoAuthenticationProvider();
        authenticationProvider.setPasswordEncoder(encoder);
        authenticationProvider.setUserDetailsService(service);
        authenticationProvider.setAuthoritiesMapper(new SimpleAuthorityMapper());
        return authenticationProvider;
    }

    @Bean
    public PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); }

    @Bean
    public SecurityContextRepository securityContextRepository() { return new HttpSessionSecurityContextRepository(); }
}
```

Рисунок 46 – Клас-провайдер екземплярів класів безпеки

Був створений інтерфейс сервісу, який забезпечує роботу з модулем безпеки. Таким чином, клієнтські модулі будуть відгороджені від роботи з модулем безпеки напряму (рис. 47).

```
public interface UserSecurityService {

    DiaryUser getCurrentUser();

    boolean isLoggedIn();

    AuthenticationDto login(String login, String password);

    void logout();
}
```

Рисунок 47 – Сервіс роботи з модулем безпеки

Був створений фільтр запитів, який перевіряє наявність авторизаційного заголовку. У випадку, якщо запит має авторизаційний заголовок, користувач буде аутентифікований і зможе отримати доступ до захище-

них ресурсів. Цей клас є прикладом шаблону ланцюг обов'язків. Це можна визначити за останнім рядком у методі фільтрації, який викликає наступний фільтр у ланцюгу.

Модуль Client

Був створений ряд контролерів, які надають доступ до сервісів застосунку. Кожний клас контролеру був анотований анотацією @RestController, яка помічає біни контролери та позначає, що кожен метод цього класу повинен бути пов'язаний з тілом відповіді. За допомогою анотації @RequestMapping на рівні класу був вказаний контекстний шлях для кожного з контролерів. Анотації @GetMapping @PostMapping @PutMapping й інші є аналогами анотації @RequestMapping з відповідним методом запиту (рис. 48).

```
@RestController
@RequestMapping(value = "/documents", produces = MediaType.APPLICATION_JSON_VALUE)
@RequiredArgsConstructor
public class DocumentController {
    private final DocumentFacade documentFacade;

    @GetMapping
    public Page<DocumentDto> getDocuments(@PageableDefault Pageable pageable) {
        return documentFacade.getDocuments(pageable);
    }

    @PostMapping
    public Long createDocument(@RequestBody @Validated DocumentCreateDto createDto) {
        return documentFacade.createDocument(createDto);
    }

    @GetMapping("/{id}")
    public DocumentDto getDocument(@PathVariable("id") Long documentId) {
        return documentFacade.getDocument(documentId);
    }

    @GetMapping("/search")
    public Page<DocumentDto> findDocuments(
            @PageableDefault Pageable pageable,
            @RequestParam("part") String part) {
        return documentFacade.findDocuments(pageable, part);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Object> updateDocument(@PathVariable("id") Long documentId,
                                                 @RequestBody @Validated DocumentCreateDto documentDto) {
        documentFacade.updateDocument(documentId, documentDto);
        return ResponseEntity.ok().build();
    }

    @PostMapping("/{id}/delete")
    public ResponseEntity<Object> dropDocument(@PathVariable("id") Long documentId) {
        documentFacade.dropDocument(documentId);
        return ResponseEntity.ok().build();
    }
}
```

Рисунок 48 – Контролер роботи з сервісами документів

За шаром контролерів знаходить шар фасадів, який працює з відповідними сервісами та маперами (рис. 49).

```java
@Component
@RequiredArgsConstructor
public class DocumentFacade {
    private final DocumentationService documentationService;
    private final DocumentDtoMapper documentDtoMapper;
    private final DocumentCreateDtoMapper documentCreateDtoMapper;
    private final UserSecurityService userSecurityService;

    @Transactional(readOnly = true)
    public Page<DocumentDto> getDocuments(Pageable pageable) {
        return documentationService.getDocuments(pageable)
                .map(documentDtoMapper::map);
    }

    @Transactional(readOnly = true)
    public DocumentDto getDocument(Long documentId) {
        return documentDtoMapper.map(documentationService.getDocument(documentId));
    }

    @Transactional
    public Long createDocument(DocumentCreateDto documentDto) {
        Document map = documentCreateDtoMapper.map(documentDto);
        map.setUserCreate((ClientUser) userSecurityService.getCurrentUser());
        map.setCreateTime(LocalDateTime.now());
        return documentationService.createDocument(map);
    }

    @Transactional
    public void updateDocument(Long documentId, DocumentCreateDto documentDto) {
        Document document = documentCreateDtoMapper.map(documentDto);
        document.setUserEdit((ClientUser) userSecurityService.getCurrentUser());
        document.setEditTime(LocalDateTime.now());
        documentationService.updateDocument(documentId, document);
    }

    @Transactional
    public void dropDocument(Long documentId) { documentationService.deleteDocument(documentId); }

    @Transactional(readOnly = true)
    public Page<DocumentDto> findDocuments(Pageable pageable, String part) {
        return documentationService.getAllByPart(pageable, part)
                .map(documentDtoMapper::map);
    }
}
```

Рисунок 49 – Клас фасаду документів

Був створений ряд DTO (Data Transfer Object) класів [9], які використовуються для комунікації у контролерах. Таким чином можна конфігурувати дані, які буде отримувати клієнт застосунку та додавати перевірки для цих даних. Декларації перевірки даних додаються за допомогою анотацій, які надаються у бібліотеці javax.validation. Запуск перевірки виконується анотацією @Validated. Приклад застосування наведено на рис. 50.

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class DocumentCreateDto {
    @NotEmpty
    private String header;

    @NotEmpty
    private String content;
}
```

Рисунок 50 – Приклад DTO-класу для редагування документу

Щоб відобразити екземпляри класів сутностей у вигляді DTO класів були створені ряд класів за шаблоном Mapper.

```
@Component
public class DocumentCreateDtoMapper {
    public Document map(DocumentCreateDto documentDto) {
        Document document = new Document();
        document.setHeader(documentDto.getHeader());
        document.setContent(documentDto.getContent());
        return document;
    }
}
```

Рисунок 51 – Приклад Mapper-документу з DTO на сутність бази даних

Наступним шаром за фасадом є шар сервісів. Ці класи надають інтерфейс роботи з репозиторієм (рис. 52).

Завдяки анотації @Transactional методи класів будуть виконані у транзакції бази даних. Таким чином, якщо буде викинуто виняток, дані не будуть збережені у базі даних. Були створені класи конфігурації.

Клас AppConfig має конфігурацію Spring застосунку: вказує пакети сканування бінів, пакети сутностей та репозиторіїв тощо. Клас SecurityConfig наслідує клас WebSecurityConfigurerAdapter та містить у собі налаштування модулю безпеки.

```
@Service
@RequiredArgsConstructor
public class DocumentationService {
    private final DocumentRepository documentRepository;

    @Transactional(readOnly = true)
    public Page<Document> getDocuments(Pageable pageable) { return documentRepository.findAll(pageable); }

    @Transactional(readOnly = true)
    public Document getDocument(Long documentId) {
        return documentRepository.findById(documentId)
                .orElseThrow(() -> new RuntimeException("Can't find document by id: " + documentId));
    }

    @Transactional
    public Long createDocument(Document document) {
        return documentRepository.save(document)
                .getId();
    }

    @Transactional
    public void updateDocument(Long documentId, Document updateDocument) {
        Document document = this.getDocument(documentId);
        updateDocument.setId(documentId);
        updateDocument.setUserCreate(document.getUserCreate());
        updateDocument.setCreateTime(document.getCreateTime());
        documentRepository.save(updateDocument);
    }

    @Transactional
    public void deleteDocument(Long documentId) { documentRepository.deleteById(documentId); }

    public Page<Document> getAllByPart(Pageable pageable, String part) {
        StringBuilder builder = new StringBuilder();
        return this.documentRepository.findAllByPart(pageable, part);
    }
}
```

Рисунок 52 – Сервіс документів

Модуль Back

Цей модуль містить у собі контролери для роботи із адміністраторським застосунком. За допомогою обробника шаблонів thymeleaf та набору інструментів bootstrap були створені HTML-сторінки адміністраторського застосунку. Усі шаблони були розміщені у папці ресурсів (рис. 53).

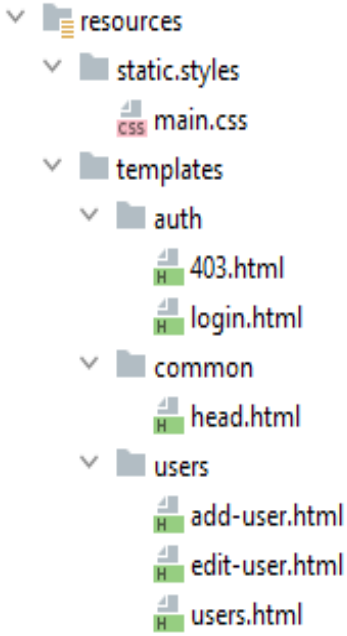


Рисунок 53 – Шаблони адміністраторського застосунку

Були створені класи контролери. Вони вирішують, який шаблон буде відображений на сторінці (рис. 54).

Модуль Back не перенавантажений бізнес-логікою, тому виклик сервісів знаходиться напряму у контролерах. Під час створення моделей для перевірки правильності були використані групи перевірок. Це дає змогу застосовувати перевірки залежно від потреби. Для цього були створені і вказані у відповідних анотаціях інтерфейси EditGroup та CreateGroup. У відповідних анотаціях @Validated були вказані необхідні інтерфейси як групи для перевірки (рис. 55).

```
@Controller
@RequestMapping("/auth")
@RequiredArgsConstructor
public class AuthController {

    private final UserSecurityService userSecurityService;

    @GetMapping("/login")
    public String userList(Model model,
                           HttpSession session) {
        if (userSecurityService.isLoggedIn()) {
            return "redirect:user/list";
        }
        model.addAttribute( s: "loginModel", new LoginModel());
        return "auth/login";
    }

    @GetMapping("/403")
    public String acessDeniedPage(Model model) { return "auth/403"; }

    @PostMapping("/login")
    public String login(Model model,
                        @ModelAttribute("loginModel") @Validated LoginModel loginModel,
                        BindingResult bindingResult) {
        try {
            userSecurityService.login(loginModel.getLogin(), loginModel.getPassword());
            return "redirect:/user/list";
        } catch (DiaryException exception) {
            model.addAttribute( s: "errorCode", exception.getCode());
            model.addAttribute( s: "errorMessage", exception.getLocalizedMessage());
            return "auth/login";
        }
    }

    @GetMapping("/logout")
    public String logout(Model model) {
        userSecurityService.logout();
        return "redirect:/auth/login";
    }
}
```

Рисунок 54 – Авторизаційний контролер

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
public class ClientUserCreateModel {

    @NotNull(groups = EditGroup.class)
    private Long id;

    @NotEmpty
    private String login;

    @NotEmpty(groups = CreateGroup.class)
    private String password;

    @NotEmpty
    private String name;

    @NotEmpty
    private String role;

    public static ClientUserCreateModel from(ClientUser user) {
        ClientUserCreateModel clientUserCreateModel = new ClientUserCreateModel();
        clientUserCreateModel.id = user.getId();
        clientUserCreateModel.login = user.getLogin();
        clientUserCreateModel.name = user.getName();
        clientUserCreateModel.role = user.getRole().getKey();
        return clientUserCreateModel;
    }
}
```

Рисунок 55 – Java-модель форми створення та редагування користувача

У конфігурації безпеки у модулі Back не було відключено створення сесії. Таким чином, аутентифікація користувача зберігається не тільки у JWT, але й у поточній сесії (рис. 56).

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authenticationProvider(authenticationProvider);

    http.csrf().disable() HttpSecurity
            .cors().and()
            .headers() HeadersConfigurer<HttpSecurity>
            .frameOptions().disable()
            .cacheControl();
    http.authorizeRequests()
            .antMatchers( ...antPatterns: "/auth/**", "/h2-console/**", "/403")
            .permitAll();
    http.authorizeRequests()
            .anyRequest()
            .hasRole("ADMIN");
    http.authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .and() HttpSecurity
            .exceptionHandling() ExceptionHandlingConfigurer<HttpSecurity>
            .accessDeniedPage("/auth/403");
    http.addFilterBefore(customUserAuthenticationFilter, UsernamePasswordAuthenticationFilter.class);
}
```

Рисунок 56 – Конфігурація безпеки запитів модулю Back

12. РЕАЛІЗАЦІЯ КЛІЄНТСЬКОЇ ЧАСТИНИ ЗАСТОСУНКУ

Клієнтська частина була реалізована за допомогою фреймворку Angular 12-ої версії на основі шаблону від CreativeTim [10]. Також були використані інструменти Bootstrap та TailwindCSS. Структурно проєкт був розподілений на модулі за доменним значенням (рис. 57).

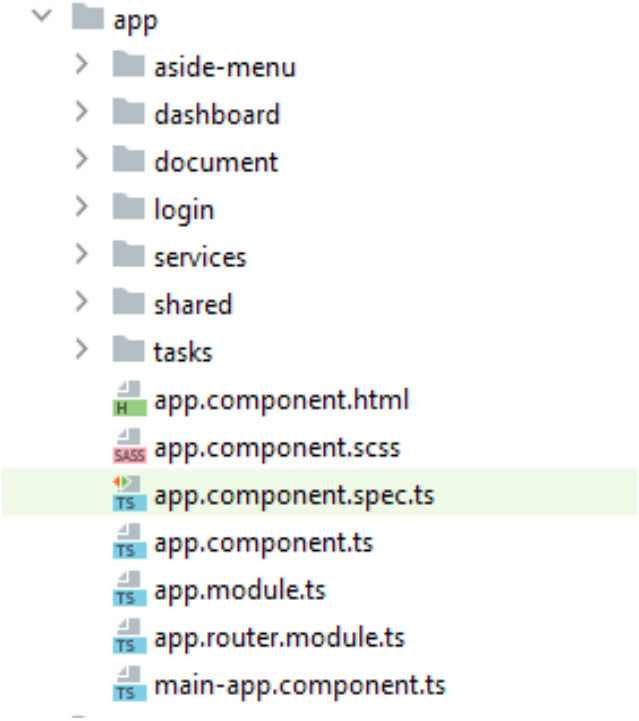


Рисунок 57 – Структура Angular-проєкту

Директорія Aside-menu містить у собі компонент навігаційного меню.

Директорія Dashboard містить у собі модуль сторінки користувача.

Директорія Dashboard містить у собі модуль сторінок документації.

Директорія login містить у собі компонент авторизації.

Директорія Services містить у собі сервіси, перехопники запитів, захист маршрутів тощо.

Директорія Shared містить у собі компоненти, які використовуються у всьому проєкті.

Директорія Tasks містить у собі модуль сторінок завдань.

За допомогою класу RouterModule були написані маршрути до вебсайту. У властивості canActivate були вказані захисники шляху, які визначають, чи може користувач отримати доступ до сторінки (рис. 58).

```
export const routes: Routes = [
    {
        path: '',
        pathMatch: 'full',
        redirectTo: 'login'
    },
    {
        path: 'login',
        canActivate: [LoggedInGuardService],
        component: LoginComponent
    },
    {
        path: 'app',
        component: MainAppComponent,
        canActivate: [AuthGuardService],
        children: [
            {
                path: '',
                pathMatch: 'full',
                redirectTo: 'dashboard'
            },
            {
                path: 'dashboard',
                loadChildren: () => import('./dashboard/dashboard.module').then(m => m.DashboardModule)
            },
            {
                path: 'tasks',
                loadChildren: () => import('./tasks/tasks.module').then(m => m.TasksModule)
            },

            {
                path: 'documents',
                loadChildren: () => import('./document/document.module').then(m => m.DocumentModule)
            },
        ]
    },

];
```

Рисунок 58 – Кореневий маршрутний модуль

Сторінка аутентифікації

Був створений компонент аутентифікації. За допомогою класу FormBuilder була створена модель форми. Таким чином, за допомогою TypeScript коду можна відслідковувати значення форми, накладати перевірки на поля форми та створювати перехопників подій. Код компоненту наведений у додатку В. Був створений оброблювач події підтвердження форми. При натисненні кнопки буде відправлений запит на перевірку аутентифікаційних даних. Якщо користувач увів вірні дані, сервер поверне токен. За допомогою нього можна користуватись захищеними ресурсами. Цей токен буде збережений у сховищі браузера і пізніше підставлятиметься за допомогою перехопника запитів (рис. 59).

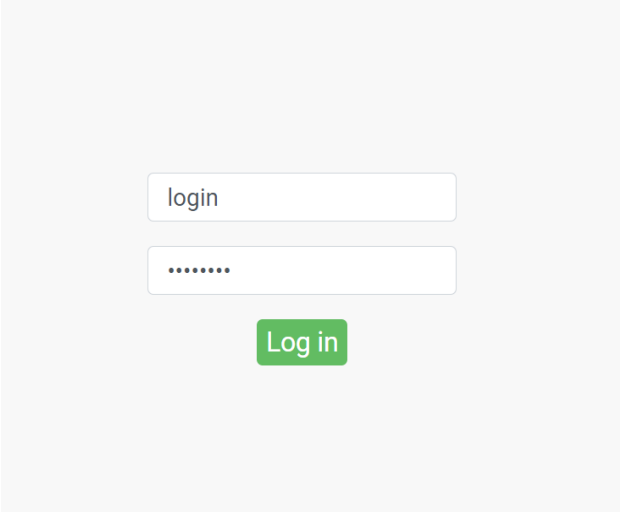


Рисунок 59 – Сторінка аутентифікації

Сторінка завдань

Після успішної авторизації, користувач потрапляє на сторінку списку поточних завдань. Тут він може перейти на сторінку створення нового завдання чи зайти у деталі одного з них. Також є можливість помітити завдання як виконане чи видалити його. Під час реалізації цього компоненту була використана директива *ngFor. Вона дозволяє копіювати певну частину HTML-шаблону для кожного елементу масиву. За допомогою директиви *ngIf був створений напис, який оповіщає користувача, якщо завдання не знайдені. У методі onInit була додана логіка,

яка завантажує доступні завдання з серверу та додає їх до масиву завдань у компоненті (рис. 60-61).

```
<div class="main-content">
    <div>
        <div class="text-xl italic text-md-center text-gray-700" *ngIf="items.length === 0">
            There is no any tasks found
        </div>
        <div class="" *ngFor="let task of items">
            <app-task-item
                    (deleteEvent)="updateView($event)"
                    [item]="task"
            ></app-task-item>
        </div>
        <div class="add-button">
            <button class="bg-green-500 hover:bg-green-700 text-white rounded-pill border-0"
            ><a routerLink="/app/tasks/add" class="text-white"><i class="bi bi-plus text-3xl"></i></a></button>
        </div>
    </div>
</div>
```

Рисунок 60 – Шаблон сторінки списку завдань

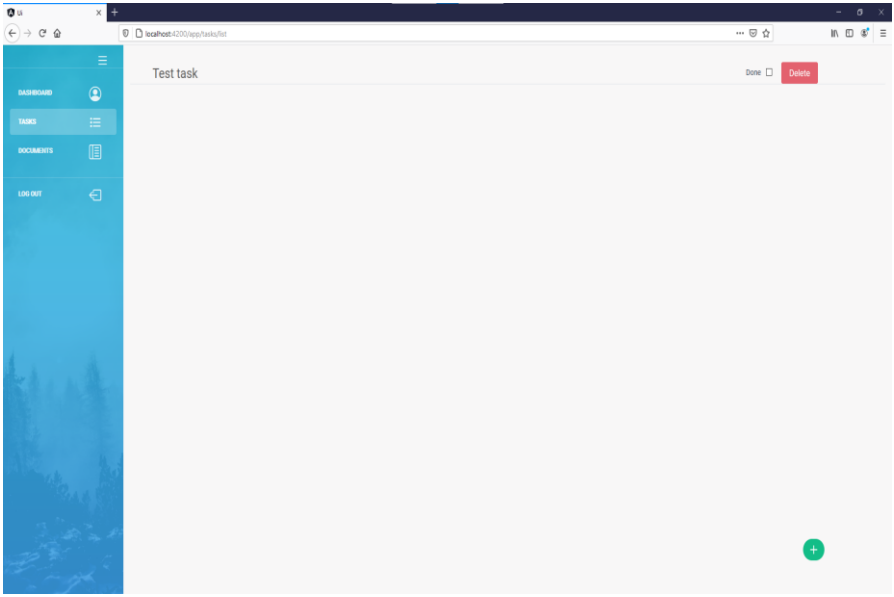


Рисунок 61 – Сторінка списку завдань

Для реалізації компоненту деталей завдання був використаний механізм двостороннього зв'язування. Таким чином, при зміні значення елементу input, відповідне значення зв'язаного об'єкта буде змінене (рис. 62).

```
<div class="row">
    <label class="col-md-4">Assignee</label>
    <span *ngIf="!editable">{{task.assignedTo}}</span>
    <select *ngIf="editable" [(ngModel)]="task.assignedTo">
        <option *ngFor="let userLogin of this.users">{{userLogin}}</option>
    </select>
</div>
```

Рисунок 62 – Приклад використання двостороннього зв'язування

Для відображення деталей завдань була використана бібліотека ngx-markdown. Це дозволяє записувати деталі завдання у форматі Markdown (рис. 63).

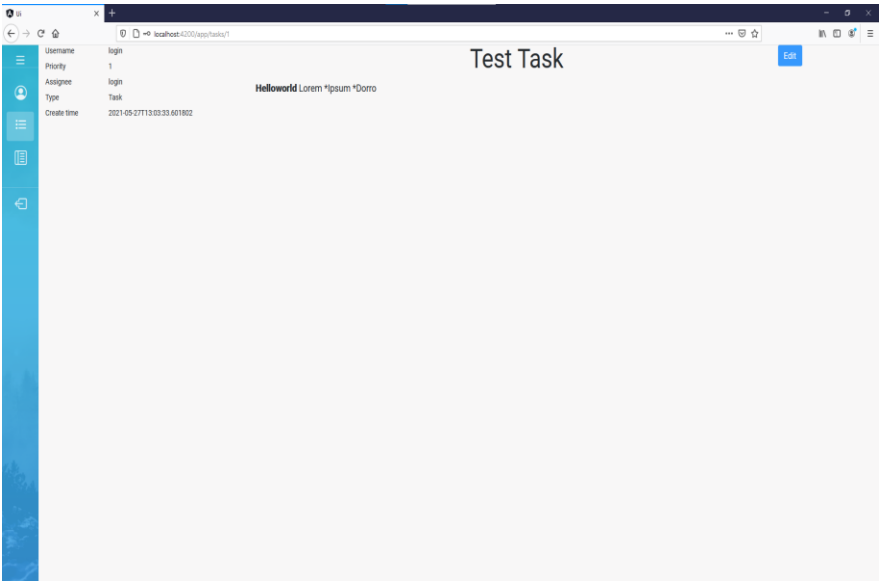


Рисунок 63 – Сторінка деталей завдання

Форма створення нового завдання була створена за допомогою класу FormBuilder (рис. 64).

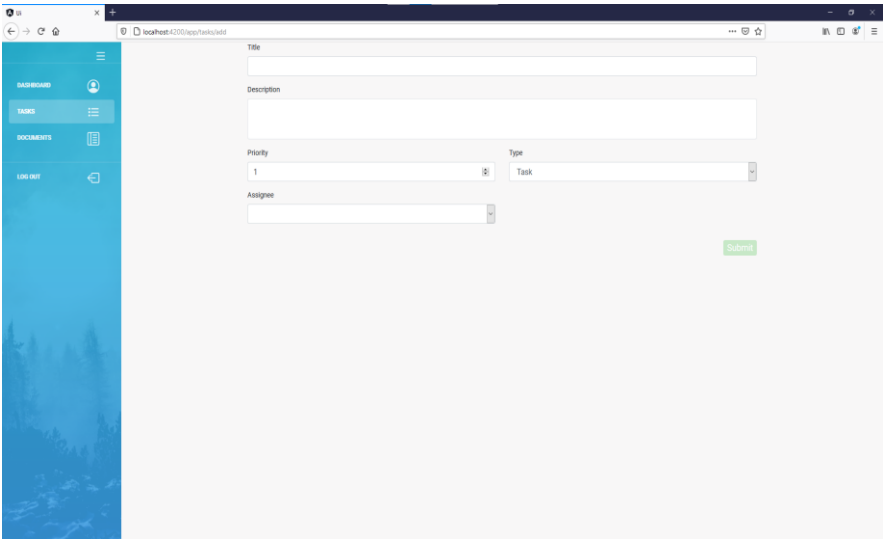


Рисунок 64 – Форма створення нового завдання

Сторінка документів

Сторінка списка документів була створена за допомогою директиви *ngFor. До компоненту було додане текстове поле для пошуку документів. За допомогою декораторів @Output та @Input була реалізована взаємодія між компонентом пошукового поля та компонентом списку завдань (рис. 65).

```
<app-custom-input
        [disabled]="searching"
        (valueChange)="search($event)"
        iconClass="bi-search"></app-custom-input>
```

Рисунок 65 – Використання компоненту пошукового поля

Для сторінки деталей документів також була використана бібліотека ngx-markdown (рис. 66).

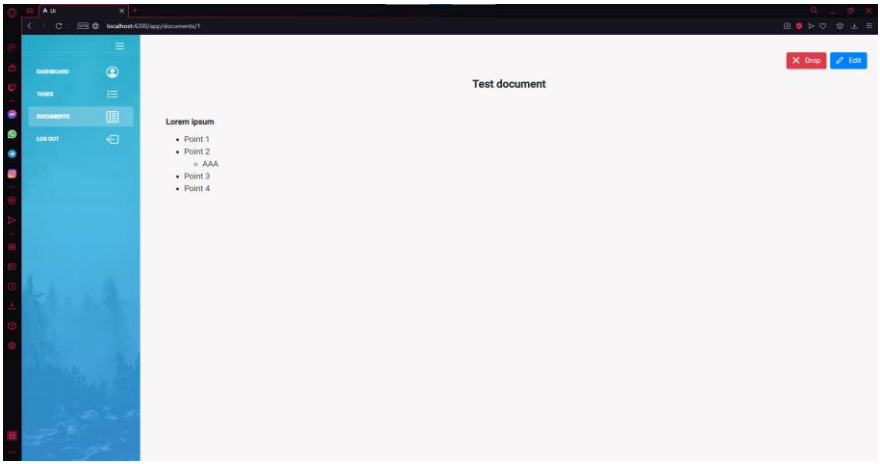


Рисунок 66 – Сторінка деталей документу

Сторінка користувача

Сторінка користувача дає можливість змінити та переглянути дані (рис. 67).

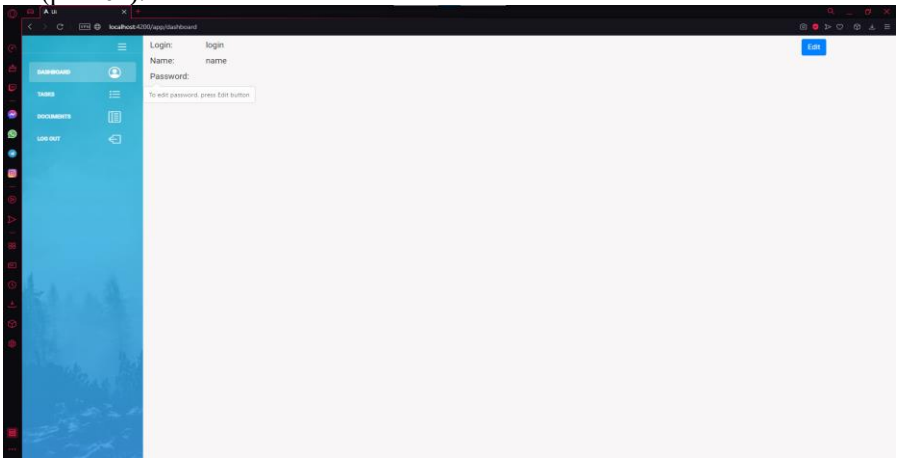


Рисунок 67 – Сторінка користувача

Для реалізації Popover-компоненту була використана бібліотека NGBootstrap [11]. Для реалізації полів вводу інформації був використаний механізм двостороннього зв’язування.

13. КОНТЕЙНЕРИЗАЦІЯ

За допомогою Docker-compose була створена конфігурація контейнерів застосунку. Конфігурація Docker-compose представлена у додатках. У параметрі build був вказаний шлях до конфігурації контейнерів, які необхідно створити. Для використання готових знімків необхідно вказати назву знімка у параметрі image. Параметром links контейнери були пов'язані між собою у одній мережі. Директорії проєкту були поміщені у контейнер за допомогою параметру volumes. Для кожного модулю застосунку був створений власний Dockerfile. Для конфігурації Java контейнерів (модулів back та client) були використані існуючи знімки openjdk. У контейнер був скопійований jar-архів із модулем застосунку та відкритий відповідний порт. Команда sleep була застосована для надання затримки старту серверної частини застосунку. Це дає час для ініціалізації бази даних. В останньому рядку у команді ENTRYPOINT були вказані команди для старту модулю застосунка. Загалом Dockerfile для модулів back та client ідентичний, окрім відкритого порту (рис. 68).

```
FROM openjdk:8-jdk-alpine

COPY ./back*.jar back.jar
COPY ./application.properties application.properties

EXPOSE 8081


CMD sleep 30
ENTRYPOINT ["java", "-Dspring.config.location=file:/application.properties", "-jar", "/back.jar"]
```

Рисунок 68 – Конфігурація контейнеру модуля back серверної частини

Для конфігурації контейнеру клієнтської частини був використаний знімок node:12.22.1-alpine3.10. Для його запуску був відкритий порт 4200 та прописана команда ng serve з відповідними параметрами (рис. 69).

```
FROM node:12.22.1-alpine3.10

RUN apk --update add curl vim bash
RUN rm -rf /var/cache/apk/*


EXPOSE 4200
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
RUN npm install -g @angular/cli
CMD ng serve --host 0.0.0.0 --port 4200
```

Рисунок 69 – Конфігурація контейнеру клієнтської частини

База даних була створена на основі знімку postgres:11.12 [12]. Для ініціалізації бази даних був створений SQL-скрипт та параметром volumes поміщений у контейнер. Для запуску контейнерів використовується команда docker-compose up -d. Зупинка контейнерів доступна командою docker-compose down –rmi local.

14. AGILE-РЕТРОСПЕКТИВА ПРОЄКТІВ

При виконанні проєкту слід проаналізувати ринок застосунків для організації часу та інформації у групах осіб. При цьому мають бути розглянуті такі типи застосунків, як органайзери, месенджери та системи управління командними проєктами. По кожному типу треба проаналізувати версії корпоративні та загальні. В кінці може бути виявлено майже повну відсутність рішень систем управління командними проєктами загального призначення.

Тому наступним етапом має бути реалізація багатофункціональної інформаційної системи організації діяльності з можливістю соціальної взаємодії, яка включає в себе основні переваги сучасних рішень та була направлена на вирішення повсякденних питань. Під час початкового етапу реалізації для найбільшої ефективності слід побудувати базову модель застосунку, засновану на класичному server-client рішенні.

Важливим етапом даного проєкту є пошук та аналіз технологій для реалізації поставлених вимог до застосунку. Доцільно проаналізувати рішення баз даних та виявити основні переваги та відмінності між реляційними, документними та графовими базами даних. Також слід проаналізувати ринок мобільної та серверної розробки, де були відібрані потрібні бібліотеки та фреймворки. Не зайвим також є аналіз рішень програмних продуктів для проєктування, у ході якого можуть бути знайдено найоптимальніші рішення для даного проєкту.

Найважливішим кроком є проєктування архітектур серверної та мобільної частин. Під час даного кроку проєктується базове представлення застосунку за допомогою wireframes та базове представлення даних за допомогою ERD. Після цього має бути спроєктовано архітектуру Android-застосунку з дотриманням основних рекомендацій від Google та реалізацію патернів, а також архітектуру серверної частини за принципами SOLID та DDD.

Кінцевим етапом є побудова автоматизованої CI/CD системи розгортання за допомогою системи контейнеризації Docker, у результаті якої має бути спроєктовано архітектуру серверних з'єднань та придбано кластер віртуальних машин на AWS, а також сконфігуровано та розгорнуто серверну частину за допомогою конфігурацій в середовищі Docker-swarm.

Поставлена задача має бути повністю виконана та протестована, серверна та мобільна частина відповідатиме заздалегідь поставленим умовам, всі поставлені функції мають бути реалізовані та працювати відмінно, всі тести пройдені з позитивним результатом. Також має бути

проведено тестування на великі навантаження, щоб переконатися у відмовостійкості рішення, результати якого можна побачити на рис. 70.

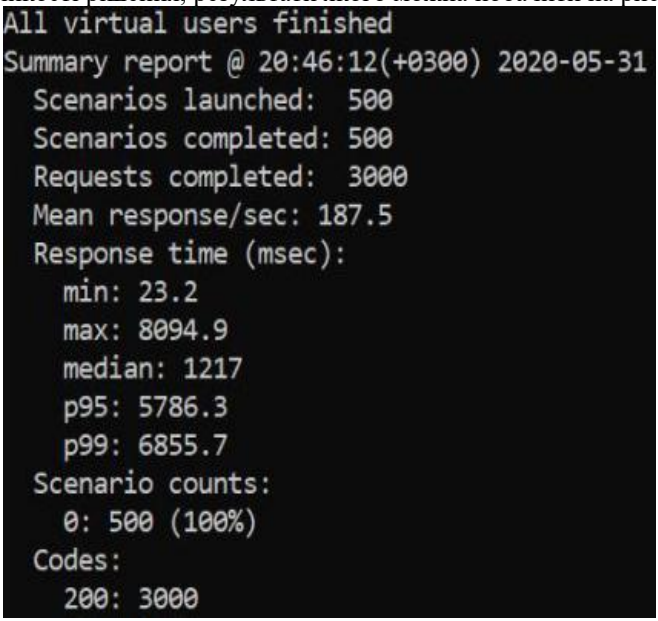


Рисунок 70 – High availability тестування серверної архітектури

У подальшому мобільну частину проєкту можна значно доопрацювати з боку продуктивності, а інтерфейс – поліпшити. Серверну частину можна розширити за допомогою сервісів RabbitMQ та Elasticsearch для покращення відмовостійкості та швидкості роботи, а всю серверну архітектуру перенести у кластер Kubernetes для автоматичного масштабування. Також можливо додавання деяких функцій у проєкт в цілому, таких як завантаження різноманітних файлів та Kanban-дошка, вбудований плеєр.

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. YouTrack: The Issue Tracking and Project Management Tool for Teams. URL : https://www.jetbrains.com/youtrack (Last accessed: 09.03.2025).
2. Explore apps for Jira. URL : https://marketplace.atlassian.com/product/jira (Last accessed: 09.03.2025).
3. 2-Way Integration for Trello and Jira. https://marketplace.atlassian.com/apps/1217147/2-way-integration-for-trello-and-jira?hosting=cloud&tab=overview (Last accessed: 09.03.2025).
4. Manage your team's work, projects, & tasks online. Asana. URL : https://asana.com (Last accessed: 09.03.2025).
5. PostgreSQL vs. SQL Server – Everything You Need to Know. URL : https://www.astera.com/knowledge-center/postgresql-sql-server/ (Last accessed: 09.03.2025).
6. Complete Guide: Inheritance strategies with JPA and Hibernate. URL : https://thorben-janssen.com/complete-guide-inheritance-strategies-jpa-hibernate/ (Last accessed: 09.03.2025).
7. Enum Mappings with Hibernate - The Complete Guide [Електронний ресурс]. – Режим доступу : URL : https://thorben-janssen.com/hibernate-enum-mappings/ (Last accessed: 09.03.2025).
8. SOLID: The First 5 Principles of Object Oriented Design. Publ. Apr. 24, 2024. URL : https://habr.com/ru/post/346016/ (Last accessed: 09.03.2025).
9. Навіщо використовують DTO. Приклади в Java-застосунках. Опубл. 12 серпня 2021 р. URL : https://dou.ua/forums/topic/34411/ дата звернення 09.03.2025).
10. Light Bootstrap Dashboard Angular: Free Bootstrap Admin Template @ Creative Tim. URL : https://www.creative-tim.com/product/light-bootstrap-dashboard-angular2 (Last accessed: 09.03.2025).
11. Angular powered Bootstrap – popover component. URL : https://ng-bootstrap.github.io/#/components/popover (Last accessed: 09.03.2025).
12. Docker Hub – postgres. URL : https://hub.docker.com/_/postgres (Last accessed: 09.03.2025).
13. Homepage – Material Design. URL: https://material.io/ (Last accessed: 09.03.2025).
14. Android Developers. URL: https://developer.android.com/ (Last accessed: 09.03.2025).
15. Griffiths Dawn, Griffiths David. Head First Android Development: A Brain-Friendly Guide. 2nd Ed. O'Reilly Media, August 19, 2017.

16. Young A. R., Meck B., Cantelon M., Oxley T., Harter T., et al. Node.js in Action 2nd Edition. Manning Publications, September 17, 2017.

17. Docs | Node.js. URL: https://nodejs.org/en/docs/ (Last accessed: 09.03.2025).

18. Despoudis T. TypeScript 5 Design Patterns and Best Practices. Packt Publishing, Limited, 2025. 424 p.

19. Arnold I. Enterprise Architecture Function. A Pattern Language for Planning, Design and Execution. Springer International Publishing, 2022. 521 p.

20. Refactoring and Design Patterns. URL: https://refactoring.guru/ (Last accessed: 09.03.2025).

21. Jones P. Mastering Docker Containers: From Development to Deployment. Walzone Press, 2025, 255 p.

22. Piper B., Clinton D. ·AWS Certified Solutions Architect Study Guide with Online Labs. Associate SAA-C02 Exam. Wiley, 2021. 464 p.

ДЛЯ НОТАТОК

ДЛЯ НОТАТОК

Виробничо-практичне видання

Методична серія. Випуск 463

Іван Сергійович БУРЛАЧЕНКО,
Володимир Юрійович САВІНОВ

СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

Методичні рекомендації
до виконання практичних та самостійних робіт

Редактор *О. Михайлова*
Комп'ютерна верстка, дизайн обкладинки *К. Гросу-Грабарчук*
Друк *С. Волинець*. Фальцювально-палітурні роботи *О. Мішалкіна.*

Підп. до друку 02.09.2025.
Формат 60х84 $^{1}/_{16}$. Папір офсет.
Гарнітура «Times New Roman». Друк різограф.
Ум. друк. арк. 6,2. Обл.-вид. арк. 2,4.
Тираж 50 пр. Зам. № 7020.

Видавець і виготовлювач: ЧНУ ім. Петра Могили.
54003, м. Миколаїв, вул. 68 Десантників, 10.
Тел.: 8 (0512) 50–03–32, 8 (0512) 76–55–81, e-mail: rector@chmnu.edu.ua.
Свідоцтво суб'єкта видавничої справи ДК № 6124 від 05.04.2018.