

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Розробка REST API сервера додатку для
поширення своєї геолокації з використанням
наскрізного шифрування»**

Виконав студент групи ЗПЗк-22-1
ТРЕТЯК Віктор Вікторович

Керівник роботи:
Морозов Андрій Васильович

Рецензент:
ЛОКТИКОВА Тамара Миколаївна

Житомир – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення
Тетяна Вакалюк

«26» лютого 2024 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ТРЕТЯК Віктор Вікторович**

Керівник роботи: **МОРОЗОВ Андрій Васильович**

Тема роботи: «Розробка REST API сервера додатку для поширення своєї геолокації з використанням наскрізного шифрування»,

затверджена Наказом закладу вищої освіти від «23» лютого 2024 р., №75/с

Вихідні дані для роботи: спосіб захищеного обміну геолокацією через мережу інтернет.

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Морозов А.В.	02.03.24	02.03.24
2	Морозов А.В.	05.04.24	05.04.24
3	Морозов А.В.	01.05.24	01.05.24

РЕФЕРАТ

Мета роботи полягає в тому щоб розробити відкритий прикладний програмний інтерфейс, у вигляді децентралізованого програмного сервера, для обміну геолокацією з використанням наскрізного шифрування.

Проект побудований на базі клієнт-серверної архітектури та дозволяє передавати повідомлення захищені наскрізним шифруванням через мережу інтернет.

Наведену інформацію в роботі можна повторно використати в інших проєктах, що потребують наскрізного шифрування для обміну повідомленнями.

Фінальну розробку можна вдосконалити чи, або повторно використати її частини для розробки схожих по змісту застосунків. На основі сервера можна розробити клієнтські застосунки під різні платформи.

Робота складається з 61 сторінок, 13 ілюстрацій, 8 таблиць і 12 використаних джерел.

КЛЮЧОВІ СЛОВА: ГЕОЛОКАЦІЯ, НАСКРІЗНЕ ШИФРУВАННЯ, СЕРВЕР

					ІПЗ.КР.Б – 121 – 24 – ПЗ			
Зм.	Арк.	№ докум.	Підпис	Дата	Розробка REST API сервера додатку для поширення своєї геолокації з використанням наскрізного шифрування	Літера	Аркуш	Аркушів
Розроб.	Третяк В.В.						3	58
Керівник	Морозов А.В.							
Н.контр.	Локтікова Т.М.					Житомирська політехніка, Група ЗІПЗк-22-1		
Затв.	Вакалюк Т.А.							

ABSTRACT

The aim of the work is to develop an open application software interface, in the form of a decentralized software server, for exchanging geolocation using end-to-end encryption.

The project is based on a client-server architecture and allows for the transmission of messages protected by end-to-end encryption over the Internet.

The information provided in this paper can be reused in other projects that require end-to-end encryption for messaging.

The final development can be improved or reused to develop similar applications. Based on the server, you can develop client applications for different platforms.

The paper consists of 61 page, 13 illustrations, 8 tables and 12 references.

KEYWORDS: END-TO-END ENCRYPTION, GEOLOCATION, SERVER

					ІПЗ.КР.Б – 121 – 24 – ПЗ	Аркуш
Зм.	Арк.	№ докум.	Підпис	Дата		4

ЗМІСТ

РЕФЕРАТ.....	3
ABSTRACT.....	4
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	6
ВСТУП.....	7
РОЗДІЛ 1. АНАЛІЗ ІНФОРМАЦІЙНОЇ БАЗИ.....	9
1.1 Постановка задачі.....	9
1.2 Аналіз аналогів.....	10
1.3 Вибір архітектури проєкту.....	12
1.4 Обґрунтування вибору інструментальних засобів та вимог до програмного забезпечення.....	14
Висновки до першого розділу.....	15
РОЗДІЛ 2. ПРОЄКТУВАННЯ.....	17
2.1 Варіанти використання системи.....	17
2.2 Опис реалізації функціонала.....	19
2.3 Структура бази даних.....	27
2.4 Проєктування та реалізація алгоритмів роботи.....	32
2.5 Налаштування розгортання.....	39
Висновки до другого розділу.....	40
РОЗДІЛ 3. ІНТЕРФЕЙС.....	42
3.1 Порядок встановлення системи.....	42
3.2 Інтерфейс адміністратора.....	44
3.3 OpenAPI документація.....	46
Висновки до третього розділу.....	49
ВИСНОВКИ.....	50
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	52
ДОДАТКИ.....	54

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

REST API - це скорочення від Representational State Transfer Application Programming Interface.

OpenAPI — вільний стандарт для документації REST API.

Баг — програмна помилка.

ORM - object-relational mapping, об'єктно орієнтоване відображення бази даних, що спрощує розробку та доступ до даних в базі даних.

Self-Host — застосунок, частіше сервер який можна підняти самостійно, для власних потреб.

E2EE — end to end encryption, теж саме, що наскрізне шифрування.

					ІПЗ.КР.Б – 121 – 24 – ПЗ	Аркуш
						6
Зм.	Арк.	№ докум.	Підпис	Дата		

ВСТУП

Актуальність роботи полягає у відсутності відкритого рішення для обміну геолокацією з використанням наскрізного шифрування. Існують централізовані сервіси, що виконують цю функцію, проте ми не можемо перевірити надійність програмного забезпечення, адже компанії не надають публічний доступ до коду застосунків. Проєкт покликаний виправити цю статистику.

Метою є розробити прикладний програмний інтерфейс з відкритою документацією та кодом. Проєкт складатиметься з:

1. Програмного сервера, яким зможуть скористуватися адміністратори та підняти особисті сервери
2. OpenAPI документації, яку зможуть вивчити розробники
3. Піднятого сервера, готового рішення від якого зможуть відштовхнутися інші розробники

Тому завданнями будуть:

1. Написати програмний сервер
2. Розгорнути основний сервер

Об'єктом дослідження є спосіб захищеного обміну геолокацією через мережу інтернет.

Предметом дослідження є використання асиметричного шифрування для поширення геолокації мережею.

Методами дослідження виступають веб програмування, об'єктно орієнтоване програмування, шифрування та клієнт-серверна архітектура.

Цей проєкт надасть можливість різним людям і компаніям, що високо цінують приватність та безпеку інтегрувати у своє життя систему обміну геолокацією. Звичайні користувачі можуть підняти сервер для особистого користування та своїх рідних, щоб тримати з ними зв'язок. Компанії можуть скористуватися протоколом та написати власні застосунки, які працюватимуть на пристроях, що потрібно відстежувати, наприклад автомобілях компанії.

					ІПЗ.КР.Б – 121 – 24 – ПЗ	Аркуш 7
Зм.	Арк.	№ докум.	Підпис	Дата		

Існуватиме і центральний сервер, яким користувачі можуть скористатися за власним бажанням.

Випускна робота складається з вступу, 3-х розділів, висновків до кожного розділу, списку використаних джерел із 12 найменувань, 8 таблиць та 13 ілюстрацій по тексту. Повний обсяг роботи становить 61 сторінок, з них 52 сторінок основного тексту.

					ІПЗ.КР.Б – 121 – 24 – ПЗ	Аркуш
Зм.	Арк.	№ докум.	Підпис	Дата		8

РОЗДІЛ 1. АНАЛІЗ ІНФОРМАЦІЙНОЇ БАЗИ

1.1 Постановка задачі

Кожен потрапляв в ситуацію коли хвилювався за своїх близьких буди з ними на відстані та хотів переконатися в їхній безпеці. Звісно завжди можна написати та спитати, але не завжди людина може одразу ж відповісти, вона просто може бути зайнята та й не завжди варто турбувати людину. Звісно інколи бувають ситуації коли просто зручно знати де зараз знаходиться близька людина, щоб зайти та привітатися з нею, звісно за взаємною згодою з обох сторін.

Проте немає застосунків з відкритим вихідним кодом, а користуватися централізованими сервісами, ще й зі закритим кодом не є гарною ідеєю, адже ми не можемо переконатися в надійності цих сервісів, в їх безпечності та приватності, а коли через ці сервіси передаються такі конфіденційні дані як ваше постійне місцеперебування, це може становити серйозну загрозу вашій безпеці.

Найкраще було б розробити застосунок у вигляді протоколу, як то «Wayland», «Matrix» чи «ActivityPub», тоді кожен зможе розробити свою серверну імплементацію чи клієнт, а також модифікувати та покращити сам протокол чи зробити щось нове на основі нього. Також люди зможуть самостійно вивчити роботу протоколу та на основі його проаналізувати розроблені додатки, щоб перевірити їх на безпечність. Протокол не залежить від одного розробника та буде жити навіть коли засновник покине проєкт, або просто перестане його підтримувати.

Таким чином, вирішено створити REST API сервер, що дозволить будь-кому в майбутньому розробити клієнтський додаток будь-якою мовою програмування та під будь-яку платформу чи операційну систему. Якщо підсумувати викладене вище, завдання полягає в розробці серверного

програмного забезпечення для поширення місцеперебування в режимі реального часу разом з автоматично генерованою специфікацією у форматі OpenAPI.

1.2 Аналіз аналогів

Під час розробки сервера будемо покладатися на аналоги, що вже існують. Нас цікавлять децентралізовані та відкриті додатки, що надаються функціонал поширення геолокації та використовують для цього наскрізне шифрування. Як аналог візьмемо протокол «Matrix». Це відкритий протокол децентралізованого обміну повідомленнями, що дозволяє листуватися з використанням наскрізного шифрування та у децентралізованому режимі, коли кожен може підняти свій особистий та незалежний сервер. Також відкритість протоколу дозволяє будь-кому створити свій власний клієнт чи сервер, а також впевнитися в надійності та приватності свого листування.



Рисунок 1.1: Логотип протоколу Matrix[1]

Однією з функцій месенджера на базі цього протоколу є поширення геолокації, проте ця функція має суттєві недоліки. Розглянемо їх краще на прикладі використання одного з багатьох клієнтів протоколу «Matrix», а саме

«ShildiChat Next». Створимо кімнату та надішлемо кілька повідомлень з геомітками. Вигляд клієнту, процес створення кімнати та вигляд геоміток продемонстровано нижче (рис 1.2).

1. В шифруванні — імплементація шифрування в клієнті має суттєвий недолік, як відсутність кешування повідомлень, із за чого підвантаження нових повідомлень вимагає багато часу, інколи це кілька хвилин. Це призводить до негативного досвіду з боку користувача. Дійсно протокол не розроблявся під конкретну таку ціль, а цей функціонал радше як додатковий та не призначений для надсилання місцеперебування на постійній основі.
2. Карта — карту з розташуванням не можна закріпити внизу чату із за чого потрібно постійно гортати в гору, аби знайти повідомлення з картою, це негативно відображається на досвіді користувача.
3. Сповіщення — із за того, що специфікація поширення геолокації є додатковою, не всі клієнти вміють з нею коректно працювати та створюють сповіщення кожен раз коли приходить оновлення розташування.
4. Поганий інтернет — протокол не має гарної оптимізації в умовах поганого з'єднання з інтернетом, тож потребує високі вимоги, а коли з'єднання різко гіршає то ми можемо зустріти баг, коли повідомлення не може відправитися та із за цього блокує чергу відправлення наступних повідомлень.
5. Відсутність історії — на карті ми можемо бачити місцеперебування людини в конкретний момент, проте не можемо побачити її маршрут та де перебував її пристрій востаннє перед тим як вона зникла зі зв'язку, наприклад.

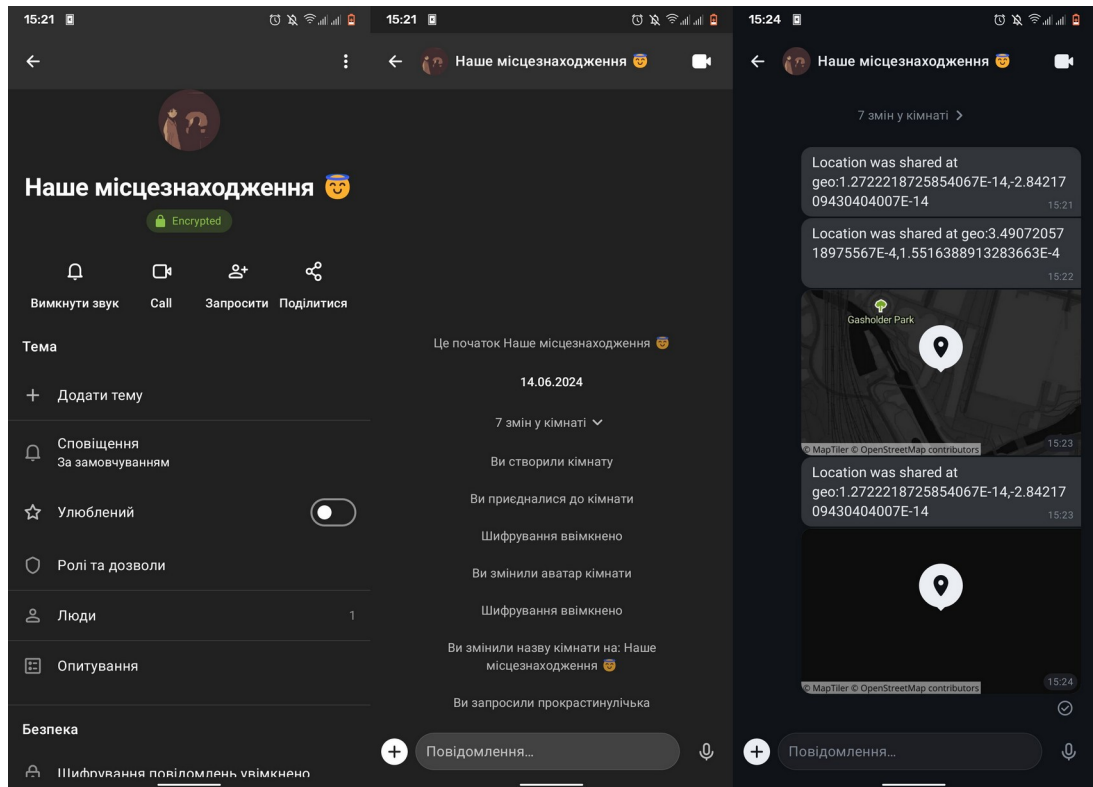


Рисунок 1.2: «Додаток SchildiChat Next»

З цього протоколу ми можемо запозичити такі критерії:

1. Відкритість протоколу — дає змогу вільно розробляти клієнти та сервери, або перевірити ті, що вже існують на захищеність, що дає 99.9% впевненість в надійності.
2. Наскрізне шифрування — шифрування забезпечує повну захищеність листування, коли навіть адміністрація не може побачити вміст.
3. Децентралізація — можливість підняти власний сервер.

1.3 Вибір архітектури проєкту

Проект матиме клієнт серверну архітектуру та буде побудований на фреймворку Django REST Framework, який детальніше розглянемо в наступному параграфі. Фреймворк має чітку структуру на якій можна швидко і зручно побудувати REST API. У фреймворк входять кілька основних елементів

які працюють по конвеєру передаючи запит далі та далі, а потім відповідь навпаки, це дозволяє писати послідовний і зрозумілий та масштабований код. В основні компоненти фреймворку входять:

1. «Views», що відповідають за поверхневу логіку
2. «Serializers», що відповідають за перетворення даних в JSON і навпаки
3. «Models», що відповідають за абстрактне відображення бази даних
4. «Urls», визначають маршрут за адресою

Крім того, загальна архітектура проєкту складається з кількох рівнів:

1. Клієнтський рівень, що є прикладним програмним інтерфейсом, документація до нього, що дасть розробникам зручний спосіб вивчити та скористуватися програмним забезпеченням, а також графічним інтерфейсом, панеллю адміністратора для зручного користування адміністраторам сервера.
2. Серверний рівень, побудований на Django REST Framework, який обробляє запити від клієнта, виконує бізнес-логіку і взаємодіє з базою даних. Цей рівень є ядром системи, де відбувається основна обробка даних.
3. Рівень бази даних, система управління базами даних (PostgreSQL чи Sqlite3 на вибір), яка зберігає всі дані, необхідні для функціонування додатка. Взаємодія з базою даних здійснюється через ORM (Object-Relational Mapping), що надається Django на якому побудований Django REST Framework.

Використання Django REST Framework дозволяє розробляти ефективні та масштабовані додатки, зручні у підтримці та розширенні. Завдяки чіткій розділеності клієнтського та серверного рівнів, а також інтеграції з базою даних, проєкт забезпечить високу гнучкість і зручність в розробці та підтримці.

1.4 Обґрунтування вибору інструментальних засобів та вимог до програмного забезпечення

Серед інструментів оберемо мову програмування Python, фреймворк Django REST Framework та базу даних PostgreSQL[11].

Python, це гнучка, сучасна високорівнева, об'єктно орієнтована мова програмування спрямована на швидку розробку та гнучкість. Разом з цим ця мова доволі швидка та потужна для опрацювання запитів до сервера, тож на ній написані Django, та його форк Django REST Framework.

Django REST Framework було вибрано враховуючи такі його переваги:

1. Server-Side Rendering (SSR), що дозволяє генерувати HTML на сервері та відправляти готові сторінки клієнту. Це забезпечує швидке завантаження сторінок, покращуючи продуктивність і досвід користувачів, особливо на повільних з'єднаннях або старих пристроях. Крім того, SSR сприяє покращенню SEO, оскільки пошукові системи можуть краще індексувати повністю рендерені сторінки. Завдяки вбудованим шаблонам і потужним можливостям рендерингу, Django є чудовим вибором для створення високопродуктивних і SEO-дружніх вебдодатків.
2. Django забезпечує автоматично згенеровану, повністю налаштовану адміністративну панель, яка дозволяє розробникам швидко створювати зручні інтерфейси для управління контентом і даними. Ця адмінпанель інтегрується з ORM, підтримує аутентифікацію та авторизацію, налаштовується під конкретні потреби проєкту, що значно прискорює розробку та зменшує витрати на створення окремих панелей управління. Це робить Django ідеальним вибором для швидкого розгортання та ефективного управління вебдодатками.
3. Django ORM надає високий рівень абстракції над базою даних, що дозволяє працювати з даними через об'єкти Python, спрощуючи та

прискорюючи процес розробки. Завдяки вбудованій системі міграцій, безпеки, з різними базами даних та інтеграції з іншими компонентами фреймворку.

4. Влаштована локалізація, що дозволяє зручно та швидко перекладати додаток на різні мови.

Для зберігання даних обрано PostgreSQL, оскільки це потужна, надійна та масштабована система керування базами даних з відкритим вихідним кодом. PostgreSQL підтримує складні запити, транзакції, збережені процедури та інші розширені функції, що робить її ідеальною для використання в проєктах, де потрібна висока продуктивність та надійність.

Проте в користувачів також буде змога обрати Sqlite3 базу даних, яка є значно простішою в користуванні та дозволить зменшити обсяг сервера, адже це файлова база даних і для неї не буде потрібно підіймати окремий сервер.

Завдяки інтеграції цих баз даних з Django ORM, робота стає ще простішою, проте залишається ефективною.

Висновки до першого розділу

Перший розділ цієї роботи висвітлив проблему відсутності відкритого рішення для обміну геолокацією з використанням наскрізного шифрування. Було проаналізовано централізовані та децентралізовані сервіси що вже існують, які не надають публічного доступу до свого коду, що ставить під сумнів їх надійність та безпеку. Визначено, що для забезпечення високого рівня приватності та безпеки необхідно розробити власний програмний інтерфейс з відкритою документацією та кодом.

Визначною метою проєкту є створення REST API сервера для обміну геолокацією в реальному часі з відкритою специфікацією OpenAPI.

Аналіз аналогів, зокрема протоколу «Matrix», що дозволило визначити важливі критерії для нашого проєкту: відкритість протоколу, наскрізне шифрування та децентралізація. Було виявлено недоліки в реалізації поширення геолокації у клієнті «ShildiChat Next», які включають проблеми з шифруванням, відсутність кешування повідомлень, незручність користування картою, неадекватне сповіщення та відсутність історії місцеперебування.

Вибір архітектури проєкту, де було обрано клієнт-серверну архітектуру на базі фреймворку Django REST Framework. Було детально розглянуто основні компоненти фреймворку та його переваги для реалізації нашого проєкту.

Обґрунтування вибору інструментальних засобів та вимог до програмного забезпечення, де було обрано мову програмування Python, фреймворк Django REST Framework та базу даних PostgreSQL. Було підкреслено переваги цих інструментів, включаючи швидкість розробки, потужність обробки запитів, зручність генерації HTML на сервері, налаштовану адміністративну панель та високий рівень абстракції над базою даних.

Проаналізувавши проблему, було чітко визначено мету та завдання проєкту. На основі мети було проаналізовано аналоги, виділено їхні переваги та недоліки, що дозволило вибрати відповідну архітектуру та інструменти для досягнення поставленої мети. Таким чином, даний проєкт надасть можливість різним людям та компаніям інтегрувати систему обміну геолокацією, що забезпечує високий рівень приватності та безпеки.

РОЗДІЛ 2. ПРОЄКТУВАННЯ

2.1 Варіанти використання системи

Для початку вирішимо як користувачі користуватимуться нашим додатком. Уявимо себе в ролі цільової аудиторії та вирішимо, що нам важливо мати в додатку.

Отже, це в першу чергу незалежне self-host рішення, що надає можливість будь-кому стати адміністратором власного сервера, а також використовувати чужий сервер замість централізованого. Система є майже самостійною і не потребує регулярної модерації, адміністратор потрібний лише для того, щоб підняти сервер і налаштувати його. Проте для адміністратора все ж присутня адміністративна панель. Звичайному користувачу ж доступні всі функції для поширення своєї геолокації захищеним шляхом, навіть на чужому сервері. Отже, ми можемо поділити користувачів на звичайних та адміністраторів. Для демонстрації взаємодії користувачів системи, було створено діаграму варіантів використання (рис 2.1).

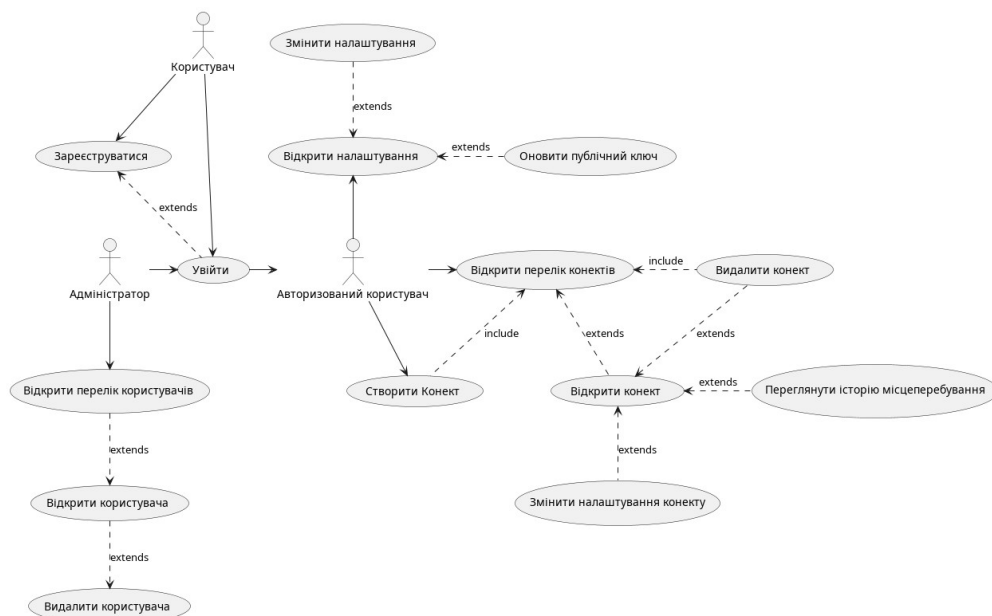


Рисунок 2.1: Діаграма варіантів використання

Для зручності опишемо прецеденти в наступних таблицях (див. Табл. 2.1-табл. 2.6).

Таблиця 2.1

Опис прецеденту «Зареєструватися»

Назва прецеденту	Зареєструватися
Актори	Користувач
Опис прецеденту	Користувачі можуть зареєструватися на сервері.

Таблиця 2.2

Опис прецеденту «Увійти»

Назва прецеденту	Увійти
Актори	Користувач, Адміністратор
Опис прецеденту	Актори можуть увійти в систему та отримати доступ до інших прецедентів.

Таблиця 2.3

Опис прецеденту «Відкрити перелік конектів»

Назва прецеденту	Відкрити перелік конектів
Актори	Авторизований користувач, Адміністратор
Опис прецеденту	Актори можуть переглянути список раніше створених конектів, відкрити конект, змінити його налаштування чи видалити.

Таблиця 2.4

Опис прецеденту «Відкрити перелік користувачів»

Назва прецеденту	Відкрити перелік користувачів
Актори	Адміністратор
Опис прецеденту	Адміністратори можуть переглядати користувачів та видаляти за потреби.

Таблиця 2.5

Опис прецеденту «Відкрити налаштування»

Назва прецеденту	Відкрити налаштування
Актори	Адміністратор, Авторизований користувач
Опис прецеденту	Актори мають можливість

	переглядати свої налаштування, змінювати їх, а також змінювати свій публічний ключ.
--	---

Таблиця 2.6

Опис прецеденту «Створити конект»

Назва прецеденту	Створити конект
Актори	Адміністратор, Авторизований користувач
Опис прецеденту	Актори мають можливість створити новий конект, після чого їх буде переадресовано на сторінку всіх конектів.

2.2 Опис реалізації функціонала

Для початку потрібно розгорнути Django проєкт, зробити це можна командою «./manage.py startproject kwim», після чого ініціалізувати git репозиторій командою «git init». Після створення проєкту першим чином налаштуємо «settings.py». Це відправна точка для додатка, потребує одноразового налаштування та визначає як працюватиме сервер.

```
from datetime import timedelta
from pathlib import Path
from environs import Env
```

Імпортуємо необхідні модулі для подальшого використання.

```
env = Env()
env.read_env(".env")
```

Модуль environs дозволяє зручно та швидко імпортувати зміни середовища.

```
BASE_DIR = Path(__file__).resolve().parent.parent
```

Дізнаємося корневий шлях проєкту.

```
SECRET_KEY = env("DJANGO_SECRET_KEY")
DEBUG = env.bool("DJANGO_DEBUG", default=False)
```

Імпортуємо основні ключі для роботи фреймворку.

```
ALLOWED_HOSTS: list = [
    ...
```

```
]
```

Цей список потрібно заповнити доменами чи IP адресами, через які буде доступ до сервера.

```
INSTALLED_APPS = [  
    "django.contrib.admin",  
    "django.contrib.auth",  
    "django.contrib.contenttypes",  
    "django.contrib.sessions",  
    "django.contrib.messages",  
    "django.contrib.staticfiles",  
  
    # REST Framework  
    "rest_framework",  
    "knox",  
]
```

Нам потрібні будуть основні додатки Django, а також окремо імпортуємо Django REST Framework та модуль Knox токенів.

```
MIDDLEWARE = [  
    "django.middleware.security.SecurityMiddleware",  
    "django.contrib.sessions.middleware.SessionMiddleware",  
    "django.middleware.common.CommonMiddleware",  
    "django.middleware.csrf.CsrfViewMiddleware",  
    "django.contrib.auth.middleware.AuthenticationMiddleware",  
    "django.contrib.messages.middleware.MessageMiddleware",  
    "django.middleware.clickjacking.XFrameOptionsMiddleware",  
]
```

Зі стандартного списку проміжних додатків викинемо зайві.

```
ROOT_URLCONF = "kwim.urls"
```

```
TEMPLATES = [  
    {  
        "BACKEND":  
        "django.template.backends.django.DjangoTemplates",
```

```

        "DIRS": [],
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ],
        },
    ],
]

```

WSGI_APPLICATION = "kwim.wsgi.application"

Автоматично генеровані, стандартні налаштування шаблонів нам підійдуть.

```

DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": BASE_DIR / "db.sqlite3",
    }
}

```

Налаштування підключення до бази даних.

```

AUTH_PASSWORD_VALIDATORS = [
    {
        "NAME":
        "django.contrib.auth.password_validation.UserAttributeSimilarityVa
lidator",
    },
    {
        "NAME":
        "django.contrib.auth.password_validation.MinimumLengthValidator",
    },
    {

```

```

        "NAME":
        "django.contrib.auth.password_validation.CommonPasswordValidator",
    },
    {
        "NAME":
        "django.contrib.auth.password_validation.NumericPasswordValidator"
    },
]

```

Валідатори для пароля користувача.

```

TIME_ZONE = env("DJANGO_TIME_ZONE", default="Europe/Kyiv")
...
USE_TZ = True

```

Налаштуємо параметри часу.

```

STATIC_ROOT: Path = BASE_DIR / "staticfiles/"
STATIC_URL: str = "static/"

```

Визначимо директорію та посилання до статичних файлів (HTML/CSS/JS).

```

DEFAULT_AUTO_FIELD = "django.db.models.BigAutoField"
...
REST_FRAMEWORK: dict = {
    "DEFAULT_SCHEMA_CLASS":
    "drf_spectacular.openapi.AutoSchema",
    "DEFAULT_AUTHENTICATION_CLASSES": (
        "knox.auth.TokenAuthentication",
    )
}
...
REST_KNOX: dict = {
    "AUTH_TOKEN_CHARACTER_LENGTH": 64,
    "TOKEN_TTL": timedelta(days=356),
    "USER_SERIALIZER": "users.serializers.UserSerializer",
    "TOKEN_LIMIT_PER_USER": 6,
    "AUTO_REFRESH": False,
    "MIN_REFRESH_INTERVAL": 60,
}

```

```

    "AUTH_HEADER_PREFIX": "Token"
}

```

Перевизначимо влаштовані налаштування фреймворку та токенів.

Коли два користувачі хочуть почати обмінюватися своїм місцеперебуванням потрібно утворити певне з'єднання між ними назовемо це «конектом», від англійського слова «connect».

Опишемо базовий серіалізатор для моделі конекту. Перевизначимо методи «get_inviter» та «get_member», щоб повертати ім'я користувача замість його внутрішнього ключа. Повертаємо також вибірку з симетричних ключів та останніх повідомлень/подій.

```

class ConnectSerializer(serializers.ModelSerializer):
    inviter: serializers.SerializerMethodField =
serializers.SerializerMethodField()

    member: serializers.SerializerMethodField =
serializers.SerializerMethodField()

    keys: KeySerializer = KeySerializer(many=True,
read_only=True, source="keys_of_connect")

    last_messages: EventSerializer =
EventSerializer(many=True, read_only=True,
source="events_of_connect")

class Meta:
    model = Connect

    fields: tuple = ("id", "inviter", "member",
"created_at", "keys", "last_messages")

    def get_inviter(self, instance: Connect) -> str: # noqa
        return f"{instance.inviter.username}" # noqa

    def get_member(self, instance: Connect) -> str: # noqa
        return f"{instance.member.username}" # noqa

```

До кожного конекту буде прив'язано певний ключ шифрування, цим ключем ми шифруватимемо повідомлення з геолокацією, яке далі називатимемо

подією. Опишемо серіалізатор також для симетричного ключа. Перевизначимо метод «get_recipient» з тією ж метою що і попередні методи.

```
class KeySerializer(serializers.ModelSerializer):
    recipient: serializers.SerializerMethodField | str =
serializers.SerializerMethodField(read_only=True)

    class Meta:
        model = Key
        fields: tuple = ("id", "recipient", "key",
"created_at", "read")

    def get_recipient(self, instance: Connect) -> str: # noqa
        return f"{instance.recipient.username}" # noqa
```

Зашифровані повідомлення з координатами надіслані користувачами назвемо подіями. Напишемо серіалізатор для надсилання події. Координати перевірити на жаль неможливо, тому просто зберігатимемо дані в форматі Base64. Користувача беремо із запиту, та перевіряємо чи він є членом конекту.

```
class SendEventSerializer(serializers.ModelSerializer):
    class Meta:
        model = Event
        fields: tuple = ("lat", "long", "connect",
"created_at")
        read_only_fields: tuple = ("created_at",)

    def validate(self, attrs: dict) -> dict:
        """
        1. Get sender from context.
        2. Check if sender is a member of connect.
        """

        attrs["sender"] = self.context["request"].user

        if attrs["sender"] not in (
```

```

        attrs["connect"].member,
        attrs["connect"].inviter
    ):
        raise serializers.ValidationError(detail=_("You
are not a member of connect."))

    return attrs

```

Щоб додати автоматично генеровану документацію (OpenAPI[5]), скористаємося модулем drf-spectacular[4]. Після встановлення пакета, потрібно налаштувати «settings.py» та «urls.py» в головному додатку проєкту.

В «settings.py» потрібно додати додаток, та додати словник з налаштуваннями.

```

INSTALLED_APPS = [
    ...
    "drf_spectacular",
    "drf_spectacular_sidecar",
    ...
]

SPECTACULAR_SETTINGS: dict = {
    "SWAGGER_UI_DIST": "SIDECAR",
    "SWAGGER_UI_FAVICON_HREF": "SIDECAR",
    "REDOC_DIST": "SIDECAR",
    "TITLE": "The kwim API",
    "DESCRIPTION": "kwim is the app for sharing location with
E2EE",
    "VERSION": "0.12.1",
    "SERVE_INCLUDE_SCHEMA": False,
}

```

Версія буде змінюватися поступово з розробкою і форматоватиметься згідно з семантичним версіюванням[6].

Далі налаштуємо «urls.py», вказавши шлях до документації.

```

from drf_spectacular.views import (
    SpectacularAPIView, SpectacularSwaggerView

```

```

)
from django.urls import path

urlpatterns = [
    # Swagger docs
    path("api/schema/", SpectacularAPIView.as_view(),
name="schema"),
    path("api/schema/swagger-ui/",
SpectacularSwaggerView.as_view(url_name="schema"), name="swagger-
ui"),
    ...
]

```

Налаштуємо локалізацію. Для цього додамо конфігураційні рядки в «settings.py».

```

...
LANGUAGE_CODE = env("DJANGO_LANGUAGE_CODE", default="uk-ua")
...
USE_I18N = True
...
LOCALE_PATHS: list = [
    BASE_DIR / "locale/"
]

```

Щоб додати рядок до перекладу потрібно імпортувати метод перекладу та передати потрібний рядок.

```

from django.utils.translation import gettext_lazy as _
...
raise serializers.ValidationError(detail=_("You are not a
member of connect."))

```

Після чого потрібно згенерувати файл перекладу для української мови командою «./manage.py makemessages -l uk_UA -i venv». В директорії «locale» буде згенерований файл з розширенням «.po», потрібно змінити всі пусті рядки відповідно до перекладу, після чого скомпілювати переклад командою «./manage.py compilemessages -l uk_UA -i venv».

2.3 Структура бази даних

База даних побудована з використанням Django ORM[3], де кожна таблиця представлена як «модель», клас який описує які поля має таблиця та дозволяє діставати з бази даних записи перетворюючи їх на змінні типу даних мови Python.

Для початку створимо модель користувача (таблицю) «User», де зберігатиметься інформація про користувача та дані для входу в систему яка міститиме такі поля:

1. «username» — унікальне ім'я користувача обмежене 30 символами.
2. «is_active» — поле для позначення користувача активним або ні. Коли адміністратор зробить користувача неактивний, той не зможе увійти в обліковий запис, фактично він буде заблокованим.
3. «is_admin» — позитивне значення в цьому полі надає користувачу доступ до адміністративної панелі.
4. «first_name», «last_name» — повне ім'я користувача. Ці поля не є обов'язковими та заповнюються користувачем за бажанням.
5. «public_key» — публічний ключ користувача зберігається саме тут.
6. «connect_secret_key», «connect_failed_attempts» — поля пов'язані з конектом, про які буде згадано в наступному параграфі.

```
class User(AbstractBaseUser):
    username: models.CharField =
models.CharField(max_length=30, unique=True,
verbose_name=_("Username"))
    is_active: models.BooleanField =
models.BooleanField(default=True, verbose_name=_("Is active?"))
    is_admin: models.BooleanField =
models.BooleanField(default=False, verbose_name=_("Is admin?"))
    first_name: models.CharField = models.CharField(
        null=True, blank=True, max_length=30,
```

```

        unique=False, verbose_name=_("First name"))
    )
    last_name: models.CharField = models.CharField(
        null=True, blank=True, max_length=30,
        unique=False, verbose_name=_("Last name"))
    )
    public_key: models.TextField =
models.TextField(max_length=512, null=True, blank=True,
verbose_name=_("Public key"))
    connect_secret_key: models.CharField = models.CharField(
        max_length=7, null=False, blank=False,
        default=generate_connect_secret_key,
verbose_name=_("Connect secret key"))
    )
    connect_failed_attempts: models.PositiveSmallIntegerField
= models.PositiveSmallIntegerField(
        null=False, blank=False, default=0,
verbose_name=_("Number of failed connection attempts"))
    )

```

Створимо таблицю «Connect», що містить інформацію про з'єднання між двома користувачами та має два поля «inviter» і «member», що відрізняються лише назвою, бо є зовнішніми ключами та посилаються на таблицю користувача. Також додатково додамо поле «created_at», що зберігає дату створення конекту.

```

class Connect(models.Model):
    created_at: models.DateTimeField = models.DateTimeField(
        auto_now_add=True, null=False, blank=True,
        verbose_name=_("Created at"))
    )
    inviter: models.ForeignKey | User = models.ForeignKey(
        to=User, null=True, blank=False,
on_delete=models.SET_NULL,
        related_name="connects_as_inviter",
related_query_name="connects_as_inviter",
verbose_name=_("Inviter"))

```

```

    )
    member: models.ForeignKey | User = models.ForeignKey(
        to=User, null=True, blank=False,
on_delete=models.SET_NULL,
        related_name="connects_as_member",
related_query_name="connects_as_member", verbose_name=_("Member")
    )

```

До кожного конекту буде прив'язано один або кілька симетричних ключів, тож створимо модель для ключів «Key», що міститиме ці ключі з такими полями:

1. «created_at» — дата відправлення ключа.
2. «connect» — зовнішній ключ до конекту.
3. «recipient» — зовнішній ключ на користувача якому було адресовано ключ.
4. «key» — асиметричний ключ попередньо зашифрований публічним ключем отримувача.
5. «read» — поле що зберігає статус, чи повідомлення прочитане.

```

class Key(models.Model):
    created_at: models.DateTimeField = models.DateTimeField(
        auto_now_add=True, null=False, blank=True,
        verbose_name=_("Created at")
    )
    connect: models.ForeignKey = models.ForeignKey(
        to=Connect, null=True, blank=False,
on_delete=models.SET_NULL,
        related_name="keys_of_connect",
related_query_name="keys_of_connect", verbose_name=_("Connect")
    )
    recipient: models.ForeignKey = models.ForeignKey(
        to=User, null=True, blank=False,
on_delete=models.SET_NULL,

```

```

        related_name="connects_keys_for_user",
related_query_name="connects_keys_for_user",
verbose_name=_("Recipient")
    )
    key: models.TextField = models.TextField(
        max_length=1024, null=False, blank=False,
        verbose_name=_("Encrypted key")
    )
    read: models.BooleanField = models.BooleanField(
        null=False, default=False, verbose_name=_("Read")
    )

```

Останньою моделлю буде «Event», що містить інформацію про події, пов'язані з з'єднаннями та зберігає повідомлення надісланні користувачами. Вона має такі поля:

1. «connect» — зовнішній ключ до конекту, якому це повідомлення належить.
2. «sender» — зовнішній ключ до користувача, яким це повідомлення було відправлене.
3. «lat» — поле, що зберігає широту в зашифрованому вигляді.
4. «long» — поле, що зберігає довготу в зашифрованому вигляді.
5. «created_at» — дата відправлення повідомлення.

```

class Event(models.Model):
    connect: models.ForeignKey | Connect = models.ForeignKey(
        null=False, blank=False, to=Connect,
related_name="events_of_connect",
related_query_name="events_of_connect",
        on_delete=models.CASCADE, verbose_name=_("Connect")
    )
    sender: models.ForeignKey | User = models.ForeignKey(
        null=False, blank=False, to=User,
on_delete=models.CASCADE, verbose_name=_("Sender")
    )

```

```

lat: models.CharField = models.CharField(
    null=False, blank=False, max_length=34,
    verbose_name=_("Latitude")
)
long: models.CharField = models.CharField(
    null=False, blank=False, max_length=34,
    verbose_name=_("Longitude")
)
created_at: models.DateTimeField = models.DateTimeField(
    null=False, blank=False, auto_now_add=True,
    verbose_name=_("Created at")
)

```

Схематичне зображення повної бази даних можна побачити на (рис. 2.2).

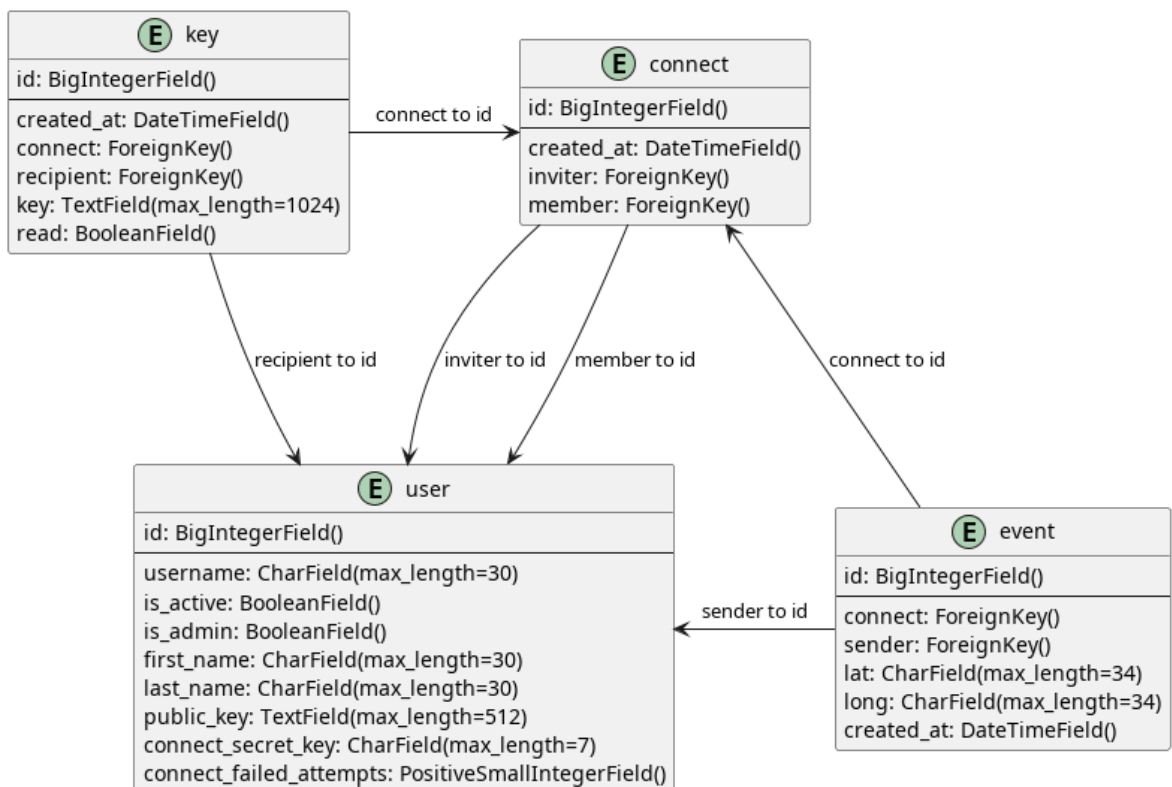


Рисунок 2.2: Схематичне зображення бази даних

2.4 Проєктування та реалізація алгоритмів роботи

Найскладнішим і найголовнішим у цьому проєкті, є реалізація наскрізного шифрування. Фактично це є кістком проєкту, адже без реалізації такого на початку, сенсу в поставленій задачі не буде.

Будь-хто знайомий з базовим шифруванням скаже, що ми можемо просто використати асиметричні RSA[7] ключі, щоб шифрувати ними повідомлення. Дійсно, за допомогою публічних і приватних ключів можна забезпечити надійну безпеку інформації, що передається через сервер, адже ключі для розшифрування ніколи не потраплять на сервер.

Проте не все так просто, якщо ми спробуємо реалізувати наскрізне шифрування, то стикнемося з одною проблемою. Уявімо ситуацію, коли люди активно користуються додатком з місяць, додаток відсилає місцеперебування кожні 5 хвилин, тоді на добу припадатиме 288 записів $((24*60)/5)$, а в місяць буде 8928 записів $(288*31)$, множимо це на 2, бо в конекті два користувачі та отримуємо фінальну цифру 17856. Якщо ми збільшимо частоту оновлення, або кількість конектів, кількість буде значно більша, а це своєю чергою впливає на кількість зайнятого місця та тривалість розшифрування. Дійсно немає сенсу тримати таку кількість записів в себе на пристрої, коли вони зберігаються на сервері. Тож припустимо наш додаток (клієнт) чистить кеш і тримає записи лише за останній місяць, користувач при цьому хоче подивитися події, що були раніше, тому клієнт вантажить їх з сервера і намагається розшифрувати. Ось тут і з'являється проблема, адже ключ для розшифрування чужих події є, а для розшифрування своїх, нема, бо вони були зашифровані чужими публічними ключами. Та ж проблема проявляється якщо ми захочемо зайти в додаток з комп'ютера, уявімо ми загубили телефон, він продовжує у фоні відправляти місцеперебування, але з комп'ютера нічого не видно, що й не дивно, повідомлення зашифровані чужими ключами.

					ІПЗ.КР.Б – 121 – 24 – ПЗ	Аркуш 32
Зм.	Арк.	№ докум.	Підпис	Дата		

Що ж, з цього можна знайти вихід, згадаємо про симетричні AES ключі, щоб зашифрувати та розшифрувати повідомлення нам потрібно мати той самий ключ. Якщо домовитися з кимось про шифрування одним AES ключем, ми зможемо обмінюватися повідомленнями в зашифрованому вигляді. Проте для того, щоб обмінятися симетричним ключем, нам треба зустрітися в реальному житті, це не зручно, бо кожен раз коли є потреба змінити ключ нам треба зустрічатися, а інколи це просто неможливо та й програмне забезпечення покликане спрощувати нам життя.

Тому ми можемо поєднати асиметричні та симетричні ключі. Шифруватимемо асиметричними ключами симетричні та передаватимемо іншому учаснику конекту, а за потреби симетричні ключі попередньо експортуватимемо з одного пристрою та імпортуватимемо на іншому. Отримаємо прив'язку конект — ключ, або декілька ключів, адже передбачено можливість зміни ключів, наприклад якщо пристрій було скомпрометовано, або просто для кращої безпеки. Асиметричні ключі ж прив'язані до самого облікового запису, на сервері зберігатиметься публічний ключ, щоб учасники мережі могли отримати його і зашифрувати симетричний ключ будь-коли, а приватний ключ зберігатиметься на самому пристрої. Пізніше цю систему можна також вдосконалити, інтегрувавши так зване перехресне підписання, це дозволить передавати симетричні ключі між пристроями без потреби експорту та імпорту.

Для того аби оновлювати ключ, напишемо необхідний серіалізатор. Додамо метод «`validate_public_key`», що з допомогою бібліотеки `rsa[2]` перевірятиме чи справжній публічний ключ завантажений до додатка. Аби уявний зловмисник не зміг завантажити новий публічний ключ зі вкраденої сесії та отримати доступ до нових симетричних ключів, додатково додамо поле «`password`» та метод «`validate_password`», що перевірятиме чи правильно був введений пароль.

```

class UserPublicKeySerializer(serializers.ModelSerializer):
    public_key: fields.CharField =
fields.CharField(max_length=512, min_length=10, required=False)
    password: fields.CharField = fields.CharField(
        write_only=True, min_length=8, max_length=24,
validators=(PasswordValidator(),),
        allow_blank=False, allow_null=False, required=True
    )

class Meta:
    model = User
    fields = (
        "public_key",
        "password"
    )

def validate_public_key(self, value: str) -> str:
    try:
        public_key: rsa.PublicKey =
rsa.PublicKey.load_pkcs1(keyfile=value.encode(encoding="utf-8"))
    except ValueError:
        raise serializers.ValidationError(detail=_("Public
key is invalid.))

    return value

def validate_password(self, value: str) -> str:
    user: User = self.context["request"].user
    if not user.check_password(raw_password=value):
        raise serializers.ValidationError(_("Password is
invalid.))

    return value

```

Щоб створити конект потрібна згода двох людей. Можна звісно зробити систему запрошень, коли ми вписуємо ім'я користувача та надсилаємо запит про

приєднання, проте в основі проєкту лежить конфіденційність і приватність, тож ми не можемо прив'язати обліковий запис, наприклад до номера телефону, щоб захиститися від зловмисників, що могли б спамити іншим користувачам та прикидатися їх знайомими.

Нехай в кожного користувача буде спеціальний код, знаючи який можна створити конект, тоді аби це зробити нам знаходиться ім'я користувача та його код. Закодуємо ці два поля в QR код і отримуємо зручний спосіб утворити конект, одна людина сканує код (запрошений), а інша показує (організатор), готово. Проте, цей код можна передати та на відстані, шляхом у якому користувачі будуть впевнені, наприклад під час відеодзвінка. Для генерації коду напишемо окрему функцію.

```
from string import ascii_lowercase
from random import randint

def generate_connect_secret_key() -> str:
    """
    Generate secret key like this "1h9-a12".
    """

    symbols: tuple = tuple([letter for letter in
ascii_lowercase] + [number for number in range(0, 10)])

    secret_key: str = ""

    for i in range(0, 6):
        if i == 3:
            secret_key = f"{secret_key}-"

        secret_key = f"{secret_key}{symbols[randint(0,
len(symbols)-1)]}"
```

```
return secret_key
```

Рахуватимемо також кількість хибних спроб, якщо він перевищує скажімо 3 хибні спроби, ми можемо сповістити учасника мережі про те, що хтось намагався підібрати код для того, щоб приєднатися, а також змінимо код цього користувача. Також можна інтегрувати можливість заблокувати створення конектів в разі такого сценарію.

Для створення конекту напишемо окремий серіалізатор, що міститиме в собі кілька валідацій.

```
class CreateConnectSerializer(serializers.Serializer):
    inviter_username: serializers.CharField =
serializers.CharField(
        min_length=3, max_length=30, allow_blank=False,
allow_null=False, required=True,
        write_only=True
    )
    inviter_secret_code: serializers.CharField =
serializers.CharField(
        min_length=7, max_length=7, allow_blank=False,
allow_null=False, required=True,
        write_only=True
    )

    def validate(self, attrs: dict) -> dict:
        """
        1. Check if inviter exist.
        2. Check if user don't create connect with himself.
        3. Check if connect already exist.
        4. Check if secret code is valid.
        """

        try:
            user: User =
User.objects.get(username=attrs["inviter_username"])
        except User.DoesNotExist: # noqa
```

```

        raise serializers.ValidationError(detail=_("User
with this username does not exist.))

        if user == self.context["request"].user:
            raise serializers.ValidationError(detail=_("You
can't create the connect with yourself.))

        try:
            connect: Connect =
Connect.objects.get(inviter=user,
member=self.context["request"].user) # noqa
            raise
serializers.ValidationError(detail=_("Connect already exist.))
        except Connect.DoesNotExist: # noqa
            pass

        try:
            connect: Connect =
Connect.objects.get(inviter=self.context["request"].user,
member=user) # noqa
            raise
serializers.ValidationError(detail=_("Connect already exist.))
        except Connect.DoesNotExist: # noqa
            pass

        if user.connect_secret_key !=
attrs["inviter_secret_code"]:
            user.connect_failed_attempts += 1
            user.save()
            raise serializers.ValidationError(detail=_("Secret
code is invalid.))

        attrs["inviter"] = user
        attrs["member"] = self.context["request"].user

        return attrs

```

На основі вище описаної інформації можна утворити наступу схему життєвого циклу, що дасть краще зрозуміти як це працює (див. рис. 2.3).

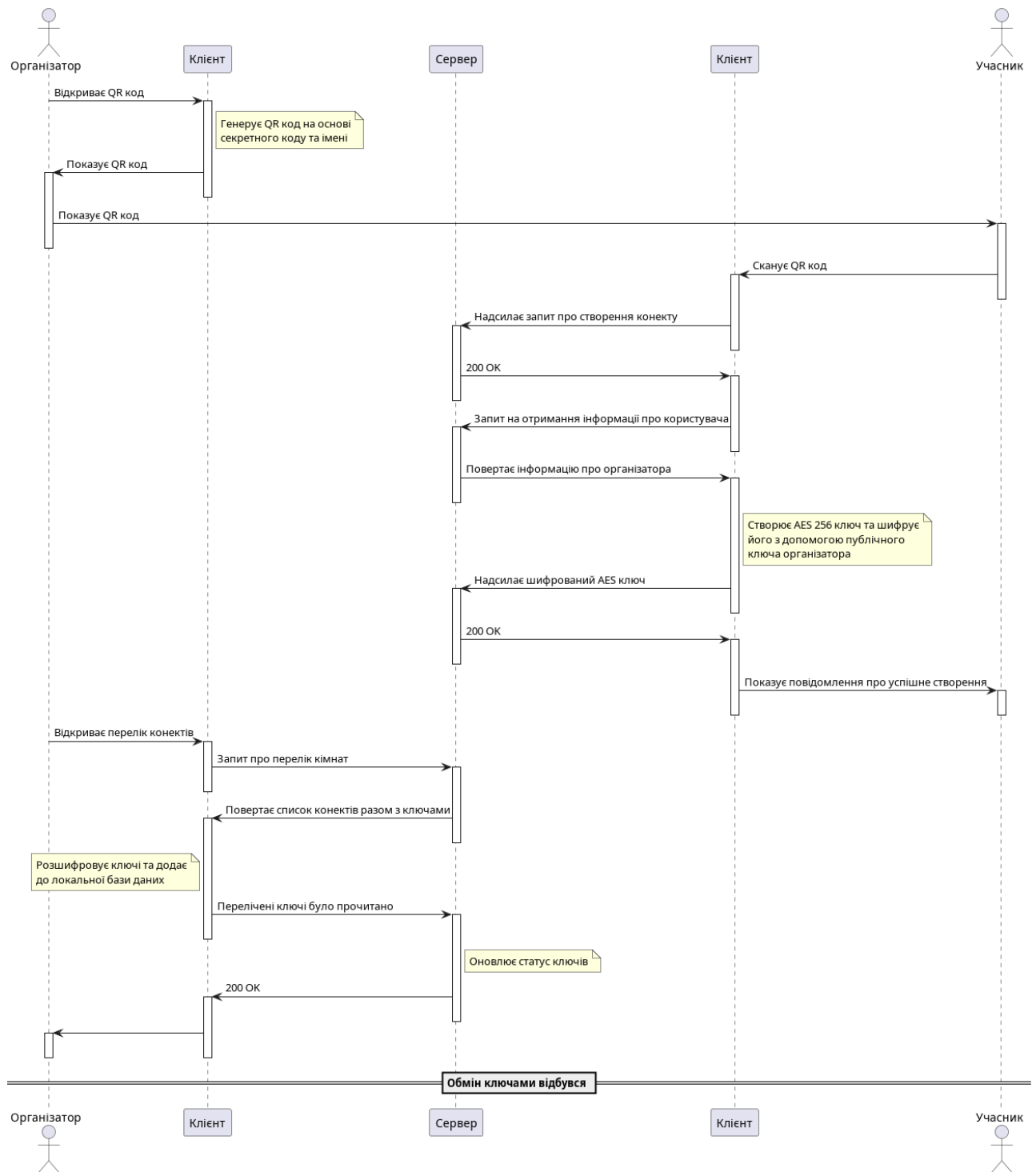


Рисунок 2.3: Схема створення конекту та обміну ключами

2.5 Налаштування розгортання

Для розгортання опишемо конфігурацію Docker[8], для збирання знімка, а також конфігурацію docker-compose[9] для підняття сервера. Конфігурація для знімка буде автентичною для будь-якої Unix подібної системи.

```
FROM "python:3.12-alpine"
```

```
WORKDIR /code/
```

```
COPY ./requirements.txt .
```

```
RUN pip install -r /code/requirements.txt
```

```
COPY . .
```

```
ENTRYPOINT python manage.py migrate && python manage.py  
collectstatic --noinput && daphne kwim.asgi:application -b 0.0.0.0  
-p 8010
```

Знімок побудований на основі дистрибутиву Linux — Alpine, що славиться своєю мінімалістичністю. В наслідуваному дистрибутиві також встановлений інтерпретатор Python. Під час збірки створюється окрема директорія, туди копіюється файл з переліком залежностей та встановлюються всі залежності, після чого копіюються всі файли проєкту в ту ж директорію. Останній рядок конфігурації вказує команду яку треба виконати при запуску знімка, а саме виконати міграції базу даних, зібрати статичні елементи та запустити daphne[10] сервер.

Створимо мінімальний приклад конфігурації docker-compose.

```
version: "3.9"
```

```
services:
```

```
  django:
```

```
    build: .
```

```

env_file: ".env"
ports:
  - "8010:8010"
volumes:
  - ".:code"
restart: unless-stopped

```

Створимо один сервіс - «django» та вкажемо знімок як крапку, це значить, що зібрати треба за інструкцією у файлі Dockerfile, що лежить в тій же директорії. Вкажемо файл віртуального оточення, бажаний порт та додамо посилання на теку з кодом. Параметр «restart» вказує на автоматичне перезавантаження докера в разі перезапуску фізичного сервера.

Цього буде достатньо, проте за бажанням можна додати локальну базу даних PostgreSQL.

Висновки до другого розділу

У другому розділі ми детально розглянули процес проєктування системи обміну геолокацією з використанням наскрізного шифрування. Основними аспектами, які були розглянуті в цьому розділі, стали варіанти використання системи, реалізація функціонала, структура бази даних, та алгоритми роботи, включаючи шифрування.

Ми визначили основні сценарії використання системи, враховуючи два типи користувачів: звичайних користувачів і адміністраторів. Для наочного демонстрації взаємодії користувачів було створено діаграму варіантів використання. Це дозволило чітко окреслити функціонал, доступний кожному типу користувача, включаючи реєстрацію, вхід, перегляд списків конектів та користувачів, налаштування і створення конектів.

Було розглянуто процес розгортання та налаштування Django-проєкту, включаючи налаштування базових компонентів, таких як settings.py, серіалізатори та модулі для автоматично згенерованої документації (OpenAPI).

Особлива увага приділялася налаштуванню аутентифікації за допомогою бібліотеки Кнох, а також реалізації перекладу інтерфейсу на українську мову.

База даних була побудована з використанням Django ORM, де кожна таблиця представлена як модель. Було розроблено моделі для користувачів, конектів, ключів та подій. Кожна модель детально описує поля, які вона містить, і взаємозв'язки між ними. Це дозволяє зберігати всі необхідні дані для забезпечення роботи системи, включаючи інформацію про користувачів, з'єднання, ключі шифрування та події з координатами.

Особливу увагу було приділено реалізації наскрізного шифрування. Для розв'язання проблеми зберігання і розшифрування великих об'ємів даних було запропоновано комбінований підхід, що поєднує асиметричні та симетричні ключі. Це дозволяє забезпечити безпеку і зручність використання системи навіть у випадку втрати пристрою або потреби змінити ключі.

Таким чином, розроблений проєкт системи обміну геолокацією з використанням наскрізного шифрування є гнучким і надійним рішенням, що відповідає вимогам сучасної безпеки даних. Впроваджені рішення дозволяють зберігати конфіденційність та цілісність інформації, а також забезпечують зручність використання системи як для звичайних користувачів, так і для адміністраторів.

РОЗДІЛ 3. ІНТЕРФЕЙС

3.1 Порядок встановлення системи

Опишемо мінімальне встановлення сервера. Для цього використаємо фізичний сервер з операційною системою Arch Linux[12] та Docker. Попередньо на сервер потрібно встановити пакети caddy та ufw, docker, docker-compose, за бажання можна доповнити встановлення іншими пакетами, або замінити перелічені аналогічним, наприклад використати nginx замість caddy. Схематичне зображення налаштованого сервера можна побачити нижче (рис. 3.1).

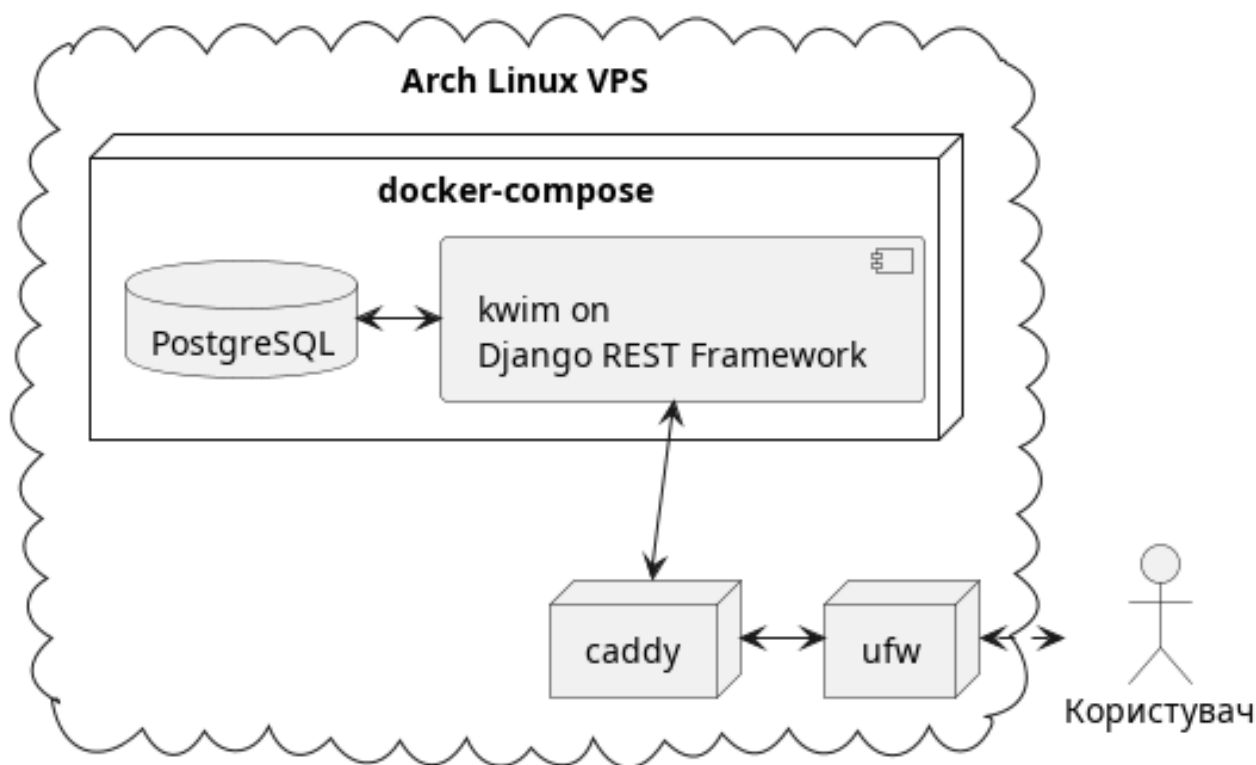


Рисунок 3.1: Схематичне зображення налаштованого сервера

На фізичному сервері клонуємо репозиторій проєкту, краще це зробити в директорію «/srv/kwim». Далі створимо файл «docker-compose.yaml», для цього скопіюємо приклад, який можна знайти в тому ж репозиторії, «docker-compose-

example.yml». В новому файлі потрібно змінити порт з «8010» на бажаний, наприклад на «8084», на цьому порті в localhost буде працювати сервер.

```
version: "3.9"

services:
  django:
    build: .
    env_file: ".env"
    ports:
      - "8084:8010"
    volumes:
      - "../code"
    restart: unless-stopped
```

Тепер потрібно зібрати код, в репозиторії вже знаходиться інструкція для Docker, тож достатньо просто ввести команду «docker compose build». Якщо команда виконалася успішно, можна запускати сервер, проте йому ще буде потрібно налаштування, що зберігаються в віртуальному середовищі, аби задати них потрібно створити «.env» файл. В репозиторії також передбачено приклад цього файлу «example.env», достатньо скопіювати його і змінити змінні оточення на свої.

Тепер запустимо сервер у фоні командною «docker compose up -d», він автоматично перезапуститься в разі перезапуску сервера, або помилки. Сервер запущено локально, тепер потрібно налаштувати обернений проксі з допомогою caddy.

Потрібно створити DNS запит до фізичного сервера. Після чого можна налаштовувати caddy, змінивши файл за шляхом «/etc/caddy/Caddyfile».

```
{
    admin "unix//run/caddy/admin.socket"
}

kwim.forblog.live {
    handle_path /static/* {
```

```

        file_server
        root * /srv/kwim/staticfiles
    }

reverse_proxy 127.0.0.1:8084
}

import /etc/caddy/conf.d/*

```

Тепер сервер доступний за вказаним доменом. Спробуємо увійти в адміністративну панель, для цього спочатку створити першого користувача в системі, адміністратора, для цього виконаємо команду «`docker compose exec django python manage.py createsuperuser`».

3.2 Інтерфейс адміністратора

Інтерфейс адміністратора представлений у вигляді вебінтерфейсу, що автоматично генерується на основі коду моделей бази даних і форм.

Щоб потрапити до вебінтерфейсу потрібно перейти за адресою сервера додавши «`/admin/`» до посилання, ми побачимо в браузері вхід у панель адміністратора (рис. 3.2).

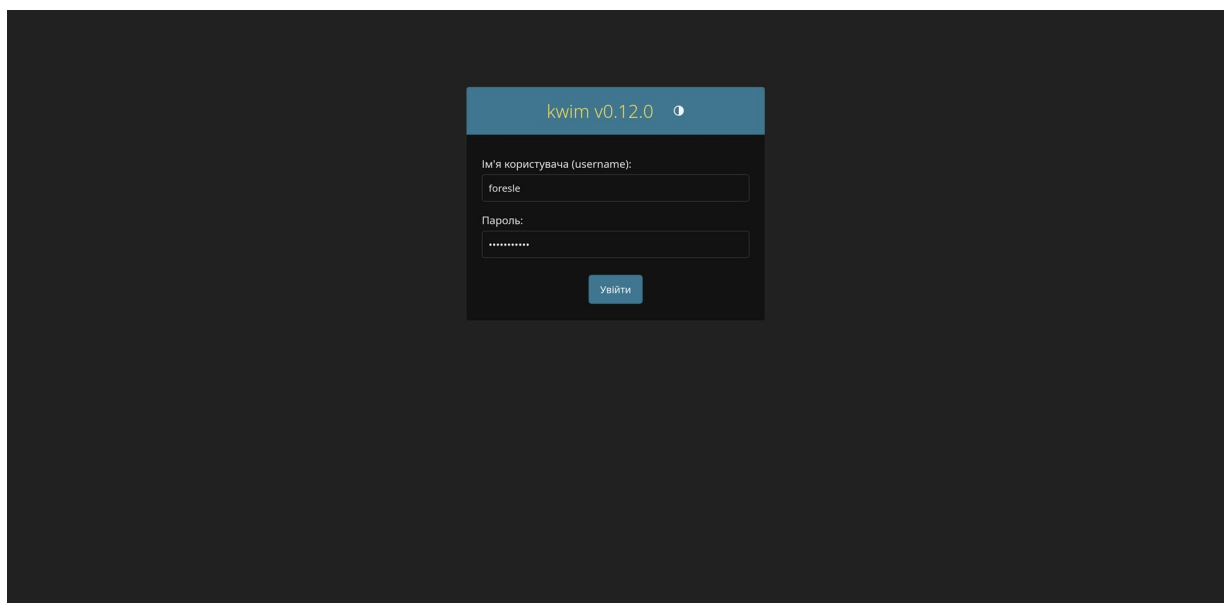


Рисунок 3.2: Сторінка входу в адміністративну панель

Вхід реалізований як форма входу з полями для пароля та імені. Над формою розміщений заголовок, що містить також актуальну версію встановленого сервера.

Увійти в панель адміністратора можна з даними користувача, що ми створили раніше командою. Це обліковий запис є першим адміністратором, за допомогою нього можна створити також інших адміністраторів вже з адміністративної панелі.

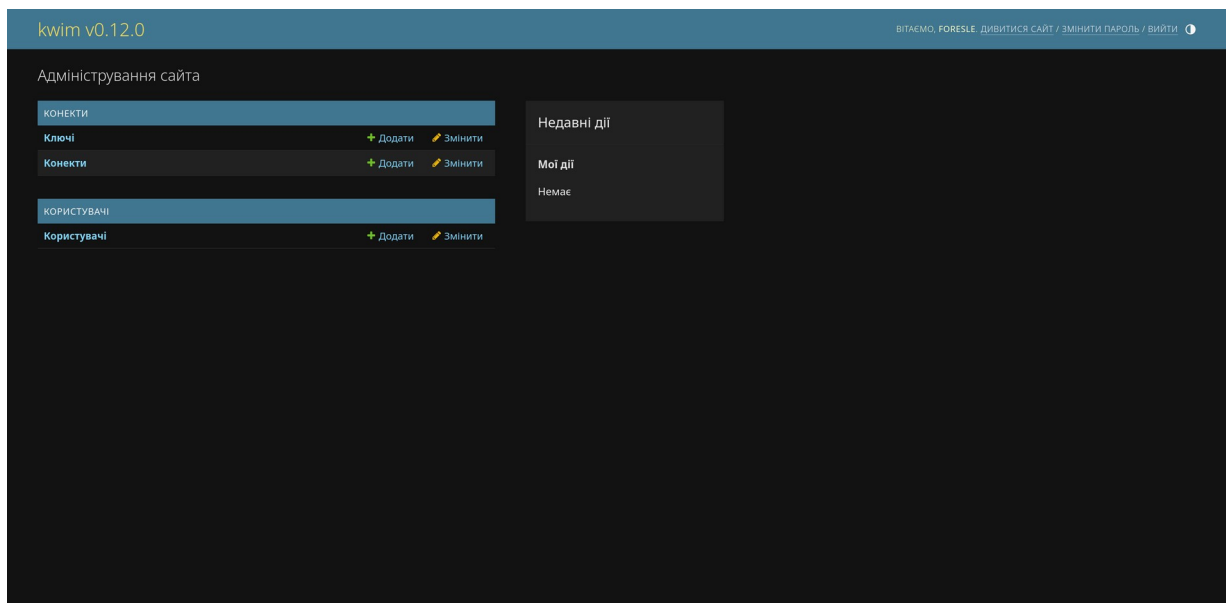


Рисунок 3.3: Головна сторінка адміністративної панелі

Після входу ми потрапимо на головну сторінку адміністративної панелі (рис. 3.3), що представлена у вигляді списку з таблиць: «Користувачі», «Конекти», «Ключі». Звідси можна вибрати потрібну нам таблицю, визначену раніше в коді та додати або змінити чи видалити запис (рис. 3.4).

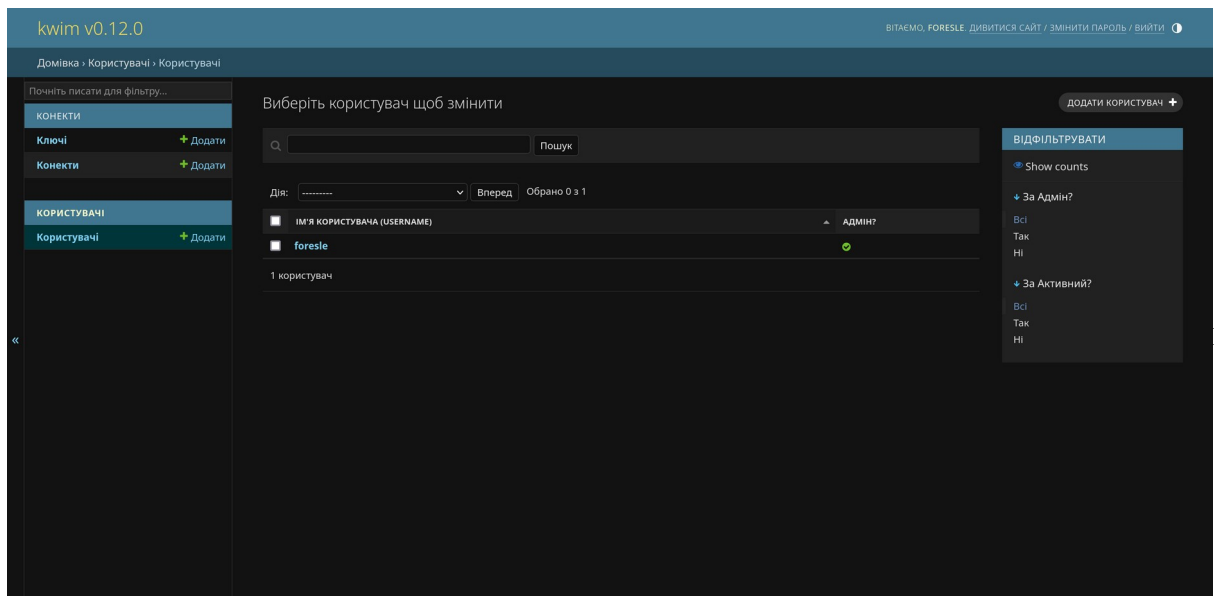


Рисунок 3.4: Сторінка переліку користувачів

Зверху кожної сторінки представлено елементи керування, такі як змінна пароля, вихід з системи, зміна кольорової схеми чи посилання на сайт, крім цього передбачено привітання для користувача. Зліва дублюється логотип з версією сервера.

Посередині знаходиться таблиця поділена на колонки визначеними параметрами, що містить пагіновані записи з таблиці. За замовчуванням в переліку є всі записи з бази даних. Щоб знайти конкретний запис, або кілька записів за конкретним полем, можна скористатися полем пошуку зверху основного елемента сторінки.

Справа передбачено зручні фільтри за заздалегідь визначеними параметрами, також записи можна сортувати натискаючи на заголовки колонок в таблиці.

3.3 OpenAPI документація

Для розробників також доступна сторінка з генерованою OpenAPI документацією, що знаходиться за адресою «api/schema/swagger-ui/» (рис. 3.5).

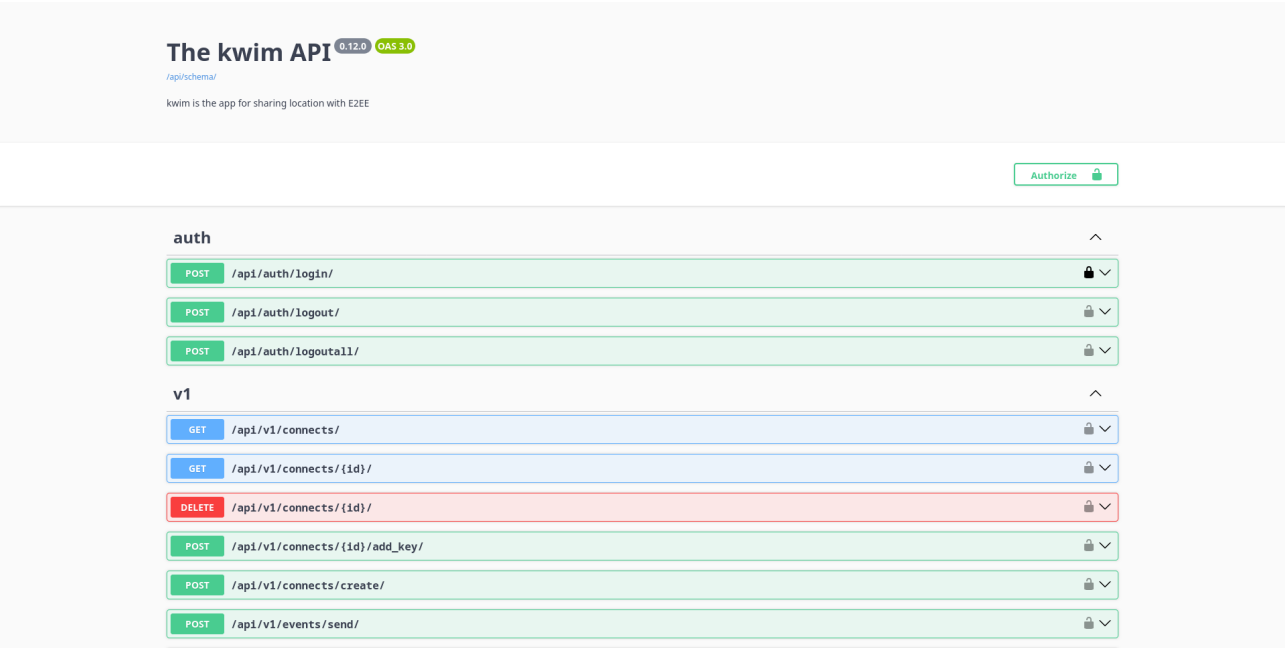


Рисунок 3.5: Сторінка OpenAPI документації

Тут розробник може побачити опис до схем та перелік всіх роутів API сервера, а також протестувати за потреби. У верхній частині можна знайти актуальну версію API та опис.

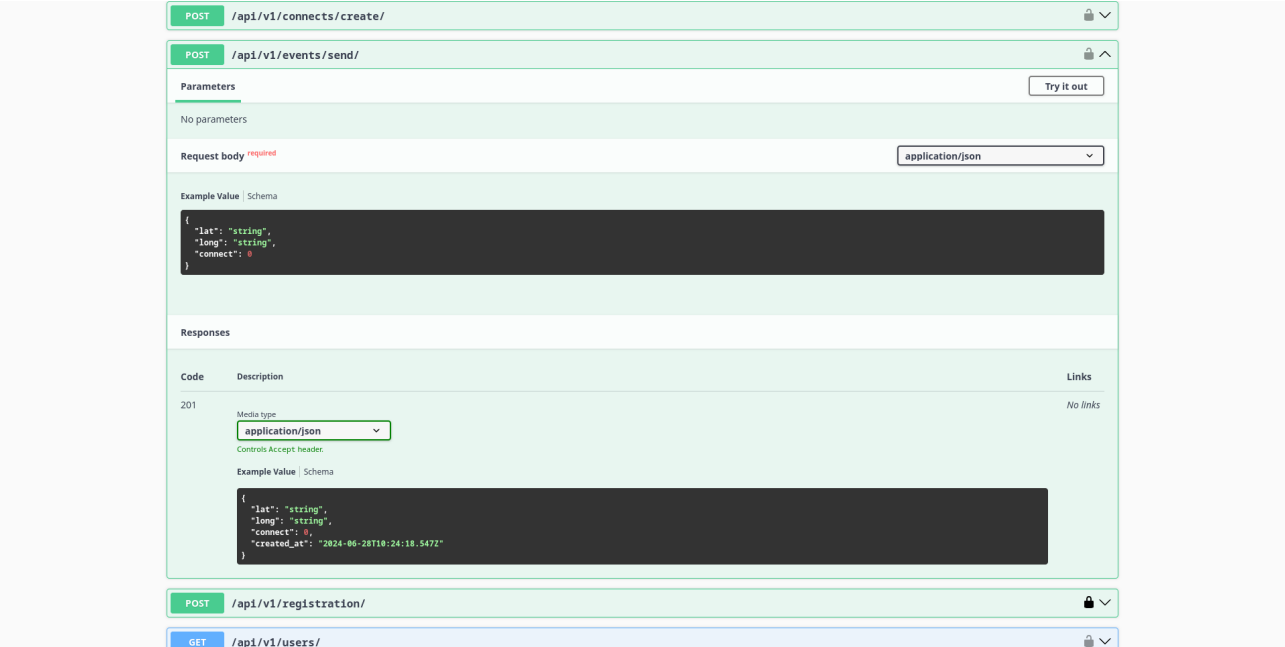


Рисунок 3.6: Документація окремого роута

Можна детально розглянути кожен роут, а також зробити запит зі зазначеними параметрами, щоб протестувати або вивчити роут на прикладі.

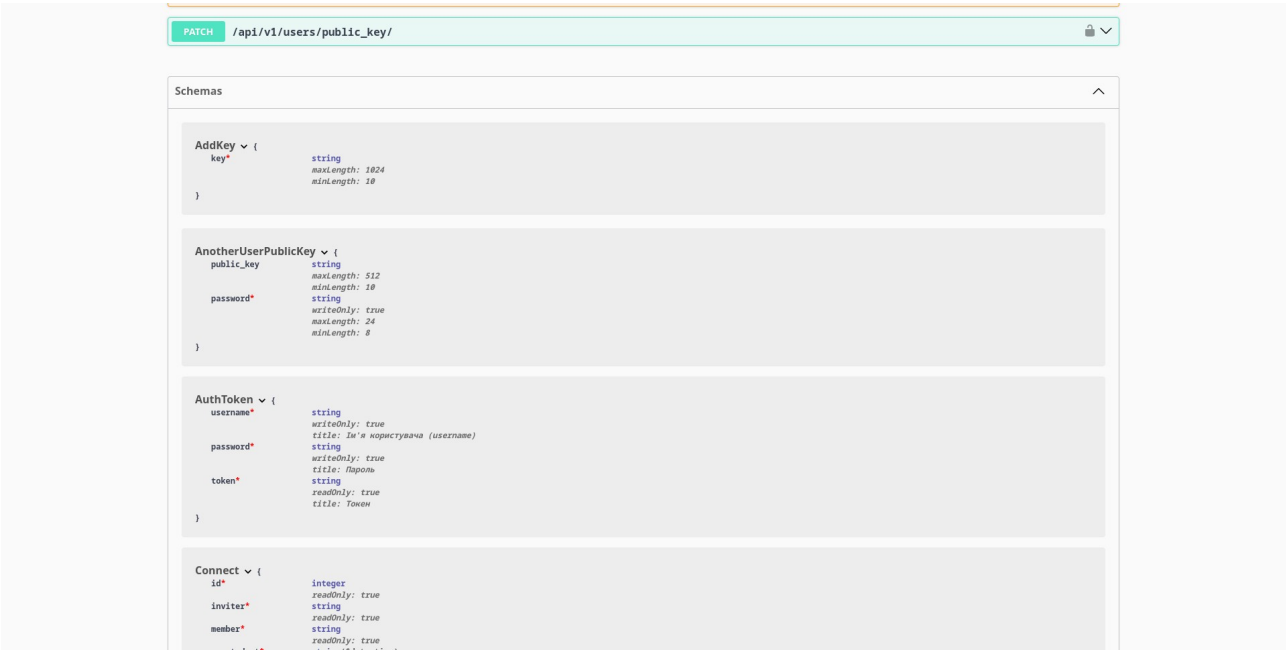


Рисунок 3.7: Сторінка схем

Схеми це відображення моделей, або таблиць у вигляді JSON об'єктів, де описано кожне поле та його обмеження, як от максимальна та мінімальна довжини вмісту і дозвіл залишити поле пустим.

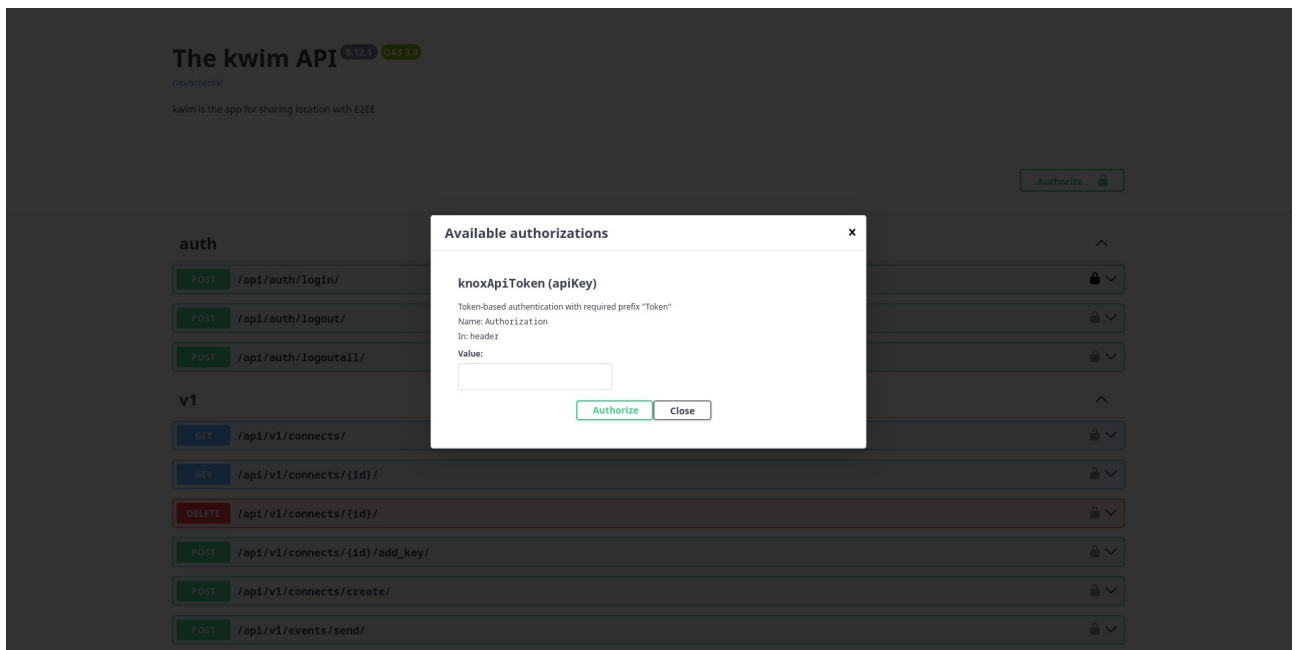


Рисунок 3.8: Сторінка введення API токена

Деякі роути вимагають додаткових прав для роботи з ними, вони позначенні піктограмою замочка у правій частині. Щоб отримати до них доступ прямо з вебінтерфейсу передбачено форму введення токена, сам токен можна отримати авторизувавшись по роуту в цьому ж інтерфейсі.

Висновки до третього розділу

В третьому розділі дипломної роботи розглянуто практичні аспекти встановлення та налаштування сервера, а також описано інтерфейс адміністратора й OpenAPI документацію.

Цей розділ демонструє практичну реалізацію ключових компонентів системи, підтверджуючи, що розроблений сервер є зручним для встановлення та налаштування, а також надає необхідні інструменти для ефективного управління і розробки. Реалізація цих компонентів забезпечує готовність системи до використання в реальних умовах, підтверджуючи її функціональність та відповідність заявленим вимогам.

ВИСНОВКИ

В ході розробки проєкту було проаналізовано аналоги та визначено завдання. Після чого було розроблено алгоритм шифрування та написана кодова база, яка після була розгорнута на фізичному сервері. В результаті була вирішена першочергова проблема, відчутність вільного і захищеного способу поширення своєї геолокації.

В результаті проведеного дослідження було розроблено відкритий програмний інтерфейс для обміну геолокацією з використанням наскрізного шифрування. На основі аналізу існуючих централізованих та децентралізованих сервісів та їхніх недоліків, особливо в аспектах безпеки та прозорості, було визначено основні критерії, яким повинен відповідати проєкт: відкритість, децентралізація та високий рівень захищеності.

Таким чином, розроблений проєкт надає ефективне рішення для обміну геолокацією, що задовольняє сучасні вимоги до безпеки та приватності. Він стане корисним інструментом як для індивідуальних користувачів, так і для компаній, що прагнуть інтегрувати захищені системи відстеження у свої процеси.

Для подальшого розвитку проєкту рекомендується:

1. Додати можливість перехресного підписання між пристроями, що дозволить швидко і захищено обмінюватися ключами та переглядати історії місцеперебування, що дозволить користувачам бачити маршрут руху власних пристроїв.
2. Розробити клієнтські додатки для різних операційних систем та пристроїв, що розширить можливості використання системи.
3. Написати автоматичні тести для виявлення та усунення потенційних вразливостей у коді.

					ІПЗ.КР.Б – 121 – 24 – ПЗ	Аркуш
						50
Зм.	Арк.	№ докум.	Підпис	Дата		

Ці рекомендації допоможуть покращити якість та функціональність проєкту, забезпечивши його відповідність найвищим стандартам безпеки та зручності використання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Офіційний сайт протоколу Matrix [Електронний ресурс] – Режим доступу до ресурсу: <https://matrix.org/>.
2. Python-RSA's documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://stuvel.eu/python-rsa-doc/>.
3. Django ORM [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.djangoproject.com/en/5.0/topics/db/queries/>.
4. drf-spectacular [Електронний ресурс] – Режим доступу до ресурсу: <https://drf-spectacular.readthedocs.io/en/latest/readme.html>.
5. OpenAPI Specification v3.1.0 [Електронний ресурс] – Режим доступу до ресурсу: <https://spec.openapis.org/oas/latest.html>.
6. Семантичне Версіонування 2.0.0 [Електронний ресурс] – Режим доступу до ресурсу: <https://semver.org/lang/uk/>.
7. Литвиненко В. Алгоритм шифрування RSA, види атак на нього. Реалізація мовою Python [Електронний ресурс] / Владислав Литвиненко – Режим доступу до ресурсу: <https://dou.ua/forums/topic/43026/>.
8. Docker Docs [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.docker.com/>.
9. Docker Compose overview [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.docker.com/compose/>.
10. daphne [Електронний ресурс] – Режим доступу до ресурсу: <https://github.com/django/daphne>.
11. PostgreSQL 17 [Електронний ресурс] – Режим доступу до ресурсу: <https://www.postgresql.org/docs/17/index.html>.

12.Arch Linux [Електронний ресурс] – Режим доступу до ресурсу:
<https://archlinux.org/>.

					ІПЗ.КР.Б – 121 – 24 – ПЗ	Аркуш
						53
Зм.	Арк.	№ докум.	Підпис	Дата		

ДОДАТКИ

					ІПЗ.КР.Б – 121 – 24 – ПЗ	Аркуш
						54
Зм.	Арк.	№ докум.	Підпис	Дата		

Код модуля конекту

```

from django.utils.translation import gettext_lazy as _
from rest_framework import serializers
from django.db.models import Q

from .models import Connect, Key
from events.models import Event
from users.models import User


class CreateConnectSerializer(serializers.Serializer):
    inviter_username: serializers.CharField =
serializers.CharField(
        min_length=3, max_length=30, allow_blank=False,
allow_null=False, required=True,
        write_only=True
    )
    inviter_secret_code: serializers.CharField =
serializers.CharField(
        min_length=7, max_length=7, allow_blank=False,
allow_null=False, required=True,
        write_only=True
    )

    def validate(self, attrs: dict) -> dict:
        """
        1. Check if inviter exist.
        2. Check if user don't create connect with himself.
        3. Check if connect already exist.
        4. Check if secret code is valid.
        """

        try:

```

```

        user: User =
User.objects.get(username=attrs["inviter_username"])
        except User.DoesNotExist: # noqa
            raise serializers.ValidationError(detail=_("User
with this username does not exist.))

        if user == self.context["request"].user:
            raise serializers.ValidationError(detail=_("You
can't create the connect with yourself.))

        try:
            connect: Connect =
Connect.objects.get(inviter=user,
member=self.context["request"].user) # noqa
            raise
serializers.ValidationError(detail=_("Connect already exist.))
        except Connect.DoesNotExist: # noqa
            pass

        try:
            connect: Connect =
Connect.objects.get(inviter=self.context["request"].user,
member=user) # noqa
            raise
serializers.ValidationError(detail=_("Connect already exist.))
        except Connect.DoesNotExist: # noqa
            pass

        if user.connect_secret_key !=
attrs["inviter_secret_code"]:
            user.connect_failed_attempts += 1
            user.save()
            raise serializers.ValidationError(detail=_("Secret
code is invalid.))

        attrs["inviter"] = user
        attrs["member"] = self.context["request"].user

```

```

        return attrs

    def create(self, validated_data) -> Connect:
        connect: Connect = Connect(
            inviter=validated_data["inviter"],
            member=validated_data["member"]
        )
        connect.save()
        return connect


class KeySerializer(serializers.ModelSerializer):
    recipient: serializers.SerializerMethodField | str =
serializers.SerializerMethodField(read_only=True)

    class Meta:
        model = Key
        fields: tuple = ("id", "recipient", "key",
"created_at", "read")

    def get_recipient(self, instance: Connect) -> str: # noqa
        return f"{instance.recipient.username}" # noqa


class EventSerializer(serializers.ModelSerializer):
    class Meta:
        model = Event
        fields: tuple = ("lat", "long", "sender", "connect",
"created_at")
        read_only_fields: tuple = fields


class ConnectSerializer(serializers.ModelSerializer):
    inviter: serializers.SerializerMethodField =
serializers.SerializerMethodField()

```

```

        member: serializers.SerializerMethodField =
serializers.SerializerMethodField()

        keys: KeySerializer = KeySerializer(many=True,
read_only=True, source="keys_of_connect")

        last_messages: EventSerializer =
EventSerializer(many=True, read_only=True,
source="events_of_connect")

class Meta:
    model = Connect
    fields: tuple = ("id", "inviter", "member",
"created_at", "keys", "last_messages")

    def get_inviter(self, instance: Connect) -> str: # noqa
        return f"{instance.inviter.username}" # noqa

    def get_member(self, instance: Connect) -> str: # noqa
        return f"{instance.member.username}" # noqa

class AddKeySerializer(serializers.ModelSerializer):
    key: serializers.CharField = serializers.CharField(
        min_length=10, max_length=1024, allow_null=False,
required=True
    )

class Meta:
    model = Key
    fields: tuple = ("key",)

    def validate(self, attrs: dict) -> dict:
        """
        1. Check if connect exist and user is a member of
connect.

        2. Get a receiver.

        3. Put the recipient and connect to attr.

```

```

        """

        try:
            connect: Connect = Connect.objects.get( # noqa
                id=self.context["request"].parser_context["kwargs"]["pk"]) & (
                Q(inviter=self.context["request"].user) |
                Q(member=self.context["request"].user) # noqa
            )
        except Connect.DoesNotExist: # noqa
            raise serializers.ValidationError(detail=_("The
            connect does not exist or you don't have permission."))

        if self.context["request"].user == connect.inviter:
            recipient: User = connect.member
        else:
            recipient: User = connect.inviter

        attrs["connect"] = connect
        attrs["recipient"] = recipient

        return attrs

from rest_framework import permissions
from rest_framework import generics
from django.db.models import Q

from .serializers import CreateConnectSerializer,
ConnectSerializer, AddKeySerializer
from .models import Connect

class CreateConnectView(generics.CreateAPIView):
    permission_classes: tuple = (permissions.IsAuthenticated,)

```

```

serializer_class = CreateConnectSerializer
queryset = Connect.objects.all() # noqa

class
RetrieveDestroyConnectView(generics.RetrieveDestroyAPIView):
    permission_classes: tuple = (permissions.IsAuthenticated,)
    serializer_class = ConnectSerializer

    def get_queryset(self):
        return
Connect.objects.filter(Q(inviter=self.request.user) |
Q(member=self.request.user)) # noqa

class ListConnectView(generics.ListAPIView):
    permission_classes: tuple = (permissions.IsAuthenticated,)
    serializer_class = ConnectSerializer

    def get_queryset(self):
        return
Connect.objects.filter(Q(inviter=self.request.user) |
Q(member=self.request.user)) # noqa

class CreateKeyView(generics.CreateAPIView):
    permission_classes: tuple = (permissions.IsAuthenticated,)
    serializer_class = AddKeySerializer

    def get_queryset(self):
        return Key.objects.all() # noqa

from django.urls import path

from .views import CreateConnectView, ListConnectView,
RetrieveDestroyConnectView, CreateKeyView

```

```

app_name: str = "connects"

urlpatterns: list = [
    path("create/", CreateConnectView.as_view(), name="create-
connect"),
    path("<int:pk>/add_key/", CreateKeyView.as_view(),
name="create-key"),
    path("<int:pk>/", RetrieveDestroyConnectView.as_view(),
name="retrieve-destroy-connect"),
    path("", ListConnectView.as_view(), name="list-connect"),
]

```