Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Центр післядипломної освіти
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Програмне забезпечення підтримки діяльності ОСББ.
Серверна частина
(тема)

Виконав:
студент 4 курсу, групи ПЗПІп-22-2

Севрюков О.Ю.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Програмна інженерія
(повна назва освітньої програми)

Керівник доц. Кириченко І.В.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ З.В.Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Центр післядипломної освіти
Кафедра програмної інженерії
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми Освітньо-професійна
Освітня програма Програмна Інженерія
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Севрюкову Олегу Юрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Програмне забезпечення підтримки діяльності ОСББ. Серверна частина.

Затверджена наказом по університету від 17.06. 2024р. № 588 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 20.07.2024

3. Вихідні дані до роботи Розробити програмний модуль який би реалізовував серверну частину програмного продукту підтримки діяльності ОСББ, а також забезпечував взаємодію з клієнтською частиною використовуючи ммову програмування Python.

4. Перелік питань, що потрібно опрацювати в роботі

Вступ, аналіз предметної галузі, формування вимог до програмної системи, архітектура та проектування програмного забезпечення, опис прийнятих програмних рішень, тестування розробленого програмного забезпечення, висновки, додатки.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі	20.05.2024	*виконано*
2	Створення специфікації ПЗ	22.05.2024	*виконано*
3	Проектування ПЗ	24.05.2024	*виконано*
4	Розробка ПЗ	28.06.2024	*виконано*
5	Тестування ПЗ	30.06.2024	*виконано*
6	Оформлення пояснювальної записки	05.07.2024	*виконано*
7	Підготовка презентації та доповіді	10.07.2024	*виконано*
8	Попередній захист	12.07.2024	*виконано*
9	Нормоконтроль, рецензування	12.07.2024	*виконано*
10	Здача роботи у електронний архів	12.07.2024	*виконано*
11	Допуск до захисту у зав. кафедри	18.07.2024	*виконано*

Дата видачі завдання 06 травня 2024р.

Студент (ка)_____ Севрюков О.Ю
(підпис)

Керівник роботи _____ доц. Кириченко І.В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до кваліфікаційної роботи бакалавра, 55 стор., 13 рис., 1 табл., 10 джерел.

API, DJANGO, DJANGO REST FRAEMWORK, PYCHARM, PYTHON

Об’єкт розробки – програмний продукт об’єднання співвласників багатоквартирного будинку.

Мета розробки – створення програмного продукту для об’єднання співвласників багатоквартирного будинку.з метою автоматизації та оптимізації управління фінансами, заявками та документами, забезпечення зручного взаємодії співвласників та управління з обмеженим витратами та підвищеною ефективністю.

Метод рішення – середовище розробки PyCharm, мова програмування Python.

У результаті розробки створено ігровий програмний продукт, котрий має можливість створювати заявки, керувати мешканцями, здійснювати оплату та оповіщати про заборгованості.

API, DJANGO, DJANGO REST FRAEMWORK, PYCHARM, PYTHON

The object of development is a software product of the association of co-owners of an apartment building.

The purpose of the development is to create a software product for the association of co-owners of an apartment building with the aim of automating and optimizing the management of finances, applications and documents, ensuring convenient interaction of co-owners and management with limited costs and increased efficiency.

The solution method is the PyCharm development environment, the Python programming language.

As a result of the development, a gaming software product was created, which has the ability to create applications, manage residents, make payments and notify about debts.

Я, Севрюков Олег Юрійович, студент гр. ПЗПІп-22-2, здобувач вищої освіти на першому (бакалаврському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Програмне забезпечення підтримки діяльності ОСББ.Серверна частина.», що буде представлена до екзаменаційної комісії для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElAr KhNURE. Усі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови до допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ОСББ – Об’єднання співвласників багатоквартирного будинку

БД – База даних

СУБД – Система управління баазами даних

API – Application Programming Interface

ER – Entity-Relationship

UML – Unified modeling language

SQL – Simple query language

DRF – Django Rest Framework

CI – Continius Integration

CD – Continius Deployment

ВСТУП

Кожне домогосподарство прагне ефективно та прозоро керувати своїми потребами, бо хто як не власники квартир та прибудинкової території можуть знати поточні проблеми господарств та шляхи їх вирішення та покращення умов проживання в будинку. Довгий час це було не можливо бо законодавство не дозволяло такої діяльності, але з розвитком громадськості та соціальної відповідальності держава впровадила такий вид юридичної особи як Об'єднання співвласників багатоквартирного будинку (ОСББ).

З виникненням ОСББ власники квартир почали об'єднуватися в та реєструвати юридичні особи, приймати активну участь в управлінні домогосподарства. Але разом з тим виникла нова проблема – ефективне управління та ведення бухгалтерського обліку, проведення зустрічей для обговорень та голосувань, ведення документації та інші повсякденні справи. Довгий час для цього використовували паперові зошити тощо, що не є зручним та ефективним.

Для вирішення вищенаведених проблем та збільшення ефективності роботи ОСББ було прийнято рішення створити програмний продукт який візьме на себе ці обов'язки а саме дозволить створювати заявки на ремонт та надання інших сервісів, дозволить вести бюджет ОСББ так що кожен мешканець завжди може перевірити на що витрачаються кошти, сплачувати рахунки, спілкуватися з сусідами, отримувати сповіщення та новини від управління ОСББ.

Програмне рішення буде реалізоване з використанням мов програмування і фреймворків Python , Django та React JS. Застосунок буде складатися з веб частини та серверної частини. Данні будуть зберігатися в PostgreSQL та хмарному сховищі. Середовище розробки PyCharm та Visual Studio Code.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі

Об’єднання співвласників багатоквартирного будинку (ОСББ) - юридична особа, створена власниками квартир та/або нежитлових приміщень багатоквартирного будинку для сприяння використанню їхнього власного майна та управління, утримання і використання спільного майна [1]. ОСББ є найбільш ефективною формою управління спільним майном бо кожен власник є зацікавленою та відповідальною особою.

ОСББ функціонує наступним чином:

- кожен мешканець може приймати участь в управлінні майном шляхом голосування або обравшись в раду ОСББ;
- ОСББ може долучати прибудинкову територію в сумісну власність для ведення господарської діяльності;
- всі доходи від діяльності та платежі мешканців зараховуються на рахунки ОСББ та використовуються на обслуговування будинку;
- ОСББ самостійно обирає підрядників для здійснення ремонтних робіт чи надання послуг;
- мешканці можуть шляхом голосування приймати рішення про необхідність та графіки робіт;
- фінансова діяльність ОСББ повністю відкрита для мешканців.

За останні 10 років ОСББ створювалися дуже активно а їхня загальна кількість налічує 38606 (за даними [2]) і продовжує збільшуватися кожного року що дає доволі великий простір для розвитку програмного продукту бо кожне ОСББ стикається з однаковими проблемами. Ці проблеми доволі відомі та легко вирішуються з використанням нашого програмного забезпечення.

Головною задачею дослідження було з’ясувати з якими проблемами найчастіше стикаються мешканці. Також визначення потреб та пріоритетності задач для мешканців будинку. Аналіз показав що проблеми діляться на проблеми управління та проблеми взаємодії мешканців з ОСББ. До популярних проблем мешканців ми можемо віднести такі як погано припарковані автомобілі та пошук

власника, відсутність єдиного уніфікованого шляху комунікації, труднощі з отриманням довідок, не прозорість фінансової звітності, складність організації голосувань та зборів. До проблем управління можна віднести такі як ведення документації, оповіщення мешканців, організація голосувань, створення та ведення заявок на роботи які стосуються ОСББ.

Наш продукт покликаний вирішити ці проблеми та буде надавати можливість сплачувати рахунки, отримувати інформацію по стану бюджету, проводити голосування, спілкуватися з сусідами, публікувати оголошення та новини, отримувати різного роду виписки, замовляти додаткові послуги, отримувати доступ до відео нагляду. Прибуток буде отримуватися на основі підписки та за рахунок реклами.

1.2 Виявлення та вирішення проблем та актуалізація рішень

В процесі дослідження були проаналізовані основні конкуренти нашого рішення. Ними виявилися такі програмні продукти: Дах, Омо, Мій дім, Моє ОСББ.

Веб застосунок «Дах» має доволі лаконічний дизайн який нагадує популярний застосунок Приват24 (див.рис.1) що є дуже вигідним рішенням бо це є державний банк яким користуються громадяни України, а тому всі верстви населення так чи інакше знайдуть дизайн знайомим. «Дах» надає наступні послуги: ведення бухгалтерського обліку, сплата рахунків, робота з боржниками, розсилка квитанцій, месенджер та онлайн збори.

До переваг застосунку можна віднести;

- ведення бухгалтерського обліку;
- сплата рахунків;
- робота з боржниками;
- розсилка квитанцій;
- месенджер;
- онлайн збори.

До недоліків застосунку можна віднести;

- відсутність створювати нові послуги;

– відсутність гостьового доступу для орендарів;

– відсутність інтеграцій з охоронними системами.

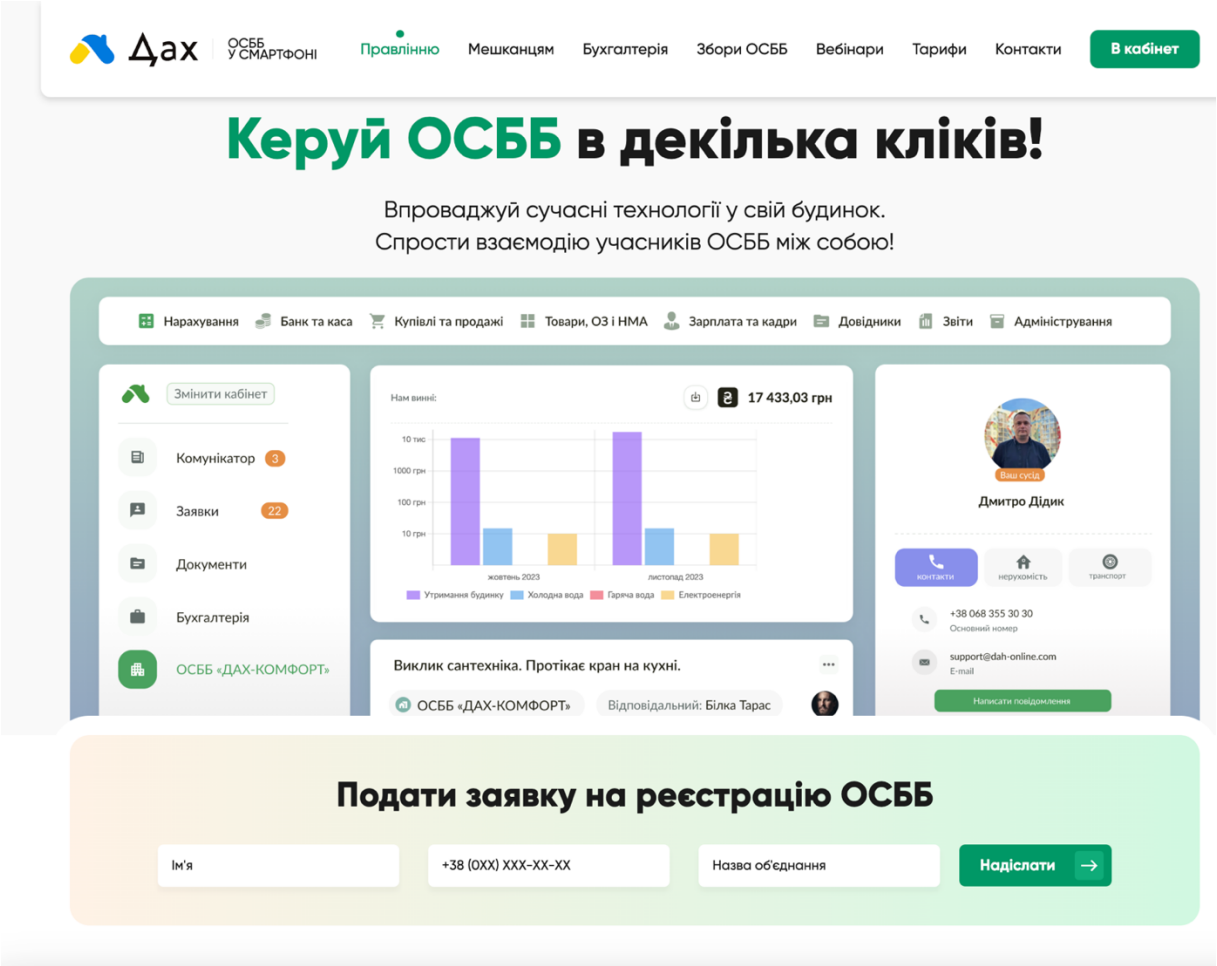


Рисунок 1 – Інтерфейс застосунку Дах[3]

Метою застосунку «ОМО» (див.рис.2) є організація та створення розумного будинку. Він дозволяє інтегрувати мешканців та будинок до системи розумного будинку та надає можливість віддаленого керування будинком. Цей продукт не є прямим конкурентом але так як ми маємо амбіції реалізувати цей фунціонал в нашому застосунку в майбутньому то мі маємо його розглянути.

До переваг застосунку можна віднести:

– інтеграція та впровадження розумного будинку;

– впровадження систем охорони.

До недоліків застосунку можна віднести:

– відсутній будь який фунціонал пов’язаний з управлінням ОСББ.

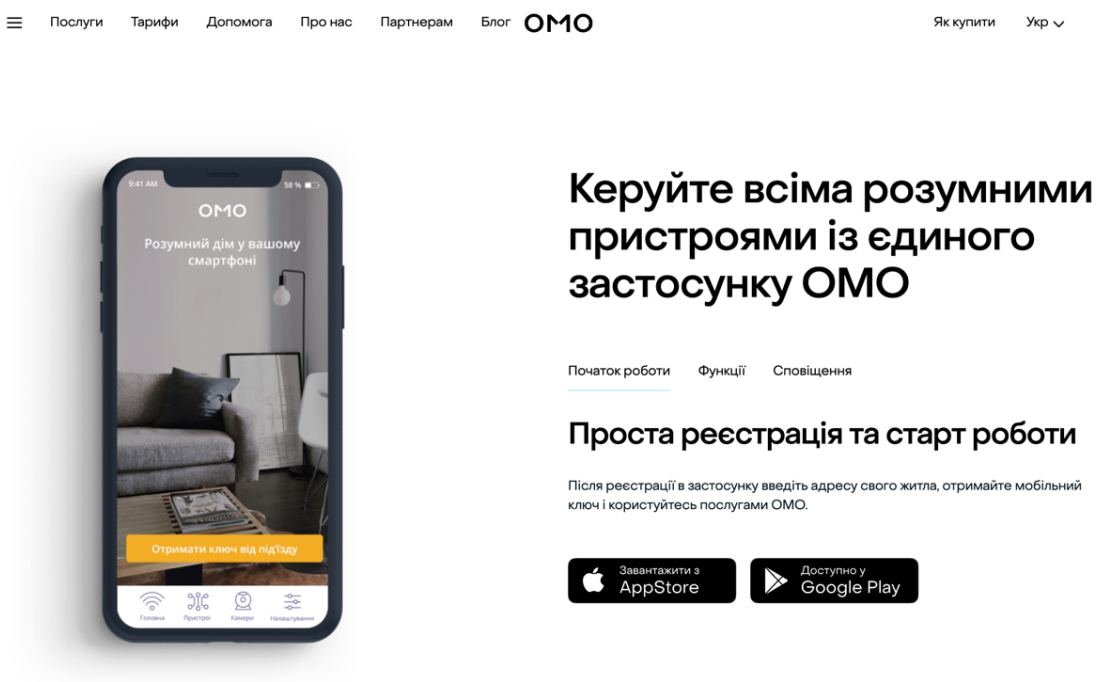


Рисунок 2 – Інтерфейс застосунку OMO [4]

«Моє ОСББ» (див.рис.3) вперш за все має на меті спростити керування ОСББ та підвищити прозорість роботи керуючої компанії за рахунок надання мешканцям доступу до фінансової інформації.

До переваг застосунку можна віднести:

– ведення бухгалтерського обліку;
– сплата рахунків;
– робота з боржниками;
– розсилка квитанцій.

До недоліків застосунку можна віднести:

– відсутність створювати нові послуги;
– відсутність гостьового доступу для орендарів;
– відсутність інтеграцій з охоронними системами;
– відсутність чатів для мешканців;
– відсутність онлайн зборів;

– відсутність голосування.

Рисунок 3 – Дизайн застосунку Моє ОСББ[5]

Продукт «Мій Дім» (див.рис.4) ми вважаємо головним конкурентом бо він має всі ті функції які ми плануємо запровадити. Він дозволяє розділяти доступи для співвласників, вести фінансову звітність, отримувати квитанції та виписки, вести комунікацію між мешканцями.

До переваг застосунку можна віднести:

– ведення бухгалтерського обліку;

– сплата рахунків;

– робота з боржниками;

– розсилка квитанцій;

– можливість створення заявок на роботи.

До недоліків застосунку можна віднести:

– відсутність створювати нові послуги;

– відсутність гостьового доступу для орендарів;

– відсутність інтеграцій з охоронними системами;

– відсутність онлайн зборів;

– відсутність голосування.

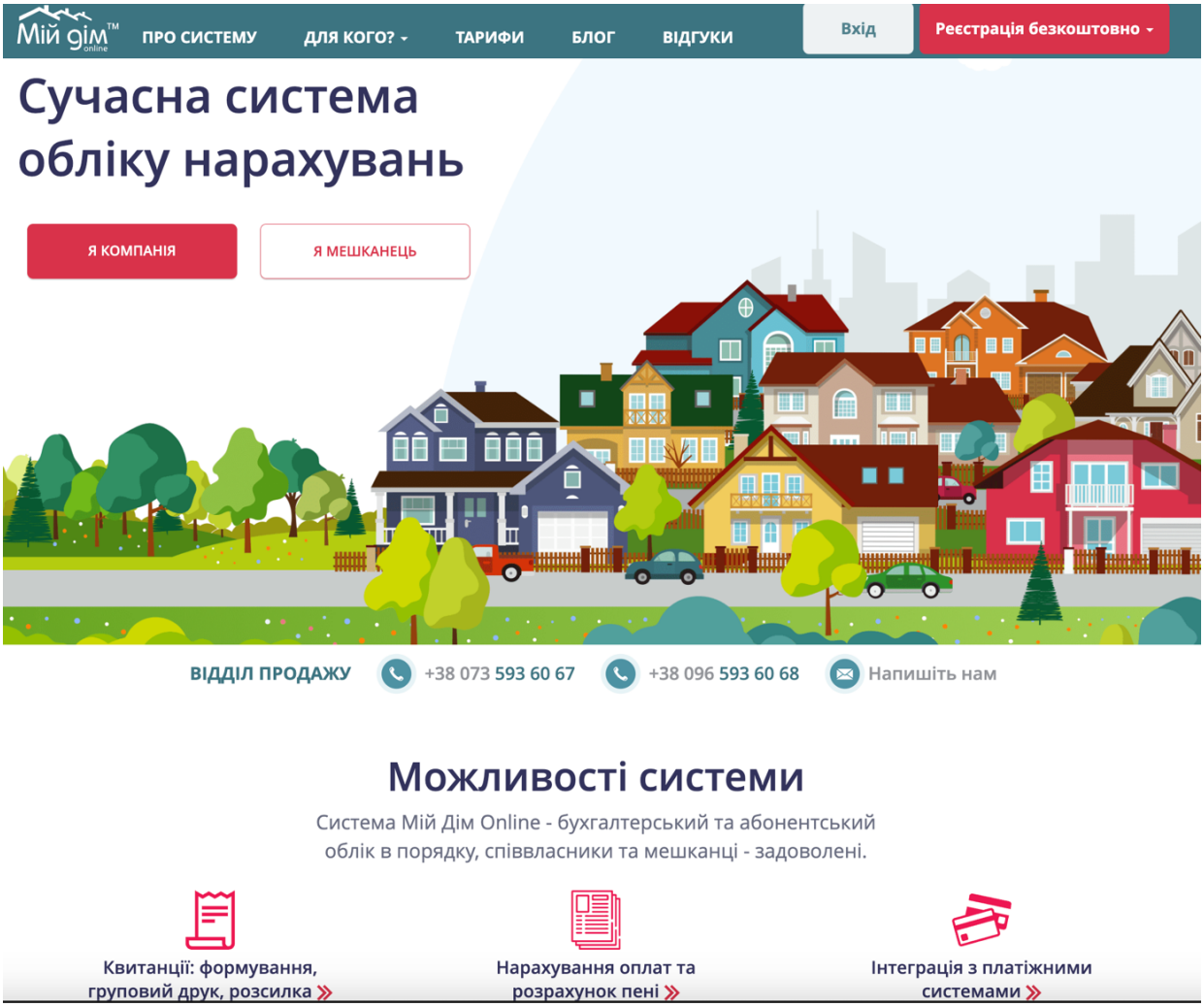


Рисунок 4 –Застосунок Мій дім [6]

Наступним кроком є порівняння всіх конкурентів з нашим продуктом за наступними критеріями:

– додавання запитів на “свої” послуги (А);

– ведення бухгалтерського обліку (Б);

– система голосування (В);

– месенджер (Г);

– звітність (Д);

– інтеграція з розумним будинком (Е);

– створення заявок на роботи (Ж).

Інформацію про порівняння наведено в таблиці 1.

Таблиця 1 – Порівняння програмних продуктів (таблиця виконана самостійно)

Програмний продукт	А	Б	В	Г	Д	Е	Ж
Мій Дім	-	+	-	+	+	-	+
ОМО	+	-	-	-	-	+	-
Моє ОСББ	-	+	-	+	-	-	-
Дах	-	+	-	+	+	-	-

Кожний програмний продукт має свої переваги та недоліки, але виходячи з аналізу функціонала конкурентів ми чітко прослідковуємо напрямки які можуть надати однозначну перевагу нашого продукту над конкурентами, а саме система внутрішньо домових чатів між мешканцями, створення заявок на роботи, онлайн голосування.

1.3 Постановка задачі

Для вирішення потреб управляючих компаній та мешканців потрібно розробити програмний продукт який надавав би наступні сервіси:

- ведення фінансового обліку;
- спілкування між мешканцями та керуючою компанією;
- спілкування між мешканцями;
- отримання новин ОСББ;
- замовлення робіт;
- сплата за послуги:
- отримання фінансових звітів;
- замовлення додаткових послуг.

Спілкування має бути організовано у вигляді месенджеру який дозволяв би створювати чат між мешканцем та мешканцем, між мешканцями будинку, поверху, та між мешканцями та керуючою компанією.

Голосування мають проводитися в онлайн формі та в ньому можуть приймати участь тільки власники квартир.

Продукт має мати багаторівневу систему прав яка дозволяла би реалізувати різні права у користувачів та доступ до різного функціоналу в залежності від ролі користувача.

Для користувачів продукту мають надаватися фінансові звіти у вигляді графіків та має бути можливість фільтрувати звітність за певний період часу.

Основною цільовою аудиторією будуть:

- люди, які хочуть створити ОСББ або доєднатися до існуючого;
- керуючі компанії;
- люди від 18 до 80 років які мають квартиру;
- люди різної статі;
- люди різної професії;

Тобто програмний продукт розрахований на максимально широку аудиторію.

За умови успішного впровадження та реклами продукту, кількість користувачів може зростати, оскільки співвласники будинків будуть шукати зручні інструменти для управління своїми обов'язками і фінансами.

Продукт має містити можливість розширення функціоналу за рахунок додавання нових корисних інструментів, таких як інтеграція зі сторонніми сервісами для покращення обслуговування, аналітика для планування бюджету тощо.

Успішне впровадження може відкрити можливості для масштабування вашого продукту на інші ринки або в інші сегменти житлового сектору.

2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

Програмний продукт буде створено у вигляді веб застосунку. Він має працювати в наступних браузерах Google Chrome та має виконувати Закон України «Про захист персональних даних» от 01.06.2010 № 2297-VI.

Програмний продукт має використовувати наступні технології:

- Python;
- Django rest framework;
- PostgreSQL;
- HTML/CSS

Продукт має містити всю інформацію про ОСББ, його керуючих, всіх проживаючих в будинку та прописаних в квартирі. Надаватиме бухгалтерські послуги, виконуватиме розсилки зі сповіщеннями про заборгованість, нагадування про сплату комунальних послуг та новини будинку.

Продукт має давати змогу замовити та сплатити додаткові послуги такі як прибирання, ремонт, замовлення питної води та інші.

Має бути реалізований механізм спілкування у вигляді чату для мешканців будинку і для спілкування з керуючими ОСББ. Також продукт має надавати можливість проводити голосування онлайн на рівні ОСББ.

Основні функціональні вимоги продукту:

- додавання, редагування та перегляд рахунків для співвласників;
- можливість автоматичної генерації рахунків на основі показників споживання послуг (вода, електроенергія тощо);
- нагадування про невиконані платежі та можливість налаштування автоматичних платежів;
- створення та відстеження заявок та звернень співвласників (ремонтні роботи, запити на інформацію тощо);
- повідомлення про статус заявок та можливість коментування для взаємодії з управлінцями;

- генерація звітів про фінансовий стан ОСББ (залишки на рахунках, доходи та витрати);
- аналітичні звіти про споживання комунальних послуг та ефективність витрат;
- завантаження та збереження важливих документів (статут ОСББ, рішення загальних зборів, договори з постачальниками);
- налаштування доступу до документів згідно з ролями користувачів (співвласники, управлінці, бухгалтер тощо).

Повинно бути реалізовано розділення за рівнем доступу до продукту. Кожен користувач має різний рівень доступу в залежності від його ролі, а саме:

а) співвласники будинків:
   1) профіль: фізичні особи або юридичні особи, які є власниками квартир у багатоквартирному будинку;
   2) потреби: можливість перегляду та оплати рахунків, створення заявок на ремонт та інші послуги, доступ до важливих документів та звітності;

б) управлінці ОСББ:
   1) профіль: представники управління ОСББ, які відповідають за організацію роботи та взаємодію з співвласниками;
   2) потреби: керування фінансами та бюджетом ОСББ, обробка заявок та робота з документами;

в) бухгалтери та фінансові аналітики:
   1) профіль: спеціалісти, які відповідають за фінансову діяльність та аналіз ОСББ;
   2) потреби: генерація фінансових звітів, контроль за оплатами та розрахунками;

г) технічна підтримка:
   1) профіль: IT-спеціалісти або підтримка продукту, які вирішують технічні проблеми та підтримують роботу платформи;
   2) потреби: вирішення технічних питань, підтримка користувачів та оновлення програмного забезпечення;

д) адміністратори:

1) профіль: адміністратори, які відповідають за налаштування та безпеку платформи;
2) потреби: керування правами доступу, моніторинг безпеки та підтримка роботи платформи.

До загальних обмежень продукту відносяться:

– забезпечення конфіденційності та безпеки особистих даних користувачів та фінансової інформації ОСББ. Важливо дотримуватися стандартів безпеки даних та захисту від несанкціонованого доступу;
– можливість інтеграції з іншими системами, такими як банківські платіжні системи, системи управління комунальними послугами тощо, для забезпечення повноцінного функціоналу продукту.

3 АРХІТЕКТУРА ТА ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 UML проектування ПЗ

Під час розробки була створена діаграма розгортання яка демонструє загальний вигляд компонентів продукту (див. рис. 5).

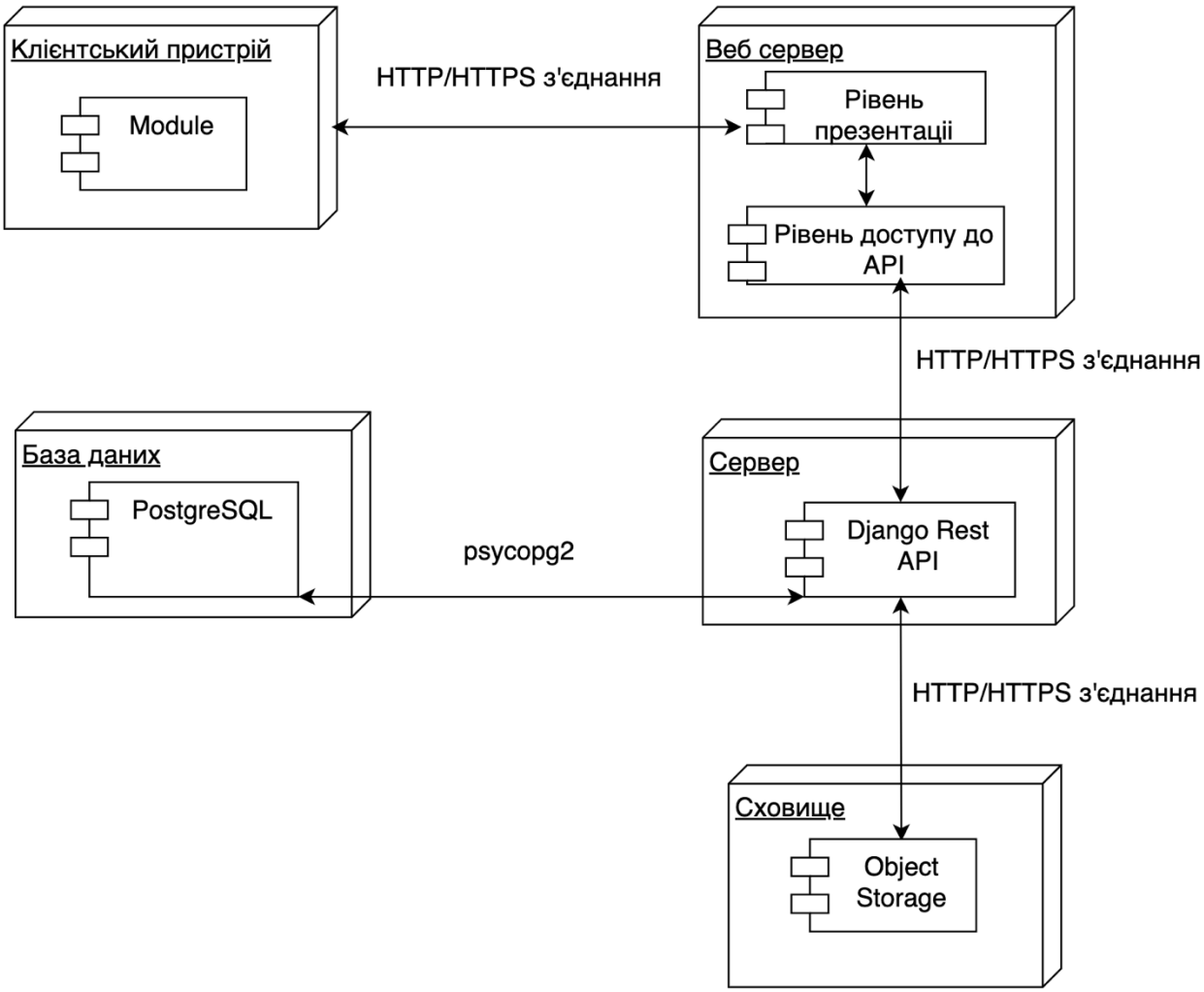


Рисунок 5 – Діаграма розгортання (рисунок виконаний самостійно)

Продукт складається з серверної частини реалізованої за допомогою мови програмування Python та фреймворку Django REST framework яка в свою чергу взаємодіє з базою даних PostgreSQL за допомогою psycopg2. Таблиці в базі даних (БД) створюються за допомогою Django моделей та реалізовані в коді у вигляді об’єктів.

Для зберігання фалів, фотографій та інших даних використовується хмарне сховище. Таким чином ці данні надійно зберігаються і дозволяють нам не витрачати простір на сервері та мати змогу використовувати масштабування як горизонтальне так і вертикальне без втрати даних.

В якості веб частини виступає програмний застосунок реалізований з допомогою ReactJS та HTML. Веб частина спілкується з серверною за допомогою HTTP/HTTPS запитів до REST API.

Однією з основних функціональних можливостей продукту є розсилка електронних листів боржникам на початку кожного календарного місяця. Для візуалізації було розроблено діаграму послідовності яка відображає цей процес (див.рис.6).

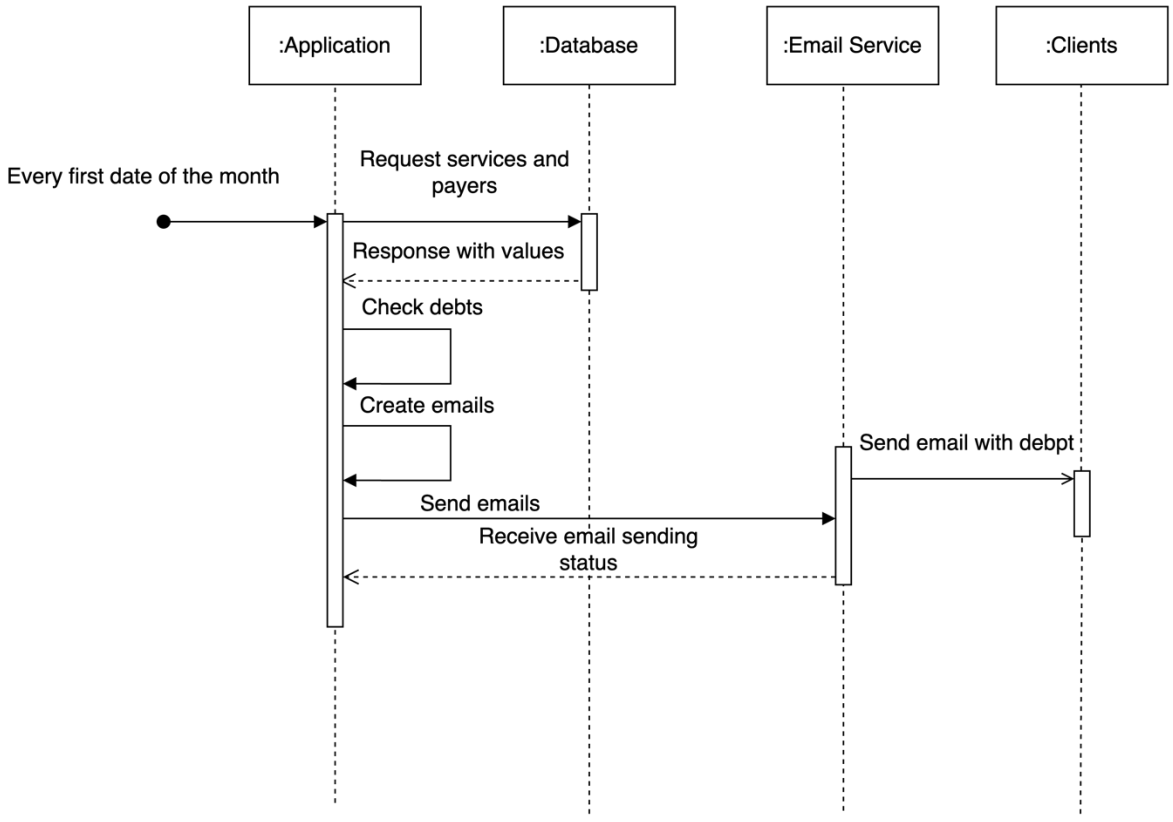


Рисунок 6 – Діаграма послідовності перевірки боргів та сповіщення (рисунок виконаний самостійно)

Діаграма послідовності зображує часові особливості передачі і прийому повідомлень об’єктами. Це є тільки послідовність дій впродовж життя об’єкту. На діаграмі відображено що при отриманні запиту на application programming interface

(API) серверу застосунок має звернутися до БД та отримати данні стосовно всіх мешканців та їх поточного стану розрахунку. Після чого відбувається перевірка на наявність заборгованості та формування електронного листа зі сповіщенням. Як тільки лист буде сформовано він буде надісланий за допомогою simple mail transfer protocol (SMTP) да доставлений користувачу. Також слід звернути увагу що ця операція не навантажує БД та не тримає відкритим підключення так як данні ми отримуємо за один запит.

Більш детально алгоритм перевірки боргів та сповіщення боржників відображений на діаграмі діяльності для (див.рис.7)

Рисунок 7 – Діаграма діяльності перевірки боргів та сповіщення (рисунок виконаний самостійно)

Діаграма діяльності ілюструє алгоритм дій необхідних для того щоб сформувати список боржників та надіслати їм сповіщення. Як видно з діаграми

функція використовує цикл для того щоб сформувати листа який буде включати всі заборгованості по всіх послугах якими користувався клієнт. У разі якщо заборгованостей немає то дія закінчується не надсилаючи електронного листа.

Другим важливим функціоналом продукту є функціонал замовлення послуг та їх оплата. Послугою може бути будь яка послуга яку надає програмний продукт, наприклад - замовлення питної води, прибирання тощо. UML діаграма послідовності для цього функціоналу описує кроки які треба виконати для оформлення та оплати замовлення (див.рис.8).

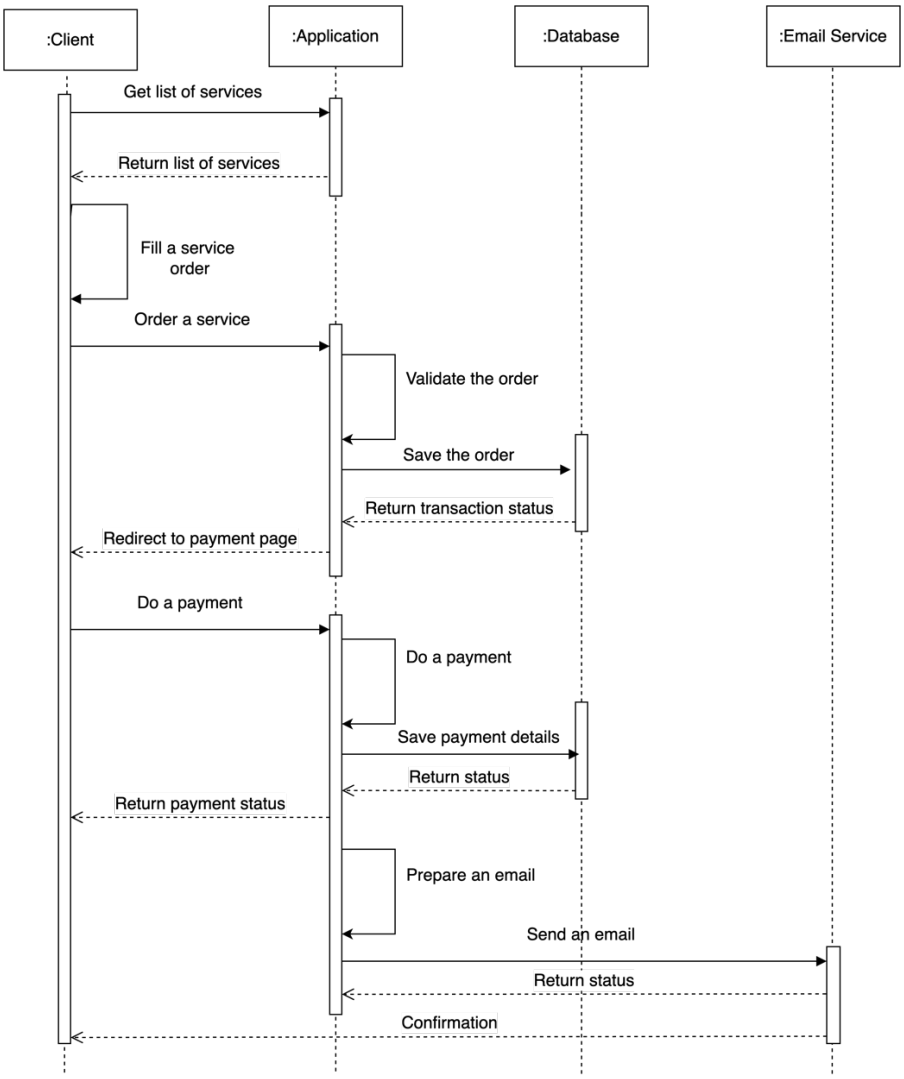


Рисунок 8 – Діаграма послідовності оформлення та оплати замовлення (рисунок виконаний самостійно)

Для виконання замовлення клієнт має перейти до відповідної форми замовлень. У формі замовлень необхідно заповнити всі поля замовлення та натиснути кнопку підтвердження. Сервер перевірить валідність заповненої форми та у разі її валідності збереже замовлення до БД та поверне команду переходу до сторінки оплати. Після Сплати замовлення клієнтом сервер збереже данні які стосуються оплати до БД та відправить електронним листом підтвердження про успішність оплати. Більш детально логіку дій ми можемо побачити на UML діаграмі діяльності (див.рис. 9).

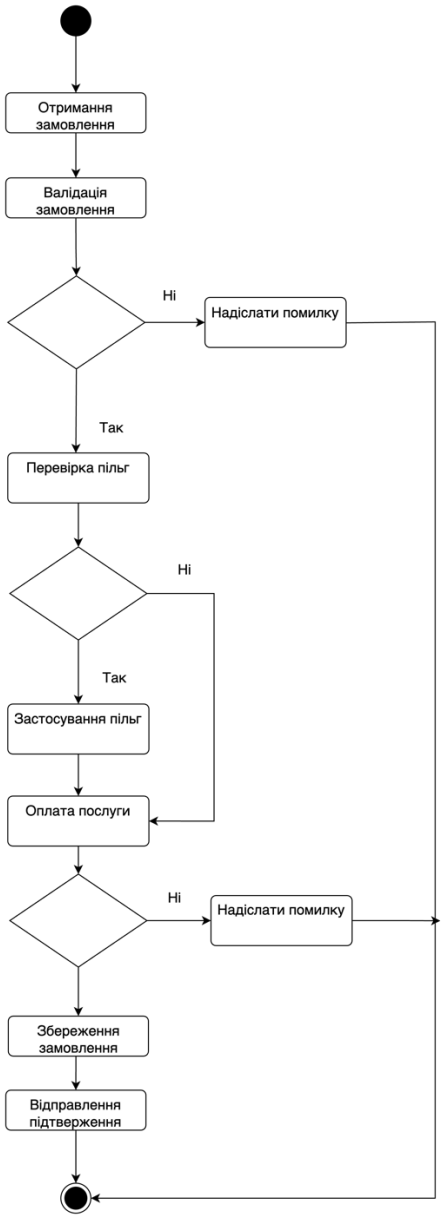


Рисунок 9 – Діаграма діяльності оформлення та оплати замовлення (рисунок виконаний самостійно)

Як видно з діаграми функція обробляє помилки та має можливість вносити зміни до ціни замовлення якщо мешканець має якісь знижки чи пільги.

3.2 Проектування архітектури програмної системи

Для побудови продукту буде використана мікро сервісна архітектура. Мікро сервісна архітектора – це така архітектура при якій компоненти система розбивається на малі компоненти які взаємодіють між собою за допомогою API та можуть розгортатися та масштабуватися незалежно одне від одного.

Продукт буде складатися з фронтенд частини та бекенд частини. Бекенд буде надавати API для взаємодії з фронтендом та надаватиме змогу розширення функціоналу за рахунок можливості розширення API. Реалізація бекенду буде виконана за допомогою Python та Django rest framework.

Кожен компонент буде розгорнутий за допомогою Docker що буде давати можливість швидкого розгортання та масштабування.

Бекенд буде використовувати з PostgreSQL для збереження даних та хмарне сховище для збереження мультимедійних даних.

На основі опитування серед різних груп людей що поділялися за віком , професією та статтю був розроблений дизайн для продукту що задовольнив би більшість вимог користувача (див. рис. 10).

3.3 Проектування структури зберігання даних

В якості БД було обрано PostgreSQL. Ця система управління базами даних (СУБД) є об’єктно-реляційною базою даних яка використовую мову запитів simple query language (SQL) для маніпулювання даними. Вона є доволі популярною та легко масштабується, має велику надійність да гарну документацію, також підтримується усіма мовами програмування.

Під час проектування БД було створено entity relationship (ER) діаграму системи (див.рис.11)

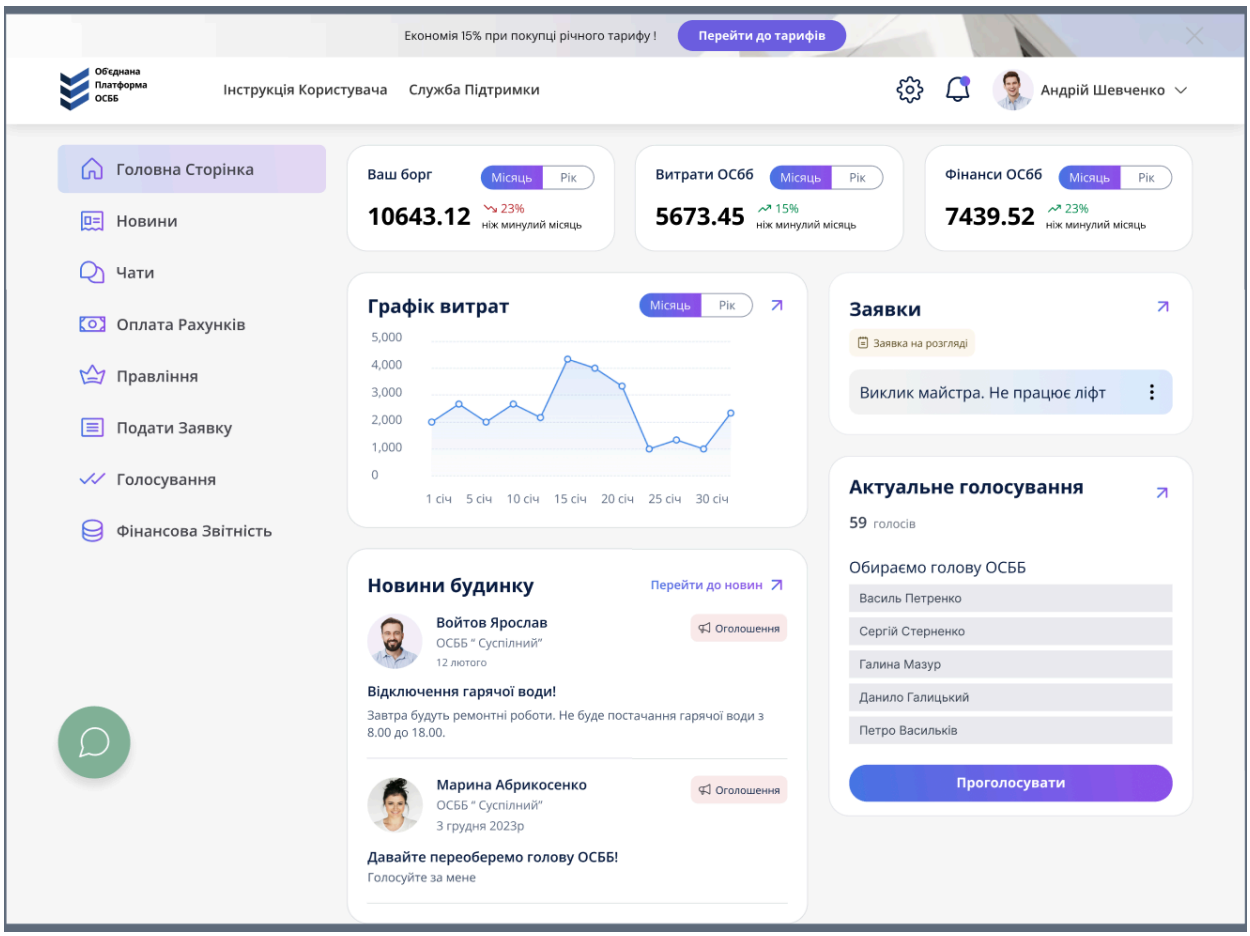


Рисунок 10 – Дизайн головної сторінки користувача (Виконано самостійно)

БД складається з 19 таблиць. Кожна таблиця містить необхідні поля для роботи з даними, розглянемо ці таблиці.

Таблиця «ОСББ» містить в собі інформацію про керуючу компанію.

Таблиця «Будинок» містить в собі інформацію про будинок доєднаний до керуючої компанії.

Таблиця «Квартира» містить інформацію про квартиру та її мешканців, в тому числі власників чи орендаторів.

Таблиця «Послуги» містить перелік послуг які доступні мешканцям.

Таблиця «Статус послуги» містить перелік статусів які можуть бути використані для послуги.

Таблиця «Ціна» містить ціну та проміжок часу за який ця ціна є актуальною.

Таблиця «Оплата» містить інформацію про оплату послуги.

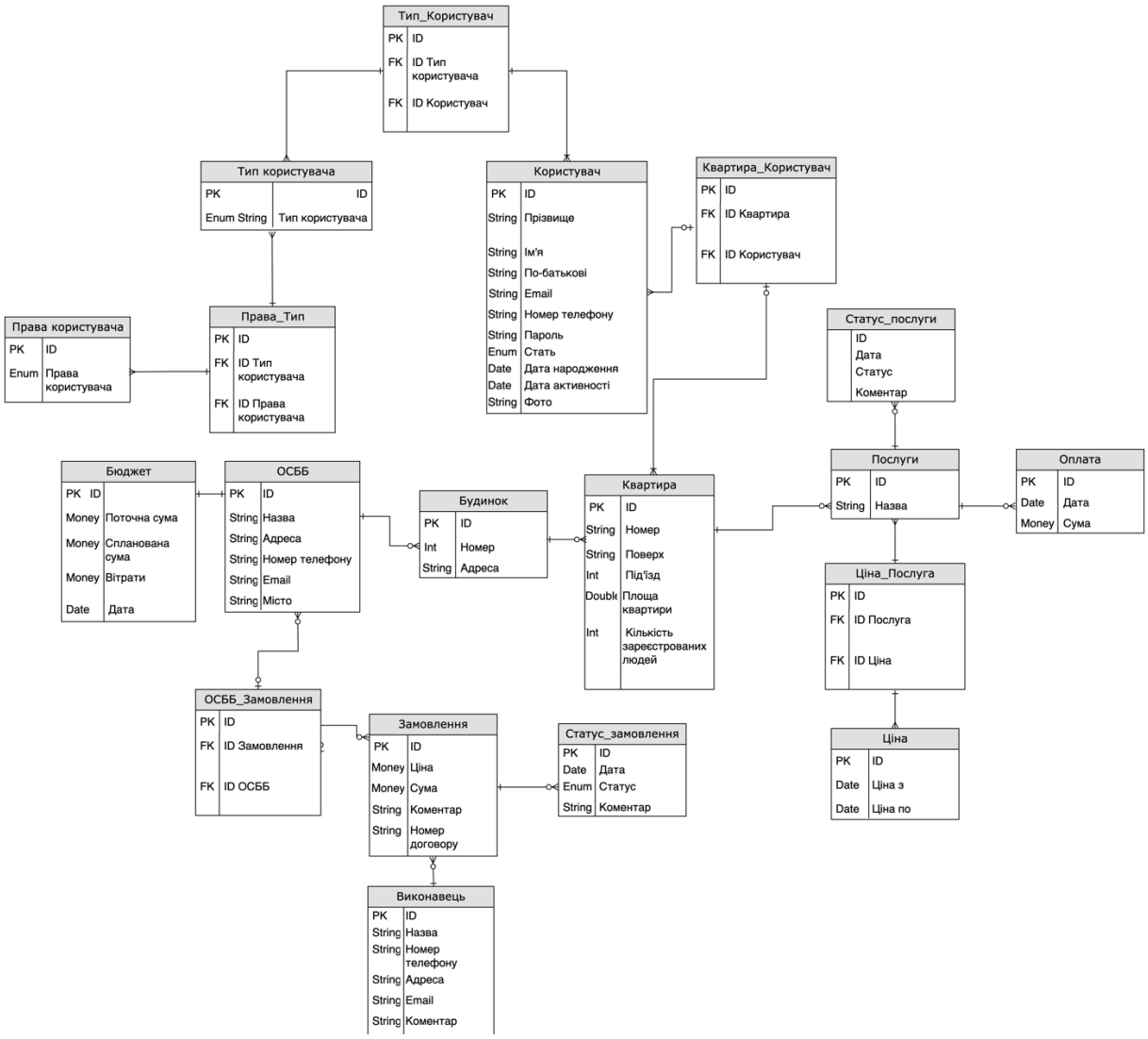


Рисунок 11 – ER діаграма системи (рисунок виконаний самостійно)

Таблиця «Користувач» містить інформацію про мешканця або користувача.

Таблиця «Тип користувача» містить інформацію про роль користувача чи то мешканець, орендатор тощо.

Таблиця «Права користувача» містить інформацію про права які має користувач.

Таблиця «Замовлення» містить інформацію про замовлення яке робить ОСББ.

Таблиця «Виконавець» містить інформацію про виконавця замовлення.

Таблиця «Статус замовлення» містить інформацію про статус замовлення.

Таблиця «Бюджет» містить інформація про поточний стан рахунку ОСББ.

В схемі БД наявні наступні зв’язки:

Зв’язок ОСББ – Будинок. ОСББ може мати багато будинків тому 1:Б.

Зв’язок ОСББ-Бюджет. ОСББ може мати один бюджет тому 1:1.

Зв’язок ОСББ-Замовлення. Б-Б виконується за допомоги проміжної таблиці ОСББ_Замовлення.

Зв’язок Замовлення-Виконавець. Б:1 бо виконавець може мати багато замовлень.

Зв’язок Замовлення-Статус. Замовлення може мати різні статуси тому 1:Б.

Зв’язок Будинок-Квартира. Будинок може мати багато квартир але квартира тільки один будинок тому 1:Б.

Зв’язок Квартира-Послуги. Багато послуг можуть бути оформлені на одну квартиру 1:Б.

Зв’язок Послуги-Статус Послуги. Кожна послуга може мати багато статусів 1:Б.

Зв’язок Послуги-Оплата. Кожна послуга може мати багато оплат 1:Б.

Зв’язок Послуги-Ціна. Кожна послуга може мати різну ціну в той час як ціна може бути актуальна для різних послуг. Б:Б виконуються за допомогою проміжної таблиці Ціна-Послуга.

Зв’язок Квартира-Користувач. Кожна квартира може декілька користувачів так само кожен користувач може бути власником різних квартир. Б:Б виконуються за допомогою проміжної таблиці Квартира_Користувач.

Зв’язок Користува-Тип користувача. Користувач може мати різні ролі так само як різні ролі можуть бути надаі багатьом користувачам Б:Б. Виконується за допомогою проміжної таблиці Тип_Користувач.

Зв’язок Тип користувача – Права користувача. Б:Б. Виконується за допомогою проміжної таблиці Тип_Права бо у одного типу можуть бути різні права так само і теж саме у зворотному напрямку.

Django підтримує структуру БД за допомогою вбудованого механізму варіювання для таблиць та міграцій. Django відслідковує зміни в моделях та створює нову міграцію якщо зміни є. Після створення міграція застосовується до БД. Для зберігання міграцій використовуються файли з розширенням «.ру». Таким

чином, Django забезпечує верифікацію бази даних через систему міграцій, що дозволяє легко відслідковувати зміни в моделях і гарантує, що структура бази даних завжди відповідає визначеним моделям.

3.4 Приклад методів та алгоритмів

Кожен раз при створенні замовлення йому автоматично виставляється статус «Очікує оплати». Після проведення оплати перевіряється чи покриває ця оплата всю вартість сервісу чи це є часткова оплата, якщо покриває всю то встановлюються статус «Сплачено», якщо ні то статус залишається незмінним але додається коментар до статусу про часткову оплату.

Даний фунціонал реалізується за допомогою допоміжної функції яка змінює поведінку серіалізатора під час POST запиту. Приклад функції наведено нижче:

```
def update_order_status(self):
    total_service_cost = sum([os.service.prices.latest('date').price for os
in self.orderservice_set.all()])
    total_payments = sum([payment.sum for payment in self.payments.all()])

    if total_payments >= total_service_cost:
        status = OrderStatus.Status.PAID
        OrderStatus.objects.update_or_create(
            order=self,
            defaults={'status': status, 'comment': 'Payment completed.',
'date': timezone.now()}
        )
    else:
        status = OrderStatus.Status.AWAITING_PAYMENT,
        OrderStatus.objects.update_or_create(
            order=self,
            defaults={'status': status, 'comment': 'Awaiting full
payment.', 'date': timezone.now()}
        )
```

Кожного першого дня місяця перевиряється чи є заборгованності по замовленням і якщо є не сплачені то формується лист в якому вказується які замовлення не сплачені та надсилається мешканцям квартири в якої є борги. Приклад фунцій наведений нижче:

```
def get_user_emails_by_apartment(apartment_id):
    users = Users.objects.filter(apartment_id=apartment_id)
    emails = [user.email for user in users if user.email]
```

```
    return emails

def check_and_notify_overdue_payments():
    overdue_orders = Orders.objects.filter(statuses__status='Очікує
оплати', statuses__date__lt=timezone.now())
    overdue_orders_by_apartment = defaultdict(list)
    for order in overdue_orders:
        if order.apartment:
overdue_orders_by_apartment[order.apartment.id].append(order)
    for apartment_id, orders in overdue_orders_by_apartment.items():
        send_payment_notification(orders)

def send_payment_notification(orders):
    apartment_id = orders[0].apartment.id
    emails = get_user_emails_by_apartment(apartment_id)
    order_details = ""
    for order in orders:
        services = order.orderservice_set.all()
        pprint(services)
        service_names = ", ".join([os.service.name for os in services])
        order_details += f"- {service_names}\n"
    message = f"Your payment for the following orders is overdue. Please
make your payment immediately:\n\n{order_details}"
    send_mail(
        'Payment Overdue Notice',
        message,
        'from@example.com',  # Replace with your "from" email address
        emails,
        fail_silently=False,
    )
```

Функція check_and_notify_overdue_payments() є точкою входу де виконується запит до БД та за допомогою фільтру отримуються всі замовлення які не мають оплати в дату виконання запита. Після чого формується список квартир до яких належать ці замовлення. Для кожної квартири викликається функція send_payment_notification() яка формує листа та надсилає його за допомогою email.

Данний функціонал реалізований за допомогою планувальника задач Celery та допоміжних функцій. Celery це асинхронна черга задач яка реалізована на Python. Вона відслідковує задачі та виконує та їх обробники, та виконує їх за розкладом. Задачі зберігаються в Redis та відслідковуються обробниками. Redis це розподіленне сховеще, яке зберігає данні в оперативній пам’яті у вигляді пар ключ та значення. Приклад планувальника наведений нижче:

```
from celery import shared_task
from .utils.payment_utils import check_and_notify_overdue_payments
```

```
@shared_task
def scheduled_payment_check():
    check_and_notify_overdue_payments()
```

В даному прикладі ми оголошуємо задачу, яку Celery має виконати, а саме виконати функцію check_and_notify_overdue_payments(). Розклад виконуваної задачі знаходиться в налаштуваннях Celery. Фрагмент коду наведений нижче:

```
@app.on_after_configure.connect
def setup_periodic_tasks(sender, **kwargs):
    sender.add_periodic_task(
        crontab(hour=15, minute=0, day_of_month='1'),
        check_and_notify_overdue_payments.s(),
    )
```

Як видно з коду, задача має запускатися кожного 1-го числа кожного місяця о 15 годині дня. Даний функціонал дозволяє робити автоматичні розсилки користувачам та надавати їм відомості по боргах за допомогою електронного листа.

В свою чергу користувач або керуюча компанія можуть перевірити всі замовлення які були сплачені за обраним проміжком часу. За цей функціонал відповідає контролер відображення. Приклад наведений нижче:

```
class PaidOrdersView(generics.ListAPIView):
    serializer_class = OrderSerializer

    def get_queryset(self):
        # Get date range from request parameters
        date_from = self.request.query_params.get('date_from')
        date_to = self.request.query_params.get('date_to')

        # Convert date strings to datetime objects
        if date_from:
            date_from = datetime.strptime(date_from, '%Y-%m-%d')
        else:
            date_from = timezone.now() - timezone.timedelta(days=365)  #
Default to last year if not specified

        if date_to:
            date_to = datetime.strptime(date_to, '%Y-%m-%d')
        else:
            date_to = timezone.now()  # Default to now if not specified

        # Filter orders that have been paid within the specified date range
        queryset = Orders.objects.filter(
            payments__date__gte=date_from,
            payments__date__lte=date_to
```

```
        ).distinct()

        return queryset
```

В данному прикладі використовується метод об’єкту моделі filter() який приймає два параметра payments__date__gte та payments__date__lte. Ці параметри оголошують як саме треба фільтрувати дані, а саме gte більше ніж або дорівнюю даті та lte менше ніж або дорівнюю даті. Таким чином за допомогою API запита до сервера api/v1/orders/paid-orders/?date_from=2022-01-1&date_to=2025-01-01 сервер поверне всі сплачені послуги в заданому проміжку часу.

4 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ

4.1 База даних

Django взаємодіє з БД за допомого адаптерів. Для роботи з Postgresql був обраний Psycopg адаптер. Psycopg це найпопулярніший адаптер бази даних PostgreSQL для мови програмування Python. Його головними особливостями є повна реалізація специфікації Python DB API 2.0 і безпека потоків (кілька потоків можуть спільно використовувати одне з'єднання). Його розроблено для інтенсивних багатопоточних програм, які створюють і знищують багато курсорів і роблять велику кількість одночасних INSERT або UPDATE [7].

Для створення таблиць використовуються Django моделі що є класом який наслідує клас Models з пакету django.db.models. Кожне поле моделі відповідає стовбцю таблиці. Унікальний ідентифікатор (id) буде створюватися автоматично та з параметром auto increment. Тобто це поле не треба описувати в моделі бо така поведінка йде за замовчуванням що гарантує доступ саме до необхідних даних та не перенавантажує код моделі зайвими полями. Фрагмент моделі зображено нижче.

```
from django.db import models
from django.utils.translation import gettext_lazy as _
from django.contrib.auth.hashers import make_password, check_password
from django.contrib.auth.models import User

class Users(models.Model):
    class Sex(models.TextChoices):
        MALE = 'M', _('Male')
        FEMALE = 'F', _('Female')

    user = models.OneToOneField(User, on_delete=models.CASCADE)
    firstname = models.CharField(max_length=100)
    lastname = models.CharField(max_length=100)
    surname = models.CharField(max_length=100)
    phone = models.CharField(max_length=50)
    sex = models.CharField(max_length=1, choices=Sex.choices)
    email = models.EmailField()
    birthdate = models.DateField()
    imageUrl = models.TextField()
    password = models.CharField(max_length=100)
    lastActiveDate = models.DateField(auto_now=True)
    appartment = models.ForeignKey("appartmentApp.Appartment",
on_delete=models.CASCADE, null=True)

    def save(self, *args, **kwargs):
        self.password = make_password(self.password)
```

```
        super().save(*args, **kwargs)

    def __str__(self):
        return self.user.username
```

Кожен стовбець таблиці має свій тип даних для зберігання, а для реалізації зв’язку один до багатьох використовується властивість поля models.ForeignKey. Також є можливість задати тип поведінки при видаленні запису за допомогою властивості models.CASCADE.

Для керування змінами в БД Django використовує міграції. Вони створюються автоматично на основі змін у моделях та можуть бути застосовані або відкочені до попереднього стану за допомогою команд Django.

За допомогою файла проекту manage.py можна виконувати різноманітні операції над БД. Основні команди це makemigrations, migrate, showmigrations. Команда makemigrations відповідальна за створення файлу міграції яка буде застосована до БД та включає всі зміни які були зроблені в моделі. Приклад файлу міграції наведений нижче.

```
# Generated by Django 5.0.6 on 2024-07-08 13:15
from django.db import migrations

class Migration(migrations.Migration):
    dependencies = [
        ("usersApp", "0001_initial"),
    ]

    operations = [
        migrations.RenameField(
            model_name="users",
            old_name="mail",
            new_name="email",
        ),
    ]
```

З цього коду можна зрозуміти що міграції реалізовані за допомогою класу Migration а зміни були застосовані до полів email в моделі users. Поле dependencies демонструє що це друга міграція і вона була виконана після початкової ініціалізації БД.

Щоб застосувати зміни зроблені у моделях необхідно послідовно виконати команди makemigrations та migrate. Команда migrate застосує зміни до БД.

Застосовані зміни можна переглянути за допомогою команди showmigrations. Це дуже потужний та простий механізм який допомагає легко керувати змінами в схемі БД.

4.2 Серверна частина

Серверна частина реалізована за допомогою Django Rest Framework (DRF). Це фреймворк який допомагає створювати Web API. Він має модульну структуру та гнучку архітектуру яка добре підійде як для малих так і для доволі великих проектів.

Для створення API потрібно перетворити Django модель на більш просту модель даних, а саме лист словників. Для цього використовується серіалізатор. Серіалізатори дозволяють перетворювати складні дані, такі як набори запитів та екземпляри моделей, у власні типи даних Python, які потім можуть бути легко перетворені на JSON, XML або інші типи вмісту. Серіалізатори також забезпечують десеріалізацію, дозволяючи перетворювати розібрані дані у складні типи після попередньої перевірки вхідних даних[8]. Приклад серіалізатора наведено нижче.

```
from rest_framework import serializers
from django.contrib.auth.models import User
from .models import Users

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ['username', 'email', 'password']
        extra_kwargs = {'password': {'write_only': True}}

    def create(self, validated_data):
        user = User.objects.create_user(**validated_data)
        return user

class UsersSerializer(serializers.ModelSerializer):
    user = UserSerializer()

    class Meta:
        model = Users
        fields = '__all__'

    def create(self, validated_data):
        user_data = validated_data.pop('user')
```

```
        user = UserSerializer().create(user_data)
        users_instance = Users.objects.create(user=user, **validated_data)
        return users_instance
```

В прикладі ми оголошуємо два серіалізатори UserSerializer задля серіалізації моделі Users. Meta клас вказує модель User, яка є стандартною моделлю користувачів в Django, а також поля які треба серіалізувати. Поле password має параметр write_only який використовується для того щоб виключити його з серіалізованих даних під час читання. Метод create створює новий об'єкт моделі.

Для реалізації API також використовуються контролер views який використовується для повернення серіалізованих даних та ендпоінти за якими ці данні будуть віддаватися.

Контролер створюються за допомогою класу APIview який є базовим класом представлень та Response який відповідальний за повернення HTTP відповідей. Для виконання POST, GET, PUT, DELETE запитів використовується анотація @api_view методу з вказівкою листа з типами запитів які можуть використовуватися з цим методом.

Наприклад контролер який повертає всіх користувачів в системі буде виглядати наступним чином:

```
from rest_framework.decorators import api_view
from .models import Users
from .serializers import UsersSerializer
from rest_framework.authtoken.models import Token
from rest_framework.response import Response
from rest_framework.views import APIView
from rest_framework import status

@api_view(['GET'])
def view_items(request):

    # checking for the parameters from the URL
    if request.query_params:
        items = Users.objects.filter(**request.query_params.dict())
    else:
        items = Users.objects.all()

    # if there is something in items else raise error
    if items:
        serializer = UsersSerializer(items, many=True)
        return Response(serializer.data)
    else:
        return Response(status=status.HTTP_404_NOT_FOUND)
```

Цей код демонструю використання DRF для отримання та фільтрацій користувачів. Якщо запит отриманий через API має данні для фільтрації то наш метод виконає фільтрацію та поверне об’єкт який відповідає запиту, якщо запит не буде містити додаткових параметрів то метод поверне всіх користувачів наявних в БД. Для реалізації функціоналу фільтрації використовується вбудований метод filter() що дозволяє зручно та гнучко фільтрувати данні без використання SQL запитів.

Невід’ємною частиною API є маршрути API. Маршрути реалізуються за допомогою модулю django.urls та функції path це дозволяє описати маршрут за яким знаходиться відповідна функція для обробки даних. Приклад маршрутів наведений нижче:

```
from django.urls import path
from . import views
from .views import CustomAuthToken

urlpatterns = [
    path('create/', views.add_items, name='add-items'),
    path('all/', views.view_items, name='view_items'),
    path('update/<int:pk>/', views.update_items, name='update-items'),
    path('api-token-auth/', CustomAuthToken.as_view(),
name='api_token_auth')
]
```

Маршрут складається з url-шляху який говорить про тип дії яка буде виконуватися та посиланням на функцію контролера яка буде обробляти дану дію. DRF також надає доступ до веб де можна ознайомитися з наявними маршрутами та виконати GET, POST, PUT, DELETE запит. За це відповідає набір класів та методів BowsableAPIrender що перетворює поточний код в HTML.

Для зручності розробки таблиці БД були описані в окремих Django applications. Це дозволяє розділити велику систему на логічно згруповані компоненти що підвищую зручність та розуміння структурі проекту.

В файлі osbbProject/settings.py знаходиться перелік компонентів який буде використовуватися DRF для створення API. Приклад файлу наведено нижче:

```
INSTALLED_APPS = [
    "osbbApp",
    "buildingApp",
    "ordersApp",
    "appartmentApp",
    "usersApp",
    "rest_framework",
    'rest_framework.authtoken',
    'corsheaders',
    'ordersOsbbApp'
]
```

osbbProject це директорія проекту Django яка містить в собі всі необхідні налаштування проекту а також опис Django applications. Також в цій директорії знаходиться файл urls.py який об’єднує ендпоінти з усіх компонентів. Приклад файлу наведений нижче:

```
from django.urls import path
from django.urls import include

urlpatterns = [
name='update-items'),
    path('api/v1/osbbs/', include("osbbApp.urls")),
    path('api/v1/buildings/', include("buildingApp.urls")),
    path('api/v1/appartments/', include("appartmentApp.urls")),
    path('api/v1/orders/', include("ordersApp.urls")),
    path('api/v1/users/', include("usersApp.urls")),
]
```

Як видно з наведених прикладів та опису DRF дозволяє доволі швидко та гнучко розробити API та реалізувати контроллери для взаємодії з БД.

4.3 CI/CD частина

Для підвищення якості програмного забезпечення та його розгортання використовується Continius Integration(CI)/Continius Deployment (CD) система на базі GitLab. CD/CD дозволяє протестувати систему при потраплянні коду до системи контролю версій та доставити новий продукт на сервер після успішного тестування без залучення розробника. Також це зменшує вірогідність потрапляння на робочі сервери неробочого коду.

Нижче наведений код який дозволяє протестувати продукт та створити Docker імедж на основі даного коду:

```
default:
  image: docker:24.0.5
  services:
    - docker:24.0.5-dind

stages:
  - Validate
  - Build

validate-job:
  stage: Validate
  script:
      - apk update && apk add docker-compose
      - docker-compose up --build --force-recreate --abort-on-container-
exit --exit-code-from app

build-job:
  stage: Build
  script:
    - cd osbb/
    - docker build -t osbbapi:latest .
  only:
    - master
```

Цей код знаходиться в файлі .gitlab-ci.yml та відповідає за створення пейплайнів для тестування коду та створення Docker имеджу. Кожен раз при додаванні змін в git буде запускатися пейплайн який провалідує код за допомогою тестів (див.рис. 12). Якщо зміни будуть додані в master гілку то код провалідується та наступним завдання буде створення та публікація імеджу.

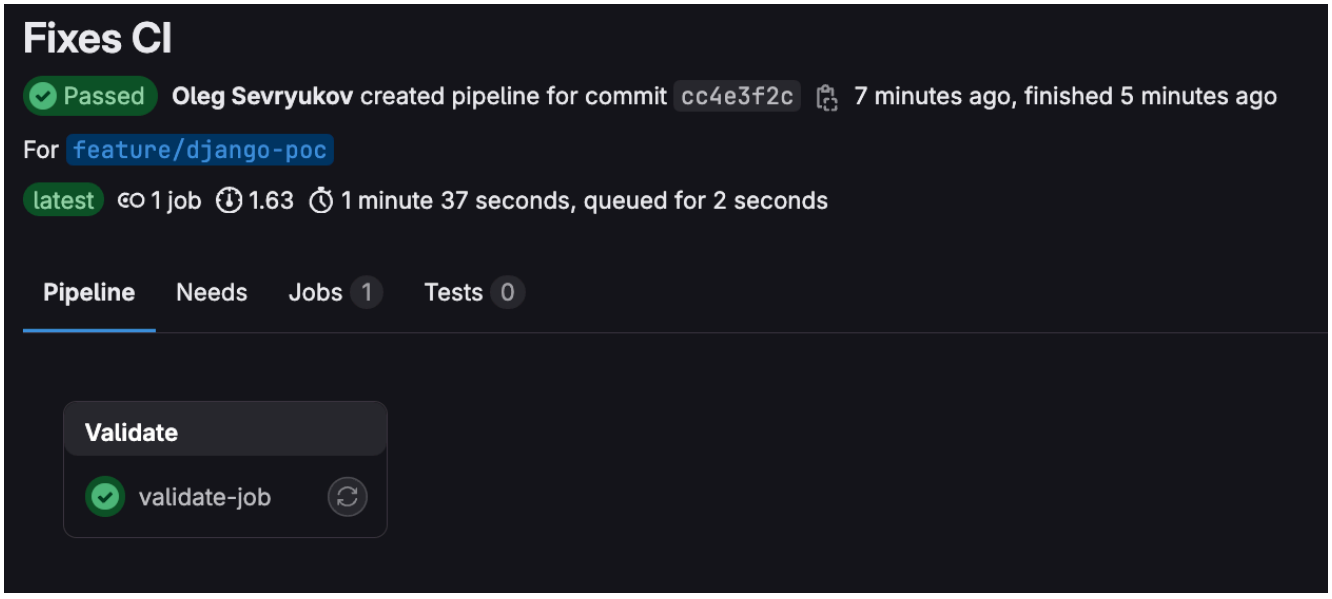


Рисунок 12 – CI/CD інтерфейс (Рисунок виконано самостійно)

Це забезпечить версіювання готового артефакту та незалежність продукту від кінцевого середовища розгортання, бо всі необхідні компоненти будуть в Docker імеджі. Також це прискорить розгортання нових версій бо система зберігання Docker імеджей оновить свої індекси і сервер зможе перезапустити сервіс використовуючи новий імедж.

5 ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5.1 Інтеграційне тестування

Інтеграційне тестування – вид тестування, при якому на відповідність вимог перевіряється інтеграція модулів, їх взаємодія між собою, а також інтеграція підсистем в одну загальну систему. Для інтеграційного тестування використовуються компоненти, вже перевірені за допомогою модульного тестування, які групуються у множини. Дані множини перевіряються відповідно до плану тестування, складеним для них, а об’єднуються вони через свої інтерфейси [9]. В контексті веб-додатків на Django, це може включати тестування взаємодій між моделями, контроллерами відображення, формами, серіалізаторами, і що важливо, зовнішніми сервісами або API.

Для написання тестів в DRF використовується клас APITestCase з модуля rest_framework.test. Цей клас розширює стандартний клас Django Client який в свою чергу є наслідником класу unittest.TestCase та дозволяє емітувати Get та POST запити та отримувати відповіді. Перевіряти валідність редіректів та статус коди. Тестувати контролери відображення проходячи по шляхам API.

В процесі тестування створюється тимчасова БД з якою будуть проводитися CRUD операції через API, створюються міграції для таблиць БД за запускаються перевірки конфігурації проекту. Наступним кроком є запуск безпосередньо тестів та відображення кількості пройдених та непройдених тестів. Останнім кроком є видалення тестової БД.

Для початкової ініціалізації тестовою БД використовується метод setUp(). Це дозволяє додати необхідні таблиці та заповнити їх тестовими даними а також створити зв’язки між таблицями БД. Фрагмент коду відповідального за це наведено нижче:

```
    def setUp(self):
        # Set up the Apartment and Service required for the Order
        from apartmentApp.models import Apartment
        # Set up the Apartment, Service, and ServicePrice
        self.apartment = Apartment.objects.create(number="101", floor=1,
entrance=1, peopleNumber=1, area=42)
```

```
        self.service = Service.objects.create(name="Cleaning",
description="Weekly cleaning service")
```

В даному фрагменті коду ми створюємо таблицю Apartment та Service і заповнюємо їх тестовими даними які будуть використані для тестування API.

Після створення тестових даних йде опис безпосередньо інтеграційних тестів. Прикладом інтеграційного тестування є наступний код:

```
def test_order_serializer(self):
    order_data = OrderSerializer(instance=self.order).data
    self.assertEqual(len(order_data['services']), 1)
    self.assertEqual(order_data['services'][0]['name'], 'Cleaning')
    self.assertTrue('Waiting for payment' in [status['status'] for status
in order_data['statuses']])
    order_data = {
        'apartment': self.apartment.id,
        'services': [{'service': self.service.id}],  # Assuming 'service'
key is correct
        'date': '2023-01-01T00:00:00Z'
    }
    order_serializer = OrderSerializer(data=order_data)
    self.assertTrue(order_serializer.is_valid(), order_serializer.errors)
    new_order = order_serializer.save()
    self.assertEqual(new_order.apartment, self.apartment)
self.assertTrue(OrderService.objects.filter(order=new_order).exists())
```

В даному тесті виконується тестовий метод test_order_serializer() який превіряє взаємодію між серіалізаторами Apartment, Orders та Service під час сторення нового замовлення. Спочатку перевіряється чи є в нас сервіс послуг з назвою Cleaning та чи відповідає id запису в таблиці очікуваному результату. Це робиться щоб бути впевненим що ініціалізовані данні відповідають даним тестування. Після початкової перевірки створюється набір даних для замовлення який буде надісланий до серіалізатору OrderSerializer де відбудеться його обробка та виконання. Останнім пунктом буде перевірка що замовлення створилося та існує.

Підчас тестування були виявленні помилки в серіалізаторах та їх полях обробки які були виправлені.

5.2 Тестування навантаження

Тестування продуктивності (тестування навантаження) – це комплекс типів тестування, метою якого є визначення працездатності, стабільності, споживання ресурсів та інших атрибутів якості додатка в умовах різних сценаріїв використання та навантажень [10]. Цей тип тестуання використовується для перевірки поведінки системи під час пікового навантаження та під час нормального очікуваного навантаження. Під час тестування створюється навантаження на систему яке імітує поведінку користувача та виконує різні операції, запити тощо. Цей тип тестування надає розробникам інформацію про стабільність системи та виявляє слабкі місця які можна усунути чи оптимізувати.

Для тестування був обраний фрейморк Locust. Даний фрейморк дозволяє протестувати систему та отримати данні у вигляді графіків через веб інтерфейс, а також має різні налаштування які можуть імітувати поведінку користувача.

Приклад коду для перевірки замовлень наведено нижче:

```
class UserBehavior(TaskSet):
    @task(2)  # You can adjust task weight here if needed
    def get_orders(self):
        # Retrieve list of orders
        self.client.get("/api/v1/orders/")

    @task(1)
    def create_order(self):
        # Data payload for creating a new order
        order_data = {
            'apartment': 1,  # Example data, replace with actual field
names and valid data
            'services': [{'service': 1}],  # Assuming the API expects a
list of services
            'date': '2023-01-01T00:00:00Z'
        }
        headers = {'content-type': 'application/json'}
        # Create a new order
        self.client.post("/api/v1/orders/create/", json=order_data,
headers=headers)
```

Цей код запускає веб інтерфейс де можна налаштувати кількість користувачів та зручно відслідкувати процес тестування. Для тестування були обрані маршрути API які повертають всі замовлення та створюють замовлення так

як вони є такими, що використовують найбільше число викликів а також мають найбільшу кількість сутностей БД.

За допомогою команди locust -f locustfile.py --host=http://127.0.0.1:8000 --users 3000 --spawn-rate 10 –autostart буде запущено навантажувальне тестування яке буде поступово збільшувати кількість користувачів на 10 з інтервалом від 1 до 5 секунд до досягнення кількість користувачів в 3000 у піковому навантаженні.

На рисунку 13 відображено результат навантажувального тестування. За результатами тестування видно що в поточній реалізації система може витримати 220 користувачів в піку які виконують одночасно запит на отримання листу замовлень та створення нового замовлення. При цьому час відповіді від серверу дорівнює 1690 мс. Таким чином було виявлено найслабкішу ланку системи, а саме БД яка має обмеження в кількості одночасних клієнтів. Для подальшого підвищення продуктивності треба проаналізувати конфігурацію БД та серверу БД.

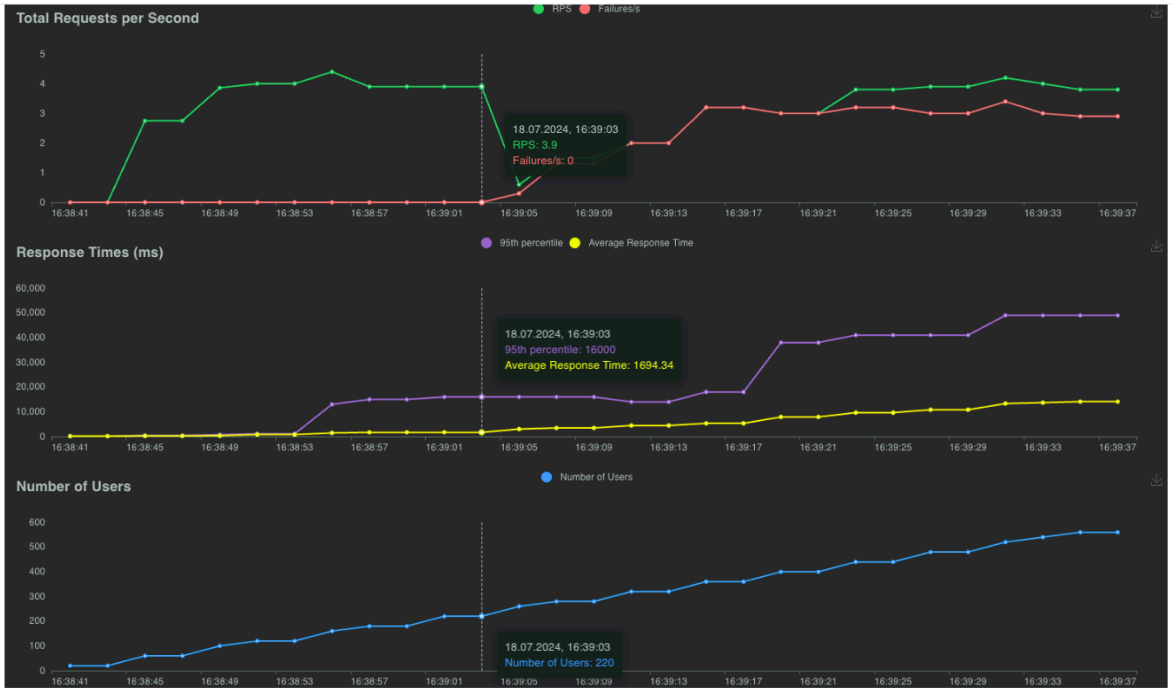


Рисунок 13 – Тестування навантаження (рисунок виконано самостійно)

Отримані результати тестування демонструють що система відповідає вимогам та витримує очікуване навантаження.

ВИСНОВКИ

Були отримані теоретичні та практичні знання з аналізу предметної галузі, побудови архітектури програмного продукту. Були набуті навички аналізу конкурентів та виявлення недоліків. Завдяки аналізу конкурентів та відкритих джерел інформації були виявлені сильні та слабкі сторони продукту. Був проведений анонімний опит потенційних користувачів та проаналізовані їх проблеми, побажання та було сформовано перелік їх потреб. Ці висновки допоможуть сформувати успішну стратегію побудови та розвитку продукту.

За допомогою UML діаграм були пропрацьовані ключові функції програмного продукту та описана логіка їх роботи та поведінки. Були визначені та проілюстровані головні активності цих фукцій. За допомогою EMR діаграми була спроектована та реалізована БД за допомогою PostgreSQL з описом всіх таблиць та зв’язків.

За допомогою діаграм та технологій Docker, Python, Django, Django rest framework, PostgreSQL було реалізовано програмний продукт який задовольняє потреби клієнтів на основі вимог до продукту.

На основі вимог та проектування було написано серверну частину, під розробки якої були закріпленні знання та навички роботи з Python, Django, DRF, PostgreSQL тощо.

Серверну частину було протестовано за допомогою інтеграційних тестів та тестів навантаження. Помилки були виправленні та знайденні місця які потребують подальшої оптимізації.

Результатом роботи є серверна частина до системи керування ОСББ. Продукт дозволяє створювати послуги, сплачувати їх та відслідковувати їх статус. З боку керування ОСББ продукт надає змогу отримувати інформацію по створених заявках опрацьовувати їх та виконувати інші операційні дії.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Закон України про ОСББ [Електронний ресурс] – URL: https://zakon.rada.gov.ua/laws/show/2866-14#Text (дата звернення: 04.06.2024)

2. Динаміка створення ОСББ в Україні [Електронний ресурс] – URL: https://osbb-ok.org.ua/storage/posts/files/541/hZ0PD1XhBjtXWeNhrhNdfYIGTYlQZ7JFYEKF8BBg.pdf. (дата звернення: 07.06.2024)

3. Застосунок «Дах» [Електронний ресурс] – URL: https://dah-online.com/ (дата звернення: 07.06.2024)

4. Застосунок «ОМО» [Електронний ресурс] – URL: https://omo.systems (дата звернення: 07.06.2024)

5. Застосунок «Моє ОСББ» [Електронний ресурс] – URL: https://moeosbb.com (дата звернення: 07.06.2024)

6. Застосунок «Мій Дім» [Електронний ресурс] – URL: https://miydimonline.com.ua/ (дата звернення: 07.06.2024)

7. Psycopg [Електронний ресурс] – URL: https://www.psycopg.org/docs/ (дата звернення 12.06.2024)

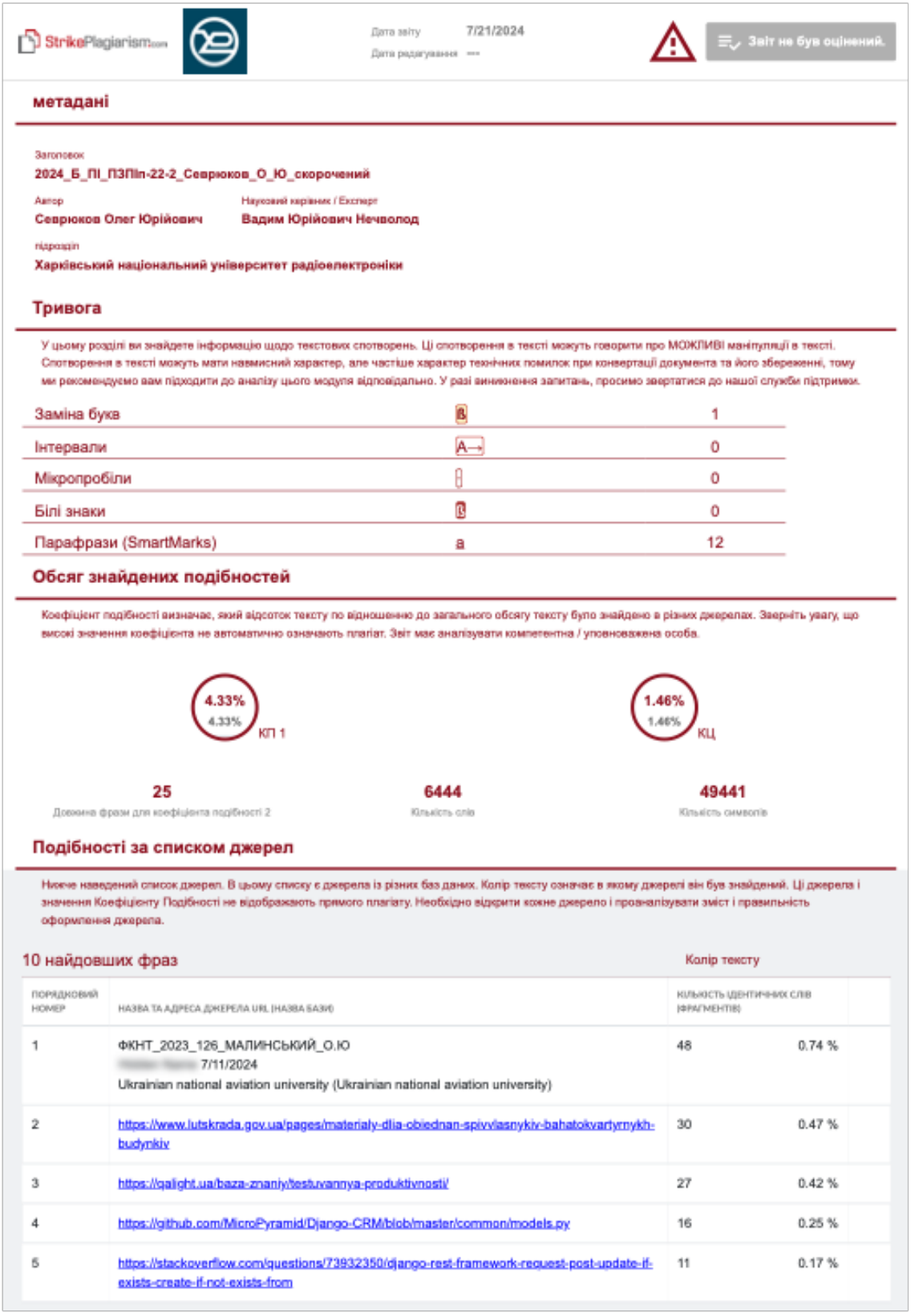
8. Django rest framework [Електронний ресурс] – URL: https://www.django-rest-framework.org/api-guide/serializers (дата звернення: 09.06.2024)

9. Інтеграційне тестування [Електронний ресурс] – URL: https://qalight.ua/baza-znaniy/integratsijne-testuvannya/ (дата звернення 18.06.2024)

10. Тестування продуктивності. URL: https://qalight.ua/baza-znaniy/testuvannya-produktivnosti/ (дата звернення 19.06.2024).

ДОДАТОК А

Звіт результатів перевірки на унікальність тексту



StrikePlagiarism.com

Дата звіту 7/21/2024
Дата редагування ---

Звіт не був оцінений.

метадані

Заголовок
2024_Б_ПІ_ПЗПіп-22-2_Севрюков_О_Ю_скорочений

Автор
Севрюков Олег Юрійович

Науковий керівник / Експерт
Вадим Юрійович Нечволод

підрозділ
Харківський національний університет радіоелектроніки

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв		1
Інтервали		0
Мікропробіли		0
Білі знаки		0
Парафрази (SmartMarks)	a	12

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.

4.33% 4.33% КП 1

1.46% 1.46% КЦ

25
Довжина фрази для коефіцієнта подібності 2

6444
Кількість слів

49441
Кількість символів

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз — Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)	
1	ФКНТ_2023_126_МАЛИНСЬКИЙ_О.Ю [redacted] 7/11/2024 Ukrainian national aviation university (Ukrainian national aviation university)	48	0.74 %
2	https://www.lutskrada.gov.ua/pages/materialy-dlia-obiednan-spivvlasnykiv-bahatokvartyrnykh-budynkiv	30	0.47 %
3	https://qalight.ua/baza-znanij/testuvannya-produktivnosti/	27	0.42 %
4	https://github.com/MicroPyramid/Django-CRM/blob/master/common/models.py	16	0.25 %
5	https://stackoverflow.com/questions/73932350/django-rest-framework-request-post-update-if-exists-create-if-not-exists-from	11	0.17 %

ДОДАТОК Б

Слайди презентації

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

Програмне забезпечення підтримки діяльності ОСББ.

Серверна частина

Виконав: ст. гр. ПЗПІп-22-2 Севрюков О.Ю.
Керівник: доц.Кириченко І.В.

1

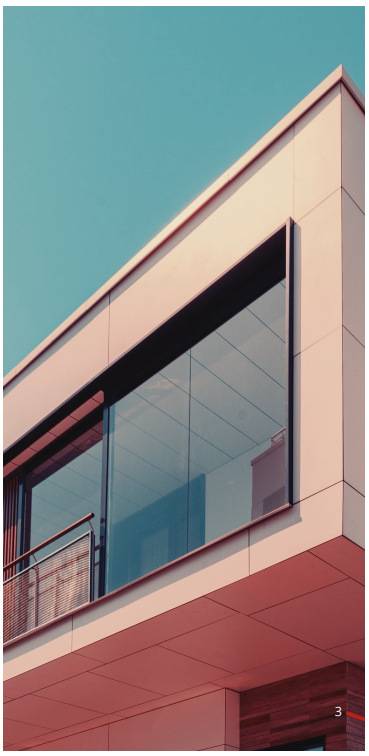
МЕТА РОБОТИ

Створення програмного продукту для об’єднання співвласників багатоквартирного будинку з метою автоматизації та оптимізації управління фінансами, заявками та документами, забезпечення зручного взаємодії співвласників та управління з обмеженим витратами та підвищеною ефективністю.

2

Проблеми предметної галузі

- Паперовий менеджмент діяльності ОСББ
- Відсутність єдиного каналу комунікацій між власниками та керівництвом ОСББ
- Ведення бухгалтерського обліку
- Голосування
- Інтеграція с систами “Розумний будинок”



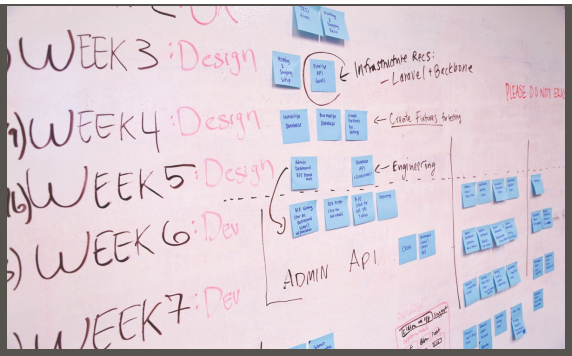
Аналіз конкурентів

Програмний продукт	Додавання запитів на “свої” послуги	Ведення бухгалтерського обліку	Система голосування	Месенджер	Звітність	Інтеграція з розумним будинком	Створення заявок на роботи
Мій дім	-	+	-	+	+	-	+
OMO	+	-	-	-	-	+	-
МОЄОСББ	-	+	-	+	-	-	-
Дах	-	+	-	+	+	-	-

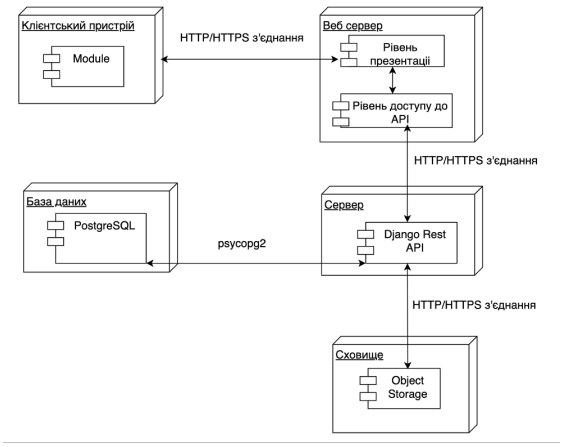
Постановка задачі

Для вирішення потреб управляючих компаній та мешканців потрібно розробити програмний продукт який надавав би наступні сервіси:

- ведення фінансового обліку;
- повідомлення боржникам;
- замовлення робіт;
- сплата за послуги:
- отримання фінансових звітів;
- замовлення додаткових послуг.



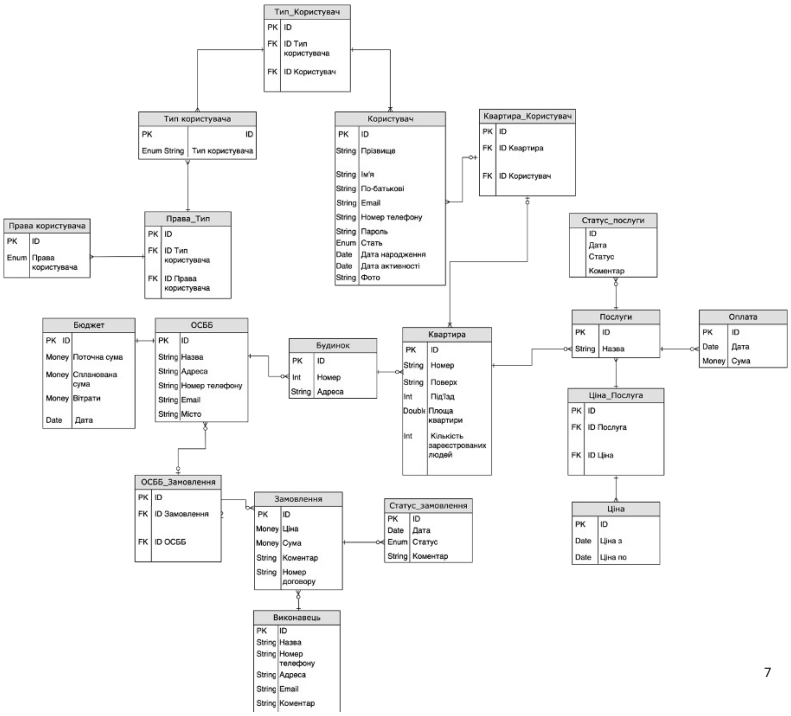
5



Діаграма розгортання

6

ER-діаграма



Створення схеми БД

```python
services = models.ManyToManyField(Service, through='OrderService', related_name='orders')
apartment = models.ForeignKey( to: "apartmentApp.Apartment", on_delete=models.SET_NULL, null=True, related_name='orders')
date = models.DateTimeField(default=timezone.now)

1 usage (1 dynamic)  Oleh Sevriukov *
def update_order_status(self):
    total_service_cost = sum([os.service.prices.latest('date').price for os in self.orderservice_set.all()])
    total_payments = sum([payment.sum for payment in self.payments.all()])

    if total_payments >= total_service_cost:
        status = OrderStatus.Status.PAID
        OrderStatus.objects.update_or_create(
            order=self,
            defaults={'status': status, 'comment': 'Сплачено.', 'date': timezone.now()}
        )
    else:
        status = OrderStatus.Status.AWAITING_PAYMENT,
        OrderStatus.objects.update_or_create(
            order=self,
            defaults={'status': status, 'comment': 'Очікує на повну оплату.', 'date': timezone.now()}
        )
    Oleh Sevriukov
def __int__(self):
    return self.id
```

```python
class OrderSerializer(serializers.ModelSerializer):
    services = OrderServiceSerializer(many=True, source='orderservice_set')
    statuses = OrderStatusSerializer(many=True, read_only=True)  # Assuming there can be multiple statuses
    payments = OrderPaymentSerializer(many=True, read_only=True)  # To see all payments related to the order
    Oleh Sevriukov
    class Meta:
        model = Orders
        fields = ['id', 'apartment', 'services', 'payments', "statuses", "date"]

    Oleh Sevriukov *
    def create(self, validated_data):
        services_data = validated_data.pop('orderservice_set', [])
        order = Orders.objects.create(**validated_data)
        for service in services_data:
            OrderService.objects.create(order=order, service_id=service.get("service").id)
        # Optionally, add a default status for the order
        status = OrderStatus.Status.AWAITING_PAYMENT,
        OrderStatus.objects.create(order=order, status=status, comment='Створено.',date=validated_data.get('date', timezone.now()))
        return order
```

Використані технології

python™

django

django REST framework

PostgreSQL

docker

GitLab

9

Діаграма активності перевірки боргів та сповіщення

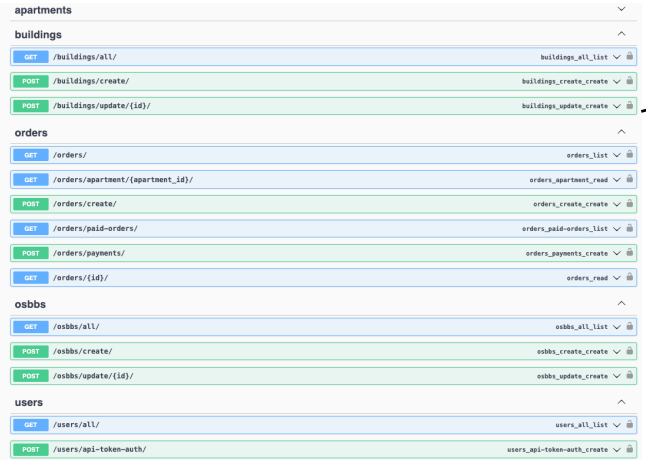
10

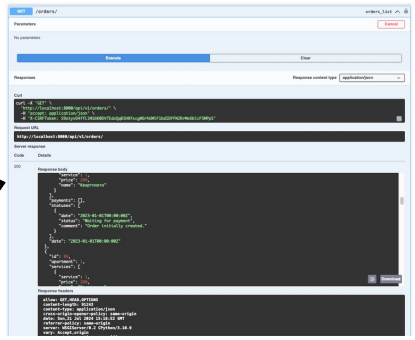
Приклад коду перевірки боргів та сповіщення

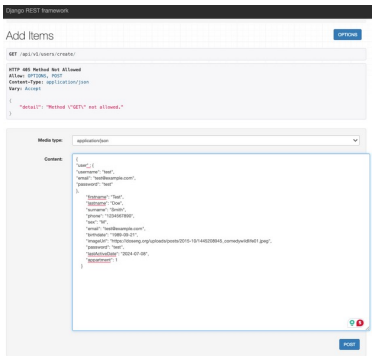
```python
1 usage  Oleh Sevriukov
def get_user_emails_by_apartment(apartment_id):
    users = Users.objects.filter(apartment_id=apartment_id)
    emails = [user.email for user in users if user.email]  # List comprehension to gather emails
    return emails

2 usages  Oleh Sevriukov
def check_and_notify_overdue_payments():
    overdue_orders = Orders.objects.filter(statuses__status='Очікує оплати', statuses__date__lt=timezone.now())
    overdue_orders_by_apartment = defaultdict(list)

    for order in overdue_orders:
        if order.apartment:
            overdue_orders_by_apartment[order.apartment.id].append(order)
    for apartment_id, orders in overdue_orders_by_apartment.items():
        send_payment_notification(orders)
```

```python
1 usage  Oleh Sevriukov *
def send_payment_notification(orders):
    apartment_id = orders[0].apartment.id
    emails = get_user_emails_by_apartment(apartment_id)

    order_details = ""
    for order in orders:
        services = order.orderservice_set.all()
        pprint(services)
        service_names = ", ".join([os.service.name for os in services])
        order_details += f"- {service_names}\n"

    message = f"Ви маєте заборгованість по рахунках. Будьласка погасіть заборгованість:\n\n{order_details}"

    pprint(message)
    send_mail(
        subject: 'Повідомлення про заборгованість',
        message,
        from_email: 'from@example.com',  # Replace with your "from" email address
        emails,
        fail_silently=False,
    )
```


Функціонал додатку







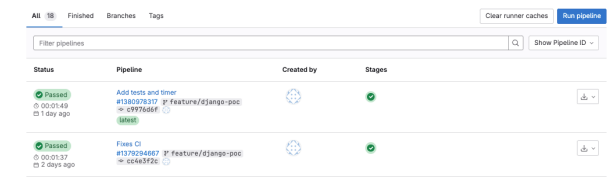
Тестування

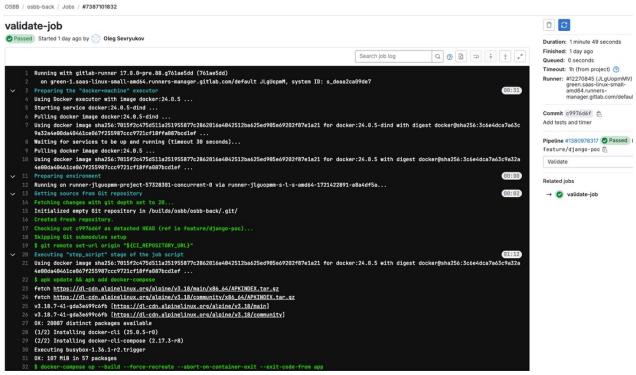
- Модульне тестування
- Інтеграційні тести
- Навантажувальне тестування
- Ручне тестування

```
System check identified no issues (0 silenced).
OrderSerializer(data={'apartment': 1, 'services': [{'service': 1}], 'date': '2023-01-01T08:00:00Z'}):
    id = IntegerField(label='ID', read_only=True)
    apartment = PrimaryKeyRelatedField(allow_null=True, queryset=Apartment.objects.all(), required=False)
    services = OrderServiceSerializer(many=True, source='orderservice_set'):
        service = PrimaryKeyRelatedField(queryset=<QuerySet [<Service: Cleaning>]>)
        price = SerializerMethodField()
        name = SerializerMethodField()
    payments = OrderPaymentSerializer(many=True, read_only=True):
        id = IntegerField(label='ID', read_only=True)
        date = DateTimeField()
        sum = DecimalField(decimal_places=2, max_digits=10, required=False)
        order = PrimaryKeyRelatedField(allow_null=True, queryset=Orders.objects.all(), required=False)
    statuses = OrderStatusSerializer(many=True, read_only=True):
        date = DateTimeField(read_only=True)
        status = ChoiceField(choices=[('Очікує оплати', 'Awaiting Payment'), ('Сплачено', 'Paid')], required=False)
        comment = CharField(style={'base_template': 'textarea.html'})
    date = DateTimeField(required=False)
..<OrderService: 3 - Cleaning>
<QuerySet [<OrderService: 3 - Cleaning>]>
('Ви маєте заборгованість по рахунках. Будьласка погасіть заборгованість:\n'
 '\n'
 '- Cleaning\n')
.
----------------------------------------------------------------------
Ran 3 tests in 0.307s

OK
Destroying test database for alias 'default'...
```

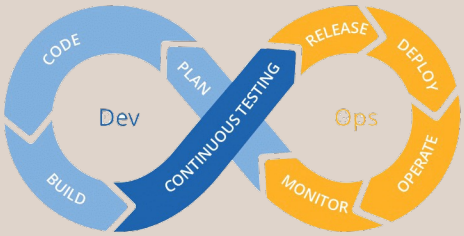
Приклад інтеграційного тесту

```python
class TestScheduledPaymentChecks(TestCase):
    def setUp(self):
        self.apartment = Apartment.objects.create(number="101", floor=1, entrance=1, peopleNumber=1, area=42)
        django_user = User.objects.create_user(
            username='testuser',
            email='test@example.com',
            password='securepassword123'
        )

        self.user = Users.objects.create(
            email='test@example.com',
            password='securepassword123',
            firstname='testuser',
            lastname='testuser',
            sex='M',
            phone='11111111',
            apartment=self.apartment,
            birthdate=timezone.now(),
            user=django_user
        )
        self.service = Service.objects.create(name="Cleaning", description="Weekly cleaning service")
        ServicePrice.objects.create(service=self.service, price=30.00, date=timezone.now() - datetime.timedelta(days=28))
        self.order = Orders.objects.create(apartment=self.apartment, date=timezone.now() - datetime.timedelta(days=10))  # Ensure the order is old enough
        OrderStatus.objects.create(
            order=self.order,
            status=OrderStatus.Status.AWAITING_PAYMENT,
            date=timezone.now() - datetime.timedelta(days=28)  # Definitely overdue
        )
        order = OrderService.objects.create(order=self.order, service=self.service)
        pprint(order)

    @patch('ordersApp.utils.payment_utils.send_mail')
    def test_overdue_payment_notification(self, mock_send_mail):
        # Run the task that should trigger the send_mail function
        scheduled_payment_check()

        # Ensure send_mail was called
        mock_send_mail.assert_called_once()

        # Retrieve the arguments with which send_mail was called
        args, kwargs = mock_send_mail.call_args

        # The message is the second positional argument
        self.assertIn('заборгованість', args[1])  # args[1] is the 'message' parameter of send_mail
```





CI/CD



15

Висновки

- Завдяки аналізу конкурентів та відкритих джерел інформації були виявлені сильні та слабкі сторони продукту.
- Був проведений анонімний опит потенційних користувачів та проаналізовані їх проблеми, побажання та було сформовано перелік їх потреб.
- За допомогою UML діаграм були пропрацьовані ключові функції програмного продукту та описана логіка їх роботи та поведінки.
- За допомогою технологій Docker, Python, Django, Django rest framework, PostgreSQL було реалізовано програмний продукт який задовольняє потреби клієнтів на основі вимог до продукту.

16