Форма №Н-6.01

Міністерство освіти і науки України
Рівненський державний гуманітарний університет
Кафедра інформаційних технологій та моделювання

Кваліфікаційна робота
за освітнім ступенем «магістр»
на тему: «Моделювання та реалізація
SaaS-платформи на базі Django Tenants і React»

Виконав:
магістр 2 курсу групи М-КН-21
спеціальності 122 «Комп'ютерні науки»
Климець Віктор Вікторович
Керівник:
к.т.н., доцент Шинкарчук Н.В.

Рівне – 2025

АНОТАЦІЯ ДО КВАЛІФІКАЦІЙНОЇ РОБОТИ

Климець В.В. Моделювання та реалізація SaaS-платформи на базі Django Tenants і React. Кваліфікаційна робота на здобуття ступеня «Магістр» за спеціальністю 122 «Комп'ютерні науки» – Рівненський державний гуманітарний університет. Рівне, 2025. 65 с.

Останнім часом, дедалі популярнішими стають багатокористувацькі SaaS-платформи (Software as a Service), які дозволяють надавати програмне забезпечення як послугу через інтернет. Це ефективна альтернатива традиційним локальним системам, оскільки забезпечує масштабованість, гнучкість та економію ресурсів для бізнесу. SaaS-платформи дозволяють кільком користувачам або організаціям спільно використовувати одну інфраструктуру, зберігаючи при цьому ізоляцію даних і налаштувань, що підвищує ефективність та знижує витрати на розробку та підтримку.

Організація SaaS-систем передбачає використання спеціалізованої архітектури, такої як multi-tenant, де один екземпляр додатка обслуговує багатьох клієнтів. Для реалізації backend-частини часто застосовується фреймворк Django з розширенням Django Tenants, яке забезпечує ізоляцію баз даних для кожного тенанта. Користувачі отримують доступ до платформи через веб-браузер з можливістю персоналізації та інтеграції з хмарними сервісами. Для забезпечення безпеки використовуються механізми аутентифікації, шифрування та контролю доступу.

В кваліфікаційній роботі проведено огляд теоретичних основ багатокористувацьких SaaS-систем, розглянуто методологію їхньої розробки. Детально описано архітектуру та проєктування SaaS-платформи, а також практичну реалізацію: створення та впровадження з використанням Django Tenants і React. Подано рекомендації щодо вибору оптимальних інструментів та оцінки ефективності розробленої платформи.

ANNOTATION TO THE QUALIFICATION THESIS

Klymets V.V. Modeling and Implementation of a SaaS Platform Based on Django Tenants and React. Qualification thesis for obtaining the Master's degree in the specialty 122 «Computer Science» – Rivne State University of the Humanities. Rivne, 2025. 65 p.

Recently, multi-user SaaS platforms (Software as a Service) have been gaining increasing popularity, as they enable software to be provided as a service via the Internet. This is an effective alternative to traditional on-premises systems, as it ensures scalability, flexibility, and resource efficiency for businesses. SaaS platforms allow multiple users or organizations to share a single infrastructure while maintaining isolation of data and configurations, which increases efficiency and reduces development and maintenance costs.

The organization of SaaS systems involves the use of specialized architectures, such as multi-tenant architecture, where a single application instance serves multiple clients. To implement the backend part, the Django framework with the Django Tenants extension is often used, which ensures database isolation for each tenant. Users access the platform through a web browser with options for personalization and integration with cloud services. Security is ensured through authentication mechanisms, encryption, and access control.

In the qualification thesis, a review of the theoretical foundations of multi-user SaaS systems is conducted, and the methodology of their development is examined. The architecture and design of the SaaS platform are described in detail, as well as its practical implementation: development and deployment using Django Tenants and React. Recommendations are provided regarding the selection of optimal tools and the evaluation of the efficiency of the developed platform.

ЗМІСТ

ВСТУП

Актуальність роботи. Моделювання та реалізація SaaS-платформ є однією з найбільш актуальних галузей сучасних інформаційних технологій. Сьогодні підприємства та організації різного масштабу все частіше обирають SaaS-рішення для управління бізнес-процесами, комунікації та зберігання даних, оскільки вони забезпечують гнучкість, доступність з будь-якого пристрою та економію ресурсів. Багатокористувацькі архітектури дозволяють обслуговувати велику кількість клієнтів на єдиній платформі, оптимізуючи витрати на інфраструктуру і спрощують адміністрування.

Водночас розробка та підтримка таких платформ супроводжується значними викликами, серед яких забезпечення безпеки даних, масштабованість системи та управління багаторівневим доступом користувачів. Сучасні технології, такі як Django Tenants для багатокористувацької архітектури та React для фронтенд-інтерфейсу, дозволяють створювати високопродуктивні, надійні та гнучкі SaaS-рішення, що відповідають сучасним потребам ринку.

Мета роботи: дослідження, моделювання та практична реалізація багатокористувацької SaaS-платформи на основі Django Tenants і React, а також формування обґрунтованих рекомендацій щодо побудови ефективної багатокористувацької архітектури та організації даних.

Об’єктом дослідження є багатокористувацька SaaS-платформа як програмно-технічна система, що забезпечує роботу великої кількості користувачів із централізованим доступом до ресурсів та даних.

Інструментом дослідження є фреймворк Django, бібліотеки Django Tenants, React.

Предметом дослідження є технології реалізації багатокористувацької архітектури, методи зберігання даних у multi-tenant середовищі та інтеграція фронтенд-інтерфейсу з бекендом за допомогою React і Django Tenants.

Завдання дослідження. Для досягнення поставлених цілей потрібно:

1. Вивчити теоретичні основи багатокористувацьких SaaS-систем.

2. Проаналізувати особливості multi-tenant архітектури та її моделі зберігання даних.
3. Розглянути методології проєктування багатокористувацьких платформ.
4. Описати архітектуру та компоненти SaaS-платформи на основі Django Tenants і React.
5. Реалізувати прототип багатокористувацької платформи та перевірити його функціональність.
6. Провести оцінку масштабованості, безпеки та гнучкості реалізованої платформи.
7. Надати практичні рекомендації щодо впровадження та розвитку багатокористувацьких SaaS-рішень.

Апробація кваліфікаційної роботи. Результати виконання кваліфікаційної роботи, окремі її аспекти та одержані узагальнення і висновки були оприлюднені на звітній науковій конференції викладачів, співробітників і здобувачів вищої освіти Рівненського державного гуманітарного університету за 2024 рік (м. Рівне, 2025).

Структура роботи. Робота складається зі вступу, чотирьох розділів, висновків та списку використаних джерел. У першому розділі розглядаються теоретичні основи SaaS-систем та особливості multi-tenant архітектури. Другий розділ присвячено методології проєктування та архітектурним підходам до розробки SaaS-платформ. Третій розділ містить процес проєктування, компонентів і моделей даних реалізованої платформи. У четвертому розділі подано практичну реалізацію SaaS-платформи, включаючи приклади впровадження та інтеграції.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ SAAS-СИСТЕМ

1.1. Поняття та еволюція SaaS

Програмне забезпечення як послуга (SaaS) – це модель надання програмного забезпечення в хмарі. Постачальник хмари розробляє і підтримує хмарне прикладне програмне забезпечення, надає для нього автоматичні оновлення та доступ клієнтам через інтернет. Оплата здійснюється тільки за те, чим користуються. Постачальник загальнодоступних хмар керує всім обладнанням і традиційним програмним забезпеченням, зокрема «проміжним» і прикладним. Отже, замовники SaaS можуть суттєво скоротити витрати, розгортати, масштабувати й оновлювати бізнес-рішення швидше, ніж за підтримки локальних систем і програмного забезпечення, а також точніше прогнозувати сукупну вартість володіння [1].

У 1960-х роках було винайдено багатозадачність, що дозволило мейнфрейм-комп'ютерам обслуговувати одночасно декількох користувачів. Протягом наступного десятиліття такий підхід став основною бізнес-моделлю для обчислювальної техніки, а кластерні обчислення дозволили декільком комп'ютерам працювати разом. Хмарні обчислення з'явилися наприкінці 1990-х років, коли такі компанії, як Amazon (1994), Salesforce (1999) і Concur (1993), почали пропонувати інтернет-додатки на основі оплати за використання. Усі вони зосередилися на одному продукті, щоб захопити велику частку ринку [2].

Починаючи з Gmail у 2004 році, поштові служби стали одними з перших SaaS-продуктів, що масово продавалися споживачам. Ринок SaaS швидко зростав протягом початку XXI століття. Спочатку SaaS розглядався як технологічна інновація, але з часом став сприйматися більше як бізнес-модель. До 2023 р. SaaS став основним методом, за допомогою якого компанії постачають додатки [2].

Основними перевагами SaaS є:

1. Відсутність витрат на придбання та підтримку інфраструктури.

2. Масштабованість та гнучкість використання.
3. Централізовані оновлення програмного забезпечення.
4. Можливість швидкого впровадження в організації будь-якого масштабу.

Сьогодні цей підхід є домінуючим у сфері корпоративного та масового використання програмного забезпечення. SaaS застосовується у різних галузях:

1. CRM-системи – Salesforce, HubSpot.
2. Системи управління проєктами – Jira, Trello, Asana.
3. Електронна пошта та офісні пакети – Gmail, Outlook 365, Google Workspace, Microsoft 365.
4. Фінансові сервіси – QuickBooks, Xero.
5. Комунікаційні платформи – Slack, Zoom, Microsoft Teams.

Використання SaaS-платформи дає змогу компаніям відмовитися від великих початкових інвестицій в IT-інфраструктуру, адже застосунок, система або digital продукт постачаються за передплатою. Це означає, що, замість купівлі дорогого програмного забезпечення для сайту, компанії сплачують за послугу, що містить регулярне оновлення app, технічну підтримку і постачання security patch-ів. Такий підхід знижує витрати на розробку та обслуговування IT рішень, що дає змогу власникам компаній сконцентруватися на розвитку інших задач проєкту [3].

1.2. Архітектурні моделі SaaS

Архітектуру SaaS-платформ (Software as a Service) можна розглядати як технологію, що дозволяє створювати багато користувацьке програмне середовище в хмарі, яке можна використовувати для надання послуг різним клієнтам з мінімальними витратами на інфраструктуру. Таким чином, стає реальною можливість забезпечити ізоляцію даних, централізоване оновлення та масштабування ресурсів, дозволяючи користувачам отримувати доступ до персоналізованих сервісів з будь-якого пристрою через інтернет (рис. 1.1).

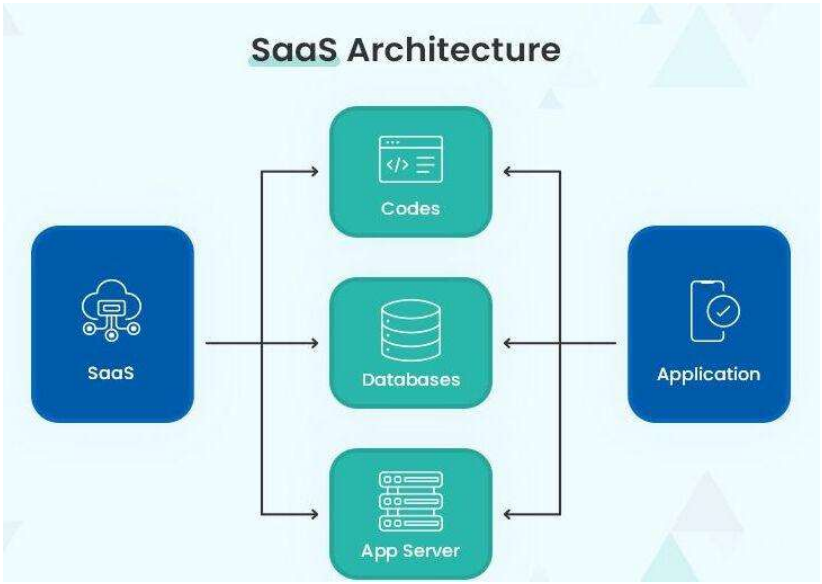

Рисунок 1.1. Загальна схема SaaS архітектури

На даний час існує кілька основних архітектурних моделей SaaS-платформ:

1. Однокористувацька (single-tenant) архітектура.
2. Багатокористувацька (multi-tenant) архітектура.

Single-tenant архітектура (рис. 1.2) – це архітектура в якій один екземпляр програмного застосунку та допоміжної інфраструктури обслуговує одного клієнта [4].

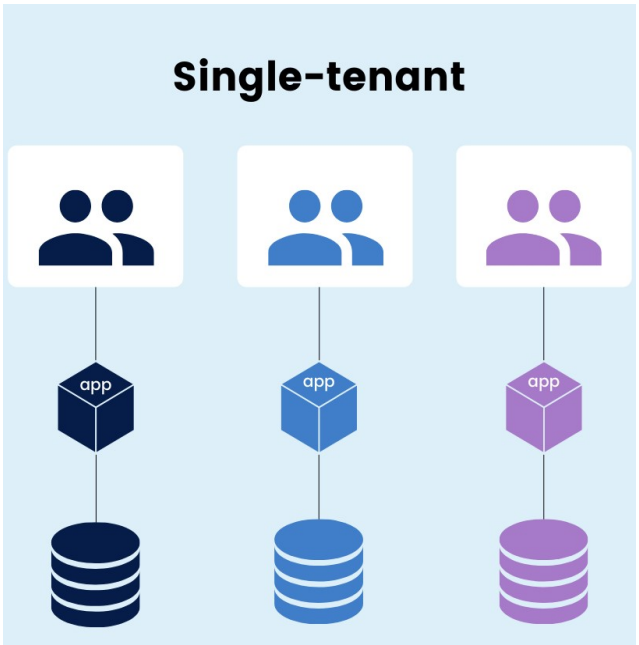

Рисунок 1.2. Single-tenant архітектура

Її можна розглядати як більш ізольовану форму розгортання, орієнтовану на індивідуальні потреби клієнта. Замість спільного використання ресурсів, для кожного тенанта створюється окрема інстанція додатка та бази даних. Користувачі взаємодіють з власною копією системи через мережеве з'єднання, таке як локальна мережа, VPN або інтернет. Крім того, основний сервер у цьому сценарії стає виділеною інфраструктурою, здатною розміщувати повну копію додатка для одного клієнта з повною кастомізацією.

Деякі спільні характеристики моделей з одним клієнтом полягають у тому, що вони, як правило, забезпечують високий рівень залучення користувачів та контролю з їхнього боку, а також надійність, безпеку та можливості резервного копіювання. Оскільки клієнти знаходяться в окремому середовищі один від одного, вони не пов'язані між собою так само, як клієнти, що використовують спільну інфраструктуру [4].

В single-tenant сервер розділяє ресурси на окремі інстанси, що містять повний набір додатків і даних, до яких клієнт отримує ексклюзивний доступ через свої пристрої. Клієнти можуть отримувати доступ до своєї інстанції з будь-яких пристроїв і в будь-якій точці, при цьому всі процеси обробки даних виконуються в ізольованому середовищі.

Клієнти підключаються до своєї інстанції за допомогою API ресурсу або аутентифікаційного сервісу, який є програмним посередником між користувачем і сервером. За допомогою single-tenant клієнти можуть запускати повну персоналізовану версію додатка на виділеній інфраструктурі, розміщеній на фізичному або хмарному сервері. Кожній інстанції буде виділено ресурси для підвищення ефективності та захисту з'єднання.

Переваги single-tenant архітектури – одноклієнтські екземпляри легко мігруються на інші хмарні сервіси. мають гнучку безпеку даних навіть якщо станеться витік даних одного клієнта з тим самим постачальником послуг, інший клієнт буде захищений від витоку, оскільки дані зберігаються в окремому екземплярі.

Оскільки всі дані клієнта є окремими, можливий великий ступінь налаштування для програмних та апаратних екземплярів. Екземпляри з одним клієнтом вважаються надійними, оскільки продуктивність базується лише на одному екземплярі, а не на багатьох від різних клієнтів [4].

Інша модель, багатокористувацька (multi-tenant) архітектура (рис. 1.3), дозволяє кільком клієнтам підключатися та використовувати спільний, але потужний додаток через мережу та одночасно працювати в ньому. Кожному тенанту надається логічно ізольований простір з персональними даними та налаштуваннями. За допомогою багатокористувацької конфігурації multi-tenant можна реалізувати за допомогою одного сервера з динамічним розподілом ресурсів.

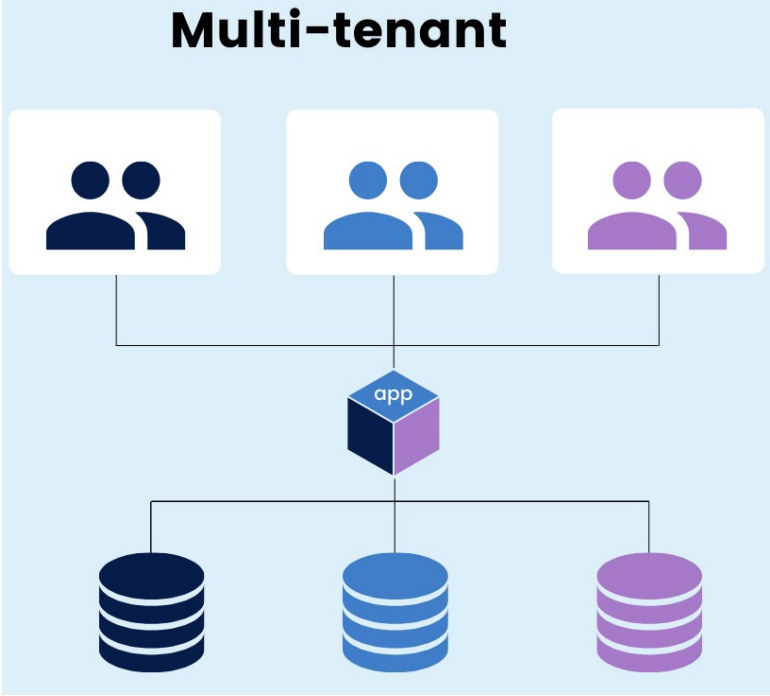

Рисунок 1.3. Схема multi-tenant архітектури

У multi-tenant архітектурі всі клієнти використовують один екземпляр додатка та, зазвичай, одну базу даних, де дані розділені логічно за допомогою схем або рядкових ідентифікаторів. Переваги включають економію ресурсів, простоту масштабування та централізоване адміністрування, що робить її ідеальною для масового ринку.

У багатокористувацький архітектурі кожен клієнт використовує одну і ту ж базу даних та програму де single-tenant всього має один екземпляр програмного забезпечення який обслуговує декількох клієнтів.

Багатокористувацька архітектура зазвичай ідеально підходить для організацій, які хочуть спростити запуск і зменшити вимоги до апаратного забезпечення. Ця архітектура стала стандартом для корпоративних середовищ SaaS.

У порівнянні з однокористувацькою архітектурою (рис. 1.4.), багатокористувацька архітектура має такі переваги:

1. Дешевша.
2. Ефективно використовує ресурси.
3. Знижує витрати на обслуговування.
4. Має більшу обчислювальну потужність.

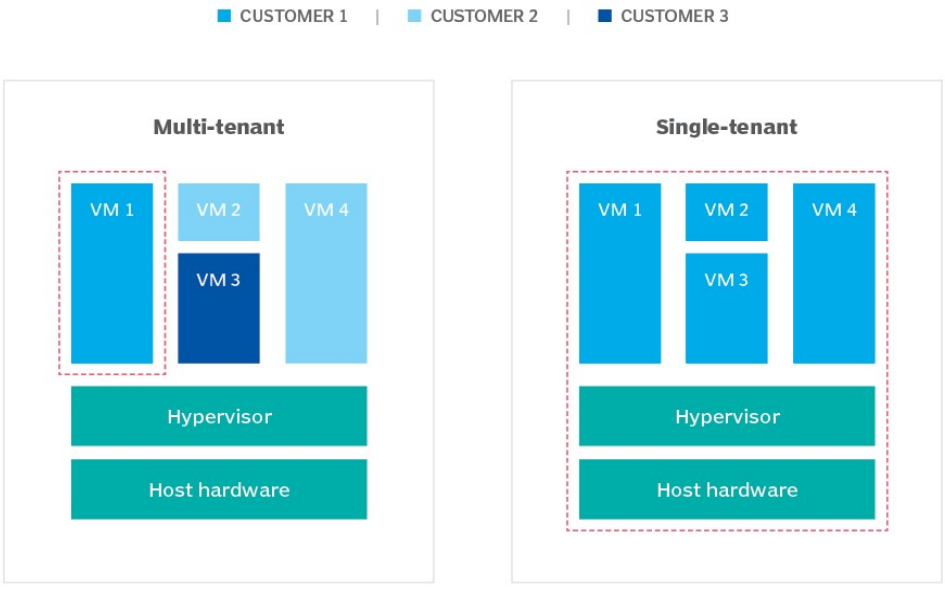

Рисунок 1.4. Порівняння multi-tenant та single-tenant

Незважаючи на те, що багатокористувацька архітектура має багато очевидних переваг над однокористувацькою, можливість внесення значних змін у програмне забезпечення обмежена, оскільки воно використовується іншими користувачами. Тому потрібно витрачати час на відповідне тестування щоб продукт відповідав вимогам клієнта [4].

1.3. Особливості багатокористувацьких SaaS-систем

Багатокористувацькі (multi-tenant) системи є основою сучасних SaaS-рішень, оскільки дозволяють обслуговувати велику кількість клієнтів на єдиній інфраструктурі, що забезпечує економію ресурсів і спрощує адміністрування. Використовуючи multi-tenant, компанії можуть надавати свої послуги багатьом користувачам без необхідності створювати окремі програми для кожної з них, що робить її розумним вибором як для бізнесу, так і для їхніх клієнтів.

Основні характеристики multi-tenant архітектур:

1. Ізоляція даних. Кожен клієнт має власний простір даних, який не перетинається з іншими користувачами системи. Ізоляція може бути реалізована різними способами: на рівні бази даних (окрема БД для кожного тенанта), на рівні схеми (окрема схема в спільній базі) або на рівні таблиць (додавання ідентифікатора тенанта до кожного запису). Вибір підходу залежить від вимог до безпеки, масштабованості та витрат на підтримку.
2. Масштабованість. Multi-tenant системи повинні забезпечувати як горизонтальне, так і вертикальне масштабування для стабільної роботи під навантаженням. Горизонтальне масштабування досягається за рахунок додавання нових вузлів або контейнерів, тоді як вертикальне – шляхом збільшення обчислювальних ресурсів існуючих сервісів. Ефективна система балансування навантаження дозволяє рівномірно розподіляти запити між різними компонентами інфраструктури.
3. Безпека. Multi-Tenant архітектура повинна забезпечувати високий рівень безпеки та ізоляції між арендаторами. Це включає в себе контроль доступу до даних, аудит доступу та ідентифікацію користувачів.
4. Багаторівневість доступу. У межах одного тенанта система підтримує різні ролі користувачів – наприклад, адміністратора, менеджера, аналітика або звичайного користувача. Це забезпечує розподіл

повноважень і підвищує безпеку, дозволяючи виконувати лише ті дії, які передбачені роллю.

5. Централізоване адміністрування. Однією з ключових переваг multi-tenant архітектури є можливість централізованого оновлення програмного забезпечення, виправлення помилок і розгортання нових функцій. Це значно спрощує технічну підтримку, адже всі клієнти отримують оновлення одночасно, без необхідності втручання у кожен інстанс окремо.
6. Гнучкість та конфігурованість. Сучасні SaaS-платформи дозволяють клієнтам налаштовувати інтерфейс, бізнес-логіку та інтеграції без порушення спільної архітектури. Це досягається через використання конфігураційних файлів, модульних плагінів або систем керування параметрами на рівні тенанта.
7. Моніторинг та аналіз. Важливо мати засоби моніторингу та аналізу для відстеження використання ресурсів кожним арендатором, виявлення аномалій та оптимізації продуктивності.

Враховуючи ці аспекти, розробники можуть створити ефективні та масштабовані Multi-Tenant системи, які задовольняють потреби багатьох користувачів з різними вимогами та відмінностями.

Multi-tenant архітектура – це підхід, за якого одна програма (кодова база) використовується для обслуговування декількох клієнтів, причому дані кожного з них ізольовані та захищені. У межах такої системи кожен тенант може мати власні налаштування, політики доступу, інтеграції та навіть власний користувацький інтерфейс [5].

Моделі реалізації multi-tenant архітектур:

1. Модель спільної бази даних (Shared Database Model) (рис. 1.5). У цьому підході всі тенанти використовують одну спільну базу даних, де записи розрізняються за допомогою спеціального ідентифікатора тенанта. Це найбільш економічна модель, оскільки потребує лише одного

екземпляра бази даних і дозволяє застосовувати оновлення одночасно для всіх користувачів.

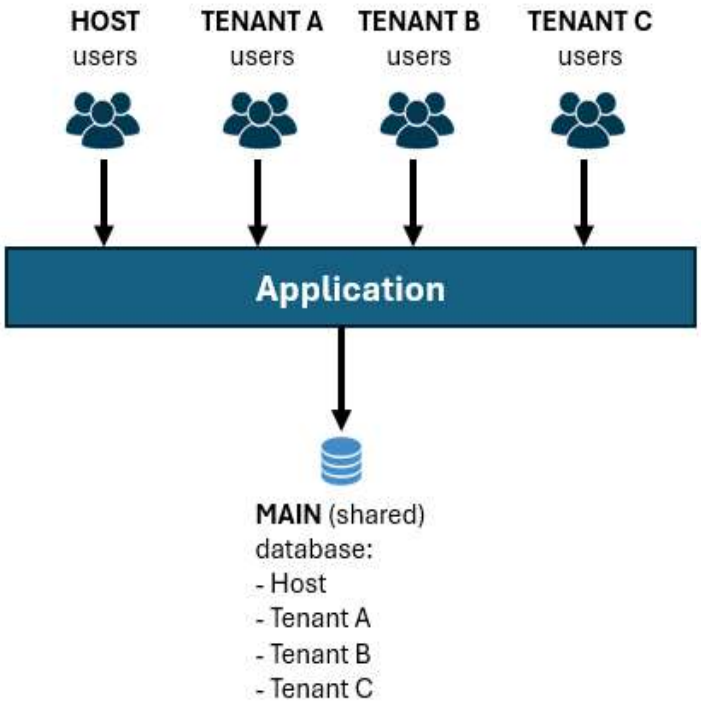

Рисунок 1.5. Shared Database Model

2. Модель одна база даних на тенант (Database per Tenant Model) (рис. 1.6). Кожен клієнт має власну окрему базу даних, що гарантує повну ізоляцію даних та стабільну продуктивність незалежно від активності інших користувачів.

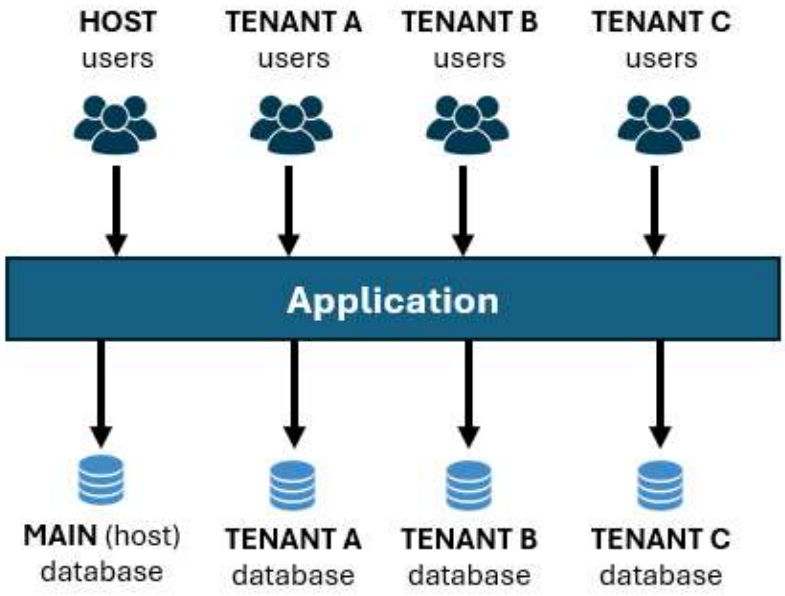

Рисунок 1.6. Database per Tenant Model

3. Гібридна модель (Hybrid Model). Гібридний підхід поєднує риси двох попередніх моделей. Наприклад, клієнти з малим обсягом даних можуть користуватися спільною базою, тоді як стратегічно важливі або високонавантажені – отримують окрему базу даних.

Побудова багатокористувацької архітектури – це процес, який потребує балансу між простотою, безпекою, продуктивністю та економічною ефективністю. Вибір конкретної моделі залежить від масштабів системи, вимог до ізоляції даних, очікуваного навантаження та доступних ресурсів.

Shared Database забезпечує найменші витрати, але потребує високого рівня контролю доступу. Database per Tenant гарантує безпеку, але ускладнює управління. Hybrid Model пропонує компроміс, дозволяючи масштабуватися з урахуванням специфічних потреб клієнтів.

Таким чином, multi-tenant архітектура є не лише технічним рішенням, а й логічним підходом до створення масштабованих, безпечних і гнучких SaaS-рішень, здатних ефективно задовольнити потреби великої кількості користувачів при оптимальних витратах на обслуговування.

1.4. Порівняння моделей зберігання даних у multi-tenant архітектурі

У multi-tenant (багатокористувацький) архітектурі головне завдання – ізолювати дані різних клієнтів так, щоб вони не перетинались, але при цьому максимально ефективно використовувати спільні ресурси.

Існує три основні моделі зберігання даних, які відрізняються рівнем ізоляції, продуктивністю та складністю підтримки.

Shared Database Model – модель зберігання коли кожен тенант використовує одну базу даних і одні й ті ж таблиці. Це досягається шляхом додавання стовпця tenant_id до кожної таблиці, яка вимагає розділення орендарів (рис. 1.7).

```sql
CREATE TABLE customers (
  id INT PRIMARY KEY,
  tenant_id INT NOT NULL,
  name VARCHAR(255),
  email VARCHAR(255),
  ...
  INDEX(tenant_id)
);
```

Рисунок 1.7. Модель зберігання Shared Database Model

Є найпростішим і найефективнішим варіантом з точки зору реалізації та обслуговування. Вона передбачає одну базу даних і спільну схему для всіх орендарів, що значно спрощує:

1. Резервне копіювання, моніторинг і оновлення системи.
2. Управління схемою – усі зміни автоматично застосовуються до всіх клієнтів одночасно.

Але при цьому має ризик витоку даних між орендарями, якщо запити не фільтруються належним чином, також на продуктивність можуть впливати «шумні сусіди» тому що універсальний підхід обмежує можливості налаштування для кожного орендаря в залежності від його потреб

Database per Tenant Model – модель зберігання коли кожен тенант має власну базу даних (рис. 1.8).

```sql
-- Create Tenant 1's database
CREATE DATABASE tenant1;

-- Create Tenant 2's database
CREATE DATABASE tenant2;
```

Рисунок 1.8. Модель зберігання Database per Tenant Model

Архітектура з окремою базою даних для кожного клієнта забезпечує найвищий рівень ізоляції та гнучкості. Вона дозволяє індивідуально налаштовувати середовище під потреби кожного клієнта, усуває ризики впливу «гучних сусідів», полегшує дотримання регіональних вимог до зберігання даних і спрощує масштабування лише тих орендарів, які цього потребують.

Однак ці переваги досягаються ціною підвищеної операційної складності:

1. Необхідно застосовувати міграції схем у кожній базі даних окремо.
2. Керування великою кількістю з’єднань ускладнюється.
3. Ресурси використовуються неефективно для невеликих орендарів,

і загалом це найдорожча модель з точки зору інфраструктури та підтримки.

Таким чином, цей підхід найкраще підходить для великих корпоративних клієнтів з високими вимогами до безпеки та повної ізоляції, але не є оптимальним для малих або середніх SaaS-рішень, де важлива економічність і простота обслуговування.

Schema-per-tenant Model - це модель яка використовує єдину базу даних, але створює окрему схему для кожного орендаря (рис. 1.9).

```
-- Create schema for Tenant 1
CREATE SCHEMA tenant1;

-- Create tables in Tenant 1's schema
CREATE TABLE tenant1.customers (
  id INT PRIMARY KEY,
  name VARCHAR(255),
  email VARCHAR(255),
  ...
);

-- Create schema for Tenant 2
CREATE SCHEMA tenant2;

-- Create tables in Tenant 2's schema
CREATE TABLE tenant2.customers (
  id INT PRIMARY KEY,
  name VARCHAR(255),
  email VARCHAR(255),
  ...
);
```

Рисунок 1.9. Модель зберігання Schema-per-tenant Model

Модель спільної бази даних з окремими схемами для кожного клієнта забезпечує краще розділення даних між клієнтами, зберігаючи при цьому економічну ефективність спільного використання однієї бази.

Такий підхід знижує ризик витоку або змішування даних, порівняно зі спільними таблицями, та дозволяє реалізувати індивідуальні налаштування для кожного орендаря (наприклад, окремі індекси, розширення чи структуру даних).

Попри збалансованість між ізоляцією та ефективністю, модель спільної бази даних з окремими схемами для кожного орендаря має і свої недоліки.

Зокрема:

1. Міграції схем потрібно виконувати для кожного орендаря окремо, що ускладнює оновлення.
2. Обмеження кількості об’єктів у базі даних (схем, таблиць, індексів) можуть створювати проблеми при великій кількості клієнтів.
3. На рівні однієї бази все ще можливі конфлікти ресурсів між орендарями.
4. Резервне копіювання та відновлення даних кожного клієнта стає складнішим.

Таким чином, цей підхід є компромісом між повною ізоляцією та простотою управління, який добре підходить для середніх SaaS-рішень, але може виявитися занадто складним при масштабуванні до сотень чи тисяч орендарів.

Вибір конкретної моделі залежить від бізнес-вимог SaaS-платформи. Якщо пріоритетом є висока безпека і індивідуальні налаштування клієнтів, доцільно використовувати database-per-tenant. Якщо важливе поєднання ізоляції і економії ресурсів – schema-per-tenant. Для великих платформ із великою кількістю клієнтів та однаковими функціональними потребами найбільш ефективною є модель shared-table (рис. 1.10).

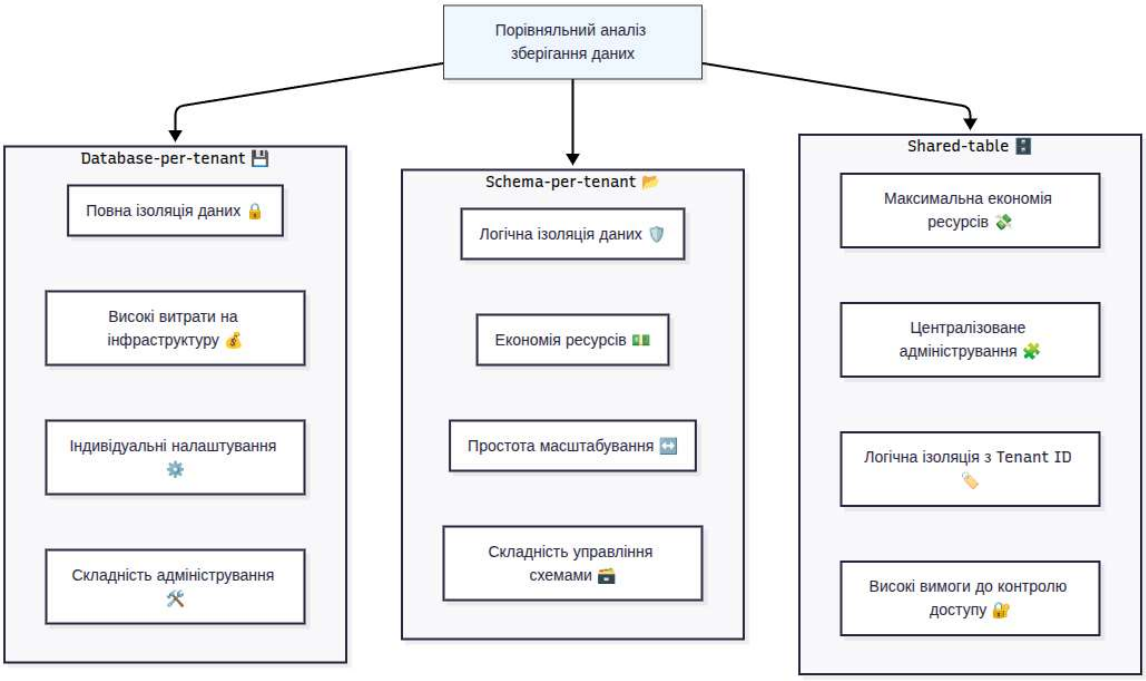

Рисунок 1.10. Порівняльний аналіз зберігання даних

1.5. Виклики та перспективи розвитку багатокористувацьких SaaS-систем

Багатокористувацькі SaaS-платформи забезпечують ефективне використання ресурсів і спрощують процес адміністрування, однак їх створення та подальша підтримка супроводжуються низкою складних завдань. Одним із ключових є гарантування безпеки даних. Оскільки одна система обслуговує велику кількість клієнтів одночасно, навіть незначна вразливість може становити загрозу для всіх користувачів. Для мінімізації таких ризиків застосовують сучасні засоби аутентифікації, шифрування інформації, аудит дій користувачів і багаторівневу систему контролю доступу.

Ще однією важливою проблемою є масштабованість. Із розширенням клієнтської бази та зростанням обсягів оброблюваних даних виникає потреба в ефективному розподілі обчислювальних і базових ресурсів, щоб уникнути перевантажень і забезпечити стабільну продуктивність. Для цього використовують оптимізацію запитів до баз даних, а також механізми горизонтального та вертикального масштабування.

Крім того, значною складністю залишається адміністрування індивідуальних налаштувань для кожного орендаря. Хоча багатокористувацька архітектура спрощує централізоване оновлення та управління, необхідність підтримувати специфічні функції або інтеграції для окремих клієнтів може ускладнювати подальший розвиток системи.

Але разом з цим SaaS-системи мають значний потенціал подальшого розвитку. Одним із ключових напрямів є підвищення їх гнучкості та здатності адаптуватися до потреб користувачів. Завдяки сучасним технологіям клієнти отримують змогу самостійно налаштовувати інтерфейс, оптимізувати бізнес-процеси та здійснювати інтеграцію з зовнішніми сервісами через API. Це забезпечує більш індивідуальний підхід і підвищує конкурентоспроможність SaaS-рішень на ринку.

Все більше компаній стимулюють попит на SaaS, впроваджуючи хмарне програмне забезпечення. У найближчі роки бізнес перейде на вертикальний підхід до SaaS, щоб оптимізувати виробництво спеціалізованого та економічно ефективного програмного забезпечення [6].

Промислові аналітики прогнозують, що до 2025 року глобальний ринок SaaS буде оцінено в 208 мільярдів доларів – стрибок, частково обумовлений зростаючим попитом на хмарні програми, недавнім асортиментом мобільних пристроїв і зростаючим попитом на інформативні дані [6].

Іншою перспективою є використання штучного інтелекту та аналітики для оптимізації роботи платформи. Автоматизація обробки даних, передбачення навантажень, адаптивне масштабування та рекомендаційні системи дозволяють підвищити ефективність роботи як користувачів, так і самої платформи.

РОЗДІЛ 2
МЕТОДОЛОГІЯ РОЗРОБКИ БАГАТОКОРИСТУВАЦЬКОЇ АРХІТЕКТУРИ

2.1. Загальний опис методології

Методологія охоплює етапи від аналізу вимог до розгортання та моніторингу системи. На початковому етапі проводиться моделювання архітектури з урахуванням типів multi-tenancy: shared database з ізоляцією на рівні схем (schema-based) для балансу між простотою та безпекою. Ключовими принципами є [7]:

1. Ізоляція даних. Кожен tenant отримує окремий PostgreSQL schema, що запобігає витокам інформації між клієнтами.
2. Масштабованість. Використання єдиної бази даних з динамічним перемиканням search path для запитів.
3. Гнучкість інтерфейсу. Фронтенд адаптується до tenant-специфічних налаштувань через API.

Ітерації розробки включають спринти по 2 тижні, з фокусом на мінімально життєздатний продукт (MVP) для одного tenant, подальше розширення на множинність та оптимізацію продуктивності.

Розробка проводиться за ітеративно-інкрементальною моделлю Agile з фіксованими двотижневими спринтами, що включають повний цикл: планування, проєктування, реалізацію, тестування, ревью та ретро. Кожен спринт орієнтований на доставку інкрементально життєздатного функціоналу, починаючи з Minimum Viable Product (MVP) для одного тенанта, з подальшим розширенням на підтримку множинності клієнтів.

Методологія базується на наборі ключових архітектурних принципів:

1. Ізоляція даних на рівні схем (Schema-per-Tenant). Кожен тенант отримує окрему схему в PostgreSQL. Middleware TenantMainMiddleware динамічно перемикає search_path, забезпечуючи повну логічну ізоляцію без фізичного розділення баз.

2. Єдина кодова база (Single Codebase). Усі тенанти обслуговуються однією інстанцією додатка, що спрощує оновлення, патчинг та управління версіями.
3. Горизонтальна масштабованість. Архітектура підтримує розподіл навантаження через кілька інстанцій бекенду за допомогою NGINX як reverse proxy та Redis для кешування tenant-конфігурацій.
4. Тенант-агностичний API. RESTful API експонує уніфіковані ендпоінти, але контекст тенанта визначається автоматично на основі домену або заголовка X-Tenant-ID.
5. Динамічна кастомізація UI. React-додаток завантажує конфігурацію тенанта (тема, логотип, доступні функції) під час ініціалізації, забезпечуючи white-label досвід.

2.2. Вибір технологічного стеку

Вибір технологій обумовлений вимогами до надійності, продуктивності та підтримки multi-tenancy. Бекенд реалізовано на Django (версія 4.2+), який забезпечує швидку розробку RESTful API та інтеграцію з PostgreSQL. Для multi-tenancy інтегровано пакет django-tenants, що підтримує schema-based підхід без значних змін у моделях Django. Фронтенд побудовано на React (версія 18+), з використанням бібліотек як React Router для маршрутизації та Axios для API-викликів, що дозволяє створювати SPA (Single Page Application) з tenant-специфічним рендерингом.

Додаткові інструменти:

1. База даних: PostgreSQL 15+ для схемної ізоляції.
2. API: Django REST Framework для серіалізації даних.
3. Деплоймент: Docker для контейнеризації та Kubernetes для оркестрації.

Цей стек обрано через його зрілість у SaaS-проєктах, де django-tenants зменшує складність на 30–50% порівняно з кастомними рішеннями.

2.3. Реалізація бекенду з використанням Django та django-tenants

Реалізація бекенду починається з конфігурації проєкту Django. Встановлюється django-tenants через pip, після чого налаштовується middleware для автоматичного виявлення тенанта за hostname запиту (наприклад, tenant1.example.com).

Ключові кроки:

1. Створення public schema. Встановлення таблиці Tenant (з полями schema_name, domain) у public schema для керування tenant'ами.
2. Моделювання даних. Tenant-специфічні моделі (наприклад, UserProfile) позначаються як TenantMixin, розміщуються в окремих schema. Shared-моделі (наприклад, глобальні конфігурації) – у public.
3. Маршрутизація. Middleware TenantMiddleware перемикає search_path на schema tenant'а, роблячи дані доступними лише для поточного клієнта.
4. API-ендпоінти. Використання DRF для tenant-aware views, з фільтрацією запитів через Tenant.objects.get(schema_name=request.tenant.schema_name).

Цей підхід забезпечує ізоляцію на рівні бази даних, з мінімальним оверхедом (менше 5% на запит). Для створення нового tenant'а реалізовано адміністративний ендпоінт, що генерує schema та домен автоматично [8].

2.4. Розробка клієнтської частини на базі React

Розробка клієнтської частини багатокористувацької архітектури на базі React (версія 18 або вища) здійснюється з урахуванням принципів модульності, реактивності та адаптивності до специфічних вимог кожного тенанта. Використання React як основи фронтенду обґрунтоване його високою продуктивністю, підтримкою віртуального DOM, розвиненою екосистемою

компонентів та можливістю ефективного управління станом у складних SPA-додатках. У контексті multi-tenancy клієнтська частина повинна забезпечувати динамічне завантаження конфігурації тенанта, кастомізацію інтерфейсу, безпечну автентифікацію та ізоляцію даних на рівні користувацького досвіду.

Розробка проводиться з дотриманням принципів Clean Architecture та Component-Driven Development (CDD), що сприяє створенню часто використовуваних, тестувальних та масштабованих компонентів. Проєкт ініціалізується за допомогою Vite – сучасного інструменту складання, який забезпечує швидке гаряче перезавантаження (HMR) та оптимізований продакшн-білд.

Клієнтська частина розробляється як модульна односторінкова програма, де компоненти адаптуються до тенантів за допомогою контексту (React Context API). Основні етапи:

1. Ініціалізація проєкту. Створення додатка з використанням інструменту Vite для швидкого складання, інтеграція React Router для динамічної маршрутизації (наприклад, /dashboard з параметрами тенанта).
2. Виявлення тенанта. Під час завантаження додатка надсилається запит до API /tenants/ для отримання конфігурації (тем, прав доступу) на основі імені хоста.
3. Розробка компонентів. Використання хуків (useEffect, useContext) для завантаження тенант-специфічних даних; наприклад, компонент Dashboard рендерить інтерфейс на основі налаштувань тенанта (tenant.settings).
4. Забезпечення безпеки. Застосування JWT-токенів з ідентифікатором тенанта для авторизації, з перехоплювачами Axios для додавання заголовків тенанта до запитів.

React забезпечує реактивність інтерфейсу, що дозволяє кастомізацію користувацького досвіду без перезавантаження сторінки, що є критичним для multi-tenant сценаріїв. Оптимізація включає розділення коду (code splitting) для зменшення розміру пакету на 20–30%.

Проєкт створюється з використанням Vite та шаблону React + TypeScript для забезпечення типізації та полегшення підтримки коду

Структура директорій побудована за принципом feature-sliced design, що групує компоненти, логіку та стилі за функціональними модулями:

```
src/
├── app/          - Глобальний стан, провайдери, маршрутизація
├── entities/     - Бізнес-сутності (User, Organization)
├── features/     - Функціональні можливості (Auth, Dashboard)
├── shared/       - Часто використовуванні компоненти, утиліти
├── widgets/      - Складові блоки сторінок (Sidebar, Header)
├── pages/        - Сторінки додатка
├── processes/    - Багатокрокові сценарії (onboarding)
└── tenants/      - Тенант-специфічні адаптації
```

Така структура полегшує масштабування та ізоляцію логіки для різних тенантів.

Для глобального стану використовується Zustand або Redux Toolkit з тенант-ізольованими сторями. JWT-токени зберігаються в httpOnly cookies для підвищення безпеки.

Автентифікація включає:

1. Логін через субдомен (tenant1.app.com).
2. Перевірку домену на бекенді.
3. Видачу JWT з tenant_id у payload.
4. Автоматичне перенаправлення на відповідний дашборд.

Оптимізація продуктивності в проєкті включає:

1. Code Splitting: Використання React.lazy та Suspense для маршрутів.
2. Bundle Analysis: vite-bundle-visualizer для контролю розміру.
3. Memoization: React.memo, useMemo, useCallback.
4. Virtual Scrolling: Для великих списків (react-window).

Інтеграція здійснюється через RESTful API з конфігурацією міждоменного обміну ресурсами (CORS) у Django. Клієнтська частина взаємодіє з серверною за

протоколом HTTPS, передаючи інформацію про тенанта в заголовках (наприклад, X-Tenant-Schema). Ключові аспекти:

1. Шар API. Серверна частина реалізує ендпоінти на кшталт /api/v1/{tenant}/resources/, з middleware для валідації тенанта.
2. Синхронізація даних. Використання WebSockets (через Django Channels) для оновлень у реальному часі в межах тенантів.
3. Розгортання. Окремі контейнери для серверної частини (Gunicorn + Nginx) та клієнтської (статичний сервер Nginx), NGINX як зворотним проксі для маршрутизації за доменами.

Така інтеграція мінімізує затримки та забезпечує безперервний користувацький досвід [9].

Таким чином, клієнтська частина на React забезпечує повну адаптивність до multi-tenant середовища, високу продуктивність, безпеку та гнучкість кастомізації, що є критичним для SaaS-продуктів.

2.5. Тестування та валідація

Тестування та валідація є критичними етапами методології, що забезпечують надійність, безпеку та продуктивність багатокористувацької архітектури. Процес охоплює багаторівневий підхід, включаючи модульне, інтеграційне, наскрізне тестування, а також перевірку продуктивності та безпеки. Для multi-tenant систем особливий акцент робиться на перевірці ізоляції даних між тенантами, щоб уникнути витоків інформації та забезпечити незалежність клієнтів. Тестування інтегрується в CI/CD пайплайн (наприклад, за допомогою GitHub Actions або Jenkins), що дозволяє автоматизувати запуск тестів на кожному коміті або pull request. Критерії успіху включають досягнення 99,9% доступності, часу відповіді менше 100 мс на тенанта, покриття коду не нижче 85% та відсутність критичних вразливостей.

Модульне тестування фокусується на ізольованій перевірці окремих компонентів. Для серверної частини застосовується фреймворк pytest у комбінації

з django-tenants, що дозволяє симулювати множинні схеми та перевіряти ізоляцію даних. Django-tenants надає спеціальні класи, такі як TenantTestCase, для автоматичного створення тенантів під час тестів, без потреби в ручному налаштуванні.

Наприклад, базовий тест на ізоляцію даних між тенантами з використанням TenantTestCase (рис. 2.1):

```python
from django_tenants.test.cases import TenantTestCase
from django_tenants.test.client import TenantClient
from accounts.models import User

class IsolationTestCase(TenantTestCase):  new *
    def setUp(self):  new *
        super().setUp()
        self.client = TenantClient(self.tenant)

    def test_data_isolation(self):  new *
        User.objects.create(name='Tenant1 User')
        self.assertEqual(User.objects.count(),  second: 1)

        # Перевірка, що дані не доступні в іншому тенанті (симуляція через окремий тест-кейс)
        with self.assertRaises(User.DoesNotExist):
            User.objects.get(name='Tenant2 User')
```

Рисунок 2.1. Приклад модульного тестування

Для швидкого виконання тестів застосовується FastTenantTestCase (рис. 2.2), де схема створюється лише один раз, а дані очищаються після кожного тесту. Це зменшує час виконання на 50–70% у великих тестах, але вимагає обережності з управлінням станом [10].

```python
from django_tenants.test.cases import FastTenantTestCase
from django_tenants.test.client import TenantClient
from accounts.models import User

class IsolationTestCase(FastTenantTestCase):  new *
    def setUp(self):  new *
        super().setUp()
        self.client = TenantClient(self.tenant)

    def test_data_isolation(self):  new *
        User.objects.create(name='Tenant1 User')
        self.assertEqual(User.objects.count(),  second: 1)

        with self.assertRaises(User.DoesNotExist):
            User.objects.get(name='Tenant2 User')
```

Рисунок 2.2. Приклад модульного тестування з FastTenantTestCase

Для клієнтської частини використовується Jest з React Testing Library для тестування компонентів, з акцентом на тенант-специфічні адаптації, такі як динамічне завантаження конфігурацій.

Наприклад, тест компонента Dashboard з mock API для тенанта (рис. 2.3):

```
import { render, screen } from "@testing-library/react";
import { TenantProvider } from "./TenantContext";
import Dashboard from "./Dashboard";

test("renders tenant-specific dashboard", () => {
  const mockTenant = { theme: "dark", name: "Tenant1" };
  render(
    <TenantProvider value={mockTenant}>
      <Dashboard />
    </TenantProvider>
  );
  expect(screen.getByText(`Welcome to ${mockTenant.name} Dashboard`)).toBeInTheDocument();
});
```

Рисунок 2.3. Приклад компонентного тестування

Інтеграційне тестування перевіряє взаємодію між компонентами, зокрема API-ендпоінтами та базою даних. Використовується pytest з django-tenants для симуляції запитів у контексті тенантів з TenantClient для автоматичного встановлення домену.

Наприклад, тест на інтеграцію API з перевіркою ізоляції (рис. 2.4):

```
from django.urls import reverse
from django_tenants.test.cases import TenantTestCase
from django_tenants.test.client import TenantClient

class APITestCase(TenantTestCase):  new *
    def setUp(self):  new *
        super().setUp()
        self.client = TenantClient(self.tenant)

    def test_api_isolation(self):  new *
        response = self.client.post(reverse('create_profile'), data: {'name': 'Test User'})
        self.assertEqual(response.status_code,  second: 201)
        # Перевірка, що дані збережені лише в поточній схемі
        self.assertEqual(User.objects.count(),  second: 1)
```

Рисунок 2.4. Приклад інтеграційного тестування

Наскрізне (end-to-end) тестування симулює реальний користувацький досвід, включаючи взаємодію фронтенду з бекендом (рис. 2.5). Застосовується Cypress для автоматизації сценаріїв, таких як автентифікація в multi-tenant середовищі, з перевіркою на перехресні витоки даних. Cypress інтегрується з Django через запуск локального сервера та симуляцію запитів [1].

```
describe("Multi-Tenant Authentication", () => {
  it("logs in as tenant user", () => {
    cy.visit("https://tenant1.example.com/login");
    cy.get('input[name="username"]').type("user@tenant1.com");
    cy.get('input[name="password"]').type("password");
    cy.get('button[type="submit"]').click();
    cy.url().should("include", "/dashboard");
    cy.contains("Welcome to Tenant1 Dashboard");
  });
});
```

Рисунок 2.5. Приклад наскрізного тестування

Тестування продуктивності проводиться з використанням Locust для симуляції навантаження від 1000+ одночасних користувачів, з фокусом на multi-tenant сценарії, такі як перемикання схем під навантаженням. Locust дозволяє визначати поведінку користувачів у Python-коді (рис. 2.5).

Наприклад, базовий Locust файл для тестування API:

```
from locust import HttpUser, task, between


class TenantUser(HttpUser):  new *
    wait_times = between( min_wait: 1,  max_wait: 5)

    @task  new *
    def get_dashboard(self):
        self.client.get("/api/v1/dashboard",
                        headers={"X-Tenant-Schema": "tenant1"})
```

Рисунок 2.5. Приклад тестування продуктивності

Аудит безпеки включає використання OWASP ZAP для динамічного сканування вразливостей, таких як SQL-ін’єкції, XSS та витоки даних між

тенантами. ZAP налаштовується як проксі для перехоплення трафіку під час ручного та автоматизованого тестування [12].

Наприклад, базова конфігурація ZAP для сканування:

1. Запуск ZAP як проксі (localhost:8080).
2. Налаштування браузера або Cypress для використання проксі.
3. Виконання активного скану на URL: https://example.com/api.

Додатково застосовуються статичні аналізатори, як Bandit для Python-коду. Критерії: відсутність високорівневих вразливостей за OWASP Top 10.

У висновку, запропонована методологія забезпечує ефективну розробку масштабованої multi-tenant системи, придатної для впровадження в промисловому середовищі.

РОЗДІЛ 3
ПРОЄКТУВАННЯ SAAS ПЛАТФОРМИ

3.1. Модель даних і логіка ізоляції користувачів (тенантів)

У контексті розробки багатокористувацької SaaS-платформи на базі Django Tenants застосовується архітектурний підхід ізоляції даних за схемою бази даних (schema-per-tenant), що забезпечує повне логічне та фізичне розділення інформації між окремими клієнтами (тенантами). Така модель відповідає вимогам безпеки, масштабованості та відповідності нормативним стандартам (наприклад, GDPR, ISO/IEC 27001) [13]. Використовується багато схема архітектура PostgreSQL, у якій:

1. Схема public містить глобальні, спільні для всіх тенантів дані, зокрема метадані про клієнтів, доменні прив'язки та системні налаштування.
2. Приватні схеми (одна на кожен тенант) зберігають усі бізнес-дані, специфічні для відповідної організації: користувачів, проєкти, документи, транзакції тощо.

Модель тенанта яка буде знаходитись у публічній схемі (рис. 3.1):

```python
from django_tenants.models import TenantMixin
from django.db import models

class Client(TenantMixin):  new *
    name = models.CharField(max_length=100, unique=True)
    schema_name = models.CharField(max_length=63, unique=True, editable=False)
    paid_until = models.DateField()
    on_trial = models.BooleanField(default=True)
    created_on = models.DateField(auto_now_add=True)

    auto_create_schema = True
    auto_drop_schema = False
```

Рисунок 3.1. Модель тенанта (клієнта)

Модель Client у цьому прикладі використовується для реалізації багатокористувацької архітектури з бібліотекою django-tenants. Вона представляє

одного клієнта або орендаря (tenant) системи, який має власну схему бази даних у PostgreSQL. Це дозволяє ізолювати дані різних клієнтів один від одного.

Клас Client наслідує TenantMixin, який надає базову логіку для роботи з тенантами. Поле name зберігає назву клієнта і є унікальним, щоб не було двох клієнтів з однаковим ім'ям. Поле schema_name зберігає назву схеми PostgreSQL, яка відповідає цьому клієнту. Це поле теж унікальне й недоступне для редагування вручну, адже його створює система при додаванні нового тенанта.

Поле paid_until визначає дату, до якої клієнт оплатив підписку. Це зручно для контролю доступу до сервісу – після закінчення цього терміну можна призупиняти роботу клієнта або обмежувати функціональність. Поле on_trial показує, чи перебуває клієнт у пробному періоді. Поле created_on автоматично зберігає дату створення клієнта при його додаванні в базу.

Атрибут auto_create_schema = True означає, що після створення об'єкта Client система автоматично створить нову схему в базі даних для цього клієнта. Атрибут auto_drop_schema = False означає, що при видаленні клієнта його схема не буде автоматично видалятися – це запобігає втраті даних, якщо потрібно зберегти резервну копію або зробити повторне підключення.

У практиці роботи з django-tenants ця модель визначає базову одиницю багатокористувацької системи. Коли створюється новий клієнт у базі створюється окрема схема, у якій будуть свої таблиці, записи та дані. Це дозволяє кожному клієнту мати незалежне середовище, навіть якщо вони користуються однією й тією ж Django-аплікацією.

Даний підхід реалізовано за допомогою бібліотеки Django Tenants, яка інтегрує механізм SEARCH_PATH PostgreSQL з ORM Django, забезпечуючи автоматичне перемикання контексту бази даних під час виконання запитів [14].

Схема public містить глобальні, спільні для всіх тенантів дані, зокрема метадані про клієнтів, доменні прив'язки та системні налаштування. Теннант

специфічні моделі , що належать конкретному клієнту, позначаються наслідковують класс TenantSpecificModel (рис. 3.2):

```python
from django.db import models


class Project(TenantSpecificModel):  new *
    name = models.CharField(max_length=200)
    description = models.TextField(blank=True)
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):  new *
        return self.name
```

Рисунок 3.2. Приклад теннант специфічної моделі

Модель Project у цьому прикладі є звичайною моделлю, яка належить до конкретного тенанта. Вона наслідує клас TenantSpecificModel, що є частиною системи django-tenants і використовується для створення моделей, дані яких ізольовані між різними схемами бази даних.

Це означає, що кожен клієнт (tenant), визначений через модель Client, має власну копію таблиці Project у своїй схемі. Таким чином, проєкти одного клієнта не змішуються з проєктами іншого – навіть якщо назви чи ідентифікатори збігаються.

Поле name зберігає назву проєкту, а description містить текстовий опис і може бути порожнім, оскільки вказано blank=True. Поле created_at автоматично заповнюється датою та часом створення проєкту, коли запис додається в базу.

Отже, Project – це типовий приклад моделі даних, що належить до певного клієнта у багатокористувацькому середовищі. Кожен tenant має власну таблицю project, а додавання, редагування чи видалення записів відбувається тільки в межах його схеми.

Механізм маршрутизації запитів у багатотенантній архітектурі SaaS-платформи реалізовано за допомогою TenantMainMiddleware, який забезпечує динамічне визначення активного тенанта на основі аналізу HTTP-запиту та

автоматичне перемикання контексту бази даних до відповідної схеми PostgreSQL [14].

TenantMainMiddleware це ключовий компонент Django Tenants для автоматичної маршрутизації запитів до правильної схеми бази даних (тенанта) на основі домену або субдомену який включає:

1. Аналіз HTTP-запиту (субдомен, заголовок, токен).
2. Визначення відповідного schema_name з таблиці Domain.
3. Встановлення connection.schema_name та search_path.
4. Усі подальші запити до БД виконуються в ізольованому контексті.

У багатотенантних SaaS-додатках, побудованих на основі Django Tenants, критичною є вимога ізоляції даних між клієнтами (тенантами). Для виконання операцій із даними в межах конкретної схеми, навіть поза HTTP-запитом (наприклад, у скриптах, Celery-задачах, міграціях чи адмінці), використовується менеджер контексту schema_context [14].

Цей приклад демонструє, як у django-tenants можна тимчасово переключитися на конкретну схему бази даних і виконати дії лише в її контексті (рис. 3.3).

```
from django_tenants.utils import schema_context
from project.models import Project

with schema_context('tenant_acme_corp'):
    Project.objects.create(name="CRM Migration")
```

Рисунок 3.3. Переключення контексту в django-tenants

Функція schema_context() – це контекстний менеджер, який дозволяє виконати певний блок коду всередині конкретної схеми (тенанта), без зміни глобального контексту застосунку.

У наведеному прикладі відбувається наступне:

1. schema_context(‘tenant_acme_corp’) тимчасово перемикає підключення до схеми tenant_acme_corp. Це означає, що всі операції з моделями Django (створення, запити, оновлення, видалення) у цьому блоці

будуть виконані в межах саме цієї схеми, а не у public або будь-якій іншій.

2. Всередині цього контексту викликається: Project.objects.create(name="CRM Migration"). Цей рядок створює новий запис у таблиці project, але саме в схемі tenant_acme_corp. Інші тенанти не побачать цього запису, адже кожен має власну копію таблиці.
3. Після виходу з блоку with контекст автоматично повертається до попередньої схеми (зазвичай public), тому наступні операції вже не впливають на дані tenant_acme_corp.

Використання цієї конструкції є обов'язковим і єдиним безпечним способом виконання операцій із даними конкретного тенанта у фоновому режимі. Воно забезпечує:

1. Повну ізоляцію даних.
2. Відсутність витоків контексту.
3. Сумісність із Celery, командами, скриптами та тестами.

Такий підхід є фундаментальним принципом безпечної багатотенантної архітектури та відповідає вимогам zero-trust та data sovereignty.

Запропонована модель даних із застосуванням схемної ізоляції забезпечує повну автономність кожного тенанта, відповідає принципам zero-trust архітектури та є рекомендованим підходом для комерційних SaaS-платформ середнього та великого масштабу. Використання Django Tenants абстрагує складність управління схемами, дозволяючи розробнику зосередитися на бізнес-логіці.

3.2. Розробка API інтерфейсу для взаємодії між клієнтом і сервером

Розробка інтерфейсу прикладного програмування (API) є центральним компонентом архітектури SaaS-платформи, забезпечуючи структуровану, безпечну та масштабовану взаємодію між клієнтською частиною (React) та серверною логікою (Django). У рамках даної роботи використано RESTful-підхід із

застосуванням Django REST Framework (DRF), що гарантує відповідність стандартам сучасних веб-сервісів.

API розроблено з урахуванням таких ключових принципів:

1. RESTful-архітектура: Використання HTTP-методів (GET, POST, PUT, PATCH, DELETE) та статус-кодів.
2. Ізоляція даних: Кожен запит виконується в контексті схеми поточного тенанта (автоматично через TenantMainMiddleware).
3. Версіонування: /api/v1/ – для забезпечення зворотної сумісності.
4. Формат обміну: JSON (з можливістю розширення до MessagePack при високих навантаженнях).

Технологічний стек:

1. Django REST Framework – для серіалізації, валідації, автентифікації та рендерингу.
2. DRF SimpleJWT – для токенної автентифікації (access/refresh tokens).
3. Django Filter + Django REST Framework Filters – для розширеної фільтрації та пошуку.
4. drf-spectacular – автоматична генерація OpenAPI 3.0 схеми (Swagger UI, ReDoc).

Для забезпечення повнофункціонального управління проєктами в межах API-інтерфейсу реалізовано ViewSet на основі Django REST Framework, що підтримує операції CRUD (створення, читання, оновлення, видалення), фільтрацію, пошук, сортування та автоматичну ізоляцію даних у контексті поточного тенанта [16].

Серіалізатор (ProjectSerializer) (рис. 3.4)

Рисунок 3.4. Приклад серіалізатора (ProjectSerializer)

Цей приклад показує, як за допомогою Django REST Framework створюється серіалізатор для моделі Project.

Серіалізатор перетворює об’єкти Django у зручний формат (наприклад JSON) для передачі через API, а також може виконувати зворотну операцію – створювати або оновлювати об’єкти з отриманих даних [15].

Клас ProjectSerializer наслідує ModelSerializer, який автоматично створює поля на основі моделі Project.

Поле owner оголошено як ReadOnlyField із параметром source=‘owner.email’. Це означає, що в API це поле буде доступне лише для читання і відображатиме адресу електронної пошти власника проєкту, але не може бути змінене через запит.

Саме поле owner очікується в моделі Project як зв’язок з користувачем (ймовірно, через ForeignKey до User).

Поле task_count створено за допомогою SerializerMethodField, що дозволяє обчислювати значення динамічно. У цьому випадку метод get_task_count повертає кількість пов’язаних завдань для конкретного проєкту – передбачається, що в моделі Task є зв’язок ForeignKey із Project із related_name=‘tasks’. Таким чином, виклик obj.tasks.count() підраховує кількість пов’язаних завдань.

У внутрішньому класі Meta зазначено модель Project і список полів, які мають бути включені до серіалізації: id, name, description, status, owner, created_at, task_count. Це означає, що при отриманні даних через API користувач бачить всю основну інформацію про проєкт, включно з власником, статусом, датою створення та кількістю завдань.

ProjectViewSet (рис. 3.5) є центральним компонентом API-інтерфейсу, що забезпечує повнофункціональне управління сутністю Project у межах RESTful-архітектури. Реалізовано на основі Django REST Framework (DRF) з урахуванням вимог ізоляції даних, безпеки, продуктивності та масштабованості.

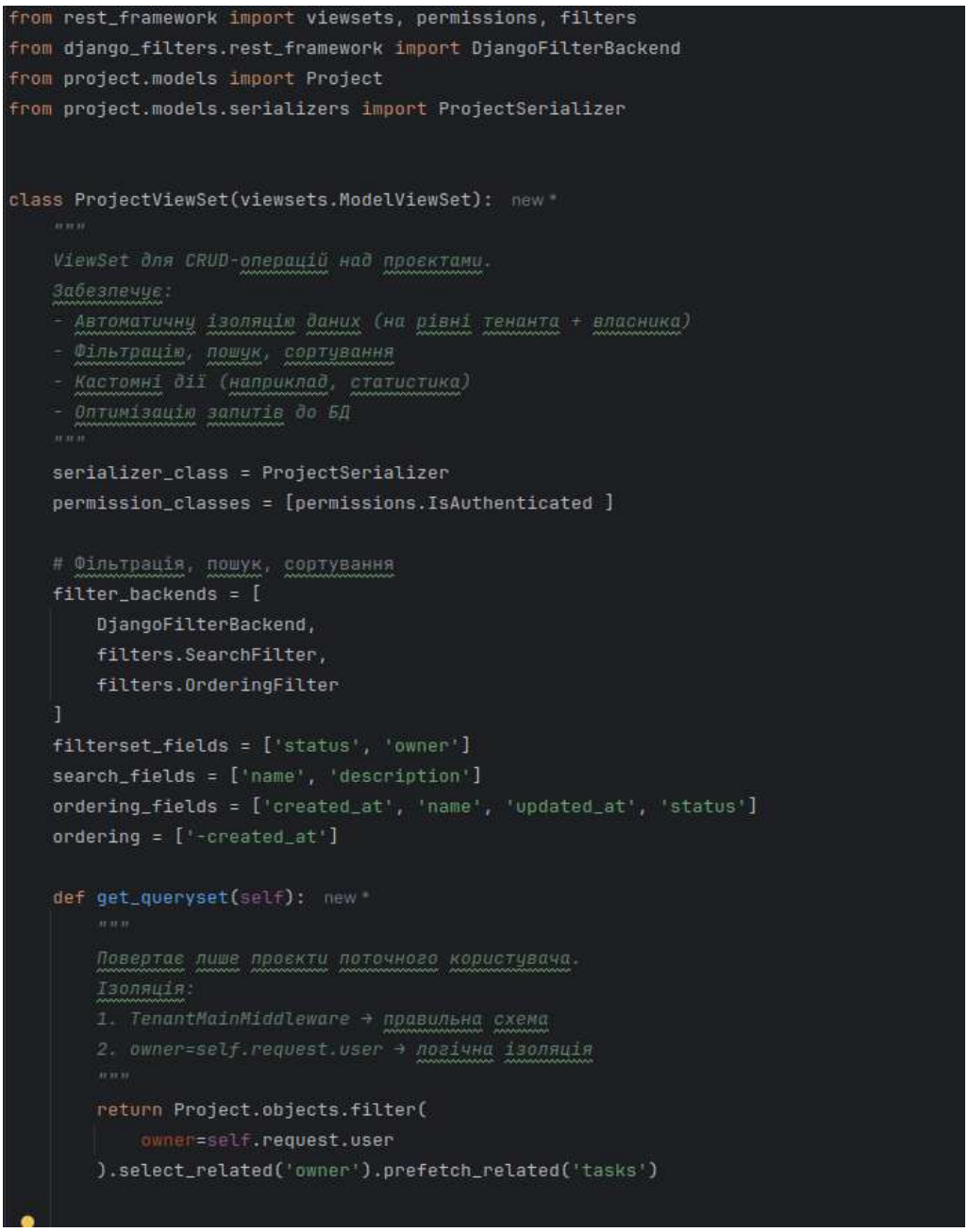

Рисунок 3.5. Приклад ViewSet (ProjectViewSet)

Цей клас ProjectViewSet є повноцінним REST API-контролером для роботи з моделлю Project у системі, побудованій на Django REST Framework і django-tenants. Він відповідає за всі CRUD-операції (створення, отримання, оновлення, видалення проєктів), а також реалізує ізоляцію даних, фільтрацію, пошук і сортування.

Клас наслідує ModelViewSet, тому автоматично отримує стандартні методи list, retrieve, create, update, partial_update і destroy. Це означає, що він може обробляти всі типові REST-запити до ресурсу /projects/.

Поле serializer_class вказує, що для перетворення об’єктів у JSON і назад використовується ProjectSerializer. Параметр permission_classes = [permissions.IsAuthenticated] означає, що API доступне лише для автентифікованих користувачів.

Далі визначено набір бекендів для фільтрації, пошуку та сортування.

1. DjangoFilterBackend дозволяє робити точну фільтрацію за полями, переліченими у filterset_fields, тобто за статусом і власником проєкту.
2. SearchFilter забезпечує текстовий пошук за назвою й описом, тому запит типу ?search=crm знайде всі проєкти, у яких згадується це слово.
3. OrderingFilter дозволяє сортувати результати за будь-яким полем, зазначеним у ordering_fields – наприклад, за датою створення, назвою, статусом або датою оновлення. За замовчуванням результати відсортовані у спадному порядку за created_at, тобто спочатку відображаються найновіші проєкти.

Метод get_queryset визначає, які записи доступні поточному користувачу. Він повертає лише ті проєкти, власником яких є користувач, що зробив запит (owner=self.request.user). Це забезпечує логічну ізоляцію даних – навіть усередині одного тенанта користувач бачить тільки власні проєкти. Фізичну ізоляцію даних на рівні бази забезпечує TenantMainMiddleware, який під час кожного запиту підключає правильну схему бази даних для поточного тенанта.

Також у запит додаються оптимізації: select_related(‘owner’) підтягує пов’язані об’єкти власника за один SQL-запит, а prefetch_related(‘tasks’) ефективно завантажує пов’язані завдання, щоб уникнути додаткових запитів під час серіалізації або відображення статистики. Це помітно покращує продуктивність при роботі з великими наборами даних.

Отже, цей ProjectViewSet поєднує всі важливі аспекти для побудови сучасного багатокористувацького REST API: ізоляцію даних на рівні тенанта та користувача, зручні інструменти фільтрації і сортування, а також оптимізовані запити до бази даних.

ProjectViewSet є еталонним прикладом реалізації RESTful-ендпоінта у багатотенантній SaaS-платформі. Він демонструє:

1. Повну інтеграцію з Django Tenants (схемна + логічна ізоляція).
2. Відповідність принципам REST та DRY.
3. Високу продуктивність завдяки оптимізації запитів.
4. Гнучкість та розширюваність через кастомні дії.

Ця реалізація може бути шаблоном для інших сутностей (Task, Invoice, Team тощо) і є рекомендованою практикою при розробці масштабованих SaaS-додатків.

3.3. Реалізація системи аутентифікації, авторизації та розмежування доступу

Система автентифікації, авторизації та розмежування доступу є критичним компонентом безпеки багатотенантної SaaS-платформи. У рамках даної роботи реалізовано сучасну, безпечну та масштабовану архітектуру, що відповідає стандартам OWASP ASVS, GDPR та ISO/IEC 27001. Використано JWT-автентифікацію, рольову модель доступу (RBAC) та об’єктно-орієнтоване розмежування з автоматичною ізоляцією на рівні тенанта.

Архітектурні принципи безпеки:

1. Zero Trust. Кожен запит перевіряється незалежно.
2. Least Privilege. Користувач отримує мінімальні необхідні права.
3. Defense in Depth. Багаторівневий захист (JWT, RBAC, Object-level).
4. Stateless Authentication. JWT-токени, без сесій.
5. Tenant Isolation. Автоматична ізоляція через TenantMainMiddleware.

Технологічний стек:

1. JWT-автентифікація. djangorestframework-simplejwt.
2. Користувачі. Кастомна модель User (унаслідувана від AbstractUser).
3. Ролі. Кастомне поле role + Group-based permissions.
4. CORS. django-cors-headers.
5. Rate Limiting. DRF Throttling.

6. Логування. django-auditlog.

У багатотенантній архітектурі SaaS-платформи модель користувача (рис. 3.6) є ключовою сутністю, що забезпечує ідентифікацію, автентифікацію, авторизацію та ізоляцію даних на рівні окремого клієнта (тенанта). Розроблена модель ґрунтується на кастомізації стандартної моделі Django (AbstractUser) з інтеграцією механізмів схемної ізоляції через django-tenants, що відповідає принципам data sovereignty, least privilege та zero-trust architecture.

```python
class User(AbstractUser, TenantSpecificModel):  new *
    """
    Кастомна модель користувача, що належить конкретному тенанту.
    """
    ROLE_CHOICES = [
        ('admin', 'Адміністратор'),
        ('manager', 'Менеджер'),
        ('member', 'Учасник'),
    ]

    email = models.EmailField(unique=True)
    role = models.CharField(max_length=20, choices=ROLE_CHOICES, default='member')
    is_active = models.BooleanField(default=True)
    date_joined = models.DateTimeField(auto_now_add=True)

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['username']

    def __str__(self):  new *
        return self.email
```

Рисунок 3.6. Приклад моделі користувача

Основна ідея полягає в тому, що кожен користувач зберігається в межах свого тенанта – тобто, у спільній базі даних, але з прив’язкою до конкретного клієнта. Таким чином, користувачі з різних організацій (тенантів) не можуть бачити чи використовувати облікові записи одне одного.

Модель містить кілька важливих полів. Поле email зроблено унікальним і використовується як основне поле для автентифікації (USERNAME_FIELD = ‘email’). Це означає, що користувач входить у систему за адресою електронної пошти, а не за стандартним username. Поле role визначає роль користувача в

системі – адміністратор, менеджер або учасник. Це дає змогу реалізувати гнучке керування доступом до функцій і даних. Поле is_active показує, чи активний обліковий запис, а date_joined автоматично зберігає дату реєстрації користувача.

Завдяки наслідуванню від TenantSpecificModel, усі записи цієї моделі мають приховане посилання на поточного тенанта. Це забезпечує логічну ізоляцію користувачів між різними схемами без необхідності дублювання таблиць. Тобто база залишається спільною, але кожен користувач пов'язаний лише зі «своїм» клієнтом.

JSON Web Token (JWT) є стандартизованим механізмом бездержавної автентифікації, що забезпечує масштабованість, безпеку та гнучкість у розподілених системах. У рамках даної роботи реалізовано та налаштовано (рис. 3.7) JWT-автентифікацію з використанням двофакторних токенів (access + refresh), HttpOnly cookies, чорного списку (blacklist) та ротації токенів [17].

```python
from datetime import timedelta

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ],
}

SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=60),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=7),
    'ROTATE_REFRESH_TOKENS': True,
    'BLACKLIST_AFTER_ROTATION': True,
    'UPDATE_LAST_LOGIN': True,

    'ALGORITHM': 'HS256',
    'VERIFYING_KEY': None,

    'AUTH_HEADER_TYPES': ('Bearer',),
    'AUTH_HEADER_NAME': 'HTTP_AUTHORIZATION',
    'USER_ID_FIELD': 'id',
    'USER_ID_CLAIM': 'user_id',

    'AUTH_TOKEN_CLASSES': ('rest_framework_simplejwt.tokens.AccessToken',),
    'TOKEN_TYPE_CLAIM': 'token_type',
    'JTI_CLAIM': 'jti',

    # Кастомні claims
    'TOKEN_OBTAIN_SERIALIZER': 'users.serializers.CustomTokenObtainPairSerializer',
}
```

Рисунок 3.7. Налаштування djangorestframework-simplejwt

Рольова модель контролю доступу (Role-Based Access Control, RBAC) є стандартизованим підходом до управління привілеями в інформаційних системах, що забезпечує гнучке, масштабоване та безпечне розмежування прав на основі ролей користувачів. У рамках даної роботи реалізовано розширений RBAC із

підтримкою ієрархії ролей, об'єктно-орієнтованого контролю та автоматичної ізоляції на рівні тенанта.

Ієрархія ролей:

1. Admin. усі операції в тенанті , керування користувачами, налаштуваннями, проєктами.
2. Manager. CRUD проєктів, задач, запрошення, керування командою
3. Member. читання та створення задач

Реалізація ролей у моделі користувача (рис. 3.8):

```python
from django.utils.translation import gettext_lazy as _
from django.contrib.auth.models import AbstractUser
from django.db import models


class User(AbstractUser, TenantSpecificModel):  new *
    ROLE_CHOICES = [
        ('admin', _('Адміністратор')),
        ('manager', _('Менеджер')),
        ('member', _('Учасник')),
    ]

    role = (models.CharField(
        max_length=20,
        choices=ROLE_CHOICES,
        default='member'))

    db_index = True

    def has_role(self, role_name):  new *
        """Перевірка ролі з урахуванням ієрархії."""
        hierarchy = {'admin': 3, 'manager': 2, 'member': 1}
        return hierarchy.get(self.role, 0) >= hierarchy.get(role_name, 0)
```

Рисунок 3.8. Приклад реалізації ролей у моделі User

Реалізація класів дозволів (рис. 3.9):

```python
class RolePermission(BaseTenantPermission):  3 usages  new *
    """
    Базовий клас для рольових перевірок.
    """
    required_role = None
    role_hierarchy = {'admin': 3, 'manager': 2, 'member': 1}

    def has_permission(self, request, view):  new *
        if not super().has_permission(request, view):
            return False
        user_role_level = self.role_hierarchy.get(request.user.role, 0)
        required_level = self.role_hierarchy.get(self.required_role, 0)
        return user_role_level >= required_level


class IsAdmin(RolePermission):  new *
    required_role = 'admin'


class IsManagerOrHigher(RolePermission):  new *
    required_role = 'manager'


class IsMemberOrHigher(RolePermission):  new *
    required_role = 'member'
```

Рисунок 3.9. Приклад реалізації класів дозволів

Наведений код реалізує механізм рольової авторизації користувачів у багатокористувацькому середовищі з підтримкою багатотенантності, що базується на спадкуванні від базового класу BaseTenantPermission.

Запропонована структура демонструє застосування принципів об’єктно-орієнтованого програмування, зокрема:

1. Спадкування – створення спеціалізованих класів на основі узагальненого базового.
2. Інкапсуляція логіки перевірки доступу в єдиній точці (базовому класі).
3. Поліморфізм – використання єдиного методу has_permission із різними умовами для різних типів доступу.

Код є прикладом реалізації рольової моделі контролю доступу (Role-Based Access Control, RBAC), адаптованої до багатотенантного середовища, що підвищує безпеку та керованість системи розподілу прав користувачів.

Бібліотека djangorestframework-simplejwt є спеціалізованим пакетом для фреймворку Django REST Framework (DRF), призначеним для реалізації аутентифікації користувачів за допомогою токенів формату JSON Web Token (JWT). Вона забезпечує сучасний, безпечний та масштабований механізм аутентифікації у веб-сервісах, побудованих за архітектурою REST (Representational State Transfer).

Основні компоненти бібліотеки:

1. Access Token (токен доступу) – короткочасний токен, що надає користувачеві доступ до ресурсів API. Має обмежений час життя (типово кілька хвилин).
2. Refresh Token (токен оновлення) – використовується для отримання нового токена доступу без необхідності повторної автентифікації. Забезпечує триваліший період дії (години або дні).
3. View-класи (TokenObtainPairView, TokenRefreshView, TokenVerifyView) – готові представлення для отримання, оновлення та перевірки токенів.

4. Система налаштувань (SIMPLE_JWT) – конфігураційний об’єкт, який дозволяє змінювати параметри безпеки, тривалість життя токенів, алгоритми підпису (наприклад, HS256, RS512) та інші аспекти аутентифікації.

Бібліотека djangorestframework-simplejwt реалізує концепцію автентифікації на основі токенів, що відповідає сучасним вимогам до розподілених систем, SaaS-платформ та мобільних застосунків.

Її архітектура поєднує принципи безпеки, продуктивності та автономності клієнта, що робить її ефективним інструментом для побудови RESTful API.

Бібліотека є реалізацією парадигми stateless authentication, яка мінімізує використання серверних ресурсів і забезпечує високий рівень контролю над життєвим циклом аутентифікаційних токенів.

Бібліотека django-auditlog – це модуль для фреймворку Django, який забезпечує автоматичне ведення журналу змін (аудит) у моделях бази даних. Вона реалізує механізм аудиту на рівні ORM (Object-Relational Mapping), фіксуючи створення, оновлення та видалення об’єктів у структурованій формі, що дозволяє відстежувати історію змін даних у системі.

Основною метою django-auditlog є забезпечення прозорості та трасованості операцій з даними, що є ключовою вимогою для інформаційних систем із підвищеними стандартами безпеки, таких як фінансові, медичні чи адміністративні платформи.

Бібліотека інтегрується безпосередньо в модель Django, не потребуючи змін у бізнес-логіці або додаткового коду для логування подій.

Бібліотека використовує сигнали Django (pre_save, post_save, pre_delete, post_delete), щоб автоматично фіксувати будь-які зміни у вибраних моделях.

Для цього достатньо зареєструвати модель у системі аудиту (рис. 3.9):

```
from auditlog.registry import auditlog
from django.db import models

class Client(models.Model):  new *
    name = models.CharField(max_length=255)
    email = models.EmailField()

auditlog.register(Client)
```

Рисунок 3.9. Приклад застосування django-auditlog

Після цього будь-яке створення, оновлення чи видалення запису в таблиці Client буде автоматично занесено до журналу AuditlogEntry, який містить такі ключові поля:

1. action – тип події (create, update, delete).
2. timestamp – час виконання операції.
3. actor – користувач, який ініціював зміну.
4. changes – серіалізоване представлення полів, що були змінені.
5. remote_addr – IP-адреса джерела запиту (якщо доступна).

Бібліотека django-auditlog є прикладом практичної реалізації концепції Data Provenance (походження даних), яка у науковій літературі розглядається як ключовий елемент інформаційної безпеки та цілісності даних.

Вона впроваджує принципи accountability (підзвітності) та traceability (трасованості) у програмних системах, побудованих на Django.

З точки зору інженерії програмного забезпечення, django-auditlog реалізує аспектно-орієнтований підхід (Aspect-Oriented Programming, AOP), де аудит розглядається як поперечний аспект, що не впливає на основну логіку додатка, але підвищує його надійність, безпечність і контрольованість [18].

РОЗДІЛ 4. ПРАКТИЧНА РЕАЛІЗАЦІЯ SAAS-ПЛАТФОРМИ

4.1. Актуальність та мета дослідження

У сучасних SaaS-системах комунікація між користувачами часто реалізується за допомогою традиційних HTTP-запитів, що не забезпечують миттєвого обміну повідомленнями. Це призводить до затримок у передачі даних, зниження інтерактивності та ефективності командної роботи.

Особливо гостро ця проблема проявляється у багатокористувацьких середовищах, де одночасно взаємодіє значна кількість клієнтів, а вимоги до масштабованості та безпеки залишаються високими.

Відсутність універсального рішення, яке поєднує реальний час (WebSockets), багатокористувацьку архітектуру (SaaS) та ізоляцію даних між орендарями (Tenants), ускладнює розробку таких систем.

Метою розробки є створення веб-сервера чату на основі Django, Django Tenants, Django Channels та DRF, який забезпечує:

1. Миттєвий обмін повідомленнями між користувачами через WebSockets.
2. Багатокористувацьку архітектуру SaaS із розмежуванням даних між орендарями.
3. Безпечну аутентифікацію користувачів за допомогою токенів (Simple JWT) та хешування паролів.
4. Гнучку модель чат-кімнат із можливістю створення приватних каналів доступу.
5. Автоматичне логування повідомлень для кожного чатруму.
6. Динамічне відображення станів користувачів (вхід/вихід, активність) у реальному часі.

Отримати прототип SaaS-платформи, який може бути основою для корпоративних систем спілкування, внутрішніх CRM-чатів або навчальних платформ з інтерактивною взаємодією користувачів.

4.2. Налаштування проєкту та бази даних

Для забезпечення відтворюваності та ізоляції залежностей проєкт розробляється з використанням віртуального середовища Python. Налаштування виконується у наступній послідовності (рис. 4.1):

```
# Створення віртуального середовища
python -m venv venv
source venv/bin/activate  # Linux/macOS
# або
venv\Scripts\activate     # Windows

# Встановлення залежностей
pip install --upgrade pip
pip install django==5.0 \
            djangorestframework \
            django-channels[websockets] \
            django-tenants \
            daphne \
            psycopg2-binary \
            channels-redis \
            python-jose[cryptography] \
            passlib[bcrypt]
```

Рисунок 4.1. Налаштування віртуального середовища Python

У файлі settings.py реалізовано підтримку багатотенантної архітектури (рис. 4.2):

```
from pathlib import Path

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

# Багатотенантність
PUBLIC_SCHEMA_URLCONF = 'saas_platform.urls_public'
SHARED_APPS = [
    'django_tenants',
    'tenants',
    'django.contrib.contenttypes',
    'django.contrib.auth',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.admin',
]

TENANT_APPS = [
    'django.contrib.contenttypes',
    'chat',
    'rest_framework',
    'rest_framework_simplejwt',
]

INSTALLED_APPS = SHARED_APPS + TENANT_APPS

TENANT_MODEL = "tenants.Client"
TENANT_DOMAIN_MODEL = "tenants.Domain"

# Маршрутизація бази даних
DATABASE_ROUTERS = ['django_tenants.routers.TenantSyncRouter']

# Налаштування бази даних
DATABASES = {
    'default': {
        'ENGINE': 'django_tenants.postgresql_backend',
        'NAME': 'saas_chat_db',
        'USER': 'saas_user',
        'PASSWORD': 'secure_password_2025',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Рисунок 4.2. Налаштування settings.py

У дослідницьких цілях використано PostgreSQL замість SQLite для забезпечення коректної роботи django-tenants з схемами. SQLite підтримується лише в режимі розробки з обмеженнями.

Також потрібно налаштувати інтерфейс між веб-сервером та Python-додатками ASGI (рис. 4.3). Використання ASGI дозволяє досягти O(1) масштабування за кількістю одночасних з'єднань при фіксованій кількості потоків, на відміну від WSGI, де масштабування пропорційне кількості потоків/процесів (O(n)). Це критично важливо для SaaS-платформ з високим рівнем паралелізму, де тисячі користувачів можуть одночасно підтримувати WebSocket-з'єднання.

```
from django.core.asgi import get_asgi_application
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.auth import AuthMiddlewareStack
from django_tenants.middleware import TenantMainMiddleware

os.environ.setdefault( key: 'DJANGO_SETTINGS_MODULE',  value: 'saas_platform.settings')

application = ProtocolTypeRouter({
    'http': get_asgi_application(),
    'websocket': TenantMainMiddleware(
        AuthMiddlewareStack(
            URLRouter(
                chat.routing.websocket_urlpatterns
            )
        )
    ),
})
```

Рисунок 4.3. ASGI-конфігурація для Django Channels

Фреймворк Django Channels розширює Django, додаючи підтримку ASGI, зберігаючи при цьому сумісність із синхронним кодом через інтеграцію з asyncio. Channels вводить концепцію канальних шарів (channel layers), які забезпечують:

1. Асинхронну маршрутизацію повідомлень між WebSocket-клієнтами.
2. Групову розсилку (group send) для реалізації чат-кімнат, сповіщень, онлайн-статусів.
3. Інтеграцію з Django ORM через database_sync_to_async.
4. Підтримку фонових завдань (background tasks).

У контексті даного SaaS проєкту, Django Channels є єдиним механізмом, що забезпечує реал-тайм взаємодію у межах ізольованих тенантів.

Міграції та створення публічної схеми:

1. Створення міграцій: python manage.py makemigrations.
2. Застосування міграцій до публічної схеми: python manage.py migrate_schemas –shared.
3. Створення тенанта: python manage.py create_tenant.

4.3. Налаштування фронтенду та додавання компонентів React

Даний розділ присвячено практичній реалізації клієнтської частини системи з використанням сучасного стеку технологій: Vite як інструменту збірки, TypeScript для статичної типізації та React з бібліотекою компонентів Chakra UI для створення адаптивного, доступного та естетично привабливого інтерфейсу.

Початкові команди для створення фронтенд інтерфейсу (рис. 4.4)

```
npm create vite@latest frontend -- --template react-ts
cd frontend
npm install @chakra-ui/react @emotion/react @emotion/styled framer-motion axios
```

Рисунок 4.4. Команди для додавання фронтенду

Науково-технічне обґрунтування вибору стеку:

1. Vite. Забезпечує миттєвий HMR (Hot Module Replacement), швидку збірку за допомогою ESBuild та підтримку сучасних можливостей ECMAScript без транспіляції.
2. TypeScript. Гарантує статичну типізацію моделей даних (повідомлення, користувачі, чат-кімнати), зменшує кількість помилок на етапі виконання.
3. Chakra UI. Реалізує Design System з вбудованою підтримкою тем, доступності , адаптивності компонентів з коробки (Input, Modal, Toast тощо).

Компонент LoginForm (рис. 4.5) виконує перший етап аутентифікації – перевірку існування імені користувача (username) у базі даних. Це відповідає вимогам системи: на вхідній сторінці юзер задає свій юзернейм. Якщо такого немає

в базі даних, то запропонувати користувачеві задати пароль і зберегти його в базі. Якщо є – запросити пароль. Такий двоетапний підхід (username → password) є обґрунтованим з точки зору:

1. Безпеки: уникає витоку інформації про існування облікового запису при неправильному паролі.
2. UX: зменшує когнітивне навантаження – користувач вводить лише один поле за раз.
3. Гнучкості: дозволяє динамічно перемикатися між реєстрацією та входом.

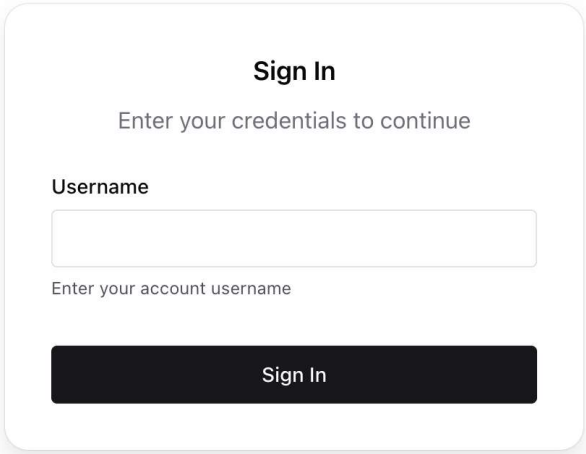

Рисунок 4.5. Компонент входу LoginForm

Технічна реалізація (рис. 4.6):

1. useForm<LoginData>. Статична типізація форми через generic-інтерфейс LoginData забезпечує компільовану безпеку типів та автодоповнення в IDE.

2. yupResolver(schema). Декларативна валідація на основі JSON-схеми. Дозволяє формалізувати правила (наприклад, username: string().required().min(3)) та повторно використовувати їх при валідації.
3. useUsernameStore(). Використання Zustand як менеджеру станів замість Redux – зменшує складність та підвищує продуктивність.
4. isSubmitting + <Spinner />. Візуальна індикація асинхронної операції запобігає повторним відправкам.

```
export const LoginForm = () => {
  const {
    register,
    handleSubmit,
    formState: { errors, isSubmitting },
  } = useForm<LoginData>({
    resolver: yupResolver(schema),
  });
  const { checkUserName } = useUsernameStore();

  const onSubmit = handleSubmit((data) => checkUserName(data.username));

  return (
    <form onSubmit={onSubmit}>
      <VStack gap={6} align="stretch">
        <Box textAlign="center">
          <Heading size="lg" mb={2}>
            Sign In
          </Heading>
          <Text color="gray.500">Enter your credentials to continue</Text>
        </Box>
        <Field.Root invalid={!!errors.username}>
          <Field.Label>Username</Field.Label>
          <Input {...register("username")} />
          {errors.username ? (
            <Field.ErrorText>{errors.username.message}</Field.ErrorText>
          ) : (
            <Field.HelperText>Enter your account username</Field.HelperText>
          )}
        </Field.Root>
        <Button type="submit" colorScheme="blue" size="md" mt={2}>
          {isSubmitting ? <Spinner size="sm" /> : "Sign In"}
        </Button>
      </VStack>
    </form>
  );
};
```

Рисунок 4.6. Технічна реалізація компоненту LoginForm

Коли користувач вводить ім’я (username) у форму входу, фронтенд не одразу надсилає його на сервер. Замість цього він використовує спеціальний менеджер стану – useUsernameStore, побудований на бібліотеці Zustand, і API-шар – AccountsAPI (зв’язок із сервером). Ось як це працює покроково.

Форма (LoginForm) використовує react-hook-form. Після натискання кнопки викликається checkUserName(data.username) (рис. 4.7)

Цей store зберігає:

1. username – поточне ім’я.
2. exists – чи є такий користувач (true / false / undefined).
3. error – помилка, якщо щось пішло не так.

Він один на всю програму – тому всі компоненти можуть читати актуальний стан.

```
async checkUserName(username) {
  try {
    set({ error: null });

    const response = await AccountsAPI.checkUserName(username);
    set({ exists: response.exists, error: null, username });
  } catch (err: unknown) {
    let message: string | Record<string, string[]> | null = null;

    if (err instanceof AxiosError) {
      const data = err.response?.data;

      if (!data) {
        message = err.message;
      } else if ("detail" in data) {
        // DRF usually returns this for top-level errors
        message = data.detail;
      } else {
        // Field-specific validation errors
        message = data as Record<string, string[]>;
      }
    } else if (err instanceof Error) {
      message = err.message;
    } else {
      message = "Something went wrong. Please try again.";
    }

    // Always show toast for unexpected or top-level errors
    if (typeof message === "string" && message) {
      toaster.create({
        type: "error",
        title: "Error",
        description: message,
        closable: true,
      });
    }

    set({ error: message });
  }
```

Рисунок 4.7. Технічна реалізація checkUserName

Фронтенд не напряму говорить із сервером. Він використовує:

1. Zustand – щоб пам’ятати, що відбувається.
2. AccountsAPI – щоб надсилати запити.

Після того як користувач вводить свій юзернейм та пароль на вхідній сторінці чат-платформи (рис. 4.8):

1. Система перевіряє наявність користувача в базі даних (через ORM Django.
2. Якщо користувача немає, пропонується створити акаунт із введенням пароля, який зберігається у вигляді хеша.
3. Якщо користувач існує, система порівнює введений пароль з хешем у базі даних.

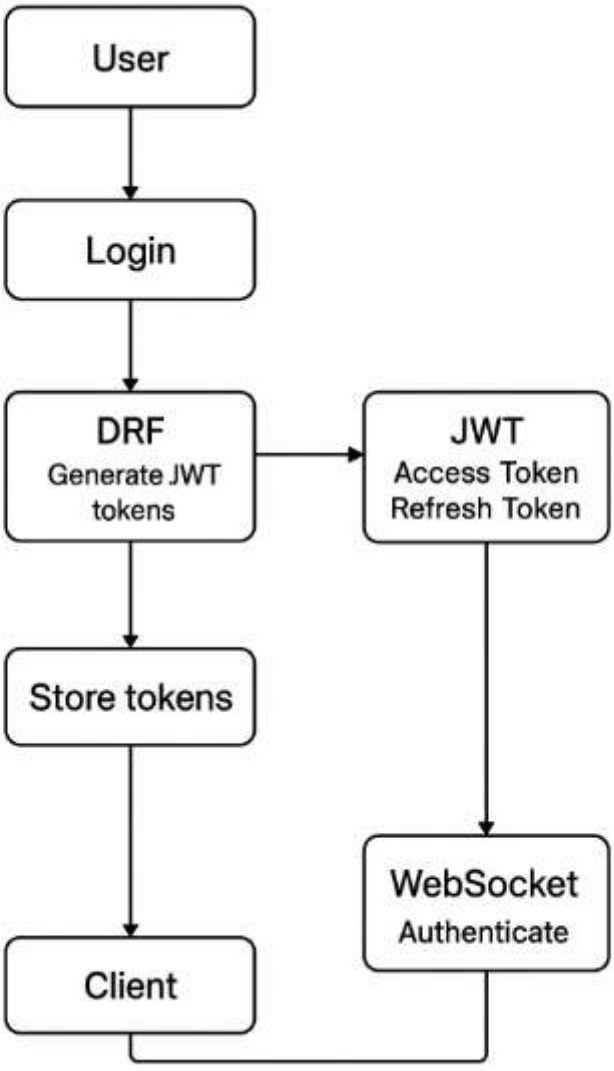

Рисунок 4.8. Схема аутентифікації

Після успішної аутентифікації виконується створення JSON Web Token (JWT) за допомогою django-simple-jwt:

1. Access Token – короткостроковий токен, який використовується для авторизації запитів до WebSocket-сервера та REST API.
2. Refresh Token – довгостроковий токен, який дозволяє отримувати новий Access Token без повторного входу користувача.

Для забезпечення сесійної взаємодії користувача з чатом токени зберігаються на клієнті:

1. Access Token – в пам’яті браузера або у secure cookie для авторизації WebSocket-підключень.
2. Refresh Token – у secure cookie або локальному сховищі (LocalStorage/SessionStorage), використовується для оновлення Access Token при його закінченні терміну дії.

Після збереження токенів клієнт ініціює підключення до WebSocket-сервера:

1. Access Token передається як параметр авторизації у заголовку або URL під час підключення.
2. Сервер, отримавши токен, перевіряє його валідність через django-simple-jwt.
3. У разі успішної перевірки користувач підключається до чат-руму, отримує унікальний колір нікнейму та статус “онлайн”.

Access Token має короткий термін дії, тому клієнт періодично виконує запит на /api/token/refresh/ з Refresh Token, щоб отримати новий Access Token.

Сервер обробляє оновлення токенів, забезпечуючи безперервну авторизацію та безпеку сесії користувача.

Чат-кімната – це серце реального часу у нашій SaaS-платформі. Користувач бачить повідомлення, статус підключення інших, може писати, бачити свій колір нікнейму та отримувати сповіщення про входи/виходи.

Мета компонента (рис. 4.9):

1. Підключення до WebSocket за room_id і JWT-токеном.
2. Отримання та відображення повідомлень у реальному часі.

3. Надсилання повідомлень.
4. Системні сповіщення.
5. Візуальне виділення користувачів кольором.

```tsx
export const ChatRoom = ({ roomId, token }: { roomId: string; token: string }) => {
  const [input, setInput] = useState('');
  const { messages, sendMessage, isConnected, messagesEndRef } = useChatWebSocket({
    roomId,
    token,
  });

  const handleSend = () => {
    if (input.trim() && isConnected) {
      sendMessage(input.trim());
      setInput('');
    }
  };

  return (
    <VStack h="100vh" p={4} spacing={4}>
      <Flex w="100%" justify="space-between" align="center">
        <Text fontWeight="bold">Room #{roomId}</Text>
        <Flex align="center" gap={2}>
          {isConnected ? (
            <Box w={2} h={2} bg="green.500" borderRadius="full" />
          ) : (
            <Box w={2} h={2} bg="red.500" borderRadius="full" />
          )}
          <Text fontSize="sm" color={isConnected ? 'green.600' : 'red.600'}>
            {isConnected ? 'Online' : 'Offline'}
          </Text>
        </Flex>
      </Flex>

      <Box
        flex={1}
        w="100%"
        overflowY="auto"
        p={4}
        bg="gray.50"
        borderRadius="md"
        border="1px solid"
        borderColor="gray.200"
      >
        {messages.length === 0 ? (
          <Text color="gray.500" textAlign="center">
            Start Conversation...
          </Text>
        ) : (
          messages.map((msg, i) => (msg.system ? <MessageItem key={i} msg={msg} /> : <MessageItem key={i} msg={msg} />))
        )}
        <div ref={messagesEndRef} />
      </Box>

      <Flex w="100%" gap={2}>
        <Input
          value={input}
          onChange={(e) => setInput(e.target.value)}
          onKeyPress={(e) => e.key === 'Enter' && handleSend()}
          placeholder="Write a message..."
          isDisabled={!isConnected}
        />
        <Button onClick={handleSend} colorScheme="blue" isDisabled={!isConnected || !input.trim()}>
          {isConnected ? 'Send' : <Spinner size="sm" />}
        </Button>
      </Flex>
```

Рисунок 4.9. Технічна реалізація ChatRoom

Особливості реалізації:

1. Колір нікнейму: передається з бекенду в WebSocket-повідомленні.
2. Системні повідомлення: system: true, італік, сірий колір.
3. Автопрокрутка: scrollIntoView({ behavior: "smooth" }).
4. Статус підключення: зелена/червона крапка.
5. Відключення при помилці: onclose, onerror.

Компонент ChatRoom – це повноцінна реал-тайм чат-кімната, яка:

1. Працює миттєво завдяки WebSocket.
2. Безпечна – через JWT у query string.
3. Зручна – автопрокрутка, статус, кольори.
4. Масштабована – легко розширити додати файли, реакції тощо.

4.4. Багатотенантна архітектура SaaS-платформи чату

Реалізується на основі бібліотеки django-tenants, яка забезпечує ізоляцію даних шляхом використання окремих схем бази даних для кожного клієнта. У публічній схемі зберігаються лише метадані про тенантів – назви організацій, доменні імена, дати оплати та статус пробного періоду. Усі прикладні сутності, такі як користувачі, чат-кімнати, повідомлення та журнали активності, розміщуються виключно в схемах відповідних тенантів. Така організація гарантує повну інформаційну ізоляцію: жоден клієнт не має доступу до даних іншого, навіть на рівні бази даних.

Автоматичне визначення тенанта здійснюється на етапі обробки HTTP- та WebSocket-запитів за допомогою спеціалізованого проміжного шару TenantMainMiddleware. При надходженні запиту система аналізує доменне ім’я (наприклад, acme.chat.local) і зіставляє його з відповідною схемою бази даних. Цей механізм діє як на рівні REST API, так і в реальному часі – при підключенні до WebSocket. Завдяки інтеграції з ASGI-сервером Daphne, перемикання контексту схеми відбувається до виконання будь-якої логіки, що забезпечує коректну роботу ORM та канальних шарів у межах правильного тенанта.

Створення нового тенанта є централізованою адміністративною операцією, що включає реєстрацію об’єкта клієнта в публічній схемі, створення відповідного домену та автоматичну ініціалізацію нової схеми бази даних з застосуванням усіх необхідних міграцій. Після цього тенант стає повністю автономним: у його схемі розгортаються таблиці користувачів, чат-кімнат і повідомлень, а всі подальші запити автоматично спрямовуються в ізольоване середовище. Такий підхід

дозволяє масштабувати платформу лінійно – додавання нового клієнта не впливає на продуктивність існуючих.

Інтеграція з клієнтською частиною враховує архітектурні особливості багатотенантності. У режимі розробки використовується проксі-сервер Vite, який перенаправляє запити на відповідний субдомен, тоді як у продакшені цю функцію виконує Nginx, що забезпечує коректну передачу заголовка Host і підтримку WebSocket-з'єднань. Таким чином, фронтенд залишається універсальним, а логіка ізоляції повністю інкапсулюється на стороні бекенду.

Безпека та контроль доступу посилюються завдяки тому, що всі операції з даними – від аутентифікації до збереження повідомлень у лог-файлах – виконуються в контексті конкретної схеми.

Адміністративна панель, доступна через публічну схему, дозволяє керувати підписками, переглядати статистику та блокувати тенанти за необхідності, зберігаючи при цьому повну ізоляцію операційних даних.

На архітектурному рівні django-channels інтегрується через ASGI-додаток, де ProtocolTypeRouter розподіляє вхідні з'єднання: HTTP-запити спрямовуються до стандартного Django, тоді як WebSocket-з'єднання проходять через стек проміжних шарів, включаючи TenantMainMiddleware та AuthMiddlewareStack. Це дозволяє ізолювати реал-тайм трафік на рівні тенанта – кожен клієнт підключається до своєї схеми бази даних, і всі операції (збереження повідомлень, перевірка доступу) виконуються в правильному контексті. Така інтеграція гарантує, що навіть при тисячах одночасних з'єднань система зберігає коректність даних і безпеки.

Ключовим елементом є асинхронний консьюмер ChatConsumer, який відповідає за обробку WebSocket-з'єднань у конкретній чат-кімнаті. При підключенні користувача консьюмер виконує перевірку доступу через has_access(), присвоює унікальний колір нікнейму на поточну сесію, додає користувача до групового каналу (chat_<room_id>) та надсилає системне сповіщення про приєднання. При відправці повідомлення воно зберігається в базі даних через database_sync_to_async, записується в окремий лог-файл для кімнати, а потім розсилається всім учасникам групи за допомогою group_send. Аналогічно

обробляється вихід користувача. Такий підхід забезпечує O(1) масштабування за кількістю клієнтів у кімнаті завдяки використанню Redis як бекенду канального шару.

Канальний шар на основі Redis виступає як високопродуктивний брокер повідомлень, що дозволяє координувати взаємодію між різними інстансами сервера у разі горизонтального масштабування. Повідомлення, надіслані одним клієнтом, потрапляють у Redis, звідки миттєво розсилаються всім іншим учасникам тієї ж групи. Використання channels-redis також підтримує механізми presence – відстеження онлайн-статусу, що відображається у списку кімнат.

Логування повідомлень у файли реалізовано асинхронно в межах консьюмера: після збереження в базі даних повідомлення записується у файл виду logs/room_{id}.log з міткою часу та нікнеймом. Це відповідає вимогам аудиту та дозволяє аналізувати історію комунікації без навантаження на основну базу даних. Оскільки лог-файли створюються в межах тенанта (шлях залежить від контексту схеми), зберігається повна ізоляція між клієнтами.

Отже, django-channels у цьому проєкті є інфраструктурним ядром нашого чату, що забезпечує:

1. Асинхронну, ізольовану та безпечну взаємодію у багатотенантному середовищі.
2. Масштабованість до десятків тисяч одночасних з’єднань.
3. Повну інтеграцію з Django ORM, JWT-аутентифікацією та файловим логуванням.

Отже, реалізована багатотенантна архітектура відповідає сучасним вимогам до SaaS-платформ: забезпечує масштабованість, безпеку, гнучке адміністрування та прозору інтеграцію з реальним часом, роблячи систему готовою до комерційного розгортання з підтримкою тисяч незалежних клієнтів.

ВИСНОВКИ

В першому розділі кваліфікаційної роботи проаналізовано теоретичні основи SaaS-систем. Розглянуто поняття SaaS та його еволюцію від простих веб-додатків до складних хмарних платформ із підтримкою тисяч клієнтів. Охарактеризовано особливості багатокористувацьких систем, зокрема вимоги до ізоляції даних, безпеки, продуктивності та персоналізації.

В другому розділі досліджено та проаналізовано методологію розробки SaaS-архітектури. Запропоновано ітеративно-інкрементальну методологію на основі принципів Agile та Domain-Driven Design, що забезпечує гнучке проєктування, швидке прототипування та контрольовану інтеграцію змін. Обґрунтовано вибір технологічного стеку: Django, PostgreSQL (для бекенду) і Vite, TypeScript, React (для фронтенду) з урахуванням вимог до масштабованості, безпеки та продуктивності.

У третьому розділі досліджено та розроблено архітектуру багатокористувацької SaaS-платформи з акцентом на ізоляцію даних, безпеку та ефективну взаємодію між клієнтом і сервером. Спроєктовано модель даних з використанням Django ORM та багатотенантної архітектури на основі django-tenants. Реалізовано логіку ізоляції користувачів (тенантів) через окремі схеми бази даних PostgreSQL, де кожна організація має власне ізольоване середовище.

У четвертому розділі розроблено та реалізовано SaaS-платформу приватного чату з підтримкою багатотенантної архітектури на основі фреймворку Django. Здійснено повну інтеграцію бібліотек django-channels, django-tenants, djangorestframework та django-simple-jwt, що забезпечило функціонування асинхронних WebSocket-з’єднань, ізоляцію даних між клієнтами, аутентифікацію та авторизацію. Реалізовано зберігання повідомлень у базі даних та логування у файли з прив’язкою до конкретної чат-кімнати.

Розроблено клієнтську частину на основі Vite, TypeScript та React 18 з використанням бібліотеки компонентів Chakra UI. Створено модульну архітектуру з розділенням логіки, компоненти та API-шар. Реалізовано форму входу з

двоетапною перевіркою, а також повнофункціональний чат з підтримкою реального часу, автопрокруткою, кольоровим виділенням нікнеймів та системними сповіщеннями.

Інтегровано багатотенантність за допомогою django-tenants з використанням PostgreSQL-схем. Кожен клієнт (тенант) має ізольовану схему бази даних, автоматично визначається за субдоменом через TenantMainMiddleware. Забезпечено коректну роботу WebSocket у межах конкретного тенанта завдяки інтеграції з ASGI-стеком та channels. Налаштовано маршрутизацію, міграції, створення тенантів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Що таке SaaS? | Oracle Україна. URL: https://www.oracle.com/ua/applications/what-is-saas/ (дата звернення: 22.09.2025).
2. Software as a service - Wikipedia. URL: https://en.wikipedia.org/wiki/Software_as_a_service (дата звернення: 24.09.2025).
3. Що таке SaaS. URL: https://wedex.com.ua/blog/shho-take-saas-programne-zabezpechennya-yak-posluga/#anchor-%d0%bf%d0%b5%d1%80%d0%b5%d0%b2%d0%b0%d0%b3%d0%b8-%d1%82%d0%b0-%d0%b2%d0%b8%d0%b3%d0%be%d0%b4%d0%b8-saas (дата звернення: 24.09.2025).
4. What is Single-Tenancy? How’s it Different from Multi-Tenancy. URL: https://www.techtarget.com/searchcloudcomputing/definition/single-tenancy (дата звернення: 02.10.2025).
5. Complete Guide to Multi-Tenant Architecture. URL: https://medium.com/@seetharamugn/complete-guide-to-multi-tenant-architecture-d69b24b518d6 (дата звернення: 06.10.2025).
6. Які тенденції у галузі SaaS? Огляд ринку на 2024-2030 роки. URL: https://payproglobal.com/uk/%D0%B2%D1%96%D0%B4%D0%BF%D0%BE%D0%B2%D1%96%D0%B4%D1%96/%D1%89%D0%BE-%D1%82%D0%B0%D0%BA%D0%B5-%D1%82%D0%B5%D0%BD%D0%B4%D0%B5%D0%BD%D1%86%D1%96%D1%97-%D1%82%D0%B0-%D1%96%D0%BD%D0%BD%D0%BE%D0%B2%D0%B0%D1%86%D1%96%D1%97-saas/ (дата звернення: 06.10.2025).
7. Welcome to django-tenants documentation. URL: https://django-tenants.readthedocs.io/en/latest/ (дата звернення: 15.10.2025).
8. Evolving django-multitenant to build scalable SaaS apps on Postgres. URL: https://www.citusdata.com/blog/2023/05/09/evolving-django-multitenant-to-build-scalable-saas-apps-on-postgres-and-citus/ (дата звернення: 08.10.2025).

9. Building a Multi-Tenant SaaS Platform with React + Django. URL: https://www.linkedin.com/posts/ashishbhosale2748_saas-multitenantarchitecture-django-activity-7341839778587267072-1Cpi (дата звернення: 08.10.2025).

10. Tests – django_tenants dev documentation. URL: https://django-tenants.readthedocs.io/en/latest/test.html (дата звернення: 10.10.2025).

11. How to Use Cypress for End-to-End Testing Your React Apps. URL: https://www.freecodecamp.org/news/cypress-for-end-to-end-testing-react-apps/ (дата звернення: 11.10.2025).

12. Django Security - OWASP Cheat Sheet Series. URL: https://cheatsheetseries.owasp.org/cheatsheets/Django_Security_Cheat_Sheet.html (дата звернення: 12.10.2025).

13. Вимоги GDPR та стандарт ISO 27001. URL: https://www.sim-networks.com/ukr/blog/gdpr-requirements-and-iso-27001-standard (дата звернення: 13.10.2025).

14. Using django-tenants – django_tenants dev documentation. URL: https://django-tenants.readthedocs.io/en/latest/use.html (дата звернення: 14.10.2025).

15. Serializers - Django REST framework. URL: https://www.django-rest-framework.org/api-guide/serializers/ (дата звернення: 15.10.2025).

16. Viewsets - Django REST framework. URL: https://www.django-rest-framework.org/api-guide/viewsets/ (дата звернення: 15.10.2025).

17. Getting started – Simple JWT. URL: https://django-rest-framework-simplejwt.readthedocs.io/en/latest/getting_started.html (дата звернення: 17.10.2025).

18. Аспектно-орієнтоване програмування. URL: https://uk.wikipedia.org/wiki/%D0%90%D1%81%D0%BF%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D1%96%D1%94%D0%BD%D1%82%D0%BE%D0%B2%D0%B0%D0%BD%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F (дата звернення: 20.10.2025).