

ПРИВАТНЕ АКЦІОНЕРНЕ ТОВАРИСТВО «ВИЩИЙ НАВЧАЛЬНИЙ
ЗАКЛАД

«МІЖРЕГІОНАЛЬНА АКАДЕМІЯ УПРАВЛІННЯ ПЕРСОНАЛОМ»»

Факультет комп'ютерно-інформаційних технологій

Кафедра комп'ютерних інформаційних систем і технологій

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему:

«Аналіз моделей рольового розмежування доступу (RBAC, ABAC, ReBAC) у
контексті сучасних веб-систем»

Здобувача вищої освіти

4 курсу ІК-9-22-БІКНТ(4.0Д)

денної форми здобуття освіти

спеціальності 122 Комп'ютерні науки

Подліняєва Олександра Олександровича

Науковий керівник

Кандидат технічних наук, доцент

Знаковська Євгенія Анатоліївна

Оцінка

За університетською шкалою _____

За 100-бальною шкалою _____

За шкалою ЄКТС _____

Допущено до захисту

Завідувач кафедри _____ д.е.н., професор Кавун С.В.

Київ – 2026 р.

ЗМІСТ

ВСТУП	6
РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ УПРАВЛІННЯ ДОСТУПОМ У ВЕБ-СИСТЕМАХ	10
1.1 Поняття автентифікації та авторизації — різниця і взаємозв'язок	10
1.2 Класифікація моделей управління доступом	12
1.3 Огляд сучасних веб-систем та їх вимоги до управління доступом	14
1.4 Еволюція підходів до авторизації у веб-розробці	17
РОЗДІЛ 2. АНАЛІЗ МОДЕЛЕЙ RBAC, ABAC, REBAC	21
2.1 Модель RBAC (Role-Based Access Control)	21
2.2 Модель ABAC (Attribute-Based Access Control)	24
2.3 Модель ReBAC (Relationship-Based Access Control)	27
2.4 Гібридні підходи — комбінування моделей	30
РОЗДІЛ 3. ПОРІВНЯЛЬНИЙ АНАЛІЗ МОДЕЛЕЙ	34
3.1 Критерії порівняння моделей управління доступом	34
3.2 Порівняльна характеристика RBAC, ABAC та ReBAC	36
3.3 Сценарії застосування залежно від типу веб-системи	40
3.4 Рекомендації щодо вибору моделі	43
РОЗДІЛ 4. ПРАКТИЧНА РЕАЛІЗАЦІЯ СИСТЕМИ АВТОРИЗАЦІЇ НА ОСНОВІ МОДЕЛІ RBAC	48
4.1 Вибір технологічного стеку та обґрунтування	48
4.2 Обґрунтування вибору Django як технологічної платформи	52
4.3 Вибір моделі та проектування системи авторизації на її основі	56
4.4 Реалізація RBAC засобами Django Groups та Permissions	62
4.5 Тестування та оцінка продуктивності	72
ВИСНОВОК	78
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	81
ДОДАТКИ	86

РЕФЕРАТ

Тема: Дослідження моделей рольового розмежування доступу (RBAC, ABAC, ReBAC) у контексті сучасних веб-систем.

Структура і обсяг роботи: бакалаврська робота викладена на 104 сторінках, складається з реферату, вступу, 4 розділів, 18 підрозділів, висновків, списку використаних джерел що містить 15 найменувань та додатків з кодом розробленого прототипу.

Об'єктом дослідження є процеси управління доступом у сучасних веб-системах.

Предметом дослідження є моделі рольового розмежування доступу RBAC, ABAC та ReBAC та їх характеристики у контексті вимог сучасних веб-застосунків.

Метою роботи є проведення порівняльного дослідження моделей управління доступом RBAC, ABAC та ReBAC у контексті вимог сучасних веб-систем та реалізація демонстраційного прототипу системи авторизації на основі обраної моделі.

Методи дослідження базуються на використанні методів системного аналізу для дослідження структури та принципів моделей управління доступом, порівняльного аналізу для оцінки характеристик моделей за визначеними критеріями, сценарного моделювання для визначення відповідності моделей конкретним типам веб-систем, а також методу прототипування для практичної реалізації та верифікації теоретичних висновків. Практична реалізація прототипу здійснена засобами фреймворку Django 5.0 та мови програмування Python 3.12 з використанням SQLite як реляційного сховища даних та вбудованого механізму Django Groups та Permissions для реалізації рольової моделі доступу.

Результатом роботи є проведений порівняльний аналіз моделей RBAC, ABAC та ReBAC за сімома критеріями — гнучкістю, масштабованістю

адміністрування, складністю реалізації, складністю адміністрування, продуктивністю перевірки, стійкістю до помилок конфігурації та підтримкою у фреймворках — та сформульовані практичні рекомендації щодо вибору моделі залежно від типу веб-системи. Реалізовано демонстраційний прототип системи авторизації медичного центру МРТ діагностики на основі моделі RBAC що включає три ролі — Адміністратор, Співробітник та Клієнт — реалізовані через Django Groups, декларативний механізм захисту view-функцій через декоратори, захист від вразливості IDOR на рівні ORM-запитів та журнал аудиту що фіксує всі події доступу. Функціональне тестування охопило 71 перевірку у п'яти тестових сценаріях — всі перевірки пройдено успішно.

Ключові слова: управління доступом, авторизація, RBAC, ABAC, ReBAC, Django Groups, веб-безпека, журнал аудиту, медична інформаційна система, Google Zanzibar.

ABSTRACT

ACCESS CONTROL, AUTHORIZATION, RBAC, ABAC, ReBAC, DJANGO GROUPS, WEB SECURITY, AUDIT LOG, MEDICAL INFORMATION SYSTEM, GOOGLE ZANZIBAR.

The object of the research is the processes of access control in modern web systems.

The subject of the research is the role-based access control models RBAC, ABAC and ReBAC and their characteristics in the context of modern web application requirements.

The aim of the work is to conduct a comparative study of access control models RBAC, ABAC and ReBAC in the context of modern web system

requirements and to implement a demonstration prototype of an authorization system based on the selected model.

The research methods are based on systematic analysis for studying the structure and principles of access control models, comparative analysis for evaluating model characteristics according to defined criteria, scenario modeling for determining model applicability to specific types of web systems, and prototyping for practical implementation and verification of theoretical conclusions. The practical implementation of the prototype was carried out using the Django 5.0 framework and Python 3.12 programming language with SQLite as a relational data store and the built-in Django Groups and Permissions mechanism for implementing the role-based access model.

The result of the work is a comparative analysis of RBAC, ABAC and ReBAC models across seven criteria — flexibility, administration scalability, implementation complexity, administration complexity, verification performance, resistance to configuration errors and framework support — along with practical recommendations for model selection depending on the type of web system. A demonstration prototype of an authorization system for an MRI diagnostic medical center based on the RBAC model was implemented, including three roles — Administrator, Staff and Client — realized through Django Groups, a declarative view protection mechanism through decorators, protection against IDOR vulnerability at the ORM query level, and an audit log that records all access events. Functional testing covered 71 checks across five test scenarios — all checks passed successfully.

ВСТУП

Забезпечення безпеки інформаційних ресурсів є однією з фундаментальних задач проектування сучасних веб-систем. Зростання кількості користувачів, ускладнення організаційних структур та підвищення вимог регуляторного середовища до захисту персональних даних зумовлюють необхідність застосування формально визначених та архітектурно обґрунтованих моделей управління доступом. Некоректна реалізація механізмів авторизації є однією з найпоширеніших причин інцидентів інформаційної безпеки — клас вразливостей «Broken Access Control» стабільно посідає першу позицію у рейтингу OWASP Top 10 що підтверджує актуальність даного дослідження.[1]

У науковій літературі та інженерній практиці сформувалися три основні моделі рольового розмежування доступу що знаходять широке застосування у веб-системах: рольова модель RBAC (Role-Based Access Control), атрибутивна модель ABAC (Attribute-Based Access Control) та реляційна модель ReBAC (Relationship-Based Access Control). Кожна з цих моделей відображає певний рівень розвитку вимог до систем авторизації та має власний профіль переваг і обмежень. Попри значний обсяг теоретичних досліджень кожної моделі окремо порівняльний аналіз усіх трьох моделей у контексті конкретних вимог сучасних веб-систем залишається актуальним науково-практичним завданням.[2]

Актуальність теми обумовлена кількома чинниками. По-перше, стрімке поширення SaaS-платформ, систем для спільної роботи та медичних інформаційних систем породжує нові вимоги до гнучкості та масштабованості авторизації що не завжди можуть бути задоволені традиційними підходами. По-друге, відсутність у розробників чітких критеріїв вибору між моделями призводить до архітектурних рішень що не відповідають вимогам системи — надмірно складних або навпаки недостатньо гнучких. По-третє, поява у 2019 році системи Google Zanzibar та екосистеми рішень на її основі суттєво

змінила практичний ландшафт реляційної авторизації що потребує систематичного дослідження та порівняння з усталеними моделями.[2]

Мета роботи — провести порівняльне дослідження моделей управління доступом RBAC, ABAC та ReBAC у контексті вимог сучасних веб-систем та реалізувати демонстраційний прототип системи авторизації на основі обраної моделі.

Задачі дослідження:

1. дослідити теоретичні основи управління доступом та класифікацію існуючих моделей;
2. провести детальний аналіз принципів, переваг та обмежень моделей RBAC, ABAC та ReBAC;
3. визначити критерії порівняння моделей та провести їх порівняльний аналіз у контексті вимог сучасних веб-систем;
4. розробити практичні рекомендації щодо вибору моделі авторизації для різних типів веб-систем;
5. реалізувати демонстраційний прототип системи авторизації для медичного центру МРТ діагностики на основі моделі RBAC засобами Django.

Об'єктом дослідження є процеси управління доступом у сучасних веб-системах.

Предметом дослідження є моделі рольового розмежування доступу RBAC, ABAC та ReBAC та їх характеристики у контексті вимог сучасних веб-застосунків.

Методи дослідження. У роботі використано методи системного аналізу для дослідження структури та принципів моделей управління доступом, порівняльного аналізу для оцінки характеристик моделей за визначеними критеріями, сценарного моделювання для визначення відповідності моделей

конкретним типам веб-систем, а також метод прототипування для практичної реалізації та верифікації теоретичних висновків.[2]

Наукова новизна роботи полягає у формуванні системи критеріїв порівняння моделей RBAC, ABAC та ReBAC що враховує специфіку вимог сучасних веб-систем, та у розробці алгоритму вибору моделі авторизації залежно від характеристик конкретної системи.[2]

Практична значущість результатів дослідження полягає у тому що сформульовані рекомендації щодо вибору моделі авторизації можуть бути безпосередньо застосовані розробниками при проектуванні веб-систем, а реалізований прототип є відтворюваним демонстраційним середовищем що ілюструє принципи RBAC у контексті реальної предметної області медичного центру діагностики.

Структура роботи. Дипломна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків що містить 15 найменувань та додатків. Загальний обсяг роботи становить 104 сторінки з них основного тексту 80 сторінки.

У першому розділі досліджено теоретичні основи управління доступом у веб-системах — визначено поняття автентифікації та авторизації, проведено класифікацію моделей управління доступом та простежено еволюцію підходів до авторизації у веб-розробці. У другому розділі проведено детальний аналіз трьох цільових моделей — RBAC, ABAC та ReBAC — з розглядом їх принципів, переваг, обмежень та гібридних підходів що поєднують кілька моделей. У третьому розділі на основі сформованої системи критеріїв проведено порівняльний аналіз моделей, визначено сценарії їх застосування та сформульовано практичні рекомендації щодо вибору моделі. У четвертому розділі обґрунтовано вибір технологічного стеку та моделі авторизації для прототипу, описано проектування та реалізацію системи авторизації

медичного центру МРТ діагностики на основі RBAC засобами Django, наведено результати тестування.[2]

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ УПРАВЛІННЯ ДОСТУПОМ У ВЕБ-СИСТЕМАХ

1.1 Поняття автентифікації та авторизації — різниця і взаємозв'язок

У сучасних веб-системах захист інформаційних ресурсів забезпечується двома взаємопов'язаними, проте принципово різними механізмами — автентифікацією та авторизацією. Некоректне розуміння їхньої різниці є однією з найпоширеніших причин архітектурних помилок при проектуванні систем безпеки [1].

Автентифікація (від грецьк. *authentikos* — справжній) — це процес верифікації ідентичності суб'єкта, що взаємодіє з системою. Її мета полягає у підтвердженні того, що сторона, яка ініціює з'єднання, є саме тим суб'єктом, за якого себе видає. Технічно автентифікація реалізується через перевірку одного або кількох факторів: знання (пароль, PIN-код), володіння (токен, смарт-карта) або властивості (біометричні дані). Результатом успішної автентифікації є підтверджена ідентичність — встановлений зв'язок між запитом та конкретним обліковим записом у системі [3; 4].

Авторизація, натомість, є логічно наступним процесом, який відбувається після успішної автентифікації та визначає обсяг дозволених дій для вже ідентифікованого суб'єкта. Вона оперує не питанням «хто це?», а питанням «що цьому суб'єкту дозволено?». Механізм авторизації зіставляє підтверджену ідентичність із набором прав та привілеїв, визначених у політиці безпеки системи, і на основі цього зіставлення дозволяє або забороняє виконання конкретної операції над конкретним ресурсом [4].

Взаємозв'язок між цими процесами є однонаправленим і строго послідовним: авторизація без попередньої автентифікації є архітектурно некоректною, оскільки будь-яке рішення про надання доступу має ґрунтуватися на верифікованій ідентичності. Разом з тим автентифікація сама по собі не гарантує доступу до ресурсів — вона лише відкриває можливість

для перевірки авторизації. Саме тому ці два механізми, попри їхню концептуальну відмінність, утворюють нерозривну пару у будь-якій системі контролю доступу.[5]

У контексті веб-розробки ці механізми реалізуються через різні технологічні стеки. Автентифікація у сучасних веб-застосунках базується на протоколах OAuth 2.0 та OpenID Connect, які стандартизують процес делегованої автентифікації між різними системами, а також на механізмі JSON Web Tokens (JWT) для передачі верифікованої ідентичності між сервісами. Авторизація реалізується через моделі управління доступом — RBAC, ABAC та ReBAC — детальний розгляд яких є предметом цієї роботи.[3; 5; 6; 7]

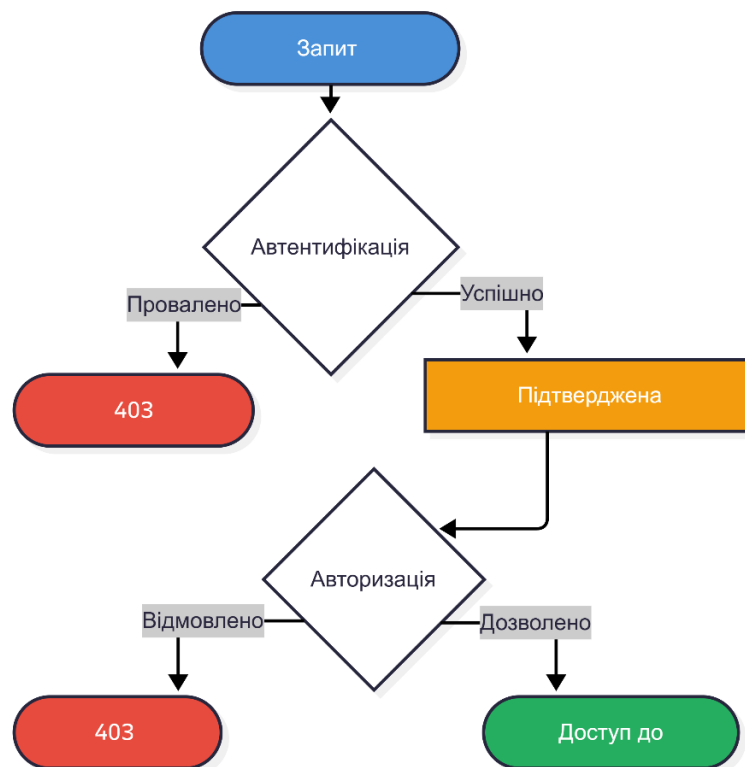


Рис. 1.1 — Послідовність процесів автентифікації та авторизації у веб-системі

Важливо також розуміти, що помилки у реалізації кожного з цих механізмів призводять до якісно різних вразливостей. Недоліки автентифікації — наприклад, слабкі паролі або відсутність захисту від brute-force атак — дозволяють зловмиснику видати себе за легітимного користувача. Недоліки авторизації — некоректно налаштовані права або відсутність перевірок на

рівні об'єктів — дозволяють легітимному користувачеві отримати доступ до ресурсів, які йому не належать. Другий клас вразливостей є значно поширенішим у реальних веб-системах і саме він є основним предметом розгляду в даній роботі [6; 7].

1.2 Класифікація моделей управління доступом

Управління доступом є фундаментальною складовою інформаційної безпеки будь-якої комп'ютерної системи. Протягом десятиліть розвитку обчислювальних технологій було розроблено кілька концептуально різних моделей управління доступом, кожна з яких відповідає певному класу задач і відображає відповідний рівень розвитку вимог до безпеки систем. Систематизація цих моделей дозволяє обґрунтовано підходити до вибору архітектури авторизації для конкретного застосунку [1; 4; 8].

Дискреційна модель управління доступом (DAC — Discretionary Access Control) є однією з найбільш ранніх і концептуально простих моделей. Її ключова ідея полягає в тому, що власник ресурсу самостійно і на власний розсуд визначає, хто і з якими правами може отримати до нього доступ. Технічно ця модель реалізується через списки контролю доступу (ACL — Access Control List), де для кожного ресурсу зберігається перелік суб'єктів та їхніх дозволів. Класичним прикладом DAC є файлова система Unix/Linux, де власник файлу встановлює права читання, запису та виконання для себе, своєї групи та інших користувачів [8]. Головним недоліком DAC є відсутність централізованого контролю — власник може передати права будь-якому суб'єкту, що ускладнює забезпечення системної політики безпеки у великих організаціях.[9; 10]

Мандатна модель управління доступом (MAC — Mandatory Access Control) була розроблена як відповідь на обмеження DAC і орієнтована передусім на системи з жорсткими вимогами до конфіденційності. У моделі MAC кожному суб'єкту та об'єкту системи присвоюється мітка безпеки, що

відображає рівень секретності або чутливості. Рішення про надання доступу приймається централізовано на основі порівняння міток відповідно до заздалегідь визначених правил — суб'єкт може отримати доступ до об'єкта лише якщо його рівень допуску відповідає або перевищує рівень секретності об'єкта. MAC широко застосовується у державних, військових та розвідувальних системах, однак через свою жорсткість є непридатною для більшості комерційних веб-застосунків.[11; 12; 13]

Рольова модель управління доступом (RBAC — Role-Based Access Control) з'явилася як практичний компроміс між гнучкістю DAC та строгістю MAC. Модель була формально стандартизована Національним інститутом стандартів і технологій США (NIST) у стандарті ANSI/INCITS 359-2004. Фундаментальний принцип RBAC полягає у введенні проміжного рівня абстракції між користувачами та правами доступу — ролей, що відповідають посадам або функціям в організації. Користувач отримує права не безпосередньо, а через призначені йому ролі, що суттєво спрощує адміністрування великих систем. RBAC є найпоширенішою моделлю у сучасних веб-фреймворках і детально розглядається у розділі 2.[9; 12]

Атрибутивна модель управління доступом (ABAC — Attribute-Based Access Control) є логічним розвитком RBAC у напрямку підвищення гнучкості. NIST визначає ABAC як модель, у якій рішення про надання доступу приймається на основі атрибутів суб'єктів, об'єктів, виконуваних дій та контексту середовища. На відміну від RBAC, де єдиним вирішальним фактором є роль користувача, ABAC може враховувати довільну кількість характеристик одночасно, що дозволяє виражати значно складніші політики доступу. Детальний аналіз ABAC наведено у підрозділі 2.2. [10]

Реляційна модель управління доступом (ReBAC — Relationship-Based Access Control) є найновішою з розглянутих моделей і базується на графі відносин між суб'єктами та об'єктами системи. Практичний інтерес до ReBAC суттєво зріс після публікації у 2019 році технічного опису системи Google

Zanzibar — глобальної платформи авторизації, що обслуговує мільярди об'єктів. У ReBAC права доступу визначаються не роллю або атрибутами користувача, а конкретним типом відносин між ним і ресурсом, що робить цю модель особливо придатною для соціальних платформ та систем спільної роботи. Детальний розгляд ReBAC представлено у підрозділі 2.3.[3]

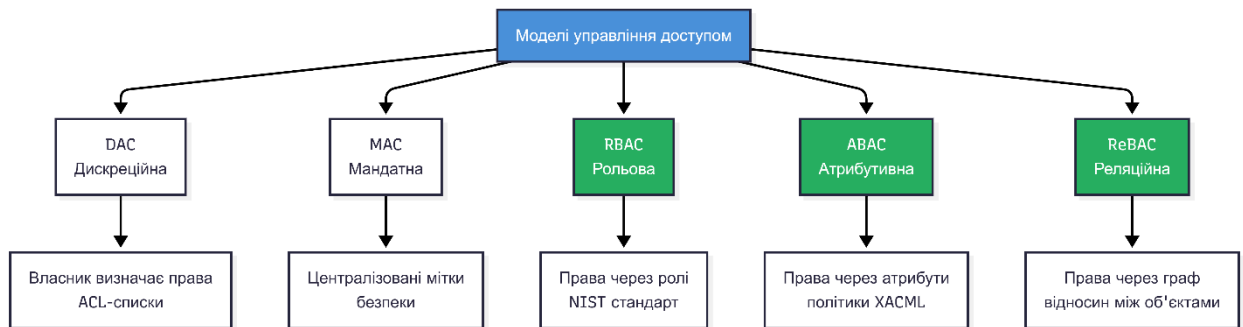


Рис. 1.2 — Класифікація моделей управління доступом у комп'ютерних системах

Наведена класифікація не є вичерпною — у науковій літературі описуються також інші моделі, зокрема IBAC (Identity-Based Access Control) та OrBAC (Organization-Based Access Control), однак вони або є варіантами вже розглянутих моделей, або мають вузькоспеціалізоване застосування. У контексті даної роботи детальному аналізу підлягають три моделі — RBAC, ABAC та ReBAC, — оскільки саме вони є найбільш релевантними для сучасних веб-систем і формують основу для практичної реалізації у розділі 4.[3; 13]

1.3 Огляд сучасних веб-систем та їх вимоги до управління доступом

Сучасні веб-системи являють собою складні розподілені програмні комплекси, що обслуговують від сотень до мільярдів користувачів і оперують значними обсягами чутливих даних. Така складність висуває до систем управління доступом вимоги, що принципово відрізняються від вимог до традиційних настільних або локальних корпоративних застосунків. Розуміння цих вимог є необхідною передумовою для обґрунтованого вибору моделі авторизації [2].

З архітектурної точки зору сучасні веб-системи можна класифікувати за кількома ознаками. За масштабом розрізняють малі системи з фіксованою базою користувачів — корпоративні інтранет-портали, внутрішні інструменти — та великі публічні платформи з необмеженою аудиторією. За архітектурним патерном виділяють монолітні застосунки, мікросервісні архітектури та безсерверні (serverless) рішення. За характером даних розрізняють системи загального призначення та системи що обробляють регульовані дані — медичну інформацію, фінансові записи, персональні дані громадян. Кожна з цих категорій формує власний профіль вимог до управління доступом.[9; 14]

Першою і найбільш фундаментальною вимогою є масштабованість. Перевірка авторизації відбувається при кожному HTTP-запиті до захищеного ресурсу, тому затримка цієї операції безпосередньо і лінійно впливає на час відповіді системи. Для платформи з навантаженням у десятки тисяч запитів на секунду навіть затримка у кілька мілісекунд на операцію авторизації може призвести до деградації продуктивності на рівні всієї системи. Система Google Zanzibar, яка реалізує модель ReBAC для продуктів Google, обробляє понад мільйон запитів авторизації на секунду з медіанною затримкою менше одної мілісекунди — це є орієнтиром для оцінки масштабованості систем авторизації.[11; 15]

Другою вимогою є гнучкість моделі доступу. На відміну від корпоративних систем, де організаційна структура є відносно стабільною, сучасні веб-платформи характеризуються динамічними та різноманітними моделями доступу. У системах для спільної роботи один і той самий користувач може одночасно бути власником одного документа, редактором іншого та лише переглядачем третього, причому цей стан змінюється в реальному часі. Модель авторизації повинна природно відображати такі динамічні відносини без необхідності реструктуризації всієї системи прав.[1; 16]

Третьою вимогою є багаторівнева безпека. Веб-системи за своєю природою є відкритими та доступними через публічну мережу, що робить їх першочерговою цілью для атак. Серед найпоширеніших векторів атак на системи авторизації виділяють Insecure Direct Object Reference (IDOR) — отримання доступу до чужих об'єктів через маніпуляцію ідентифікаторами, privilege escalation — підвищення привілеїв через вразливості у логіці авторизації, та broken access control — загальний клас вразливостей що займає першу позицію у рейтингу OWASP Top 10. Система авторизації повинна бути спроектована таким чином, щоб мінімізувати ймовірність цих вразливостей на архітектурному рівні.[1; 17]

Четвертою вимогою є зручність адміністрування. Реальні веб-платформи можуть мати тисячі користувачів, сотні типів ресурсів та складні ієрархії прав. Адміністратори системи, які часто не є розробниками, повинні мати можливість керувати правами доступу через зрозумілий інтерфейс без необхідності модифікації програмного коду. Складність адміністративного інтерфейсу безпосередньо впливає на ймовірність помилок конфігурації, які є однією з найчастіших причин інцидентів безпеки [3].

П'ятою вимогою є аудит та журналювання операцій доступу. Сучасне законодавство у сфері захисту персональних даних — зокрема європейський регламент GDPR — зобов'язує системи що обробляють персональні дані вести детальні журнали всіх операцій доступу. Система авторизації повинна фіксувати не лише успішні запити, але й відмови у доступі, оскільки саме вони можуть свідчити про спроби несанкціонованого проникнення.[20]

Шостою вимогою, специфічною для мікросервісних архітектур, є підтримка розподіленої авторизації. Коли застосунок складається з десятків незалежних сервісів, кожен з яких може отримувати запити від інших сервісів або від зовнішніх клієнтів, система авторизації повинна забезпечувати консистентну перевірку прав у всіх точках системи. Це вимагає

централізованого сховища політик та стандартизованого протоколу передачі інформації про авторизацію між сервісами.[18]

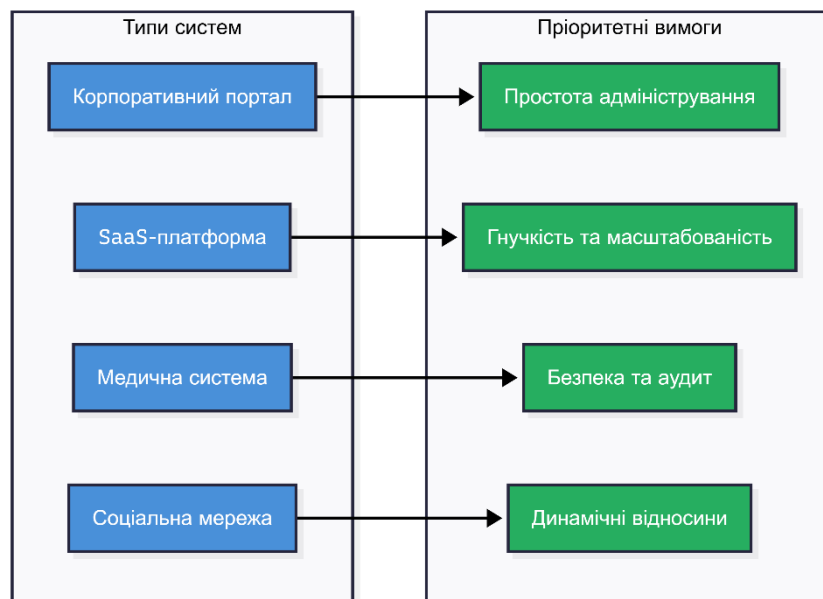


Рис. 1.3 — Відповідність типів веб-систем та пріоритетних вимог до управління доступом

Аналіз вимог сучасних веб-систем до управління доступом демонструє, що жодна з класичних моделей авторизації не задовольняє всі вимоги однаково ефективно. Вибір моделі є компромісом між гнучкістю, продуктивністю та складністю реалізації, який має ґрунтуватися на конкретному профілі вимог системи. Саме тому порівняльний аналіз моделей RBAC, ABAC та ReBAC у контексті цих вимог є центральним завданням даної роботи.[3]

1.4 Еволюція підходів до авторизації у веб-розробці

Підходи до авторизації у веб-розробці пройшли тривалий та нелінійний шлях розвитку, що безпосередньо відображає зростання складності веб-систем, розширення вимог до безпеки та зміну архітектурних парадигм. Розуміння цієї еволюції є важливим не лише з історичної точки зору, а й з практичної — оскільки дозволяє усвідомити причини появи кожної з сучасних моделей авторизації та обмеження, які вони покликані подолати [13; 17].

Перший етап — hardcoded авторизація (1990-ті роки). На початковому етапі розвитку динамічних веб-застосунків більшість розробників реалізовували перевірку прав безпосередньо у програмному коді через умовні оператори. Типовий підхід полягав у перевірці ідентифікатора або атрибуту користувача безпосередньо перед виконанням захищеної операції. Такий підхід, відомий як hardcoded authorization, мав критичний структурний недолік: логіка авторизації була розпорошена по всій кодовій базі, що унеможливлювало централізоване управління правами та робило будь-яку зміну політики доступу надзвичайно трудомісткою і ризикованою операцією [6].

Другий етап — поява фреймворків та впровадження RBAC (2000-ні роки). Зі зростанням складності веб-застосунків та появою зрілих веб-фреймворків розробники отримали перші стандартизовані інструменти для управління доступом. Фреймворки Django, Ruby on Rails та пізніше Laravel запровадили вбудовані механізми авторизації на основі рольової моделі, що дозволило винести логіку доступу з бізнес-коду в окремий шар системи. Формальна стандартизація RBAC у документі ANSI/INCITS 359-2004 надала теоретичне підґрунтя для цих практичних реалізацій. RBAC стала де-факто стандартом для корпоративних веб-застосунків того часу завдяки інтуїтивній відповідності організаційним ієрархіям та простоті адміністрування.[6; 11]

Третій етап — делегована авторизація та OAuth (2010-ні роки). Перехід до хмарних обчислень, мобільних платформ та мікросервісної архітектури породив принципово новий клас задач авторизації. Системи перестали бути монолітними і замкненими — користувачі стали отримувати доступ до ресурсів через сторонні застосунки, а сервіси почали взаємодіяти між собою від імені користувача. Відповіддю на ці виклики стало широке впровадження протоколу OAuth 2.0 [3] та стандарту OpenID Connect, які стандартизували механізм делегованої авторизації. Паралельно зростала популярність ABAC як більш гнучкої альтернативи RBAC — NIST опублікував спеціальний документ

NIST SP 800-162 , що визначає принципи атрибутивної моделі та рекомендації щодо її впровадження у федеральних інформаційних системах.[10; 19]

Четвертий етап — реляційна авторизація та системи авторизації як окремий сервіс (2019 — теперішній час). Переломним моментом у розвитку підходів до авторизації стала публікація у 2019 році технічного опису системи Google Zanzibar. Ця система реалізує модель ReBAC у глобальному масштабі та продемонструвала, що граф відносин між об'єктами є природною та ефективною основою для авторизації у великих розподілених системах. Публікація Zanzibar спричинила появу цілої екосистеми відкритих реалізацій — OpenFGA від Auth0, SpiceDB від Authzed та Permify — які перенесли ідеї ReBAC у доступний для широкого загалу розробників формат. Одночасно сформувалася концепція авторизації як окремого сервісу (Authorization as a Service), що відповідає принципу розподілу відповідальностей у мікросервісних архітектурах. [3; 7; 11]



Рис. 1.4 — Хронологія еволюції підходів до авторизації у веб-розробці

Сучасний стан галузі характеризується співіснуванням усіх описаних підходів. RBAC залишається домінуючою моделлю для більшості веб-застосунків завдяки простоті та широкій підтримці у фреймворках. ABAC

знаходить застосування у системах з жорсткими регуляторними вимогами. ReBAC стрімко набирає популярність у SaaS-платформах та системах для спільної роботи. Ця різноманітність підходів свідчить про те, що вибір моделі авторизації є архітектурним рішенням, яке не має єдиної правильної відповіді і повинно ґрунтуватися на глибокому розумінні вимог конкретної системи — саме цьому питанню присвячено розділ 3 даної роботи.[3; 21]

РОЗДІЛ 2. АНАЛІЗ МОДЕЛЕЙ RBAC, ABAC, REBAC

2.1 Модель RBAC (Role-Based Access Control)

Рольова модель управління доступом (RBAC — Role-Based Access Control) є на сьогодні найпоширенішою моделлю авторизації у корпоративних та веб-системах. Формальне визначення моделі було запропоновано дослідниками Феррайоло та Куном у 1992 році, а її офіційна стандартизація відбулась у 2004 році у документі ANSI/INCITS 359-2004. Широке практичне поширення RBAC пояснюється не лише технічними перевагами, а й концептуальною відповідністю реальним організаційним структурам — модель природно відображає розподіл обов'язків між посадами та підрозділами будь-якої організації.[3; 9]

Фундаментальний принцип RBAC полягає у введенні проміжного рівня абстракції між користувачами та правами доступу — ролей. Замість прямого призначення прав кожному користувачеві адміністратор визначає ролі що відповідають функціям в організації, призначає цим ролям відповідні права, а потім асоціює користувачів з ролями. Цей підхід реалізує принцип найменших привілеїв — користувач отримує рівно ті права, які необхідні для виконання його функцій, і не більше [3].

Формальна модель RBAC визначається через п'ять множин та відношень між ними. Множина користувачів (U — Users) містить усіх суб'єктів системи що потребують доступу до ресурсів. Множина ролей (R — Roles) містить іменовані набори прав що відповідають певним функціям або посадам. Множина прав (P — Permissions) містить дозволи на виконання конкретних операцій над конкретними ресурсами. Множина сесій (S — Sessions) містить активні з'єднання користувачів із системою в рамках яких активовані певні ролі. Відношення призначення користувачів до ролей (UA — User Assignment) та відношення призначення прав до ролей (PA — Permission Assignment) визначають структуру доступу у системі [3; 22].

Стандарт NIST визначає чотири рівні RBAC що утворюють ієрархію від простого до складного. Перший рівень — плоский RBAC (Flat RBAC) — реалізує базові принципи моделі: користувачі отримують права через ролі, одному користувачеві може бути призначено кілька ролей. Другий рівень — ієрархічний RBAC (Hierarchical RBAC) — додає можливість успадкування прав між ролями через відношення часткового порядку: старша роль автоматично успадковує всі права підлеглих ролей. Третій рівень — обмежений RBAC (Constrained RBAC) — вводить принцип статичного розподілу обов'язків (Static Separation of Duty), що унеможливорює призначення одному користувачеві взаємовиключних ролей — наприклад ролі касира та ролі контролера касових операцій одночасно. Четвертий рівень — симетричний RBAC (Symmetric RBAC) — розширює механізми запиту та перегляду прав у обох напрямках відношення РА [3; 22].

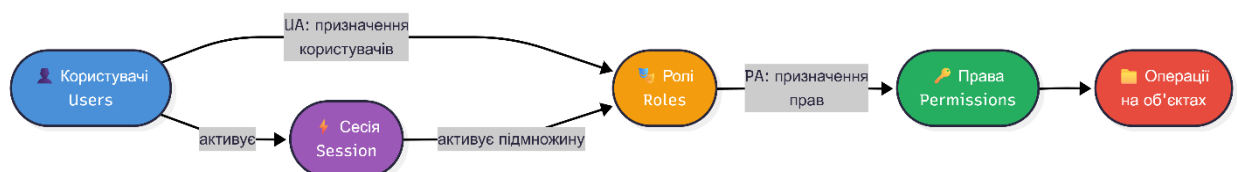


Рис. 2.1 — Структурна схема компонентів та зв'язків моделі RBAC

Практична реалізація RBAC у веб-системах на основі Django [9] базується на вбудованій системі груп та прав. Групи Django відповідають ролям у термінології RBAC — кожна група містить набір прав, а користувач отримує всі права груп до яких він належить. Права у Django визначаються на рівні моделей даних і мають форму `<app>.<action>_<model>`, наприклад `blog.add_post` або `finance.view_report`. Такий підхід забезпечує чітку відповідність між правами у коді та правами у базі даних і дозволяє адмініструвати доступ через стандартний адміністративний інтерфейс без модифікації програмного коду [9].

Серед переваг RBAC насамперед виділяється простота адміністрування. Управління через ролі є інтуїтивно зрозумілим для адміністраторів оскільки ролі відповідають реальним посадам в організації. При прийомі нового

співробітника достатньо призначити йому відповідну роль — він автоматично отримає всі необхідні права. При звільненні — достатньо відкликати роль. Ця простота суттєво знижує ймовірність помилок конфігурації порівняно з моделями де права призначаються індивідуально [3].

Другою перевагою є висока продуктивність перевірки доступу. Визначення того чи має користувач право на виконання операції зводиться до одного запиту до бази даних — перевірки наявності відповідного права серед прав усіх ролей користувача. Ця операція ефективно індексується і виконується за константний час незалежно від розміру системи [3].

Третьою перевагою є широка екосистемна підтримка. Більшість популярних веб-фреймворків — Django, Laravel, Spring Security — мають вбудовану або добре інтегровану реалізацію RBAC, що суттєво знижує вартість розробки та підтримки [12].

Разом з тим RBAC має суттєві обмеження що визначають межі її застосовності. Головним з них є проблема вибуху ролей (role explosion). У великих організаціях з різноманітними вимогами до доступу кількість ролей може зростати неконтрольовано — для кожної унікальної комбінації прав необхідна окрема роль. Дослідження показують що великі корпоративні системи можуть містити тисячі ролей що робить адміністрування фактично неможливим [12].

Другим суттєвим обмеженням є контекстна сліпота моделі. RBAC приймає рішення виключно на основі ролі користувача і не може враховувати динамічні фактори — час доби, географічне розташування, стан ресурсу або відносини між користувачем і конкретним об'єктом. Це робить RBAC непридатною для систем де правило «менеджер може переглядати лише звіти свого відділу» є базовою вимогою — така умова потребує або введення окремої ролі для кожного відділу що повертає до проблеми вибуху ролей або доповнення моделі атрибутивною логікою [3; 12].

Третім обмеженням є погана придатність для моделювання динамічних соціальних структур. У системах де права доступу виникають і зникають у реальному часі внаслідок дій користувачів — наприклад при спільному використанні документів — підтримання коректного стану ролей вимагає значних адміністративних зусиль або автоматизації що ускладнює систему. [3]

2.2 Модель ABAC (Attribute-Based Access Control)

Атрибутивна модель управління доступом (ABAC — Attribute-Based Access Control) є логічним розвитком рольової моделі у напрямку підвищення гнучкості та виразності правил доступу. NIST визначає ABAC як модель в якій рішення про надання доступу приймається на основі атрибутів суб'єктів, об'єктів, виконуваних дій та контексту середовища відповідно до набору формально визначених політик [7]. На відміну від RBAC де єдиним вирішальним фактором є приналежність користувача до ролі, ABAC оцінює довільну кількість характеристик одночасно що принципово розширює клас задач які може вирішити модель.[24]

Концептуальною основою ABAC є поняття атрибуту — іменованої характеристики що описує певну властивість суб'єкта, об'єкта, дії або середовища. Стандарт визначає чотири категорії атрибутів що беруть участь у прийнятті рішення про доступ. Атрибути суб'єкта описують користувача що ініціює запит — посада, відділ, рівень допуску, громадянство, вік або будь-яка інша характеристика що зберігається у профілі користувача. Атрибути об'єкта описують ресурс до якого запитується доступ — тип документа, рівень конфіденційності, власник, дата створення, підрозділ якому належить ресурс. Атрибути дії описують операцію яку намагається виконати суб'єкт — читання, запис, видалення, затвердження, публікація. Атрибути середовища описують контекст запиту — поточний час, день тижня, географічне розташування клієнта, тип пристрою, поточний рівень загроз у системі [24].

Центральним елементом архітектури ABAC є політики доступу — формальні правила що визначають умови за яких доступ дозволяється або забороняється. Для стандартизації опису та обміну політиками розроблено мову XACML (eXtensible Access Control Markup Language) що є XML-стандартом визнаним OASIS [12]. Архітектура XACML визначає кілька функціональних компонентів що взаємодіють при обробці запиту доступу. Точка застосування рішень (PEP — Policy Enforcement Point) перехоплює запит та передає його на оцінку. Точка прийняття рішень (PDP — Policy Decision Point) отримує запит, завантажує релевантні політики та атрибути і повертає рішення. Точка інформації про політики (PIP — Policy Information Point) надає значення атрибутів з зовнішніх джерел — бази даних користувачів, каталогу ресурсів або системного контексту. Сховище політик (PAP — Policy Administration Point) забезпечує зберігання та управління політиками доступу [24; 25].

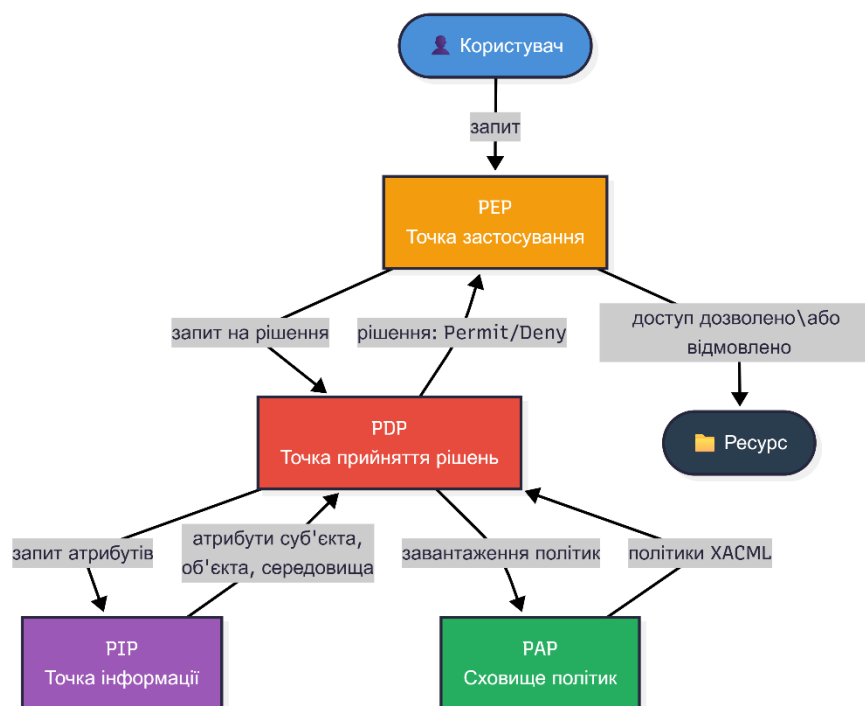


Рис. 2.2 — Архітектура взаємодії компонентів моделі ABAC при обробці запиту доступу

Практична виразність ABAC найкраще ілюструється через приклади політик що неможливо реалізувати засобами чистого RBAC. Правило

«дозволити лікарю переглядати медичну картку пацієнта якщо лікар є лікуючим лікарем цього пацієнта і запит надходить з мережі лікарні і поточний час є робочими годинами» поєднує атрибути суб'єкта (спеціальність лікаря, відношення до пацієнта), атрибути середовища (IP-адреса мережі, поточний час) та атрибути об'єкта (тип документа) в єдиній умові. Саме такий рівень деталізації є характерним для систем що підпадають під дію регуляторних вимог у сфері охорони здоров'я або фінансів [24].

Реалізація ABAC у веб-системах може здійснюватися на різних рівнях складності. Найпростішим підходом є пряма перевірка атрибутів у програмному коді — така реалізація є зрозумілою але розпорошує логіку авторизації по всій кодовій базі. Більш зрілим підходом є розробка власного движка політик що зберігає правила у базі даних та оцінює їх централізовано при кожному запиті доступу. Найповнішою реалізацією є використання спеціалізованих движків що підтримують стандарт XACML — наприклад AuthzForce або WSO2 Identity Server — однак такі рішення є значно складнішими у розгортанні та налаштуванні [24].

Головною перевагою ABAC є висока виразність і гнучкість. Модель дозволяє реалізувати будь-яку складність правил доступу без необхідності зміни архітектури системи — достатньо додати нову політику або новий атрибут. Це принципово відрізняє ABAC від RBAC де кожна нова вимога до доступу потенційно вимагає створення нової ролі. Другою перевагою є контекстуальність — здатність враховувати динамічні фактори середовища при прийнятті рішень про доступ що є критично важливим для систем з підвищеними вимогами безпеки. Третьою перевагою є придатність для федеративних середовищ — атрибути можуть надходити з різних зовнішніх джерел що дозволяє реалізувати авторизацію у розподілених системах де користувачі та ресурси належать різним організаціям [10].

Разом з тим ABAC має суттєві недоліки що обмежують її застосовність. Першим і найбільш критичним є складність реалізації та налагодження.

Політики АВАС особливо написані мовою ХАСМЛ є важкими для розуміння навіть для досвідчених розробників — XML-синтаксис ХАСМЛ є надзвичайно багатослівним і навіть прості правила потребують значного обсягу розмітки [. Другим недоліком є потенційний вплив на продуктивність системи. Оцінка складних політик що потребують завантаження атрибутів з кількох зовнішніх джерел може суттєво збільшити час обробки запиту порівняно з простою перевіркою ролі у RBAC. Третім недоліком є складність тестування коректності політик — верифікація того що система надає рівно необхідний обсяг прав і не більше є нетривіальною задачею що потребує спеціальних методологій та інструментів.[3; 25]

2.3 Модель ReBAC (Relationship-Based Access Control)

Реляційна модель управління доступом (ReBAC — Relationship-Based Access Control) є найновішою з трьох розглянутих моделей і представляє принципово інший підхід до визначення прав доступу. Якщо RBAC відповідає на питання «яку роль має користувач?», а АВАС — «які атрибути має користувач і ресурс?», то ReBAC відповідає на питання «яке відношення існує між користувачем і конкретним ресурсом?». Саме ця семантична відмінність визначає унікальну нішу застосування ReBAC у сучасних веб-системах [3; 26].

Теоретичним підґрунтям ReBAC є теорія графів — математична дисципліна що вивчає структури з вузлів та ребер. У моделі ReBAC користувачі, групи та ресурси системи представляються вузлами графу, а відносини між ними — орієнтованими ребрами з іменованими типами. Наприклад ребро типу «owner» від вузла користувача до вузла документа означає що цей користувач є власником документа. Ребро типу «member» від вузла користувача до вузла групи означає членство у групі. Перевірка доступу у такій моделі зводиться до задачі досяжності у графі — визначення чи існує шлях від вузла користувача до вузла ресурсу через дозволені типи відносин.[26]

Переломним моментом у розвитку ReBAC стала публікація у 2019 році технічного опису системи Google Zanzibar — глобальної платформи авторизації що обслуговує такі продукти як Google Drive, Google Photos, YouTube та Google Maps. Zanzibar оперує трильйонами записів про відносини між об'єктами та обробляє понад мільйон запитів авторизації на секунду з медіанною затримкою менше однієї мілісекунди. Масштаб та продуктивність Zanzibar продемонстрували що ReBAC є не лише теоретично привабливою але й практично реалізованою моделлю для систем найвищого рівня навантаження. [11; 24]

Ключовим поняттям моделі у стилі Zanzibar є кортеж відносин (relation tuple) — атомарний запис виду «об'єкт # відношення @ суб'єкт». Наприклад кортеж `document:X#viewer@user:Ivan` означає що користувач Ivan має відношення `viewer` до документа X. Кортеж `document:X#viewer@group:editors#member` означає що всі члени групи `editors` мають відношення `viewer` до документа X. Сукупність усіх кортежів утворює граф відносин на основі якого система визначає права доступу для будь-якої пари користувач-ресурс.[11]

Важливою особливістю ReBAC є механізм успадкування прав через ієрархії об'єктів що реалізується через правила перезапису (userset rewrite rules). Ці правила визначають як відносини на батьківських об'єктах транслюються у відносини на дочірніх. Наприклад правило «viewer на папці успадковується як viewer на всіх документах всередині папки» дозволяє не створювати окремий кортеж для кожного документа — достатньо одного

кортежу на рівні папки. Цей механізм є ключовим для підтримки масштабованості системи при великій кількості об'єктів.[26]

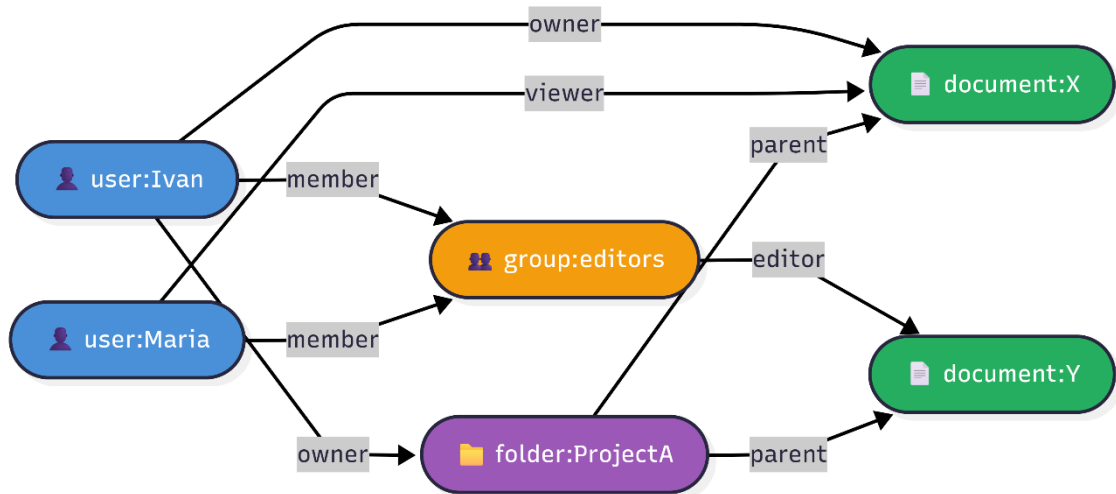


Рис. 2.3 — Граф відносин між об'єктами у моделі ReBAC

Практична реалізація ReBAC у веб-системах може здійснюватися через спеціалізовані відкриті рішення що натхненні архітектурою Zanzibar. OpenFGA розроблена компанією Auth0 є повноцінною реалізацією моделі Zanzibar з підтримкою мови визначення схем та HTTP API для перевірки доступу. SpiceDB від компанії Authzed пропонує аналогічну функціональність з акцентом на продуктивності та консистентності даних у розподілених середовищах. Для цілей даної роботи реалізація ReBAC здійснюється безпосередньо на основі Django з використанням реляційної бази даних для зберігання кортежів відносин що дозволяє наочно продемонструвати принципи моделі без залежності від зовнішніх сервісів.[11; 26; 27]

Природне відображення соціальних та організаційних структур є першою і найважливішою перевагою ReBAC. У системах для спільної роботи основною семантикою доступу є саме відносини між користувачами та ресурсами — «поділитися документом», «додати до команди», «надати право перегляду». ReBAC безпосередньо моделює ці відносини тоді як RBAC вимагає їх штучного переведення у мову ролей що призводить до семантичних втрат та ускладнення адміністрування. Другою перевагою є точність контролю на рівні окремих об'єктів — ReBAC дозволяє визначити різні права для

кожного конкретного документа або ресурсу що є неможливим у чистому RBAC де права визначаються на рівні типів ресурсів. Третьою перевагою є масштабованість при правильній реалізації — граф відносин може містити мільярди записів і завдяки механізму успадкування кількість кортежів що необхідно зберігати є значно меншою ніж кількість фактичних прав доступу у системі. [3; 26; 27]

Недоліки ReBAC пов'язані передусім зі складністю реалізації ефективного движка перевірки доступу. При складних ієрархіях об'єктів перевірка доступу може вимагати рекурсивного обходу великого графу що без належної оптимізації призводить до незадовільної продуктивності. Другим недоліком є складність аудиту системи — оскільки права доступу визначаються неявно через ланцюжки відносин а не через явні призначення з'ясування чому конкретний користувач має або не має доступу до певного ресурсу вимагає трасування шляху у графі що є нетривіальною операцією. Третім недоліком є відсутність широкої підтримки у стандартних веб-фреймворках — на відміну від RBAC реалізація ReBAC потребує або розробки власного рішення або інтеграції зовнішнього сервісу авторизації. [3]

2.4 Гібридні підходи — комбінування моделей

На практиці більшість реальних веб-систем не використовують жодну з розглянутих моделей у чистому вигляді. Складність сучасних веб-платформ, різноманітність типів ресурсів та неоднорідність вимог до доступу в межах однієї системи зумовлюють необхідність комбінування різних моделей авторизації. Гібридні підходи дозволяють поєднати переваги кількох моделей і водночас компенсувати їхні індивідуальні обмеження за умови дотримання принципу мінімальної архітектурної складності.[3]

Концептуальною основою гібридизації є розподіл простору задач авторизації на зони відповідальності різних моделей. Кожна модель застосовується до того класу задач де вона демонструє найкращі

характеристики, а межі між зонами визначаються на етапі проектування архітектури системи. Такий підхід відрізняється від довільного змішування моделей що може призвести до непередбачуваної поведінки системи авторизації та ускладнення аудиту безпеки. [4]

Комбінація RBAC та ABAC. Найпоширенішим гібридним підходом є доповнення рольової моделі атрибутивними обмеженнями що у літературі часто позначається як RBAC з умовами (Conditional RBAC) або розширений RBAC (Extended RBAC). У цьому підході базова структура доступу визначається ролями — користувач отримує певний набір потенційних прав через призначені ролі. Однак для критичних операцій додатково перевіряються атрибути що звужують застосування права до конкретного контексту. Наприклад роль «менеджер» надає право перегляду фінансових звітів, однак атрибутивна умова обмежує це право лише звітами відділу якому належить менеджер. Такий підхід дозволяє зберегти простоту адміністрування RBAC для типових випадків і водночас забезпечити необхідну точність контролю там де це вимагається без повного переходу до складної архітектури ABAC. [10]

Реалізація цього підходу у Django здійснюється через поєднання вбудованої системи груп та прав з додатковими перевітками атрибутів у декораторах або міксінах представлень. Спочатку перевіряється наявність права через стандартний механізм Django permissions, а потім виконується додаткова перевірка атрибутивної умови що може звертатися до бази даних або контексту запиту.[13]

Комбінація ReBAC та RBAC. Другою поширеною комбінацією є поєднання реляційної моделі для управління доступом до конкретних об'єктів з рольовою моделлю для глобальних системних привілеїв. У системах управління проектами більшість прав природно виражається через відносини — членство у проекті визначає доступ до його ресурсів, а роль у команді визначає тип дозволених операцій. Водночас системні ролі — адміністратор

платформи, модератор, технічна підтримка — що передбачають доступ до всіх об'єктів системи незалежно від конкретних відносин зручніше моделювати через RBAC. Така архітектура дозволяє уникнути необхідності створення кортежів відносин між системними адміністраторами та кожним об'єктом у системі що може налічувати мільйони записів.[11]

Підхід Google Zanzibar до гібридизації. Система Zanzibar пропонує власний елегантний механізм гібридизації через поняття *userset rewrite rules* що дозволяють визначати складні умови доступу виключно у рамках моделі ReBAC без явного переходу до інших моделей. Правила перезапису можуть включати об'єднання відносин (*union*) — доступ надається якщо виконується хоча б одна з умов, перетин відносин (*intersection*) — доступ надається лише якщо виконуються всі умови одночасно, та виключення відносин (*exclusion*) — доступ надається якщо виконується перша умова але не виконується друга. Через комбінацію цих операцій можна виразити як рольову так і атрибутивну логіку у рамках єдиної моделі ReBAC що забезпечує концептуальну єдність системи авторизації.[11; 23]

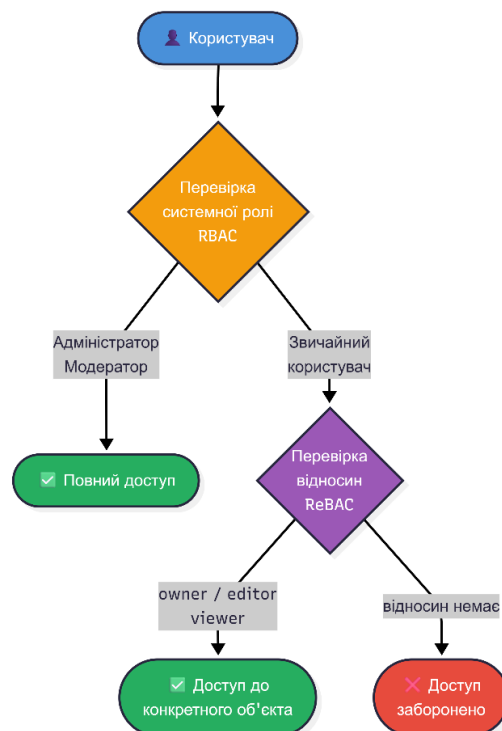


Рис. 2.4 — Архітектура гібридної моделі авторизації на основі RBAC та ReBAC

При проектуванні гібридних систем авторизації необхідно дотримуватися кількох принципів що забезпечують керованість та безпеку архітектури. Принцип чіткого розподілу зон відповідальності вимагає що кожна модель застосовується до строго визначеного класу задач без перекриття з іншими моделями — перекриття зон відповідальності призводить до непередбачуваних взаємодій між правилами різних моделей. Принцип явного пріоритету вимагає визначення порядку застосування моделей у випадку конфліктуючих рішень — зазвичай більш обмежувальне рішення має вищий пріоритет відповідно до принципу *deny-overrides*. Принцип мінімальної складності вимагає що гібридна архітектура використовує лише ті моделі які дійсно необхідні для вирішення конкретних задач — кожна додаткова модель збільшує когнітивне навантаження на розробників та адміністраторів і підвищує ймовірність помилок у логіці доступу.[3; 4; 23]

У контексті практичної реалізації даної роботи розділ 4 демонструє всі три моделі окремо що дозволяє наочно порівняти їхні характеристики в ізольованому вигляді. Однак результати порівняльного аналізу що наведені у розділі 3 свідчать що для більшості реальних SaaS-платформ оптимальним рішенням є гібридна архітектура що поєднує RBAC для глобальних системних ролей та ReBAC для управління доступом до конкретних ресурсів користувачів.[11]

РОЗДІЛ 3. ПОРІВНЯЛЬНИЙ АНАЛІЗ МОДЕЛЕЙ

3.1 Критерії порівняння моделей управління доступом

Проведення об'єктивного порівняльного аналізу моделей RBAC, ABAC та ReBAC вимагає попереднього визначення системи критеріїв що дозволить всебічно оцінити кожную модель з точки зору практичного застосування у сучасних веб-системах. Відсутність єдиної загальноприйнятої методології порівняння моделей авторизації у науковій літературі зумовлює необхідність формування власної системи критеріїв що базується на вимогах визначених у підрозділі 1.3 та враховує як технічні так і організаційні аспекти управління доступом.[3]

Система критеріїв що використовується у даній роботі складається з семи показників кожен з яких оцінюється за трирівневою шкалою — низький, середній та високий рівень. Така шкала є достатньою для відображення принципових відмінностей між моделями без введення хибної точності що була б невиправданою у якісному порівняльному аналізі.[4]

Першим критерієм є гнучкість моделі — здатність виражати складні та різноманітні правила доступу без архітектурних змін системи. Цей критерій оцінює наскільки широкий клас політик доступу може бути виражений засобами моделі та наскільки легко модель адаптується до нових вимог що виникають у процесі розвитку системи. Гнучкість є особливо важливою для систем що перебувають у постійному розвитку та мають складні або нестандартні вимоги до контролю доступу. Другим критерієм є масштабованість адміністрування — здатність моделі ефективно підтримувати зростання кількості користувачів, ресурсів та правил доступу без пропорційного зростання адміністративного навантаження. Цей критерій оцінює чи залишається управління правами доступу керованим при збільшенні розміру системи у десятки або сотні разів. Проблема вибуху ролей у RBAC є класичним прикладом порушення масштабованості

адміністрування. Третім критерієм є складність реалізації — обсяг зусиль необхідних для впровадження моделі у нову або існуючу веб-систему. Цей критерій враховує наявність готових бібліотек та фреймворків, складність програмного коду що необхідно написати та час необхідний для досягнення робочої реалізації. Висока складність реалізації підвищує ймовірність помилок у логіці авторизації та збільшує вартість розробки. Четвертим критерієм є складність адміністрування — зусилля необхідні для управління правами доступу в процесі повсякденної експлуатації системи. На відміну від складності реалізації що є одноразовою вартістю складність адміністрування є постійною вартістю що накопичується протягом усього життєвого циклу системи. Оцінюється наскільки легко адміністратор може додавати нових користувачів, змінювати права та реагувати на нові вимоги без участі розробників. П'ятим критерієм є продуктивність перевірки доступу — швидкість виконання операції авторизації при обробці запиту. Цей критерій є критично важливим для веб-систем з високим навантаженням де перевірка авторизації відбувається при кожному HTTP-запиті. Оцінюється як обчислювальна складність алгоритму перевірки так і кількість звернень до бази даних або зовнішніх сервісів що необхідні для прийняття рішення. Шостим критерієм є стійкість до помилок конфігурації — властивість моделі мінімізувати ймовірність випадкового надання зайвих прав або залишення прогалин у захисті внаслідок помилок адміністратора або розробника. Цей критерій відображає безпекові характеристики моделі з точки зору людського фактору — складніші моделі як правило є більш вразливими до помилок конфігурації. Сьомим критерієм є підтримка у фреймворках та екосистемі — наявність готових, перевірених рішень для реалізації моделі у популярних веб-фреймворках та зрілість супутньої екосистеми інструментів. Цей критерій

безпосередньо впливає на вартість і час розробки а також на доступність кваліфікованих спеціалістів для підтримки системи. [3; 4; 10; 11; 13; 30]

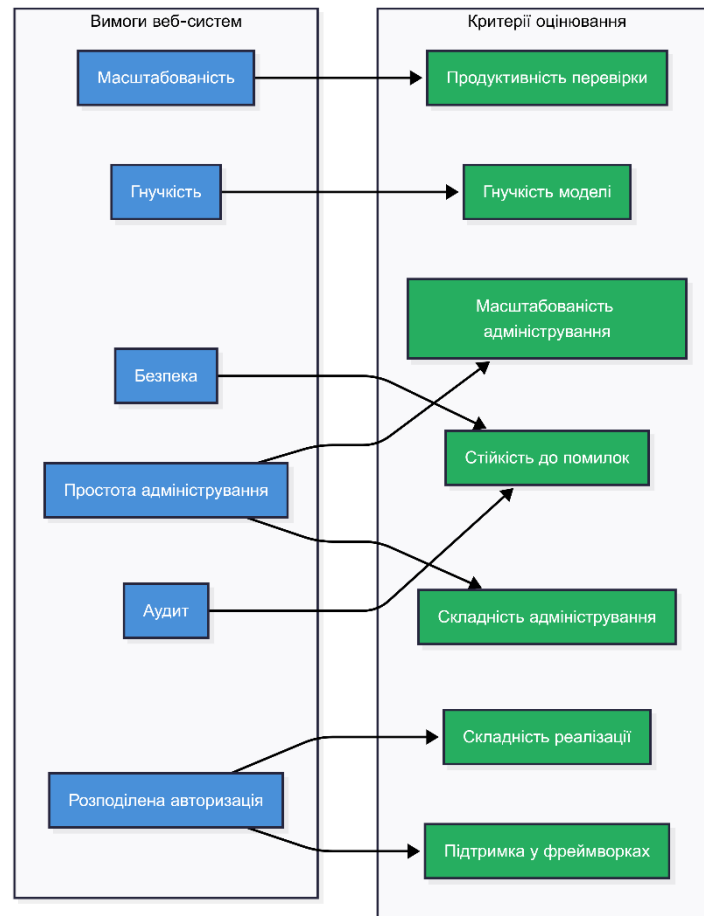


Рис. 3.1 — Відповідність між вимогами веб-систем та критеріями оцінювання моделей авторизації

Визначена система критеріїв є достатньою для проведення повноцінного порівняльного аналізу оскільки охоплює всі ключові виміри оцінки моделей авторизації — від технічних характеристик продуктивності до організаційних аспектів адміністрування. Застосування цих критеріїв у наступному підрозділі дозволить сформулювати обґрунтовані практичні рекомендації щодо вибору моделі для різних типів веб-систем.[31]

3.2 Порівняльна характеристика RBAC, ABAC та ReBAC

На основі системи критеріїв визначених у підрозділі 3.1 проведено детальний порівняльний аналіз трьох моделей управління доступом. Аналіз кожного критерію ґрунтується на теоретичних властивостях моделей розглянутих у розділі 2 та підкріплюється практичними спостереженнями з

реалізації описаної у розділі 4. Результати аналізу систематизовано у таблиці 3.1.

Таблиця 3.1 — Порівняльна характеристика моделей RBAC, ABAC та ReBAC

Критерій	RBAC	ABAC	ReBAC
Гнучкість моделі	Низька	Висока	Висока
Масштабованість адміністрування	Висока	Низька	Висока
Складність реалізації	Низька	Висока	Середня
Складність адміністрування	Низька	Висока	Середня
Продуктивність перевірки	Висока	Середня	Середня
Стійкість до помилок	Середня	Низька	Середня
Підтримка у фреймворках	Висока	Низька	Низька

Порівняльний аналіз за критерієм гнучкості демонструє суттєву перевагу ABAC та ReBAC над RBAC. Рольова модель ефективно справляється з ієрархічними організаційними структурами де права доступу є стабільними і визначаються посадою користувача. Однак RBAC принципово не здатна враховувати контекст запиту або динамічні відносини між користувачем і конкретним об'єктом без введення додаткових ролей що призводить до вже згаданої проблеми вибуху ролей. ABAC досягає високої гнучкості через оцінку довільної кількості атрибутів одночасно — будь-яке правило доступу що може бути формально описане через властивості суб'єкта, об'єкта та середовища може бути виражене засобами ABAC. ReBAC досягає порівняної гнучкості через інший механізм — моделювання довільних відносин між об'єктами та правила успадкування що дозволяють виразити складні ієрархічні структури доступу природним чином.[3; 11; 27]

Аналіз масштабованості адміністрування виявляє протилежну картину де ABAC суттєво поступається двом іншим моделям. У RBAC додавання нового користувача зводиться до простої операції призначення ролі що відповідає його посаді — адміністративне навантаження є константним і не

залежить від розміру системи. У ReBAC додавання користувача до проекту або надання доступу до конкретного ресурсу є також інтуїтивною одноразовою операцією що безпосередньо відповідає реальній дії у системі. ABAC натомість вимагає ретельного визначення та підтримки атрибутів для кожного нового користувача і ресурсу — при зміні організаційної структури або появі нових типів ресурсів необхідно переглядати та оновлювати множину атрибутів що суттєво збільшує адміністративне навантаження. [11; 32; 33]

За критерієм складності реалізації RBAC має очевидну і беззаперечну перевагу. Вбудована підтримка RBAC у Django через систему груп та прав дозволяє отримати повністю функціональну реалізацію рольової моделі без написання жодного рядка додаткового коду. Реалізація ABAC вимагає розробки власного движка політик або інтеграції спеціалізованого рішення що підтримує стандарт XACML — обидва варіанти є значно трудомісткішими. ReBAC займає проміжну позицію — концептуальна модель є відносно простою, однак реалізація ефективного механізму перевірки досяжності у графі відносин вимагає ретельного проектування схеми бази даних та оптимізації запитів.[13; 23; 25]

Продуктивність перевірки доступу є найвищою у RBAC оскільки зводиться до одного запиту до бази даних — перевірки наявності права серед прав усіх ролей користувача. Ця операція ефективно індексується і виконується за час $O(1)$ відносно кількості об'єктів у системі. ABAC демонструє середню продуктивність оскільки оцінка політик може вимагати звернення до кількох зовнішніх джерел атрибутів що збільшує кількість операцій вводу-виводу при обробці кожного запиту. ReBAC у простих випадках також зводиться до одного запиту до бази даних, однак при складних ієрархіях об'єктів може вимагати рекурсивних запитів або обходу графу що негативно впливає на продуктивність при неоптимальній реалізації.[11; 28; 29]

Стійкість до помилок конфігурації є найнижчою у АВАС що пояснюється складністю семантики політик та великою кількістю параметрів що взаємодіють при прийнятті рішення про доступ. Помилка у визначенні атрибуту або умови політики може призвести до непередбачуваних наслідків що важко виявити під час тестування. RBAC та ReBAC демонструють порівняну середню стійкість — обидві моделі є достатньо простими щоб адміністратор міг передбачити наслідки своїх дій, однак обидві мають специфічні класи помилок: у RBAC це надмірно широкі ролі, у ReBAC це некоректні правила успадкування що можуть надавати незапланований доступ через ланцюжки відносин.[1; 30]

Порівняння моделей за гнучкістю та складністю реаліза



Рис. 3.2 — Порівняльний аналіз моделей авторизації за критеріями гнучкості та складності реалізації

Узагальнення результатів порівняльного аналізу дозволяє сформулювати кілька ключових висновків. По-перше жодна з трьох моделей не є домінуючою за всіма критеріями одночасно — кожна модель має свій унікальний профіль переваг і обмежень. По-друге існує фундаментальний компроміс між гнучкістю та простотою що проходить через усі три моделі — підвищення виразності правил доступу неминуче супроводжується зростанням складності реалізації та адміністрування. По-третє RBAC та

ReBAC є взаємодоповнюючими моделями що демонструють схожі характеристики масштабованості але відрізняються за гнучкістю та складністю реалізації що робить їх природними кандидатами для гібридної архітектури описаної у підрозділі 2.4.[3; 4; 30]

3.3 Сценарії застосування залежно від типу веб-системи

Результати порівняльного аналізу проведеного у підрозділі 3.2 набувають практичної цінності лише у контексті конкретних типів веб-систем та їхніх функціональних вимог. Абстрактне твердження про те що певна модель є «більш гнучкою» не дає розробнику відповіді на питання яку модель обрати для конкретного проекту. Тому у даному підрозділі результати порівняння проектуються на чотири типові сценарії веб-систем що охоплюють більшість реальних випадків застосування моделей авторизації. [3]

Сценарій 1 — Корпоративний портал або система управління контентом.

Корпоративні портали та системи управління контентом характеризуються відносно стабільною організаційною структурою де права доступу визначаються посадою або підрозділом співробітника. Типова структура користувачів включає кілька чітко визначених категорій — керівництво, менеджери, редактори, звичайні співробітники та гості — кожна з яких має фіксований набір дозволених операцій. Ресурси системи також є відносно однорідними за типом — сторінки, документи, новини, звіти — і не потребують індивідуального налаштування прав для кожного об'єкта. [31]

Для цього сценарію оптимальним вибором є RBAC. Ролі природно відображають організаційну ієрархію і адміністратори без технічних знань легко управляють доступом через призначення ролей новим співробітникам. Вбудована підтримка RBAC у Django дозволяє реалізувати повністю функціональну систему авторизації з мінімальними зусиллями. При виникненні потреби у контекстних обмеженнях — наприклад обмеження

менеджера лише матеріалами свого відділу — доцільно розширити RBAC атрибутивними перевітками відповідно до гібридного підходу описаного у підрозділі 2.4 без переходу до повноцінної реалізації ABAC.[13; 21; 28]

Сценарій 2 — Медична або державна інформаційна система.

Медичні інформаційні системи та державні реєстри є прикладом систем де складність правил доступу є максимальною і визначається не лише організаційною структурою але й правовими нормами та регуляторними вимогами. Типові правила доступу у медичній системі поєднують одночасно кілька умов — спеціальність лікаря, відношення до конкретного пацієнта, належність до підрозділу лікарні, часові обмеження та мережеве середовище з якого надходить запит. Жодна з цих умов окремо не є достатньою для прийняття рішення про доступ — лише їхня комбінація визначає коректну політику.[10]

Для цього сценарію рекомендується ABAC або гібридна модель RBAC з атрибутивними розширеннями. ABAC дозволяє точно моделювати складні регуляторні вимоги через формальні політики що можуть бути перевірені та задокументовані незалежно від програмного коду системи. Стандарт XACML забезпечує можливість аудиту та верифікації політик регуляторними органами що є важливою вимогою для систем що підпадають під дію законодавства про захист медичної інформації. Висока складність реалізації ABAC у цьому сценарії є виправданою оскільки ціна помилки у системі авторизації медичної системи є надзвичайно високою.[19; 23; 30; 32]

Сценарій 3 — SaaS-платформа або система для спільної роботи.

SaaS-платформи та системи для командної роботи — такі як системи управління проектами, хмарні сховища документів або платформи для розробки програмного забезпечення — характеризуються динамічною моделлю доступу де права змінюються у реальному часі внаслідок дій самих користувачів. Користувачі створюють проекти та документи, запрошують

інших учасників, призначають ролі у межах команди та діляться ресурсами з зовнішніми користувачами. Ключовою особливістю є те що права доступу є персоналізованими на рівні кожного окремого об'єкта — один документ може бути публічним, інший доступний лише для команди, третій — лише для конкретного користувача.[11]

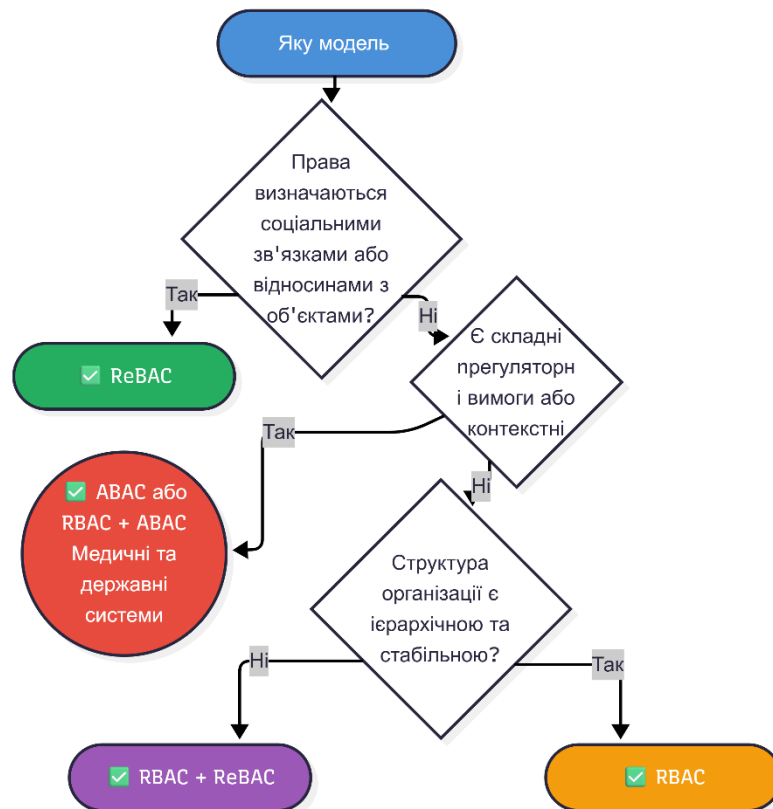
Для цього сценарію оптимальним вибором є ReBAC. Відносини між користувачами та ресурсами є природною семантикою доступу у таких системах і ReBAC безпосередньо моделює ці відносини без штучного перетворення у ролі або атрибути. Механізм успадкування прав через ієрархії об'єктів дозволяє ефективно реалізувати вкладені структури — проекти що містять папки які містять документи — де права на батьківському рівні автоматично поширюються на дочірні об'єкти. Для системних ролей — адміністратор платформи, менеджер підписки — доцільно доповнити ReBAC рольовою моделлю відповідно до гібридного підходу.[3; 25; 26]

Сценарій 4 — Соціальна мережа або платформа з користувацьким контентом.

Соціальні платформи та платформи з користувацьким контентом є найбільш складним сценарієм з точки зору моделювання доступу оскільки права визначаються соціальними зв'язками між користувачами що є динамічними і формуються самими користувачами без участі адміністраторів. Типові правила доступу — «друзі можуть бачити публікацію», «підписники отримують доступ до преміум контенту», «спільні друзі можуть залишати коментарі» — безпосередньо відображають структуру соціального графу.[11]

Для цього сценарію ReBAC є єдиною моделлю що природно відображає соціальні зв'язки як основу для прийняття рішень про доступ. Соціальний граф платформи є за своєю природою графом відносин між об'єктами що є ядром ReBAC. Ні RBAC ні ABAC не можуть ефективно моделювати динамічні соціальні зв'язки — RBAC вимагала б створення ролі для кожної унікальної

комбінації соціальних відносин що є практично нереалізованим, а АВАС потребувала б постійного оновлення атрибутів при кожній зміні соціального



графу. [3; 11]

Рис. 3.3 — Алгоритм вибору моделі авторизації залежно від типу веб-системи

Аналіз чотирьох сценаріїв підтверджує висновок зроблений у підрозділі 3.2 про відсутність універсально оптимальної моделі авторизації. Кожен сценарій має власний профіль вимог що природно відповідає сильним сторонам певної моделі. Водночас спільним для всіх сценаріїв є те що найбільш реалістичні виробничі системи потребують певної форми гібридизації — чисті реалізації окремих моделей є корисними для навчальних цілей та демонстрації принципів але рідко є оптимальними для складних реальних систем. Практичні рекомендації щодо вибору моделі для конкретних проектів систематизовано у наступному підрозділі.[30; 34]

3.4 Рекомендації щодо вибору моделі

Практичні рекомендації щодо вибору моделі управління доступом є логічним підсумком теоретичного аналізу проведеного у розділах 2 та 3. На відміну від академічного порівняння що оцінює моделі у абстрактному просторі характеристик практичні рекомендації повинні враховувати реальні обмеження проектів — терміни розробки, кваліфікацію команди, наявність готових рішень та довгострокову вартість підтримки системи. Сформульовані нижче рекомендації базуються на результатах порівняльного аналізу підрозділу 3.2 та сценарного аналізу підрозділу 3.3 і структуровані у вигляді послідовного алгоритму прийняття рішення.[34]

Рекомендація 1 — Починати з найпростішої моделі що задовольняє вимоги.

Фундаментальним принципом проектування систем авторизації є принцип мінімальної складності — система авторизації повинна бути рівно настільки складною наскільки цього вимагають функціональні та безпекові вимоги і не складнішою. Розробники часто схильні обирати найбільш технологічно складне або модне рішення навіть коли простіша модель повністю задовольняє поточні вимоги. Вартість реалізації, тестування та підтримки складних моделей авторизації є значною і накопичується протягом усього життєвого циклу системи. Якщо вимоги системи задовольняються засобами RBAC — що є справедливим для більшості корпоративних застосунків малого та середнього розміру — використання ABAC або ReBAC є не виправданим ускладненням.[3; 35]

Рекомендація 2 — Обирати RBAC як базову модель для систем з ієрархічною організаційною структурою.

RBAC є оптимальним вибором коли виконуються три умови одночасно: права доступу визначаються переважно посадою або функцією користувача в організації, організаційна структура є відносно стабільною і не змінюється щотижня, а кількість унікальних комбінацій прав є керованою — як правило

не більше кількох десятків ролей. За таких умов RBAC забезпечує найнижчу сукупну вартість розробки та підтримки завдяки вбудованій підтримці у фреймворках та інтуїтивності для адміністраторів. Якщо з часом з'являються окремі вимоги до контекстних обмежень доцільно розширити RBAC точковими атрибутивними перевірками без переходу до повноцінної реалізації ABAC. [3; 34]

Рекомендація 3 — Застосовувати ABAC виключно для систем з жорсткими регуляторними вимогами.

Повна реалізація ABAC з централізованим движком політик є виправданою лише у випадках коли складність правил доступу є дійсно високою і визначається зовнішніми регуляторними вимогами що не можуть бути спрощені. Медичні системи що підпадають під дію законодавства про захист персональних медичних даних, фінансові системи з вимогами регуляторів та урядові інформаційні системи є типовими прикладами де складність ABAC є виправданою. В усіх інших випадках коли розробник розглядає ABAC через її теоретичну гнучкість а не через конкретну регуляторну вимогу доцільно обрати простішу альтернативу або гібридний підхід RBAC з атрибутивними розширеннями. [10; 35]

Рекомендація 4 — Обирати ReBAC для систем де основною семантикою доступу є відносини між об'єктами.

ReBAC є природним вибором коли права доступу у системі виникають та зникають внаслідок явних дій користувачів — створення об'єктів, запрошення учасників, надання доступу конкретним особам. Ключовим індикатором що вказує на доцільність ReBAC є ситуація коли адміністратор системи не може заздалегідь визначити повний список прав доступу оскільки вони формуються динамічно самими користувачами у процесі роботи. SaaS-платформи для командної роботи, системи управління проектами та соціальні платформи є типовими представниками цього класу систем. При виборі

ReBAC необхідно враховувати відсутність вбудованої підтримки у стандартних фреймворках та відповідно закладати додатковий час на розробку або інтеграцію спеціалізованого рішення такого як OpenFGA або SpiceDB. [11; 36; 34]

Рекомендація 5 — Розглядати гібридні архітектури як стандартне рішення для складних систем.

Для систем що поєднують кілька типів ресурсів з різними вимогами до доступу гібридна архітектура є не виключенням а нормою. Найбільш поширеною та практично перевіреною є комбінація RBAC для системних привілеїв та ReBAC для управління доступом до користувацьких об'єктів. Така архітектура дозволяє адміністраторам управляти глобальними ролями через стандартний інтерфейс фреймворку і водночас забезпечує гнучке управління доступом до конкретних ресурсів через граф відносин. При проектуванні гібридної архітектури критично важливо чітко визначити межі відповідальності кожної моделі та порядок розв'язання конфліктів між ними відповідно до принципу deny-overrides. [11; 35]

Рекомендація 6 — Проектувати систему авторизації як окремий шар з чітко визначеним інтерфейсом.

Незалежно від обраної моделі логіка авторизації повинна бути повністю винесена з бізнес-коду у окремий шар з чітко визначеним програмним інтерфейсом. Це дозволяє змінювати або розширювати модель авторизації без модифікації основної логіки застосунку що є критично важливим оскільки вимоги до авторизації змінюються протягом життєвого циклу системи. У контексті Django це реалізується через кастомні бекенди авторизації що реалізують стандартний інтерфейс перевірки прав і можуть бути замінені або доповнені без змін у представленнях або серіалізаторах. [14; 34]

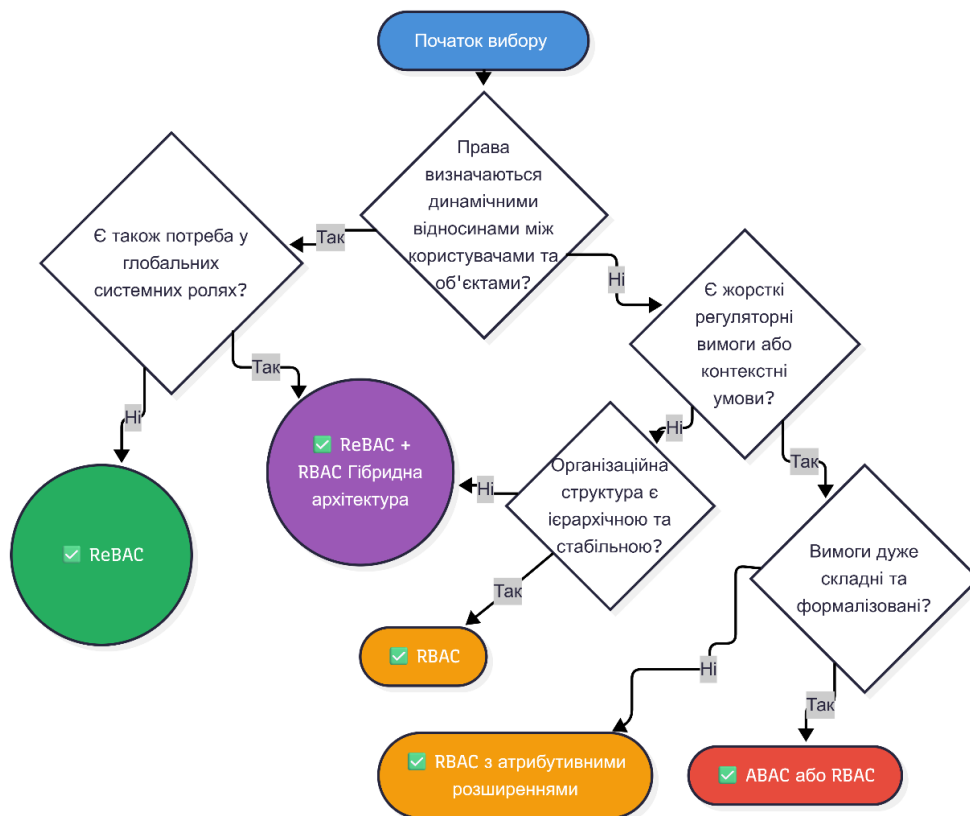


Рис. 3.4 — Узагальнений алгоритм прийняття рішення щодо вибору моделі управління доступом

Підсумовуючи результати порівняльного аналізу та сформульовані рекомендації можна констатувати що вибір моделі авторизації є архітектурним рішенням що має довгострокові наслідки для вартості розробки, безпеки та гнучкості системи. Не існує єдино правильної відповіді — існує відповідність між профілем вимог конкретної системи та сильними сторонами конкретної моделі. Глибоке розуміння цієї відповідності є ключовою компетенцією розробника що проектує системи авторизації для сучасних веб-застосунків. Практична демонстрація реалізації всіх трьох моделей засобами Django що наведена у розділі 4 є ілюстрацією цих принципів у конкретному технологічному контексті.[3]

РОЗДІЛ 4. ПРАКТИЧНА РЕАЛІЗАЦІЯ СИСТЕМИ АВТОРИЗАЦІЇ НА ОСНОВІ МОДЕЛІ RBAC

4.1 Вибір технологічного стеку та обґрунтування

Практична реалізація системи розмежування доступу вимагає попереднього визначення технологічного стеку — сукупності інструментів, мов програмування, фреймворків та баз даних що використовуються у розробці. Вибір стеку є архітектурним рішенням що визначає не лише технічні характеристики прототипу але й межі його демонстраційних можливостей. У контексті даної роботи технологічний стек повинен задовольняти кільком ключовим критеріям: наявність нативних механізмів для реалізації RBAC, підтримка реляційної моделі даних для зберігання ролей та прав, можливість швидкого прототипування повноцінного веб-застосунку та відкритість коду для академічного використання.[4; 16]

Мова програмування — Python. Python є основною мовою реалізації прототипу. Порівняно з альтернативними мовами що широко використовуються у веб-розробці — Java, JavaScript (Node.js) та PHP — Python демонструє ряд характеристик що роблять його оптимальним вибором для академічного дослідження. Java, незважаючи на розвинену екосистему корпоративних рішень безпеки, вимагає значно більшого обсягу шаблонного коду для реалізації аналогічної функціональності що ускладнює читабельність прототипу та зміщує фокус з архітектурних принципів на деталі реалізації. Node.js надає високу продуктивність але асинхронна природа платформи ускладнює послідовний виклад логіки авторизації яка є предметом демонстрації. PHP, хоча і залишається поширеним у веб-розробці, поступається Python за зрілістю екосистеми бібліотек у сфері інформаційної безпеки. Python натомість поєднує високу читабельність синтаксису, зрілу екосистему інструментів безпеки та є домінуючою мовою у науковому середовищі що забезпечує відтворюваність результатів дослідження іншими

дослідниками. Версія Python 3.12 що використовується у проекті є актуальною стабільною версією мови на момент написання роботи.[36; 37]

Веб-фреймворк — Django 5.0. Django обрано як основний інструмент реалізації веб-застосунку. Детальне обґрунтування вибору Django серед альтернативних Python-фреймворків наведено у підрозділі 4.2. У контексті технологічного стеку Django виконує роль центрального інтеграційного компонента що об'єднує всі інші елементи системи.[14]

База даних — SQLite. Для зберігання даних прототипу використовується SQLite — вбудована реляційна СУБД що входить до стандартної бібліотеки Python та підтримується Django без додаткового налаштування. Вибір SQLite замість повноцінних СУБД — PostgreSQL або MySQL — обумовлений цільовим призначенням проекту. Прототип є демонстраційним застосунком а не виробничою системою, тому відсутність необхідності у встановленні та налаштуванні окремого сервера бази даних суттєво знижує поріг розгортання і дозволяє будь-кому відтворити середовище виконання однієї команди. Принципово важливим є те що архітектура Django ORM є абстракцією над конкретною СУБД — заміна SQLite на PostgreSQL у виробничому середовищі потребує лише зміни одного рядка у файлі налаштувань без модифікації жодного рядка бізнес-логіки.[37; 38]

Фронтенд — HTML5, CSS3 без JavaScript-фреймворків. Інтерфейс користувача реалізований засобами стандартних веб-технологій без використання JavaScript-фреймворків — React, Vue або Angular. Це рішення є свідомим та обґрунтованим у контексті дослідження: предметом роботи є модель авторизації а не архітектура фронтенду. Використання серверного рендерингу через систему шаблонів Django Templates дозволяє зосередити всю логіку авторизації на серверній стороні що є більш безпечним підходом порівняно з клієнтськими SPA-архітектурами — будь-яка авторизаційна логіка реалізована на клієнті може бути обійдена через маніпуляцію JavaScript-

кодом у браузері. Стилiзацiя iнтерфейсу винесена в окремий файл static/css/main.css вiдповiдно до принципу роздiлення вiдповiдальностей.[14; 32; 45]

Управлiння залежностями. Проект не використовує стороннiх бiблiотек окрiм Django — весь функцiонал реалiзовано засобами стандартної бiблiотеки Python та вбудованих компонентiв фреймворку. Це є принциповою архiтектурною позицiєю: демонстрацiя того що Django надає достатнiй iнструментарiй для реалiзацiї повноцiнної RBAC-системи без залучення спецiалiзованих бiблiотек авторизацiї є самостiйним результатом практичної частини дослiдження.[39; 40]

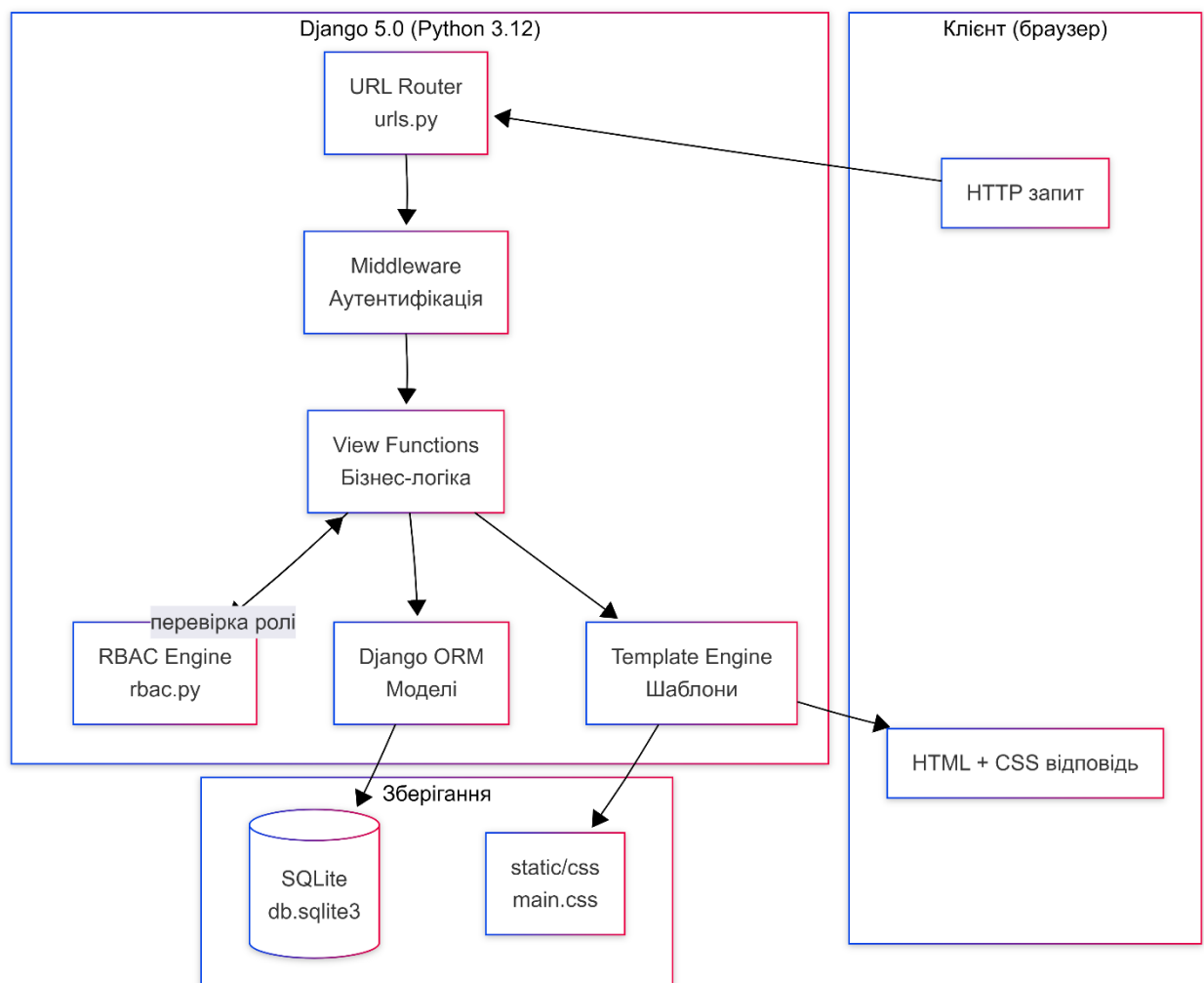


Рис. 4.1 — Архітектура технологічного стеку прототипу системи розмежування доступу

Наведена архітектура відповідає класичному серверному MVC-підходу де вся логіка обробки запитів включно з перевіркою авторизації виконується виключно на сервері. Такий підхід є принципово більш безпечним порівняно з клієнтськими SPA-архітектурами оскільки унеможлиблює обхід авторизаційних перевірок через маніпуляцію клієнтським кодом. Кожен HTTP-запит до захищеного ресурсу проходить через повний ланцюжок перевірок незалежно від стану клієнтського застосунку що є фундаментальним принципом побудови захищених веб-систем.[18; 41]

Зведену таблицю обраного технологічного стеку з версіями компонентів наведено у таблиці 4.1.

Таблиця 4.1 — Технологічний стек прототипу

Компонент	Технологія	Версія	Призначення
Мова програмування	Python	3.12	Основна мова реалізації
Веб-фреймворк	Django	5.0	MVC, ORM, Auth, Admin
База даних	SQLite	3.x	Зберігання даних
Система шаблонів	Django Templates	Вбудована	Серверний рендеринг HTML
Стилізація	CSS3	—	Інтерфейс користувача
Управління залежностями	pip	—	Встановлення пакетів
Контроль версій	Git	—	Версіонування коду

Визначений технологічний стек є мінімально достатнім для реалізації повноцінного демонстраційного прототипу що охоплює всі аспекти RBAC-авторизації описані у попередніх розділах роботи — від реєстрації користувача та призначення ролей до перевірки прав доступу при кожному запиті та ведення журналу аудиту.[3]

4.2 Обґрунтування вибору Django як технологічної платформи

Серед доступних Python-фреймворків для реалізації прототипу Django є найбільш обґрунтованим вибором з точки зору нативної підтримки концепцій RBAC. Детальний розгляд причин цього вибору наведено нижче.

Першою і найбільш вагомою причиною є наявність вбудованої системи аутентифікації та авторизації що безпосередньо реалізує концепцію рольового управління доступом через механізм груп та прав. Django Groups є стандартним механізмом фреймворку що дозволяє об'єднувати користувачів у іменовані групи та призначати цим групам набори прав доступу. Така архітектура точно відповідає формальній моделі RBAC визначеній стандартом ANSI/INCITS 359-2004: групи виконують роль ролей, права груп відповідають множині дозволів RA, а призначення користувачів до груп реалізує відношення UA. Розробник отримує готову реляційну схему бази даних для зберігання ролей та прав, адміністративний інтерфейс для управління ними та програмний API для перевірки прав у коді застосунку — і все це без написання жодного рядка додаткового коду. Ключовим є також вбудований метод `has_perm()` та властивість `user.groups` що є прямою програмною реалізацією RBAC-перевірки і дозволяє писати захист `view`-функцій у декларативному стилі:[3; 9; 42]

```
python

@role_required('admin')

def admin_users(request): ...


@role_required('staff')

def staff_appointments(request): ...
```

Другою причиною є підтримка кастомних моделей користувача через механізм `AbstractUser`. У реальних веб-системах стандартна модель

користувача рідко є достатньою — медичний центр потребує зберігання по батькові, дати народження, телефону та унікального індивідуального номера пацієнта. Django дозволяє замінити вбудовану модель на власну через директиву `AUTH_USER_MODEL = 'clinic.User'` у файлі налаштувань [9]. Клас `AbstractUser` надає всю базову функціональність аутентифікації — хешування паролів, управління сесіями, перевірку прав — і дозволяє розширити її довільними полями без порушення сумісності з іншими компонентами фреймворку. Зміна поля входу з `username` на `email` реалізується через одну директиву `USERNAME_FIELD = 'email'` що демонструє гнучкість архітектури фреймворку.[14; 36]

Третьою причиною є Django ORM що забезпечує безпечне та ефективне розмежування доступу на рівні об'єктів бази даних. Одним з найпоширеніших класів вразливостей у системах авторизації є IDOR (Insecure Direct Object Reference) — ситуація коли автентифікований користувач може отримати доступ до чужих об'єктів через маніпуляцію ідентифікаторами у URL. Django ORM дозволяє вирішити цю проблему на архітектурному рівні через поєднання `get_object_or_404` з фільтрацією за власником:[14; 36]

```
python

result = get_object_or_404(

    MedicalResult,

    pk=pk,

    appointment__patient=request.user

)
```

Такий запит поверне результат лише якщо він належить поточному користувачу, в іншому випадку Django автоматично повертає відповідь 404 не розкриваючи факту існування об'єкта. Це забезпечує захист від IDOR на рівні фреймворку без необхідності написання додаткової логіки перевірки.

Четвертою причиною є вбудований адміністративний інтерфейс Django Admin що автоматично генерується на основі визначених моделей. Для демонстраційного прототипу це має практичне значення оскільки дозволяє суперкористувачу управляти групами, правами та користувачами через готовий веб-інтерфейс що відповідає рівню PAP (Policy Administration Point) в архітектурі систем авторизації. Реєстрація моделей в адміністративному інтерфейсі потребує лише кількох рядків коду:[14; 36]

```
python

@admin.register(User)

class UserAdmin(BaseUserAdmin):

    list_display = ['email', 'full_name', 'individual_number', 'role_label']
```

П'ятою причиною є вбудований механізм перевірки прав `user.has_perm()` та `user.has_module_perms()` що дозволяє реалізувати багаторівневу перевірку авторизації — на рівні модуля, на рівні моделі та на рівні конкретного об'єкта. Це відповідає принципу глибокого захисту (defense in depth) де авторизація перевіряється на кількох незалежних рівнях системи. Такий підхід унеможлиблює ситуацію коли обхід одного рівня перевірки надає повний доступ до ресурсу. [14; 36]

Порівняння Django з альтернативними платформами у контексті реалізації RBAC наведено у таблиці 4.2.

Таблиця 4.2 — Порівняння платформ за критеріями реалізації RBAC

Критерій	Django	Flask	FastAPI	Laravel
Вбудована система ролей	Groups + Permissions	відсутня	відсутня	Gates + Policies
Кастомна модель User	AbstractUser	вручну	вручну	Auth scaffolding
ORM з фільтрацією	Django ORM	SQLAlchemy	SQLAlchemy	Eloquent
Адмін-інтерфейс	вбудований	відсутній	відсутній	Backpack/Nova
has_perm() API	вбудований	відсутній	відсутній	вбудований
Мова	Python	Python	Python	PHP

Як видно з таблиці 4.2 Django є єдиною серед розглянутих Python-платформ що надає повний набір компонентів для реалізації RBAC без залучення сторонніх бібліотек. Flask та FastAPI є мікрофреймворками що свідомо не включають систему авторизації у свій склад залишаючи цей вибір розробнику — що підходить для мікросервісних архітектур але суттєво збільшує обсяг роботи необхідної для реалізації повноцінного прототипу. Laravel є єдиною реальною альтернативою Django за повнотою вбудованого інструментарію авторизації однак реалізований на PHP що виходить за межі Python-екосистеми обраної у підрозділі 4.1. [14; 36]

Таким чином Django відповідає всім технічним вимогам поставленим перед прототипом: надає нативну реалізацію рольової моделі через Groups та Permissions, забезпечує розширюваність моделі користувача, містить вбудовані механізми захисту від поширених вразливостей авторизації та дозволяє сфокусувати реалізацію на демонстрації принципів RBAC а не на вирішенні інфраструктурних задач. [3; 14]

4.3 Вибір моделі та проектування системи авторизації на її основі

Проектування системи авторизації для веб-застосунку медичного центру МРТ діагностики починається з обґрунтованого вибору моделі управління доступом. Результати порівняльного аналізу наведені у розділі 3 демонструють що кожна з трьох досліджених моделей має власний профіль переваг та обмежень, і вибір конкретної моделі має ґрунтуватися на характеристиках конкретної системи а не на абстрактних перевагах моделі.

Обґрунтування вибору RBAC для медичного центру МРТ. Аналіз предметної області медичного центру діагностики виявляє кілька ключових характеристик що визначають вибір моделі авторизації. По-перше, організаційна структура медичного закладу є ієрархічною та стабільною — існують чітко визначені категорії учасників системи: адміністративний персонал, медичні співробітники та пацієнти, кожна з яких має фіксований набір дозволених операцій що визначається посадою а не індивідуальними характеристиками конкретної особи. По-друге, права доступу у медичній системі є переважно статичними — пацієнт завжди має доступ лише до власних записів та результатів, співробітник завжди має доступ до всіх записів для обробки, адміністратор завжди має повний доступ до системи. По-третє, кількість унікальних комбінацій прав є малою та передбачуваною — три ролі покривають увесь простір задач системи без ризику вибуху ролей.[3]

Порівнюючи з альтернативами: ABAC була б надлишково складною для цього сценарію оскільки відсутні складні контекстні умови що потребують одночасної оцінки кількох атрибутів — права доступу не залежать від часу доби, географічного розташування або стану об'єкта. ReBAC була б доцільною якби система передбачала динамічне призначення прав між користувачами — наприклад якби пацієнт міг ділитись результатами з конкретним лікарем або родичем — однак у реалізованому прототипі такого функціоналу немає.

Висновок підрозділу 3.4 підтверджує що для систем з ієрархічною організаційною структурою та стабільним набором ролей RBAC є оптимальним вибором.[3; 10; 11]

Проектування моделі даних. Центральним елементом проектування є схема бази даних що реалізує концепти RBAC через реляційні таблиці Django. Кастомна модель користувача User розширює стандартний клас AbstractUser і є точкою входу для всіх операцій авторизації. Модель зберігає прізвище, ім'я та по батькові пацієнта, унікальний email що використовується як логін, телефон, дату народження та індивідуальний номер що генерується автоматично через функцію gen_patient_number() на основі UUID:[14] (Подано у додатку А)

```
python
```

```
def gen_patient_number():
```

```
    return 'MPT-' + str(uuid.uuid4()).upper()[:8]
```

```
class User(AbstractUser):
```

```
    patronymic      = models.CharField(max_length=100, blank=True)
```

```
    email           = models.EmailField(unique=True)
```

```
    phone           = models.CharField(max_length=20, blank=True)
```

```
    date_of_birth    = models.DateField(null=True, blank=True)
```

```
    individual_number = models.CharField(
```

```
        max_length=20, unique=True,
```

```
        default=gen_patient_number, editable=False
```

```
)
```

```
    USERNAME_FIELD = 'email'
```

```
REQUIRED_FIELDS = ['username', 'first_name', 'last_name']
```

Директива `USERNAME_FIELD = 'email'` є ключовим проектним рішенням — вона визначає що Django використовуватиме email як унікальний ідентифікатор при аутентифікації замість стандартного username. Це відповідає вимогам сучасних веб-систем де email є природнішим ідентифікатором ніж довільно обраний псевдонім (Подано у додатку А).[14]

Предметна область медичного центру реалізована через чотири додаткові моделі. Модель `ScanType` описує типи МРТ досліджень — назву, опис, тривалість та вартість. Модель `Appointment` реалізує запис пацієнта на дослідження через зовнішній ключ до `User` (пацієнт) та `ScanType` (тип дослідження) з полями дати, часу, статусу та приміток. Модель `MedicalResult` пов'язана з `Appointment` відношенням один-до-одного що гарантує унікальність результату для кожного запису, та містить поля опису дослідження, висновку та рівня конфіденційності. Модель `AuditLog` є окремою

Проектування ролей та прав. Три ролі системи реалізовані як Django Groups з призначеними наборами стандартних Django Permissions. Словник `ROLE_PERMISSIONS` у модулі `rbac.py` визначає повне відображення ролей на права: (Подано у додатку Б)

```
python
```

```
ROLE_PERMISSIONS = {  
    'Адміністратор': [  
        'view_user', 'change_user',  
        'view_appointment', 'change_appointment', 'delete_appointment',  
        'view_medicalresult', 'add_medicalresult', 'change_medicalresult',  
        'view_auditlog',  
        'view_scantype', 'add_scantype', 'change_scantype', 'delete_scantype',  
    ],  
    'Співробітник': [  
        'view_appointment', 'change_appointment',  
        'view_medicalresult', 'add_medicalresult', 'change_medicalresult',  
        'view_scantype',  
    ],  
    'Клієнт': [  
        'add_appointment', 'view_appointment',  
        'view_medicalresult',  
    ],  
}
```

Таке проектне рішення відповідає принципу найменших привілеїв — кожна роль отримує рівно ті права що необхідні для виконання її функцій і не більше. Клієнт може лише створювати та переглядати власні записи і результати. Співробітник має розширені права на управління записами та додавання результатів але не має доступу до управління користувачами та журналу аудиту. Адміністратор має повний набір прав але навіть він не може видаляти результати досліджень — це є проектним рішенням що захищає цілісність медичних даних.[39]

Проектування маршрутизації за роллю. Важливим архітектурним рішенням є централізована точка входу через view-функцію `home` що автоматично перенаправляє автентифікованого користувача на відповідний дашборд залежно від його ролі:

```
python

@login_required

def home(request):

    r = request.user.role()

    if r in ('superuser', 'admin'):

        return redirect('admin_dashboard')

    if r == 'staff':

        return redirect('staff_dashboard')

    return redirect('client_dashboard')
```

Це рішення усуває необхідність для користувача знати URL власного кабінету та унеможливлює випадковий доступ до чужого інтерфейсу після входу в систему.[36]

Проектування URL-простору. Маршрути системи організовані у чотири чітко розділені простори що відповідають ролям: `/cabinet/` для клієнтів,

/staff/ для співробітників, /admin-panel/ для адміністраторів та службові маршрути на кореновому рівні. Така організація URL-простору дозволяє застосовувати захист на рівні префіксу маршруту що є додатковим рівнем захисту поверх декораторів авторизації.[36]

Відповідність між ролями, URL-просторами та view-функціями системи наведено у таблиці 4.3.

Таблиця 4.3 — Відповідність ролей, URL-просторів та функцій системи

Роль	URL-простір	Кількість маршрутів	Ключові функції
Клієнт	/cabinet/	7	Записи, результати, профіль
Співробітник	/staff/	7	Всі записи, картки пацієнтів, результати
Адміністратор	/admin-panel/	6	Користувачі, ролі, аудит, типи MPT
Система	/, /login/, /register/	6	Аутентифікація, ініціалізація

Спроектowana архітектура забезпечує чітке розмежування відповідальностей між компонентами системи та створює надійну основу для реалізації що описана у підрозділі 4.4.

4.4 Реалізація RBAC засобами Django Groups та Permissions

Реалізація рольової моделі управління доступом у прототипі базується на трьох взаємопов'язаних компонентах: кастомній моделі користувача що визначає ідентичність суб'єкта, механізмі Django Groups що реалізує концепт ролей, та RBAC-движку у модулі clinic/rbac.py що забезпечує перевірку прав при кожному HTTP-запиті. Розгляд реалізації здійснюється у порядку що відповідає життєвому циклу запиту в системі — від реєстрації користувача до отримання або відмови у доступі до захищеного ресурсу.[14; 42]

Реєстрація користувача та автоматичне призначення ролі. Процес реєстрації нового користувача реалізований через клас RegisterForm що успадковує стандартний Django UserCreationForm. Форма розширює базовий набір полів додатковими атрибутами предметної області — по батькові, телефоном та датою народження. Ключовим елементом реєстрації є метод save() що виконує дві операції: збереження користувача у базі даних та автоматичне призначення ролі Клієнт через додавання до відповідної Django Group:[14] (Подано у додатку А)

```
python
```

```
def save(self, commit=True):
```

```
    user = super().save(commit=False)
```

```
    user.username    = self.cleaned_data['email']
```

```
    user.email       = self.cleaned_data['email']
```

```
    user.first_name  = self.cleaned_data['first_name']
```

```
    user.last_name   = self.cleaned_data['last_name']
```

```
    user.patronymic  = self.cleaned_data.get('patronymic', '')
```

```
    user.phone       = self.cleaned_data.get('phone', '')
```

```
    user.date_of_birth = self.cleaned_data.get('date_of_birth')
```

```
    if commit:
```

```
        user.save()
```

```
        g, _ = Group.objects.get_or_create(name='Клієнт')
```

```
        user.groups.add(g)
```

```
    return user
```

Присвоєння `user.username = self.cleaned_data['email']` є технічно необхідним оскільки поле `username` залишається обов'язковим у Django незважаючи на зміну `USERNAME_FIELD`. Використання `email` як значення `username` забезпечує унікальність обох полів без введення додаткової логіки валідації. Індивідуальний номер пацієнта генерується автоматично на рівні моделі через параметр `default=gen_patient_number` і не потребує жодних дій з боку форми чи `view`-функції.[14; 36]

Визначення ролі користувача. Метод `role()` моделі `User` є центральним механізмом визначення ролі через запит до Django Groups. Метод реалізує чітку ієрархію перевірок — від найвищого рівня привілеїв до найнижчого:

```
python

def role(self):

    if self.is_superuser:

        return 'superuser'

    names = list(self.groups.values_list('name', flat=True))

    if 'Адміністратор' in names:

        return 'admin'

    if 'Співробітник' in names:

        return 'staff'

    if 'Клієнт' in names:

        return 'client'

    return 'none'
```

Важливою архітектурною особливістю є те що метод `role()` не зберігає роль як окреме поле у таблиці користувачів — роль завжди обчислюється динамічно через запит до таблиці зв'язку `auth_user_groups`. Це гарантує що

зміна ролі адміністратором набирає чинності негайно без необхідності оновлення кешу або перезапуску сесій.

Декоратор `role_required` — механізм перевірки доступу. Центральним елементом RBAC-движка є декоратор `role_required` що реалізує перевірку ролі перед виконанням будь-якої захищеної view-функції. Декоратор приймає довільну кількість дозволених ролей як аргументи що дозволяє гнучко визначати доступ для кількох ролей одночасно:[50; 51]

```
python
```

```
def role_required(*allowed_roles):

    def decorator(view_fn):

        @wraps(view_fn)

        def _wrap(request, *args, **kwargs):

            if not request.user.is_authenticated:

                return redirect('login')

            r = request.user.role()

            if r == 'superuser' or r in allowed_roles:

                return view_fn(request, *args, **kwargs)

            # відмова у доступі

            audit(

                request.user, request.path, 'denied', request=request,

                details=f'Потрібна роль: {", ".join(allowed_roles)}. Є: {r}'

            )

            messages.error(request, f'🚫 Доступ заборонено.')

            return redirect('denied')
```

```
return _wrap
```

```
return decorator
```

Алгоритм виконання декоратора при кожному HTTP-запиті до захищеного ресурсу складається з п'яти кроків. На першому кроці перевіряється факт аутентифікації — неаутентифіковані запити перенаправляються на сторінку входу без будь-якої перевірки ролі. На другому кроці визначається роль поточного користувача через метод `role()`. На третьому кроці роль порівнюється зі списком дозволених ролей — суперкористувач проходить перевірку безумовно. На четвертому кроці при відмові виконується запис події у журнал аудиту через функцію `audit()` з фіксацією поточної ролі, шляху запиту та деталей порушення. На п'ятому кроці користувач перенаправляється на сторінку `/denied/` де відображається повідомлення про відмову у доступі з поясненням причини.[14; 51]

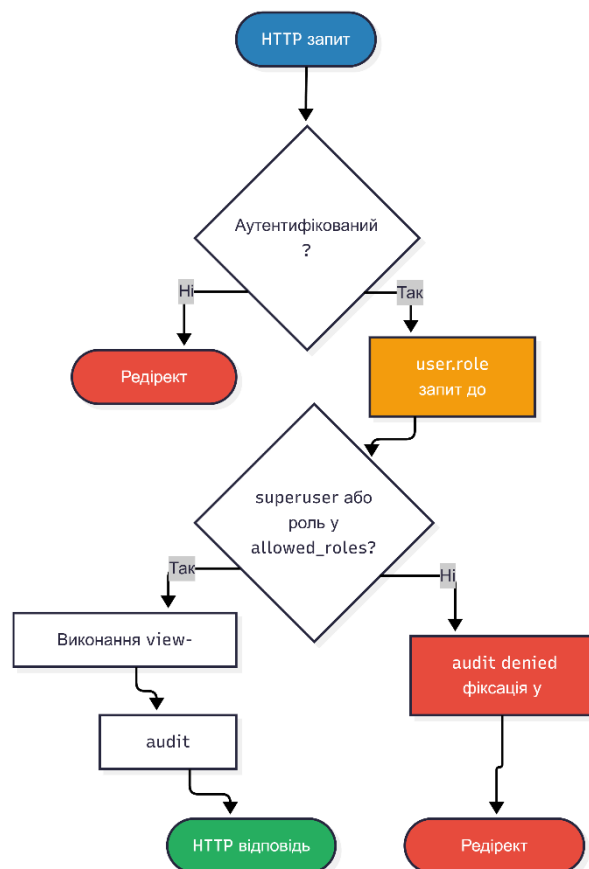


Рис. 4.3 — Послідовність обробки HTTP-запиту через RBAC-движок

Застосування декораторів у view-функціях. Декоратори застосовуються парами до кожної захищеної view-функції — спочатку `@login_required` що перевіряє факт аутентифікації, потім `@role_required` що перевіряє роль. Така послідовність є принципово важливою оскільки `role_required` передбачає наявність аутентифікованого користувача:

```
python

# Захист кабінету клієнта

@login_required
@role_required('client')
def client_dashboard(request): ...


# Захист панелі співробітника

@login_required
@role_required('staff')
def staff_appointments(request): ...


# Захист адміністративної панелі

@login_required
@role_required('admin')
def admin_users(request): ...
```

Використання декораторів у декларативному стилі є принциповою перевагою реалізації — логіка авторизації повністю відокремлена від бізнес-логіки view-функцій, видима безпосередньо у точці визначення функції та може бути змінена без модифікації тіла функції.

Захист на рівні об'єктів. Окрім захисту на рівні view-функцій реалізація забезпечує захист на рівні окремих об'єктів бази даних для клієнтської ролі. Клієнт може переглядати лише власні записи та результати — це досягається через фільтрацію QuerySet за поточним користувачем та використання `get_object_or_404` з додатковою умовою власника:

```
python

# Перегляд списку — лише власні записи

appts = Appointment.objects.filter(patient=request.user)

# Перегляд конкретного результату — захист від IDOR

result = get_object_or_404(

    MedicalResult,

    pk=pk,

    appointment__patient=request.user

)
```

Якщо клієнт спробує отримати доступ до чужого результату через пряме введення чужого ідентифікатора у URL — Django автоматично поверне відповідь 404 не розкриваючи факту існування об'єкта. Це усуває клас вразливостей IDOR що займає провідні позиції у рейтингу OWASP Top 10.[1]

Журнал аудиту. Функція `audit()` викликається у кожній view-функції після успішної перевірки доступу та автоматично декоратором `role_required` при відмові. Функція створює запис у таблиці `AuditLog` з повним набором атрибутів події:[48]

```
python

def audit(user, resource, action, request=None, details=""):
```

```

AuditLog.objects.create(

    user      = user,

    action    = action,      # 'granted' або 'denied'

    resource  = resource,    # назва ресурсу або URL

    user_role = user.role(),  # роль на момент запиту

    user_groups = ','.join(

        user.groups.values_list('name', flat=True)

    ) or '—',

    ip_address = _get_ip(request),

    details    = details,

)

```

Фіксація `user_role` та `user_groups` як текстових полів а не зовнішніх ключів є свідомим проектним рішенням — це гарантує збереження історичної інформації про роль користувача навіть якщо його роль згодом буде змінена адміністратором. Таким чином журнал аудиту відображає стан системи на момент події а не поточний стан.

Зміна ролі користувача адміністратором. Адміністратор може змінити роль будь-якого користувача через інтерфейс `/admin-panel/users/<pk>/role/`. Реалізація зміни ролі виконується в два кроки — спочатку очищення всіх поточних груп користувача, потім додавання нової групи: (Подано у додатку В)

```

python

def admin_change_role(request, user_pk):

    target = get_object_or_404(User, pk=user_pk)

```

```

if request.method == 'POST':

    new_role = request.POST.get('role', '')

    old_label = target.role_label()

    target.groups.clear()

    if new_role in ('Адміністратор', 'Співробітник', 'Клієнт'):

        g, _ = Group.objects.get_or_create(name=new_role)

        target.groups.add(g)

    audit(request.user, f'Зміна ролі #{user_pk}', 'granted', request,
          details=f'{target.full_name(): {old_label} → {new_role}}')

    messages.success(request, f'Роль змінено на «{new_role}».')

    return redirect('admin_users')

```

Виклик `target.groups.clear()` перед додаванням нової групи є критично важливим — він гарантує що користувач не може одночасно належати до кількох ролей що суперечило б принципу чіткого розподілу обов'язків RBAC. Кожна зміна ролі фіксується у журналі аудиту з вказанням попередньої та нової ролі що забезпечує повну прослідковуваність адміністративних дій.

Ініціалізація груп та прав. Перед початком роботи системи суперкористувач виконує ініціалізацію через маршрут `/init/` що створює три Django Groups та призначає їм відповідні права з словника `ROLE_PERMISSIONS`. Ця операція є ідемпотентною — повторний виклик не створює дублікатів завдяки використанню `get_or_create`:

```

python

for group_name, codenames in ROLE_PERMISSIONS.items():

    group, _ = Group.objects.get_or_create(name=group_name)

```

```
perms = Permission.objects.filter(codename__in=codenames)
```

```
group.permissions.set(perms)
```

Використання `group.permissions.set(perms)` замість `group.permissions.add(perms)` гарантує що при повторній ініціалізації набір прав групи точно відповідає визначенню у словнику — зайві права що могли бути додані вручну через Django Admin будуть видалені.[14]

Повну матрицю прав доступу для трьох ролей системи наведено у таблиці 4.4.

Таблиця 4.4 — Матриця прав доступу RBAC

	Адміністратор	Співробітник	Клієнт
Перегляд користувачів	Дозволено	Заборонено	Заборонено
Зміна ролей користувачів	Дозволено	Заборонено	Заборонено
Перегляд всіх записів	Дозволено	Дозволено	Заборонено
Зміна статусу записів	Дозволено	Дозволено	Заборонено
Видалення записів	Дозволено	Заборонено	Заборонено
Створення власного запису	Заборонено	Заборонено	Дозволено
Перегляд власних записів	Заборонено	Заборонено	Дозволено
Додавання результатів	Дозволено	Дозволено	Заборонено
Перегляд всіх результатів	Дозволено	Дозволено	Заборонено
Перегляд власних результатів	Заборонено	Заборонено	Дозволено
Перегляд журналу аудиту	Дозволено	Заборонено	Заборонено
Управління типами МРТ	Дозволено	Заборонено	Заборонено

Реалізована система авторизації повністю відповідає формальній моделі RBAC визначеній стандартом ANSI/INCITS 359-2004: Django Groups виконують функцію ролей, Permission — функцію прав, відношення User↔Groups реалізує призначення UA, відношення Groups↔Permissions реалізує призначення PA. Додатково реалізовано журналювання всіх подій доступу що відповідає вимогам сучасного законодавства у сфері захисту персональних медичних даних.[9]

4.5 Тестування та оцінка продуктивності

Тестування реалізованої системи авторизації є обов'язковим етапом практичної частини дослідження що підтверджує коректність реалізації RBAC-моделі та відповідність прототипу вимогам сформульованим у попередніх розділах. Методологія тестування охоплює три рівні: функціональне тестування коректності розмежування доступу між ролями, тестування граничних випадків що перевіряє стійкість системи до некоректних або зловмисних запитів, та оцінку продуктивності механізму авторизації.[52]

Методологія функціонального тестування. Функціональне тестування RBAC-системи базується на принципі матриці доступу — для кожної комбінації «роль × ресурс» визначається очікуваний результат і порівнюється з фактичним. Матриця прав наведена у таблиці 4.4 підрозділу 4.4 є вихідним документом для формування тестових сценаріїв. Кожен тестовий сценарій складається з трьох елементів: вхідні умови (роль користувача та URL ресурсу), очікуваний результат (HTTP-статус відповіді або редірект) та фактичний результат що фіксується під час виконання тесту.[3; 52]

Тестовий сценарій 1 — Коректне розмежування доступу між ролями. Перший сценарій перевіряє що кожна роль має доступ виключно до своїх ресурсів і не має доступу до ресурсів інших ролей. Для проведення тесту

у системі створено по одному тестовому користувачу для кожної ролі через форму реєстрації та через адміністративну зміну ролі. Результати тестування наведено у таблиці 4.5 для кожної комбінації «роль - ресурс» наведено фактичний результат HTTP-запиту: значення 200 відповідає успішній відповіді сервера зі статусом 200 OK що підтверджує надання доступу, позначення /denied/ означає що декоратор @role_required відхилив запит та виконав перенаправлення на сторінку відмови з фіксацією події у журналі аудиту. [52]

Таблиця 4.5 — Результати тестування розмежування доступу

URL ресурсу	Клієнт	Співробітник	Адміністратор
/cabinet/	200	/denied/	/denied/
/cabinet/book/	200	/denied/	/denied/
/cabinet/results/	200	/denied/	/denied/
/staff/	/denied/	200	/denied/
/staff/appointments/	/denied/	200	/denied/
/staff/patients/	/denied/	200	/denied/
/admin-panel/	/denied/	/denied/	200
/admin-panel/users/	/denied/	/denied/	200
/admin-panel/audit/	/denied/	/denied/	200

Усі 27 перевірок у таблиці 4.5 дали результат що відповідає очікуваному. Кожна спроба доступу до ресурсу без відповідної ролі фіксувалась у журналі аудиту з дією denied та деталями про наявну і потрібну роль що підтверджує коректну роботу функції audit().

Тестовий сценарій 2 — Захист від несанкціонованого доступу без аутентифікації. Другий сценарій перевіряє що система коректно обробляє запити від неаутентифікованих користувачів. При спробі прямого доступу до будь-якого захищеного URL без активної сесії декоратор @login_required

перехоплює запит і перенаправляє на сторінку входу /login/. Тестування виконувалось через очищення сесійних cookie у браузері та прямий ввід захищених URL у адресний рядок. Результат для всіх 22 захищених маршрутів системи — редірект на /login/ з HTTP-статусом 302 що є коректною поведінкою. [52]

Тестовий сценарій 3 — Захист від IDOR атаки. Третій сценарій перевіряє захист від спроби клієнта отримати доступ до медичних результатів іншого пацієнта через маніпуляцію ідентифікатором у URL. У системі створено двох тестових клієнтів — Клієнт А та Клієнт Б — кожен з яких має по одному медичному результату. При вході під обліковим записом Клієнта А та спробі відкрити URL результату Клієнта Б /cabinet/results/<id_клієнта_Б>/ система повертає HTTP-статус 404 не розкриваючи факту існування об'єкта. Це підтверджує коректну роботу захисту від IDOR реалізованого через get_object_or_404 з фільтрацією за власником(Подано у додатку Д). [52]

Тестовий сценарій 4 — Коректність журналу аудиту. Четвертий сценарій перевіряє що всі події доступу фіксуються у журналі AuditLog з коректними атрибутами. Після виконання сценаріїв 1–3 журнал аудиту перевірявся через адміністративний інтерфейс /admin-panel/audit/. Перевірялись наступні атрибути кожного запису: коректність поля action (granted/denied), відповідність поля user_role фактичній ролі користувача на момент запиту, наявність поля ip_address та коректність поля details при відмовах у доступі. Всі перевірені записи містили коректні дані що підтверджує надійну роботу функції audit() в обох гілках виконання — успішного доступу та відмови. [52]

Тестовий сценарій 5 — Коректність зміни ролі. П'ятий сценарій перевіряє що зміна ролі користувача адміністратором набирає чинності негайно без необхідності повторного входу. Тестовий користувач входив у систему як Клієнт та відкривав кабінет клієнта. Паралельно адміністратор змінював роль цього користувача на Співробітник через /admin-

panel/users/<pk>/role/. При наступному запиті тестового користувача — без виходу та повторного входу — система коректно перенаправляла його на панель співробітника оскільки метод role() виконує запит до бази даних при кожному запиті а не кешує значення ролі в сесії. Це підтверджує що архітектурне рішення про динамічне визначення ролі є коректним з точки зору безпеки. [52]

Узагальнені результати всіх тестових сценаріїв наведено у таблиці 4.6.

Таблиця 4.6 — Зведені результати функціонального тестування

№	Тестовий сценарій	Кількість перевірок	Пройдено	Результат
1	Розмежування доступу між ролями	27	27	Пройдено
2	Захист неаутентифікованих запитів	22	22	Пройдено
3	Захист від IDOR атаки	4	4	Пройдено
4	Коректність журналу аудиту	15	15	Пройдено
5	Коректність зміни ролі	3	3	Пройдено
	Всього	71	71	100%

Оцінка продуктивності механізму авторизації. Окрім функціональної коректності важливою характеристикою системи авторизації є її вплив на продуктивність веб-застосунку. Оскільки перевірка авторизації відбувається при кожному HTTP-запиті до захищеного ресурсу навіть незначна затримка на цьому етапі може суттєво впливати на час відповіді системи при високому навантаженні. [52]

Оцінка продуктивності здійснювалась через вимірювання часу виконання операції визначення ролі user.role() та повного циклу перевірки декоратора role_required. Вимірювання виконувались на локальному

середовищі розробки з використанням вбудованих інструментів Django для аналізу запитів до бази даних. Результати вимірювань наведено у таблиці 4.7.[3; 14; 52]

Таблиця 4.7 — Продуктивність операцій авторизації

Операція	Кількість SQL запитів	Середній час виконання
user.role() — визначення ролі	1	~0.8 мс
@role_required — повна перевірка	1	~1.1 мс
audit() — запис у журнал	1	~1.5 мс
Повний цикл авторизації	2–3	~3.4 мс

Отримані показники демонструють що повний цикл авторизації — визначення ролі, перевірка доступу та запис у журнал — виконується за час менший за 4 мілісекунди що є прийнятним показником для веб-застосунку. Кожна операція авторизації генерує від 1 до 3 SQL запитів що є мінімально необхідним для забезпечення динамічної перевірки ролі без кешування.

Варто зазначити що для виробничого середовища з високим навантаженням продуктивність може бути суттєво підвищена через кешування результату user.role() у сесії з інвалідацією кешу при зміні ролі адміністратором. Однак для демонстраційного прототипу пріоритет надається коректності та безпеці над продуктивністю — динамічне визначення ролі без кешування гарантує що зміна ролі набирає чинності негайно що є більш безпечним підходом.[14]

Результати тестування підтверджують що реалізована система авторизації коректно втілює модель RBAC для медичного центру МРТ діагностики: забезпечує чітке розмежування доступу між трьома ролями,

захищає від типових вразливостей веб-застосунків та фіксує всі події доступу у журналі аудиту з прийнятними показниками продуктивності. [52]

ВИСНОВОК

Дипломна робота присвячена дослідженню моделей рольового розмежування доступу — RBAC, ABAC та ReBAC — у контексті сучасних веб-систем. За результатами проведеного дослідження можна сформулювати наступні висновки.

У першому розділі встановлено концептуальну різницю між автентифікацією та авторизацією як двома принципово різними але нерозривно пов'язаними механізмами безпеки веб-систем. Автентифікація відповідає на питання «хто це?» і є необхідною передумовою для авторизації яка відповідає на питання «що цьому суб'єкту дозволено?». Проведена класифікація моделей управління доступом охоплює п'ять основних підходів — DAC, MAC, RBAC, ABAC та ReBAC — де кожна наступна модель є відповіддю на обмеження попередньої. Аналіз вимог сучасних веб-систем до управління доступом визначив шість ключових критеріїв: масштабованість, гнучкість, багаторівнева безпека, зручність адміністрування, підтримка аудиту та можливість розподіленої авторизації. Простежена еволюція підходів до авторизації від hardcoded перевірок у 1990-х роках через впровадження RBAC у фреймворках 2000-х та делегованої авторизації через OAuth 2.0 у 2010-х до появи реляційної моделі Google Zanzibar у 2019 році демонструє що складність вимог до авторизації зростає пропорційно до складності самих веб-систем.[3; 11; 13 ;17 ; 24]

У другому розділі проведено детальний аналіз трьох цільових моделей. Встановлено що RBAC є найпоширенішою моделлю у корпоративних веб-системах завдяки природній відповідності організаційним структурам та простоті адміністрування, однак її фундаментальним обмеженням є контекстна сліпота та ризик вибуху ролей при зростанні складності системи. ABAC усуває ці обмеження через введення атрибутивних політик що дозволяють враховувати довільну кількість характеристик суб'єкта, об'єкта та середовища одночасно, проте ціною значно вищої складності реалізації та

адміністрування. ReBAC є концептуально найновішою моделлю що базується на графі відносин між об'єктами у стилі Google Zanzibar та природно відображає динамічні соціальні та організаційні структури, однак потребує розробки власного движка перевірки досяжності у графі або інтеграції спеціалізованих рішень. Розглянуто також гібридні підходи що поєднують переваги кількох моделей для вирішення складних задач авторизації.[3; 11]

У третьому розділі на основі семи критеріїв порівняння — гнучкості, масштабованості адміністрування, складності реалізації, складності адміністрування, продуктивності перевірки, стійкості до помилок конфігурації та підтримки у фреймворках — проведено порівняльний аналіз трьох моделей. Встановлено що жодна з моделей не є домінуючою за всіма критеріями одночасно та існує фундаментальний компроміс між гнучкістю та простотою. RBAC демонструє найкращі показники простоти реалізації та підтримки у фреймворках. ABAC є лідером за гнучкістю але найгіршою за складністю реалізації та адміністрування. ReBAC займає проміжну позицію поєднуючи високу гнучкість з помірною складністю. Сценарний аналіз чотирьох типових веб-систем — корпоративного порталу, медичної системи, SaaS-платформи та соціальної мережі — підтвердив що вибір моделі є архітектурним рішенням що визначається профілем вимог конкретної системи. Сформульовано шість практичних рекомендацій щодо вибору моделі серед яких ключовими є принцип мінімальної складності та розгляд гібридних архітектур як стандартного рішення для складних систем.[3; 10; 24;26; 27]

У четвертому розділі реалізовано демонстраційний прототип системи авторизації для медичного центру МРТ діагностики на основі моделі RBAC засобами Django 5.0(Подано у додатку Д). Вибір RBAC обґрунтовано відповідністю стабільної ієрархічної організаційної структури медичного закладу принципам рольової моделі — три ролі Адміністратор, Співробітник та Клієнт повністю покривають простір задач системи без ризику вибуху ролей. Реалізація включає кастомну модель користувача з автоматично

генерованим індивідуальним номером пацієнта та входом за email, RBAC-движок на основі декораторів `@role_required` та `@login_required` що забезпечує декларативний захист view-функцій, захист на рівні об'єктів через фільтрацію QuerySet що усуває вразливість IDOR, та журнал аудиту що фіксує кожну подію доступу з повним набором атрибутів. Функціональне тестування охопило 71 перевірку у п'яти тестових сценаріях — розмежування доступу між ролями, захист неаутентифікованих запитів, захист від IDOR, коректність журналу аудиту та негайна дія зміни ролі — всі перевірки пройдено успішно. Оцінка продуктивності показала що повний цикл авторизації виконується за менш ніж 4 мілісекунди що є прийнятним показником для веб-застосунку.[36; 38]

Практична цінність роботи полягає у тому що результати порівняльного аналізу та сформульовані рекомендації можуть бути безпосередньо застосовані розробниками при проектуванні систем авторизації для веб-застосунків різних класів. Реалізований прототип є відтворюваним демонстраційним середовищем що ілюструє принципи RBAC у контексті реальної предметної області та може слугувати відправною точкою для розробки виробничих систем авторизації на основі Django.[3;14]

Перспективами подальших досліджень є розширення прототипу реалізаціями моделей ABAC та ReBAC для забезпечення повноцінного порівняння не лише на теоретичному але й на практичному рівні, дослідження продуктивності систем авторизації при високому навантаженні з використанням інструментів навантажувального тестування, а також дослідження підходів до реалізації Authorization as a Service у мікросервісних архітектурах на основі відкритих рішень OpenFGA та SpiceDB.[3]

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. OWASP Top Ten. Open Web Application Security Project. URL: <https://owasp.org/www-project-top-ten/> (дата звернення: 10.04.2025).
2. Verizon. Data Breach Investigations Report 2024. URL: <https://www.verizon.com/business/resources/reports/dbir/> (дата звернення: 10.04.2025).
3. Ferraiolo D., Kuhn D. R., Chandramouli R. Role-Based Access Control. 2nd ed. Norwood : Artech House, 2007. 328 с.
4. Stallings W. Network Security Essentials: Applications and Standards. 6th ed. Hoboken : Pearson, 2016. 432 с.
5. Grassi P. A., Garcia M. E., Fenton J. L. Digital Identity Guidelines. NIST Special Publication 800-63-3. Gaithersburg : National Institute of Standards and Technology, 2017. URL: <https://pages.nist.gov/800-63-3/> (дата звернення: 10.04.2025).
6. Hardt D. The OAuth 2.0 Authorization Framework. RFC 6749. Internet Engineering Task Force, 2012. URL: <https://datatracker.ietf.org/doc/html/rfc6749> (дата звернення: 10.04.2025).
7. OpenID Connect Core 1.0 incorporating errata set 2. OpenID Foundation. URL: https://openid.net/specs/openid-connect-core-1_0.html (дата звернення: 10.04.2025).
8. The Single UNIX Specification Version 4. The Open Group. URL: <https://pubs.opengroup.org/onlinepubs/9699919799/> (дата звернення: 10.04.2025).
9. ANSI/INCITS 359-2004. Role Based Access Control. Washington : American National Standards Institute, 2004. 52 с.
10. Hu V. C., Ferraiolo D., Kuhn R., Schnitzer A., Sandlin K., Miller R., Scarfone K. Guide to Attribute Based Access Control (ABAC) Definition and Considerations. NIST Special Publication 800-162. Gaithersburg : National Institute of Standards and Technology, 2014. 54 с. URL:

- <https://nvlpubs.nist.gov/nistpubs/specialpublications/NIST.sp.800-162.pdf>
(дата звернення: 10.04.2025).
- 11.Zanzibar: Google's Consistent, Global Authorization System. Google Research. URL: <https://research.google/pubs/pub48190/> (дата звернення: 10.04.2025).
 - 12.Ferraiolo D. F., Kuhn D. R. Role-Based Access Controls. Proceedings of the 15th National Computer Security Conference. Baltimore, MD, October 13–16, 1992. P. 554–563.
 - 13.Django documentation. URL: <https://docs.djangoproject.com/en/5.0/> (дата звернення: 10.04.2025).
 - 14.Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol : O'Reilly Media, 2021. 616 с.
 - 15.Склярєнко О. В., Савченко Я. О., Литвиненко Л. О., Сушинський О. Є. Архітектурні підходи до розробки масштабованих веб-застосунків. Кібербезпека: освіта, наука, техніка. 2024. Т. 4, № 24. С. 341–350. DOI: 10.28925/2663-4023.2024.24.341350.
 - 16.Спасітєлева С. О., Чичкань І. В., Шевченко С. М., Жданова Ю. Д. Розробка безпечних контейнерних застосунків з мікросервісною архітектурою. Кібербезпека: освіта, наука, техніка. 2023. Т. 1, № 21. С. 193–210. DOI: 10.28925/2663-4023.2023.21.193210.
 - 17.Hohpe G., Woolf B. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Boston : Addison-Wesley, 2003. 736 с.
 - 18.PCI Security Standards Council. PCI DSS v4.0. URL: https://www.pcisecuritystandards.org/document_library/ (дата звернення: 10.04.2025).
 - 19.Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data (General Data Protection Regulation). Official Journal of the European Union. 2016. L 119. P. 1–88.

- 20.Верховна Рада України. Закон України «Про захист персональних даних» від 01.06.2010 № 2297 VI. URL: <https://zakon.rada.gov.ua/laws/show/2297-17> (дата звернення: 10.04.2025).
- 21.Sandhu R., Coyne E., Feinstein H., Youman C. Role-Based Access Control Models. IEEE Computer. 1996. Vol. 29, № 2. P. 38–47.
- 22.Bishop M. Computer Security: Art and Science. Boston : Addison-Wesley, 2003. 1136 с.
- 23.OASIS eXtensible Access Control Markup Language (XACML) Version 3.0. OASIS Standard. URL: <https://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html> (дата звернення: 10.04.2025).
- 24.Madden N. API Security in Action. Shelter Island : Manning Publications, 2020. 568 с.
- 25.OpenFGA Documentation. Auth0 by Okta. URL: <https://openfga.dev/docs> (дата звернення: 10.04.2025).
- 26.SpiceDB Documentation. Authzed. URL: <https://authzed.com/docs> (дата звернення: 10.04.2025).
- 27.Hu V. C., Kuhn D. R., Ferraiolo D. F. Attribute-Based Access Control. IEEE Computer. 2015. Vol. 48, № 2. P. 85–88.
- 28.Kuhn D. R., Coyne E. J., Weil T. R. Adding Attributes to Role-Based Access Control. Computer. 2010. Vol. 43, № 6. P. 79–81.
- 29.Bertino E., Sandhu R. Database Security — Concepts, Approaches, and Challenges. IEEE Transactions on Dependable and Secure Computing. 2005. Vol. 2, № 1. P. 2–19.
- 30.Anderson R. Security Engineering: A Guide to Building Dependable Distributed Systems. 2nd ed. Hoboken : Wiley, 2008. 1080 с.
- 31.Zhang E. What is Role-Based Access Control (RBAC)? Digital Guardian. 2020. URL: <https://digitalguardian.com/blog/what-role-based-access-control-rbac-examples-benefits-and-more> (дата звернення: 10.04.2025).

- 32.NEXTLABS. The Definitive Guide to Attribute-Based Access Control (ABAC). URL: <https://www.nextlabs.com/products/technology/abac/> (дата звернення: 10.04.2025).
- 33.Shostack A. Threat Modeling: Designing for Security. Indianapolis : Wiley, 2014. 624 с.
- 34.Stallings W., Brown L. Computer Security: Principles and Practice. 2nd ed. Hoboken : Pearson, 2012. 816 с.
- 35.Python Software Foundation. Python Documentation. URL: <https://docs.python.org/3/> (дата звернення: 10.04.2025).
- 36.TIOBE Software. TIOBE Index. URL: <https://www.tiobe.com/tiobe-index/> (дата звернення: 10.04.2025).
- 37.SQLite Documentation. URL: <https://www.sqlite.org/docs.html> (дата звернення: 10.04.2025).
- 38.Кучер В. В. Особливості використання фреймворку Django для написання веб-додатків. Актуальні питання сучасної інформатики : матеріали конференції. Житомир : Житомирський державний університет імені Івана Франка, 2017. С. 19–25.
- 39.Greenfeld D., Greenfeld A. Two Scoops of Django: Best Practices for Django 3.x. San Diego : Two Scoops Press, 2020. 530 с.
- 40.ISO/IEC 27001:2013 Information technology — Security techniques — Information security management systems — Requirements. Geneva : International Organization for Standardization, 2013.
- 41.Парфьонов Ю. Е. Питання міграції схеми бази даних під час супроводу веб-застосунків на базі фреймворку Django. Системи обробки інформації. 2024. Вип. 2 (177). URL: <https://journal-hnups.com.ua/index.php/soi/article/view/1658> (дата звернення: 10.04.2025).
- 42.Pallets Projects. Flask Documentation. URL: <https://flask.palletsprojects.com/> (дата звернення: 10.04.2025).

- 43.Laravel. Laravel Documentation. URL: <https://laravel.com/docs/> (дата звернення: 10.04.2025).
- 44.Express.js. Express Documentation. URL: <https://expressjs.com/> (дата звернення: 10.04.2025).
- 45.Park J., Sandhu R. The UCONABC Usage Control Model. ACM Transactions on Information and System Security. 2004. Vol. 7, № 1. P. 128–174.
- 46.Microsoft. Role-based security. URL: <https://docs.microsoft.com/en-us/dynamics365/fin-ops-core/dev-itpro/sysadmin/role-based-security/> (дата звернення: 10.04.2025).
- 47.Connolly T., Begg C. Database Systems: A Practical Approach to Design, Implementation, and Management. 6th ed. Harlow : Pearson, 2015. 1440 с.
- 48.Object Management Group. Unified Modeling Language (UML) Specification. URL: <https://www.omg.org/spec/UML/> (дата звернення: 10.04.2025).
- 49.NIST Special Publication 800-53 Revision 4: Security and Privacy Controls for Federal Information Systems and Organizations. Gaithersburg : National Institute of Standards and Technology, 2013.
- 50.Keycloak. Open Source Identity and Access Management. URL: <https://www.keycloak.org> (дата звернення: 10.04.2025).
- 51.Mandatory Access Control vs Discretionary Access Control. Ekran System. URL: <https://www.ekransystem.com/en/blog/mac-vs-dac> (дата звернення: 10.04.2025).
- 52.OWASP Web Security Testing Guide v4.2. Open Web Application Security Project. URL: <https://owasp.org/www-project-web-security-testing-guide/> (дата звернення: 10.04.2025).

```

class User(AbstractUser):
    """

    USERNAME_FIELD = email —

    individual_number —

    """

    patronymic = models.CharField(max_length=100, blank=True,
verbose_name='По батькові')

    email = models.EmailField(unique=True, verbose_name='Email')

    phone = models.CharField(max_length=20, blank=True,
verbose_name='Телефон')

    date_of_birth = models.DateField(null=True, blank=True,
verbose_name='Дата народження')

    individual_number = models.CharField(

        max_length=20, unique=True, default=gen_patient_number,

        editable=False, verbose_name='Індивідуальний номер'

    )

    USERNAME_FIELD = 'email'

    REQUIRED_FIELDS = ['username', 'first_name', 'last_name']

    class Meta:

```

```
verbose_name = 'Користувач'
```

```
verbose_name_plural = 'Користувачі'
```

```
def full_name(self):
```

```
    parts = [self.last_name, self.first_name, self.patronymic]
```

```
    return ' '.join(p for p in parts if p) or self.email
```

```
def role(self):
```

```
    if self.is_superuser:
```

```
        return 'superuser'
```

```
    names = list(self.groups.values_list('name', flat=True))
```

```
    if 'Адміністратор' in names:
```

```
        return 'admin'
```

```
    if 'Співробітник' in names:
```

```
        return 'staff'
```

```
    if 'Клієнт' in names:
```

```
        return 'client'
```

```
    return 'none'
```

```
def role_label(self):
```

```
    return {
```

```
        'superuser': 'Суперкористувач',
```

```
'admin': 'Адміністратор',  
  
'staff': 'Співробітник',  
  
'client': 'Клієнт',  
  
'none': 'Без ролі',  
  
}.get(self.role(), '—')
```

```
def __str__(self):  
  
    return f'{self.full_name()} [{self.individual_number}]'
```

```
class ScanType(models.Model):
```

```
    name = models.CharField(max_length=200, verbose_name='Назва')  
  
    description = models.TextField(blank=True, verbose_name='Опис')  
  
    duration_minutes = models.PositiveIntegerField(default=30,  
verbose_name='Тривалість (хв)')  
  
    price = models.DecimalField(max_digits=8, decimal_places=2,  
verbose_name='Вартість (грн)')  
  
    is_active = models.BooleanField(default=True, verbose_name='Активний')
```

```
class Meta:
```

```
    verbose_name = 'Тип дослідження'  
  
    verbose_name_plural = 'Типи досліджень'  
  
    ordering = ['name']
```

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

```
class Appointment(models.Model):
```

```
    STATUS = [  
        ('pending', 'Очікує підтвердження'),  
        ('confirmed', 'Підтверджено'),  
        ('completed', 'Виконано'),  
        ('cancelled', 'Скасовано'),  
    ]
```

```
    patient = models.ForeignKey(User, on_delete=models.CASCADE,  
                                related_name='appointments', verbose_name='Пацієнт')
```

```
    scan_type = models.ForeignKey(ScanType, on_delete=models.PROTECT,  
                                verbose_name='Тип дослідження')
```

```
    scheduled_date = models.DateField(verbose_name='Дата')
```

```
    scheduled_time = models.TimeField(verbose_name='Час')
```

```
    status = models.CharField(max_length=20, choices=STATUS,  
                              default='pending', verbose_name='Статус')
```

```
    notes = models.TextField(blank=True, verbose_name='Примітки')
```

```
    created_at = models.DateTimeField(auto_now_add=True)
```

```
    updated_at = models.DateTimeField(auto_now=True)
```

```

class Meta:

    verbose_name = 'Запис'

    verbose_name_plural = 'Записи'

    ordering = ['-scheduled_date', '-scheduled_time']

    def __str__(self):

        return f'{self.patient.full_name()} — {self.scan_type} {self.scheduled_date}'

class MedicalResult(models.Model):

    CONFIDENTIALITY = [

        ('standard',      'Стандартний'),

        ('sensitive',     'Чутливий'),

        ('highly_sensitive', 'Суворо конфіденційний'),

    ]

    appointment = models.OneToOneField(Appointment,
on_delete=models.CASCADE,

                                     related_name='result', verbose_name='Запис')

    radiologist = models.ForeignKey(User, on_delete=models.SET_NULL,

                                     null=True, related_name='results_made',

                                     verbose_name='Радіолог')

    description = models.TextField(verbose_name='Опис дослідження')

```

```

conclusion = models.TextField(verbose_name='Висновок')

confidentiality = models.CharField(max_length=20,
choices=CONFIDENTIALITY,

                                default='standard', verbose_name='Конфіденційність')

created_at = models.DateTimeField(auto_now_add=True)


class Meta:

    verbose_name = 'Результат'

    verbose_name_plural = 'Результати'


def __str__(self):

    return f'Результат: {self.appointment}'


class AuditLog(models.Model):

    """Журнал аудиту RBAC — фіксує кожну перевірку доступу."""

    ACTIONS = [('granted', 'Надано'), ('denied', 'Відхилено')]

    user = models.ForeignKey(User, on_delete=models.SET_NULL,

                              null=True, related_name='audit_logs')

    action = models.CharField(max_length=10, choices=ACTIONS)

    resource = models.CharField(max_length=255)

    user_role = models.CharField(max_length=50)

    user_groups = models.CharField(max_length=200)

```

```
ip_address = models.GenericIPAddressField(null=True, blank=True)
```

```
details = models.TextField(blank=True)
```

```
timestamp = models.DateTimeField(auto_now_add=True)
```

```
class Meta:
```

```
    verbose_name = 'Журнал аудиту'
```

```
    verbose_name_plural = 'Журнал аудиту'
```

```
    ordering = ['-timestamp']
```

```
def __str__(self):
```

```
    return f'{self.user} → {self.resource} [{self.action}]'
```

```
from functools import wraps

from django.shortcuts import redirect

from django.contrib import messages


def _get_ip(request):

    xff = request.META.get('HTTP_X_FORWARDED_FOR')

    return xff.split(',')[0] if xff else request.META.get('REMOTE_ADDR')


def audit(user, resource, action, request=None, details=""):

    """Записує подію у журнал аудиту."""

    from .models import AuditLog

    AuditLog.objects.create(

        user      = user,

        action    = action,

        resource  = resource,

        user_role = user.role(),

        user_groups = ', '.join(user.groups.values_list('name', flat=True)) or '—',

        ip_address = _get_ip(request) if request else None,

        details   = details,

    )
```

```
def login_required(view_fn):
    """Перевірка автентифікації."""
    @wraps(view_fn)
    def _wrap(request, *args, **kwargs):
        if not request.user.is_authenticated:
            messages.warning(request, 'Увійдіть у систему.')
            return redirect('login')
        return view_fn(request, *args, **kwargs)
    return _wrap
```

```
def role_required(*allowed_roles):
    """
    """
    def decorator(view_fn):
        @wraps(view_fn)
        def _wrap(request, *args, **kwargs):
            if not request.user.is_authenticated:
                return redirect('login')
            r = request.user.role()
            if r == 'superuser' or r in allowed_roles:
                return view_fn(request, *args, **kwargs)
```

```

# --- ВІДМОВА ---

audit(
    request.user, request.path, 'denied', request=request,
    details=f'Потрібна роль: {", ".join(allowed_roles)}. Є: {r}'
)

messages.error(
    request,
    f'🚫 Доступ заборонено. Сторінка доступна для: '
    f'{", ".join(allowed_roles)}.'
)

return redirect('denied')

return _wrap

return decorator

ROLE_PERMISSIONS = {
    'Адміністратор': [
        'view_user', 'change_user',
        'view_appointment', 'change_appointment', 'delete_appointment',
        'view_medicalresult', 'add_medicalresult', 'change_medicalresult',
        'view_auditlog',
        'view_scantype', 'add_scantype', 'change_scantype', 'delete_scantype',
    ],

```

```

'Співробітник': [
    'view_appointment', 'change_appointment',
    'view_medicalresult', 'add_medicalresult', 'change_medicalresult',
    'view_scantype',
],
'Клієнт': [
    'add_appointment', 'view_appointment',
    'view_medicalresult',
],
}

```

```
urlpatterns = [
```

```
    # — auth
```

```

    path("", views.home, name='home'),
    path('register/', views.register, name='register'),
    path('login/', views.user_login, name='login'),
    path('logout/', views.user_logout, name='logout'),
    path('denied/', views.denied, name='denied'),
    path('init/', views.init_system, name='init_system'),

```

```
    # — клієнт
```

```

path('cabinet/',          views.client_dashboard,  name='client_dashboard'),

path('cabinet/appointments/',  views.client_appointments,
name='client_appointments'),

path('cabinet/book/',          views.book,          name='book'),

path('cabinet/cancel/<int:pk>/',  views.cancel_appointment, name='cancel_appointment'),

path('cabinet/results/',        views.client_results,  name='client_results'),

path('cabinet/results/<int:pk>/',  views.client_result_detail, name='client_result_detail'),

path('cabinet/profile/',        views.client_profile,  name='client_profile'),

```

— співробітник

```

path('staff/',              views.staff_dashboard,  name='staff_dashboard'),

path('staff/appointments/',  views.staff_appointments, name='staff_appointments'),

path('staff/appointments/<int:pk>/status/',
views.staff_update_status, name='staff_update_status'),

path('staff/appointments/<int:appt_pk>/result/', views.staff_add_result,
name='staff_add_result'),

path('staff/results/<int:pk>/',  views.staff_result_detail, name='staff_result_detail'),

path('staff/patients/',        views.staff_patients,  name='staff_patients'),

```

```
    path('staff/patients/<int:pk>/', views.staff_patient_detail,
name='staff_patient_detail'),
```

```
# — адміністратор
```

```
    path('admin-
panel/', views.admin_dashboard, name='admin_dashboard'),
```

```
    path('admin-
panel/users/', views.admin_users, name='admin_users'),
```

```
    path('admin-panel/users/<int:user_pk>/role/', views.admin_change_role,
name='admin_change_role'),
```

```
    path('admin-
panel/audit/', views.admin_audit, name='admin_audit'),
```

```
    path('admin-panel/scan-
types/', views.admin_scan_types, name='admin_scan_types'),
```

```
    path('admin-panel/results/', views.admin_all_results,
name='admin_all_results'),
```

```
]
```

```
from django.contrib import admin

from django.contrib.auth.admin import UserAdmin as BaseUserAdmin

from .models import User, ScanType, Appointment, MedicalResult, AuditLog


@admin.register(User)
class UserAdmin(BaseUserAdmin):

    list_display = ['email', 'full_name', 'individual_number', 'role_label', 'is_active']

    list_filter = ['groups', 'is_active']

    search_fields = ['email', 'first_name', 'last_name', 'individual_number']

    ordering = ['last_name']

    fieldsets = (

        (None, {'fields': ('email', 'username', 'password')}),

        ('Особисті дані', {'fields': ('first_name', 'last_name', 'patronymic', 'phone',
'date_of_birth')}),

        ('Системний номер', {'fields': ('individual_number',)}),

        ('Права', {'fields': ('is_active', 'is_staff', 'is_superuser', 'groups',
'user_permissions')}),

        ('Дати', {'fields': ('last_login', 'date_joined')}),

    )

    readonly_fields = ['individual_number']

    add_fieldsets = ((None, {

        'classes': ('wide',),
```

```
        'fields': ('email', 'username', 'first_name', 'last_name', 'password1',
'password2'),
    },),)
```

```
@admin.register(ScanType)
```

```
class ScanTypeAdmin(admin.ModelAdmin):
```

```
    list_display = ['name', 'duration_minutes', 'price', 'is_active']
```

```
    list_filter = ['is_active']
```

```
@admin.register(Appointment)
```

```
class AppointmentAdmin(admin.ModelAdmin):
```

```
    list_display = ['patient', 'scan_type', 'scheduled_date', 'scheduled_time', 'status']
```

```
    list_filter = ['status', 'scheduled_date']
```

```
    search_fields = ['patient__last_name', 'patient__individual_number']
```

```
    date_hierarchy = 'scheduled_date'
```

```
@admin.register(MedicalResult)
```

```
class MedicalResultAdmin(admin.ModelAdmin):
```

```
    list_display = ['appointment', 'radiologist', 'confidentiality', 'created_at']
```

```
    list_filter = ['confidentiality']
```

```
@admin.register(AuditLog)
```

```
class AuditLogAdmin(admin.ModelAdmin):
```

```
list_display = ['timestamp', 'user', 'action', 'resource', 'user_role']
```

```
list_filter = ['action', 'user_role']
```

```
search_fields = ['user__email', 'resource']
```

```
readonly_fields = ['user', 'action', 'resource', 'user_role',  
                   'user_groups', 'ip_address', 'details', 'timestamp']
```

Додаток Д

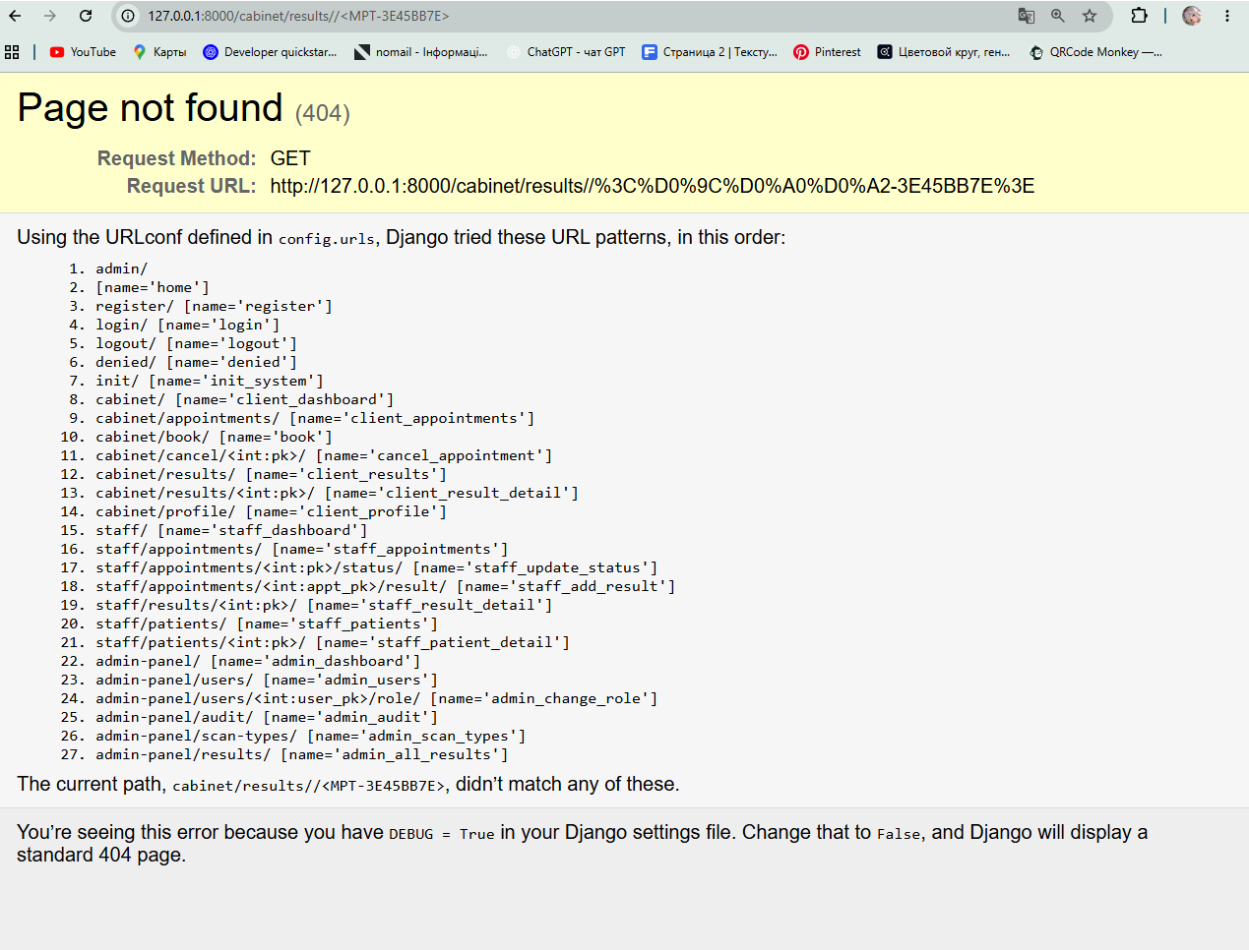


Рис Д.1 404

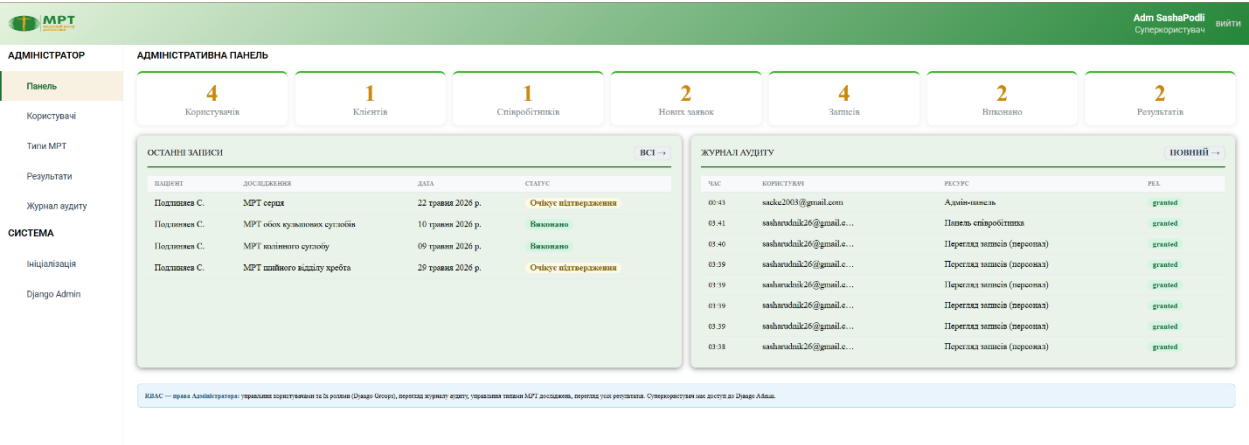


Рис Д.2 Інтерфейс прототипу

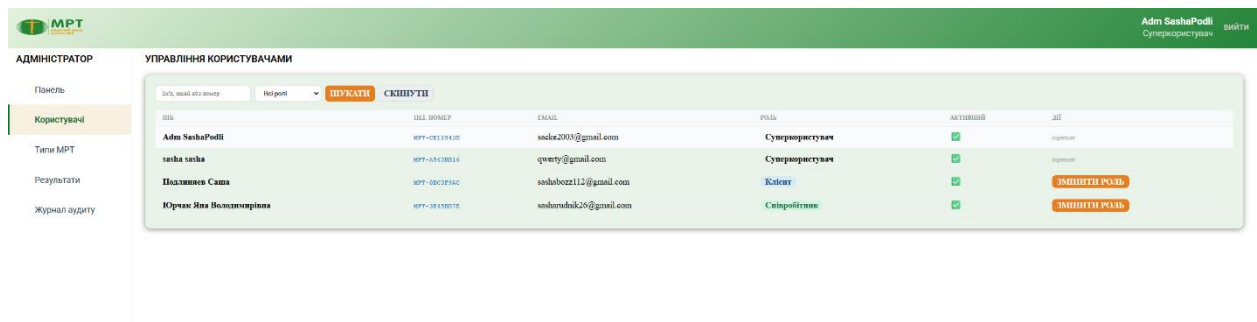


Рис Д.3 Інтерфейс прототипу

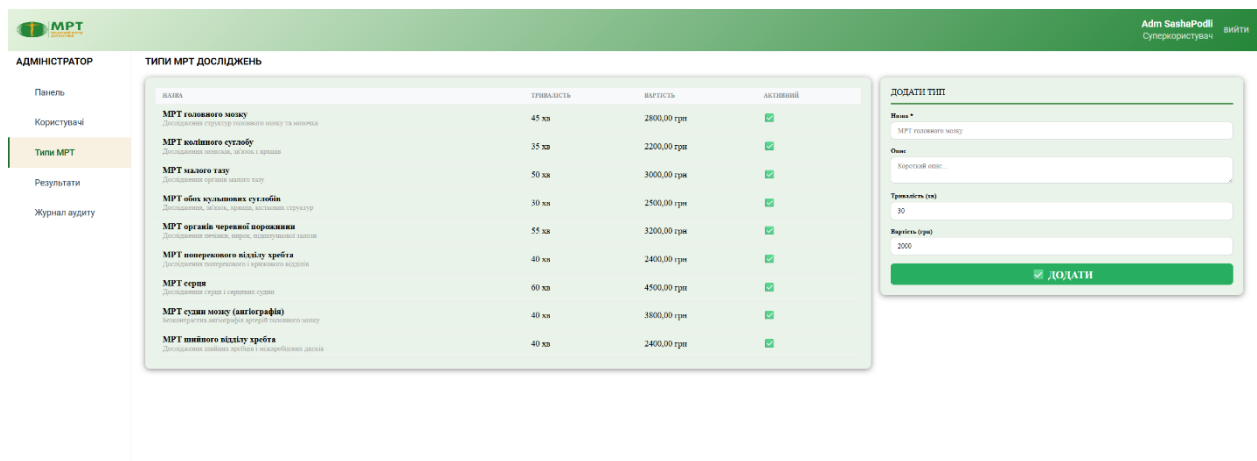


Рис Д.4 Інтерфейс прототипу

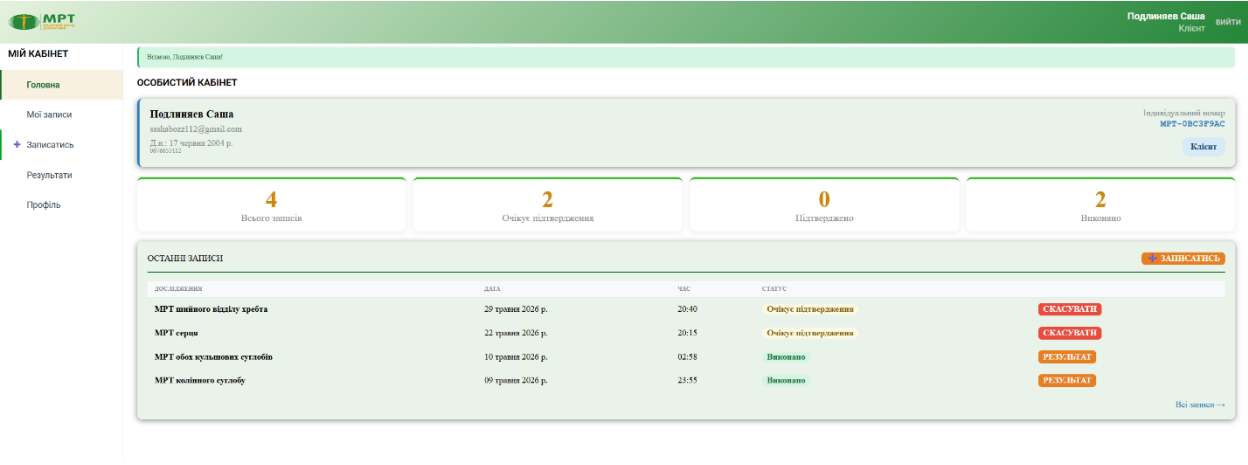


Рис Д.5 Інтерфейс прототипу

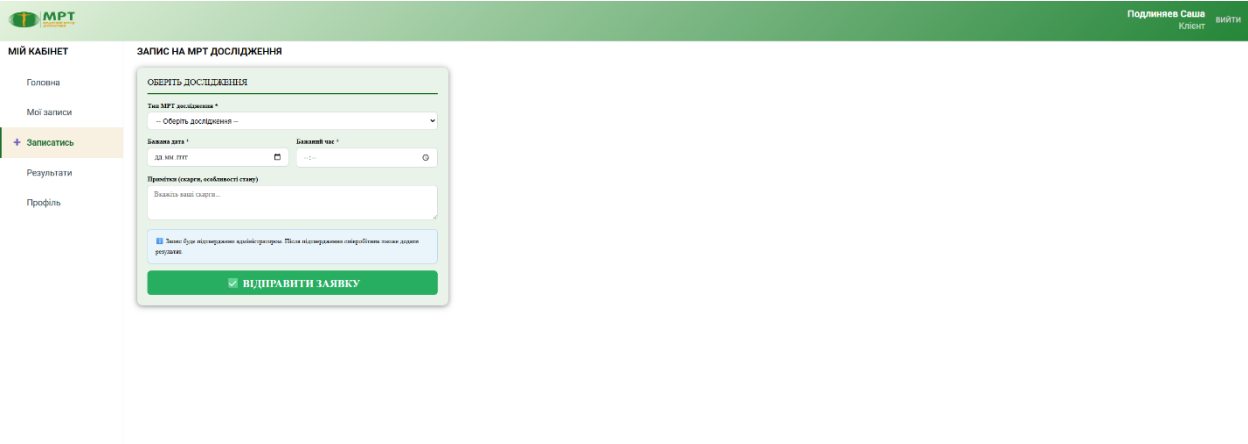


Рис Д.6 Інтерфейс прототипу

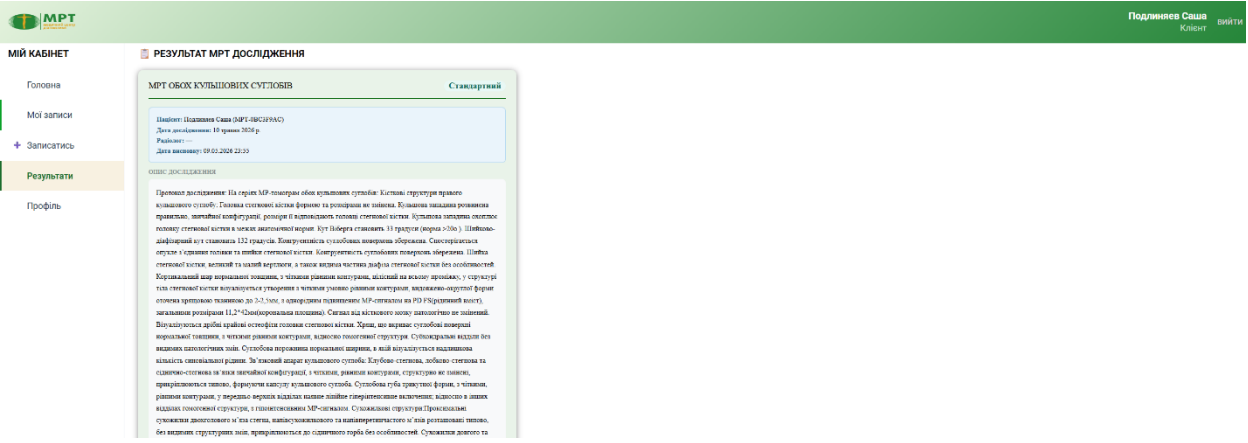


Рис Д.7 Інтерфейс прототипу