ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Цифрова платформа відкритого репозиторію для управління та публікації відкритих наборів даних»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Технології цифрового розвитку»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Владислав ЛИСЕНКО
(підпис)

Виконав: здобувач вищої освіти групи ТЦР-44

Владислав ЛИСЕНКО

Керівник: Микола ГЕРЦЮК
д-р філософії

Рецензент:

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Технологій цифрового розвитку

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Технології цифрового розвитку»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Технологій цифрового розвитку
_____ Вікторія ЖЕБКА
« ____ » _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Лисенку Владиславу Ігоровичу

1. Тема кваліфікаційної роботи: «Цифрова платформа відкритого репозиторію для управління та публікації відкритих наборів даних» керівник кваліфікаційної роботи д-р філософії, доцент Микола ГЕРЦЮК, затверджені наказом Державного університету інформаційно-комунікаційних технологій від 20 лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи 18 травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: документація репозиторію з відкритим кодом Harvard Dataverse, офіційні документації фреймворків Spring Framework та Angular, офіційні технічні ресурси технологій Amazon Web Services, приклади імплементації та джерела з гексагональної архітектури (Ports and Adapters).

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд предметної галузі та аналіз складностей технічного дизайну подібних систем.
2. Вибір технологій та проєктування веб-застосунку.

3. Імплементація серверної та клієнтських частин платформи для управління та публікації відкритих наборів даних.
4. Тестування бізнес-логіки застосунку.

5. Перелік графічного матеріалу: *презентація*
1. Аналіз аналогів
2. Вимоги до програмного забезпечення
3. Програмні засоби імплементації
4. ER Діаграма зв’язків
5. Діаграма прецедентів
6. Діаграма створення набору даних
7. Екранні форми
8. Відео презентація застосунку
9. Тестування
10. Апробація результатів дослідження
11. Висновки
6. Дата видачі завдання 20 лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	24.02-01.03.2026	Виконано
2	Аналіз та дослідження існуючих аналогів	02.03-11.03.2026	Виконано
3	Огляд концепту застосунку та деталей програмної реалізації подібних систем	12.03-21.03.2026	Виконано
4	Повноцінне проєктування застосунку	22.03-09.04.2026	Виконано
5	Імплементація серверної та клієнтської частин	10.04-20.04.2026	Виконано
6	Тестування серверної та клієнтської частин	21.04-28.04.2026	Виконано
7	Оформлення роботи: вступ, висновки, реферат	29.04-04.05.2026	Виконано
8	Розробка демонстраційних матеріалів	05.05-10.05.2026	Виконано
9	Попередній захист роботи	11.05.2026	Виконано

Здобувач вищої освіти _____ Владислав ЛИСЕНКО
(підпис)

Керівник
кваліфікаційної роботи _____ Микола ГЕРЦЮК
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 72 стор., 5 табл., 12 рис., 58 джерел.

Мета роботи – підвищення ефективності процесу шляхом свідомого зменшення простору можливостей та використання гнучкої архітектури.

Об'єкт дослідження – процеси управління та публікації відкритих наборів даних.

Предмет дослідження – цифрова платформа відкритого репозиторію, призначена для управління та публікації таких наборів даних.

Короткий зміст роботи: В ході роботи досліджено концепт репозиторіїв відкритих наборів даних та його важливість на основі таких відомих веб-застосунків, як Harvard Dataverse, Zenodo та Kaggle. На основі аналізу даних альтернатив було виявлено ключові вимоги до застосунку. На основі отриманих вимог розроблено архітектуру програмного забезпечення, яка базується на принципах гексагональної архітектури та орієнтована на гнучкість, масштабованість і спрощення подальшого розвитку системи. Реалізовано серверну частину застосунку з використанням Spring Boot, що забезпечує обробку запитів, управління користувачами, автентифікацію та взаємодію з базою даних через REST API. Клієнтську частину побудовано за допомогою Angular у вигляді SPA-застосунку з інтуїтивним інтерфейсом користувача.

У роботі реалізовано функціональність створення, редагування, зберігання та публікації наборів даних, а також механізми версіонування, повторного використання та експорту стандартизованих метаданих. У результаті розроблено платформу відкритого репозиторію даних, яка може використовуватись як готове рішення для невеликих організацій, дослідницьких груп чи освітніх установ, а також слугувати основою для подальшого розвитку й інтеграції з іншими інформаційними системами.

КЛЮЧОВІ СЛОВА: ВІДКРИТІ ДАНІ, РЕПОЗИТОРІЙ ДАНИХ, МЕТАДАНІ, ВЕРСІОНУВАННЯ, SPRING BOOT, ANGULAR, REST API.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПЗ – програмне забезпечення

API – інтерфейс прикладного програмування (Application Programming Interface)

REST API – архітектурний стиль взаємодії через HTTP (Representational State Transfer)

SPA – односторінковий застосунок (Single Page Application)

Java – об’єктно-орієнтована мова програмування

Spring Boot – фреймворк для створення серверних застосунків на Java

Angular – фреймворк для розробки клієнтських веб-застосунків

PostgreSQL – реляційна система керування базами даних

Redis – in-memory сховище даних для кешування та сесій

SQL – мова запитів до реляційних баз даних (Structured Query Language)

JPA – специфікація доступу до даних у Java (Java Persistence API)

AWS – хмарна платформа Amazon Web Services

Amazon S3 – сервіс зберігання об’єктів у хмарі

Amazon SES – сервіс для надсилання електронної пошти

DOI – цифровий ідентифікатор об’єкта (Digital Object Identifier)

UI – інтерфейс користувача (User Interface)

UX – користувацький досвід (User Experience)

ЗМІСТ

ВСТУП

На сьогоднішній день більшість світових інституцій збирають необхідні їм дані, які майже завжди піддаються широкому аналізу та вивченню з метою збору будь-яких корисних ідей, отримання цінності чи кращого розуміння середовища, в якому ці формування ведуть свою діяльність. Є доволі очевидним, що дуже велика частина даних, які вони отримують, є дуже чутливою та критичною для їх функціонування. Проте, деякі з них, з найкращими намірами для суспільства чи людства, публікують такі види даних в публічний доступ, що безумовно є дуже важливим для сучасного розвитку. У сучасну епоху, що характеризується розвитком штучного інтелекту та великих даних, доступ до ресурсів такої якості і масштабу та які можуть служити на користь тим, хто не може собі їх дозволити – сприяє появі усвідомлених ідей і рішень, що формують світ [1].

Існування спеціалізованих платформ для управління та поширення наборів даних є дуже важливим аспектом у цьому процесі. Дуже часто великі інституції мають свої особисті репозиторії, проте невеликі організації часто не мають необхідних ресурсів для створення та підтримки власної інфраструктури. У такому випадку відкриті репозиторії даних виступають ключовим інструментом. Вони забезпечують централізоване, стандартизоване та безпечне середовище, у якому дані можуть бути збережені, структуровані, перевірені та повторно використані будь-ким та з будь-якою метою.

Об'єктом дослідження даної кваліфікаційної роботи є процеси управління та публікації відкритих наборів даних.

Предметом дослідження кваліфікаційної роботи є платформа відкритого репозиторію, призначена для управління та публікації таких наборів даних.

Метою роботи є підвищення ефективності процесу шляхом свідомого зменшення простору можливостей та використання гнучкої архітектури.

Методи дослідження: Кваліфікаційна робота була виконана в два ключових етапи.

Етап перший – вивчення та аналіз аналогічних застосунків у сфері зберігання відкритих наборів даних. На цьому етапі були розглянуті платформи, такі як Harvard Dataverse, Kaggle та Zenodo, які є найвідомішими в цій галузі. На основі їх аналізу було визначено перелік функціональних та нефункціональних вимог до застосунку кваліфікаційної роботи.

Етап другий – вибір технологій та імплементація. Під час цього етапу був сформований стек технологій, орієнтований на надійність, зручність розробки та масштабованість. Для серверної частини було обрано мову Java із використанням фреймворку Spring Boot, що задовільняє необхідні умови завдяки автоконфігурації та розвиненій екосистемі, яка включає засоби для зручної роботи з безпекою, базами даних та REST API. Клієнтська частина реалізована за допомогою Angular, який забезпечує побудову SPA застосунку з структурованим інтерфейсом користувача. Дані зберігаються у реляційній базі даних, а для оптимізації роботи з сесіями автентифікації застосовано in-memory сховище. Для спрощення процесу розробки та запуску компонентів використано контейнеризацію Docker. У подальшому розгортання системи планується у хмарному середовищі Amazon Web Services із використанням відповідних сервісів для зберігання даних, обробки запитів та забезпечення стабільної роботи застосунку.

Наукова новизна виконаної роботи зумовлена розробкою цифрової платформи відкритого репозиторію для управління та публікації відкритих наборів даних, яка поєднує в собі найважливіші аспекти відомих альтернатив та в той же час оптимізована шляхом свідомого зменшення можливостей до таких, що стосуються безпосередньо наборів даних та використання гнучкої гексагональної архітектури.

Практична значущість результатів кваліфікаційної роботи полягає у можливості використання розробленої платформи як готового рішення для управління та публікації відкритих наборів даних у невеликих організаціях, дослідницьких групах та освітніх установах. Запропонований підхід дозволяє спростити процес роботи з даними, забезпечити їх централізоване зберігання, структурованість і доступність для подальшого використання. Система може бути застосована як основа для створення більш складних репозиторіїв даних або інтегрована з іншими інформаційними системами.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Репозиторії відкритих наборів даних

Репозиторії відкритих наборів даних є ключовим елементом сучасної інфраструктури роботи з інформацією, оскільки саме вони забезпечують зберігання, структуризацію та поширення даних у зручному для використання вигляді. Для розуміння їх значення важливо розглянути передумови їх виникнення, еволюцію самого поняття відкритих даних та проблематику пов’язану з їх поступом. Історичний розвиток цієї концепції дозволяє краще зрозуміти причини формування сучасних підходів до публікації інформації та ті принципи, які лежать в основі створення репозиторіїв відкритих даних.

1.1.1 Огляд історії відкритих даних

Визначення “відкриті дані” в сучасному значенні вперше з’явилося в 1995 році, проте воно має доволі багатий історичний контекст. Ідея того, що наукові дослідження повинні бути доступні для будь-кого була популяризована Робертом Кінгом Мертоном, одним з самих відомих американських соціологів 20-го століття, на початку 1940-х років [2]. Дослідження (в результаті яких отримуються дані) повинні вільно розповсюджуватись на благо суспільства.

Згодом ця ідея почала виходити за межі суто наукової спільноти. У другій половині 20-го століття поступово сформувалося розуміння того, що відкритими можуть бути не лише результати наукової роботи, а й дані, створені або зібрані державними установами, міжнародними організаціями та іншими інституціями. Із розвитком цифрових технологій та мережі Інтернет це питання набуло нового значення – якщо раніше поширення інформації було обмежене технічними можливостями, то в сучасному світі основною перешкодою стали правові, організаційні та інституційні бар’єри [3].

Таким чином поняття “відкриті дані” почало асоціюватися з доступністю інформації, можливістю її вільного використання, повторного застосування та поширення. У цьому контексті відкриті дані стали важливим інструментом прозорості, наукового обміну, інновацій та суспільного розвитку. Особливу роль у цьому відіграють репозиторії відкритих даних, адже вони забезпечують систематизоване зберігання, зручний доступ і повторне використання інформації різними групами користувачів – від дослідників і розробників до журналістів, бізнесу та громадян.

1.1.2 Роль відкритих даних в економіці та суспільному житті

Визначення впливу відкритих даних включає в себе певні нюанси та у багатьох випадках показує результати в більш ніж одній сфері впливу. Додатково, ефект відкритих даних на життя людей часто є непрямим та опосередкованим іншими широкими соціальними, політичними та економічними факторами [4]. Тим не менш, не дивлячись на ці нюанси існує чотири основних способи, якими відкриті дані впливають на суспільство.

Відкриті дані покращують державне управління, оскільки підвищують підзвітність, прозорість і ефективність роботи органів влади. Вони розширюють можливості громадян, даючи їм доступ до інформації для більш обґрунтованого прийняття рішень і для суспільної мобілізації. Також відкриті дані створюють нові економічні можливості, оскільки стимулюють інновації, розвиток сервісів, появу нових бізнес-моделей і зростання економічної активності. На останок, вони допомагають у розв’язанні суспільно важливих проблем, оскільки дають змогу аналізувати складні явища на основі даних і приймати більш ефективні рішення у таких сферах, як охорона здоров’я, безпека, екологія чи реагування на кризові ситуації.

Виходячи з вищесказаного, окрім своєї основної функції сховища даних, репозиторії відкритих даних є й інструментами економічної та суспільної взаємодії. Через публікацію наборів даних у стандартизованих, доступних форматах та чітко визначеним шляхом вони спрощують повторне використання

інформації, знижують технічні бар'єри для авторів і користувачів та створюють основу для нових аналітичних продуктів, цифрових сервісів і громадських ініціатив. Таким чином, значення самих відкритих даних для економіки й суспільства у межах цього дослідження доцільно розглядати саме через якісну організацію репозиторію та можливість для будь-кого оприлюднювати й поширювати набори даних у зручному для подальшого використання вигляді.

1.2 Огляд існуючих програмних продуктів

Сучасний ринок програмних рішень для зберігання та поширення відкритих даних представлений низкою усталених і масштабних платформ, які сформували високу довіру серед користувачів та мають значний функціональний потенціал. Такі системи, як правило, забезпечують широкий набір інструментів для роботи з даними, включаючи їх публікацію, структуризацію, пошук і довготривале зберігання. Водночас, попри загальну ефективність і популярність, кожна з цих платформ має власні особливості, сильні сторони та обмеження, які впливають на доцільність їх використання у конкретних умовах. Для обґрунтованого вибору інструментів та формування вимог до розробки нового рішення у цій сфері важливо розуміти ці переваги та недоліки.

1.2.1 Harvard Dataverse

Harvard Dataverse – це відкрита цифрова платформа для зберігання, поширення та цитування наукових даних розроблена в Інституті кількісних соціальних наук Гарвардського університету. Вона розроблена як репозиторій, який дозволяє дослідникам, університетам, організаціям та окремим користувачам публікувати набори даних у структурованому вигляді. Кожен набір даних (dataverse або dataset) може містити файли різних форматів, метадані, описові елементи та інформацію про авторство. Платформа підтримує принципи відкритої науки, забезпечуючи доступ до даних через веб-інтерфейс і API, а також інтеграцію з іншими науковими інфраструктурами [5].

Основне призначення Harvard Dataverse полягає у забезпеченні довготривалого збереження дослідницьких матеріалів і спрощенні їх повторного використання. Вона дуже широко використовується у наукових дослідженнях, зокрема в соціальних науках, економіці, медицині та інших дисциплінах, де важливо забезпечити прозорість і відтворюваність результатів. Платформа дозволяє користувачам створювати власні колекції даних, керувати доступом до них, а також надавати постійні ідентифікатори (DOI), що полегшує цитування та відстеження використання. Harvard Dataverse виступає як приклад системи, що поєднує централізоване зберігання з гнучкими інструментами управління та публікації.

Серед ключових переваг Harvard Dataverse можна виділити високу стандартизацію метаданих, підтримку міжнародних протоколів обміну даними та зручні механізми пошуку і доступу [6]. Платформа забезпечує контроль версій, що дозволяє відслідковувати зміни у наборах даних та підтримує різні рівні доступу – від повністю відкритого до обмеженого. Додатково важливими є інструменти для автоматичного цитування, аналітики використання та інтеграції з іншими системами. Це робить її ефективним рішенням для організації відкритих даних, підвищення їх доступності та забезпечення довіри до опублікованої інформації.

1.2.2 Zenodo

Zenodo – це ще одна відкрита цифрова платформа для зберігання та поширення наукових і дослідницьких результатів, створена за підтримки CERN та European OpenAIRE [7]. Вона функціонує як універсальний репозиторій, що дозволяє дослідникам, науковим групам і організаціям публікувати набори даних, статті, програмне забезпечення, звіти та інші цифрові матеріали. Платформа орієнтована на забезпечення відкритого доступу до результатів досліджень і підтримує стандартизовані підходи до опису ресурсів через метадані, що полегшує їх індексацію та пошук [8].

Основне використання Zenodo пов’язане з архівуванням і розповсюдженням наукового контенту в різних галузях, включаючи природничі, технічні та

гуманітарні науки. Як і Harvard Dataverse, ця платформа забезпечує надання постійних ідентифікаторів (DOI) для кожного завантаженого об’єкта, що робить можливим його коректне цитування у наукових роботах [9]. Користувачі можуть створювати власні колекції, керувати доступом до матеріалів, а також інтегрувати репозиторій із зовнішніми сервісами, такими як системи керування кодом чи інші наукові інфраструктури. У контексті репозиторіїв відкритих даних Zenodo виступає як друга поширена альтернатива, яка розширює можливості публікації за рахунок підтримки різних типів наукових результатів.

Серед переваг Zenodo варто відзначити універсальність, відносну простоту використання та широку підтримку різних форматів даних і цифрових об’єктів. Платформа забезпечує довготривале зберігання інформації, автоматичне створення цитувань і підтримку ліцензій відкритого доступу. Додатково вона пропонує можливості версіонування, інтеграції з такими платформами як GitHub, а також зручні інструменти для пошуку та аналізу контенту [10]. Завдяки цьому Zenodo є ефективним інструментом для організації відкритих даних і поширення результатів досліджень, сприяючи підвищенню доступності та прозорості наукової інформації

1.2.3 Інші альтернативи

Kaggle – це онлайн-платформа від Google, яка орієнтована на роботу з наборами даних, аналізом даних та машинним навчанням [11]. На відміну від класичних наукових репозиторіїв, таких як Harvard Dataverse чи Zenodo, вона не є повноцінною системою для архівування всіх типів дослідницьких результатів, але попри це, принцип її роботи безумовно заслуговує уваги по відношенню до цієї роботи. Її основний акцент зроблено саме на наборах даних, їх практичному використанні та взаємодії спільноти. Однією з найголовніших особливостей є можливість користувачів завантажувати, переглядати та обробляти дані безпосередньо в середовищі платформи, використовуючи вбудовані інструменти для аналізу, зокрема ноутбуки [12]. Підхід Kaggle орієнтований не на формальну

публікацію досліджень, а на доступність і зручність роботи з даними, що перетинається з ціллю даного проєкту.

Серед ключових переваг Kaggle можна виділити простоту використання, інтерактивність та сильну орієнтацію на користувача. Платформа забезпечує швидкий доступ до великої кількості відкритих наборів даних, а також дозволяє працювати з ними без необхідності складної попередньої підготовки чи налаштування інфраструктури. Інтегроване середовище для аналізу, підтримка спільної роботи та активна спільнота сприяють обміну знаннями та повторному використанню даних. Хоча Kaggle не є прямим аналогом класичних репозиторіїв, її фокус на наборах даних, їх доступності та простоті взаємодії відображає ряд характеристик, які є важливими для сучасних платформ відкритих даних.

1.2.4 Порівняння можливостей та недоліків

Після розгляду окремих платформ важливо провести їх узагальнене порівняння, щоб виявити спільні підходи, відмінності та обмеження. Кожне з проаналізованих рішень має власне бачення організації відкритих даних, що проявляється у функціональних можливостях, рівні складності використання, підходах до структурування інформації та підтримці взаємодії з користувачами. Жодна з платформ не є універсальною, оскільки їхні переваги часто супроводжуються певними компромісами [13]. Порівняльний аналіз цих платформ дозволяє оцінити ефективність існуючих інструментів та визначити аспекти, які можуть бути вдосконалені або оптимізовані при розробці власного рішення.

Таблиця 1.1

Аналіз альтернативних платформ

Критерій	Harvard Dataverse	Zenodo	Kaggle	OpenData (проєкт)
Орієнтація платформи	Спеціалізація на наукових даних, але з розширеним функціоналом	Універсальна платформа (дані, статті, ПЗ тощо)	Орієнтація на аналіз даних і ML	Чітка спеціалізація тільки на наборах даних

Продовження таблиці 1.1

Стандарти метаданих	Високий рівень стандартизації та галузеві метадані	Підтримка стандартів, але більш узагальнена	Обмежена, не формалізована (Schema.org, Croissant)	Фокус на основні DataCite та Dublin Core
Постійні ідентифікатори (DOI)	Підтримуються	Підтримуються	Частково (не основний фокус)	Передбачена підтримка як базова функція
Версіонування	Розвинене, але може бути складним	Підтримуєтьс я	Обмежене/неявне	Спрощене та прозоре версіонування наборів даних
Складність використання	Середня/висока через гнучкість	Середня	Низька	Низька за рахунок обмеженого функціоналу
Управління доступом	Гнучке, багаторівневе	Гнучке	Обмежене	Чітка і проста модель доступу без надлишкової складності

За результатами порівняння можна зробити висновок, що такі великі платформи, як Harvard Dataverse, Zenodo та Kaggle, забезпечують широкий набір функціональних можливостей і високу гнучкість у роботі з різними типами наукових ресурсів. Harvard Dataverse вирізняється високим рівнем стандартизації, розвиненою системою метаданих і підтримкою довготривалого збереження даних [6], а Zenodo є більш універсальним інструментом, який поєднує простоту використання з підтримкою різних типів цифрових об’єктів і зручними механізмами цитування [8]. Kaggle в свою чергу орієнтується насамперед на

практичну роботу з наборами даних, надаючи інтерактивне середовище для аналізу та активну модель взаємодії користувачів зі спільнотою [11].

Такі універсальність та масштабність мають і зворотний бік. Розширення функціоналу неминуче призводить до підвищення складності використання, перевантаженості інтерфейсу та меншої узгодженості у структурі даних. Користувачам часто доводиться працювати з надлишковими можливостями, які не є критично необхідними для базових задач публікації, опису та поширення наборів даних. Унаслідок цього основні сценарії взаємодії з репозиторієм можуть ставати менш зрозумілими, а сам процес керування даними - більш громіздким [14].

Отже, результати порівняння показують, що нова сучасна платформа є більш ефективним й функціонально насиченим рішенням, через те, що їхня орієнтація на широкий спектр завдань не завжди відповідає потребі у простому, цілісному та спеціалізованому інструменті саме для роботи з наборами даних. Саме ця обставина зумовлює необхідність окремо розглянути проблематику сучасних репозиторіїв відкритих даних та обґрунтувати доцільність більш вузькоспеціалізованого підходу.

1.2.5 Проблематика сучасних відкритих репозиторіїв

Однією з ключових проблем сучасних відкритих репозиторіїв є їхня зростаюча складність, яка зумовлена прагненням охопити максимально широкий спектр наукових потреб. Окрім роботи з наборами даних, великі платформи орієнтуються на зберігання та публікацію статей, програмного забезпечення, звітів та інших типів цифрових ресурсів. Така універсальність, хоча і розширює сферу застосування систем, водночас призводить до перевантаження функціоналу та ускладнення базових сценаріїв використання [13].

У результаті ключові процеси, пов’язані саме з наборами даних, зокрема опис метаданих, їх експорт, організація структури та версіонування, стають менш інтуїтивними та більш громіздкими. Додатково ситуацію ускладнює підтримка великої кількості стандартів метаданих, що створює надлишковість при виборі або

застосуванні конкретних схем опису. Це негативно впливає на узгодженість даних і ускладнює їх повторне використання. Таким чином, прагнення до універсальності фактично знижує ефективність роботи з основним об’єктом - наборами даних, які для більшості користувачів є центральним елементом таких систем.

У межах даної роботи пропонується альтернативний підхід, який полягає у свідомому обмеженні функціонального простору репозиторію. Замість підтримки широкого спектра типів ресурсів система фокусується виключно на наборах даних та ключових процесах, пов’язаних із їх публікацією та управлінням. Такий підхід передбачає використання обмеженого, але чітко визначеного набору можливостей, включаючи стандартизовані метадані, базові механізми версіонування, підтримку постійних ідентифікаторів та просте управління доступом. Завдяки цьому користувач отримує більш зрозуміле, передбачуване та узгоджене середовище, у якому основну увагу зосереджено саме на роботі з даними. Це дозволяє зменшити когнітивне навантаження, спростити процес публікації та підвищити ефективність використання репозиторію у практичних сценаріях

1.3 Огляд технологій та засобів імплементації

Визначивши особливості репозиторіїв відкритих даних та розуміючи основні характеристики, які входять у такі системи, наступним кроком є вибір відповідних технологій та засобів їх реалізації. Ефективність та розвиток розроблюваної платформи значною мірою залежить від правильно підібраного технологічного стеку, який має забезпечувати надійність, масштабованість, зручність розробки та подальшої підтримки. В цьому проєкті важливо враховувати як і функціональні вимоги, особливо пов’язані з обробкою, зберіганням та публікацією даних, так і нефункціональні аспекти, такі як продуктивність, безпека та гнучкість архітектури. У даному розділі розглянуто основні технології та інструменти, які було обрано для імплементації системи, а також обґрунтовано доцільність їх використання.

1.3.1 Java та Spring Boot

При виборі технологій для серверної частини важливо враховувати вимоги до надійності, масштабованості та зручності розробки. Використання мови програмування Java, яка протягом тривалого часу залишається одним із стандартів у побудові корпоративних систем є дуже доцільним у цьому випадку. Вона забезпечує стабільність, швидкість та кросплатформеність завдяки віртуальній машині JVM [15]. Це особливо важливо для системи, яка потенційно може розгортатися у різних середовищах.

Основним фреймворком був обраний Spring Boot, який значно спрощує процес створення серверних застосунків. Його ключовою перевагою є автоконфігурація, що дозволяє зменшити обсяг шаблонного коду та зосередитись на бізнес-логіці. Крім того, Spring Boot має дуже велику екосистему та надає вбудовані механізми для створення REST API, інтеграції з базами даних, реалізації безпеки та обробки запитів [16].

З позиції проєктування така комбінація технологій виглядає практичною, оскільки вона дозволяє швидко створити структурований та розширюваний бекенд, який доволі легко адаптується до змін вимог. Водночас, зріла екосистема Spring забезпечує достатню гнучкість для реалізації як простих, так і більш складних сценаріїв взаємодії з даними.

1.3.2 Сховища даних PostgreSQL та Redis

Ключовим аспектом при проєктуванні системи є вибір сховища для основних даних, оскільки саме воно визначає надійність, цілісність і ефективність роботи з інформацією. PostgreSQL була обрана як основна реляційна база даних завдяки своїй зрілості, стабільності та відповідності стандартам SQL. Вона широко використовується у високонавантажених системах і добре підходить для роботи зі структурованими даними різної складності [17].

PostgreSQL забезпечує потужні механізми транзакційності (ACID), що гарантує узгодженість і надійність даних навіть у разі збоїв. Система підтримує

ефективну індексацію, оптимізацію запитів і паралельну обробку, що дозволяє досягати високої продуктивності при роботі з великими обсягами інформації. Крім того, PostgreSQL надає широкі можливості для розширення включаючи користувацькі типи даних, функції та розширення, що дозволяють адаптувати базу даних під різні прикладні задачі [17].

З точки зору масштабування PostgreSQL також пропонує достатньо можливостей, включаючи реплікацію та розподіл навантаження. Це робить його придатним вибором навіть у випадку зростання обсягів даних або кількості користувачів. Таким чином, він виступає надійною основою для зберігання структурованої частини системи.

Окрім основного сховища, в проєкті використаний Redis як допоміжний інструмент. Він був застосований для зберігання сесій автентифікації та потенційного кешування даних у майбутньому [18]. Redis добре інтегрується з екосистемою Spring Boot, що дозволяє легко реалізувати роботу з сесіями та кешем через відповідні модулі. Завдяки роботі в оперативній пам’яті Redis забезпечує високу швидкість доступу, що дозволяє зменшити навантаження на основну базу даних та покращити загальну продуктивність системи.

1.3.3 Хмарні рішення Amazon Web Services

Оскільки основною функцією системи є зберігання наборів даних, питання вибору інфраструктури для їх розміщення є критично важливим. Воно безпосередньо впливає на вартість експлуатації, масштабованість та доступність системи. У цьому контексті доцільно розглядати використання хмарних рішень, зокрема платформи Amazon Web Services, яка надає широкий спектр сервісів для побудови подібних систем.

Для зберігання самих файлів наборів даних раціонально використовувати Amazon S3. Цей сервіс забезпечує високу надійність, практично необмежену масштабованість та модель оплати, орієнтовану на фактичне використання. Це дозволяє ефективно керувати витратами, особливо на початкових етапах розвитку

системи, а також уникнути необхідності підтримки власної файлової інфраструктури [19].

Для роботи з електронними повідомленнями доцільно застосувати Amazon SES (Simple Email Service), який може використовуватись для підтвердження реєстрації, відновлення доступу та інших сценаріїв взаємодії з користувачами [20].

Загалом, Amazon Web Services є зручним і практичним рішенням, яке дозволяє делегувати інфраструктурні завдання та зосередитись на реалізації логіки застосунку.

1.3.4 Використання Angular

Для реалізації клієнтської частини системи важливо обрати інструмент, який забезпечує створення структурованого, масштабованого та зручного інтерфейсу користувача. У цьому контексті доцільним є використання Angular як одного з провідних фреймворків для побудови односторінкових застосунків (SPA) [21].

Angular пропонує чітку архітектуру, засновану на компонентах, що дозволяє розділяти логіку інтерфейсу на незалежні частини. Це спрощує розробку, тестування та подальшу підтримку застосунку. Крім того, вбудовані механізми для роботи з HTTP-запитами, формами та маршрутизацією роблять його придатним для створення повноцінного клієнтського інтерфейсу, який взаємодіє з серверною частиною через API [22].

З точки зору проєктування Angular є доцільним вибором, оскільки він дозволяє створити узгоджений і передбачуваний користувацький досвід. Це особливо важливо для системи, орієнтованої на роботу з наборами даних, де зручність інтерфейсу безпосередньо впливає на ефективність взаємодії користувача з платформою.

Підсумовуючи, було розглянуто предметну галузь репозиторіїв відкритих наборів даних, їхнє значення для зберігання, поширення та повторного використання інформації. Також було проаналізовано роль відкритих даних у суспільстві й економіці та розглянуто існуючі платформи, зокрема Harvard Dataverse, Zenodo та Kaggle. Ці рішення мають широкий функціонал, однак через

універсальність можуть бути складними для користувачів, яким потрібна передусім зручна робота саме з наборами даних.

Окрім цього, було визначено основні технології для реалізації розроблюваної системи, якими є Java та Spring Boot для серверної частини, PostgreSQL і Redis для роботи з даними, Amazon Web Services для зберігання файлів та інфраструктурних задач, а також Angular для клієнтської частини. Таким чином, проведений аналіз підтверджує доцільність створення спеціалізованого репозиторію відкритих наборів даних, який буде простішим, зрозумілішим і більш сфокусованим на основних сценаріях роботи з даними.

2 ПРОЄКТУВАННЯ ЗАСТОСУНКУ

2.1 Функціональні та нефункціональні вимоги

Аналіз та порівняння існуючих платформ, проведені у попередньому розділі показали, що сучасні репозиторії, такі як Harvard Dataverse, Zenodo та Kaggle, хоч і поєднують широкий спектр можливостей, але саме ця універсальність часто ускладнює базові сценарії роботи з наборами даних. У межах даного проекту обрано інший підхід, що полягає у свідомому обмеженні функціоналу системи та увазі на основних процесах. Такими процесами є створення, опис, збереження, версіонування та публікація наборів даних. Цей підхід дозволить зменшити складність взаємодії з платформою та забезпечити більш узгоджене середовище для користувача. На основі вищесказаного – сформуємо функціональні та нефункціональні вимоги застосунку.

Застосунок який розробляється являє собою платформу-репозиторій, орієнтований на підтримку основних процесів управління та публікації відкритих наборів даних. Система має забезпечувати середовище для завантаження, опису, збереження, оприлюднення та подальшого використання даних відповідно до потреб користувачів.

З огляду на функціонал системи можна виділити три основні категорії користувачів.

1. *Неавтентифіковані користувачі*

Першу категорію становлять неавтентифіковані користувачі, які можуть переглядати каталог опублікованих наборів даних, відкривати сторінки окремих наборів, переглядати історію версій та експортувати метадані.

2. *Автентифіковані користувачі*

Друга категорія – автентифіковані користувачі, які отримують можливість створювати власні набори даних, редагувати їх у вигляді чернеток, додавати файли, зберігати зміни та виконувати публікацію.

3. *Власники наборів даних*

Третю категорію становлять автентифіковані користувачі, які є власниками наборів даних та для яких система додатково забезпечує можливість модифікації вже існуючих записів, перегляду поточного чернеткового стану та видаленню чернеткових версій. Такий поділ ролей відповідає логіці репозиторію, в якому публічний доступ до опублікованого вмісту поєднується з контрольованим редагуванням з боку автора.

Функціональні вимоги.

До основних функціональних вимог системи належать вимоги, пов’язані з керуванням обліковими записами користувачів. Застосунок повинен забезпечувати реєстрацію користувача із перевіркою коректності введених даних, надсилання повідомлення для підтвердження адреси електронної пошти, верифікацію облікового запису та подальшу автентифікацію лише після підтвердження електронної адреси. Також система повинна підтримувати завершення сесії, відновлення інформації про поточного користувача та відображення базових даних профілю. Ці можливості є базовою передумовою для захищеного доступу до функцій створення та редагування наборів даних.

Наступну групу становлять функціональні вимоги до роботи з наборами даних. Система повинна дозволяти створення нового набору даних у вигляді чернетки (Draft) із заповненням основних метаданих. За основу цих самих основних метаданих у застосунку взяті mandatory та широко використовувані recommended атрибути стандарту метаданих для наборів даних DataCite [23][24]. Цими полями є назва, опис, автори, мови, тематики та тип доступу. Після створення, користувач отримує чернеткову версію, яку має мати можливість редагувати, завантажувати файли підтримуваних форматів, зберігати зміни та публікувати результат. При цьому важливою вимогою є підтримка версіонування. Після першої публікації всі наступні зміни не повинні змінювати вже опубліковану версію, а мають створювати нову, чернеткову версію. Також необхідно врахувати, що зміни метаданих і зміни файлів мають мати різний вплив

на формування номера наступної версії, що забезпечить більш точне відображення характеру внесених змін.

Система повинна забезпечувати публічний доступ до опублікованих наборів даних. Для цього мають бути створені каталог наборів даних, сторінка перегляду конкретного набору та список версій з підтримкою перегляду окремої версії. Додатково застосунок повинен підтримувати експорт метаданих у найпоширеніших форматах Dublin Core та DataCite [23][25], що підвищить сумісність репозиторію з багатьма іншими інформаційними системами та відповідає практикам поширення наукових і відкритих даних. Для наборів із обмеженим доступом система має забезпечувати відкритість метаданих, але не повинна надавати стороннім користувачам доступ до завантаження файлів.

Оскільки в даному проєкті акцент зроблено на спрощенні основних сценаріїв роботи з даними, окрему увагу потрібно приділити функціональним вимогам до інтерфейсу створення та редагування наборів даних. Інтерфейс повинен підтримувати послідовний і зрозумілий перехід від створення метаданих до додавання файлів і публікації. Користувач повинен мати змогу вільно працювати з чернеткою у межах якої виконуються редагування полів метаданих, тимчасове додавання файлів, збереження та подальша публікація. Підвищення ефективності досягається шляхом зменшення простору можливостей в цих сценаріях до найбільш необхідних для предметної області.

Нефункціональні вимоги.

Ключовим аспектом нефункціональних вимог, є перш за все вимоги до безпеки. Застосунок повинен забезпечувати надійне зберігання облікових даних користувачів, перевірку прав доступу до операцій редагування, публікації та доступу до приватних або restricted-ресурсів. Також важливо гарантувати, що неавторизований користувач не зможе будь-яким чином видозмінювати чужі набори даних та отримувати доступ до чернеткових станів, які не призначені для публічного перегляду. При цьому, потрібно забезпечити обробку помилок автентифікації, конфліктних ситуацій і запитів до неіснуючих ресурсів.

Наступними по суттєвості є вимоги до цілісності та надійності даних. Система повинна гарантувати незмінність опублікованих версій, коректне збереження історії змін, узгоджене керування чернетками та перевірку коректності метаданих і файлів перед збереженням. Це особливо важливо для застосунку орієнтованого на публікацію наборів даних, який має забезпечувати довіру до опублікованої інформації.

На рівні серверної та клієнтської частини повинна бути реалізована валідація, задача якої не допускати порушення встановлених правил чи вимог, які стосуються таких аспектів, як наприклад обов’язкові поля, підтримувані формати файлів чи їх розмір. Такі елементи, як допустимі мовні коди мають перевірятись на відповідність стандарту, який використовує застосунок, в даному випадку – ISO 639-1 [26].

До нефункціональних вимог також належать вимоги щодо зручності використання та узгодженості інтерфейсу. У попередньому розділі було виявлено проблему перевантаженості сучасних універсальних платформ, тому важливо, щоб у даному застосунку інтерфейс був побудований так, щоб основні сценарії були максимально зрозумілими та прямолінійними. Форми введення мають супроводжуватися перевіркою даних та повідомленнями про помилки, поясненнями причин недоступності певних дій, індикаторами стану збереження та завантаження.

Сукупність вищесформульованих функціональних і нефункціональних вимог відображає основну ідею проєкту. На відміну від універсальних платформ, що прагнуть охопити максимально широкий спектр задач, спроектовані умови застосунку зосереджуються на ключових процесах предметної області, що дозволяє досягти більшої зрозумілості інтерфейсу та логіки роботи системи.

2.2 Моделювання сценаріїв роботи

Моделювання сценаріїв роботи застосунку використовується для формалізації основних сценаріїв взаємодії користувача з репозиторієм. Оскільки

застосунок орієнтований на підтримку повного циклу роботи з наборами даних, важливо виділити декілька ключових сценаріїв:

1. Реєстрація та автентифікація користувача
2. Створення набору даних
3. Редагування та публікація набору даних
4. Перегляд опублікованих матеріалів іншими користувачами.

Моделювання цих сценаріїв дозволяє виділити основні точки взаємодії між користувачем і програмним застосунком. Для загального розуміння цих процесів, спочатку використана діаграма прецедентів, яка відображає основні ролі користувачів та їх взаємодію з функціями системи.

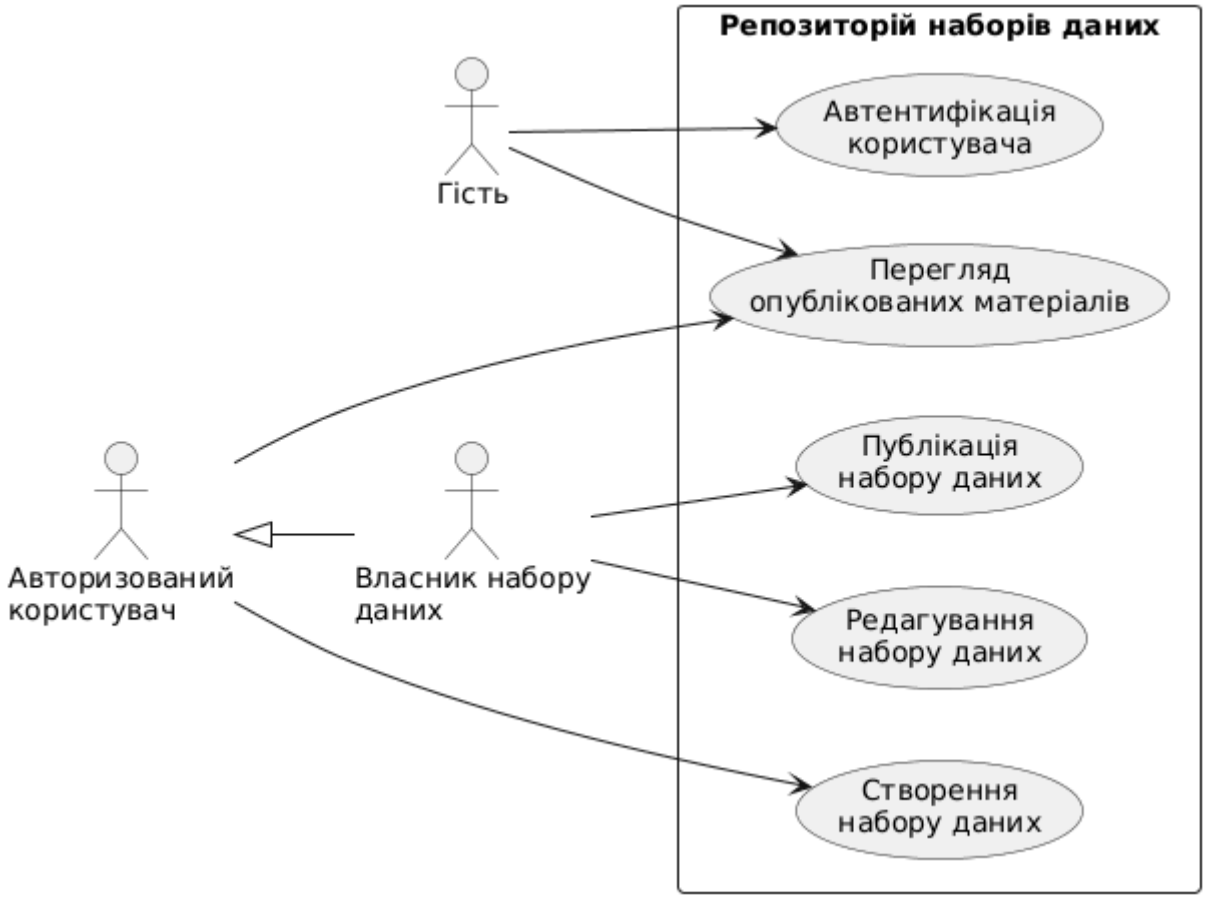


Рис. 2.1 Діаграма прецедентів системи

Сценарій реєстрації та автентифікації користувача описує початкову взаємодію користувача із системою. У ньому користувач створює обліковий запис, підтверджує адресу електронної пошти та після цього виконує вхід до системи.

Цей сценарій є обов’язковим для доступу до функцій, які пов’язані зі створенням і керуванням власними наборами даних. Хоча самі дії реєстрації та автентифікації є послідовно пов’язаними, ці процеси логічно відокремлені. Для їх подання використано дві окремі діаграми активності.

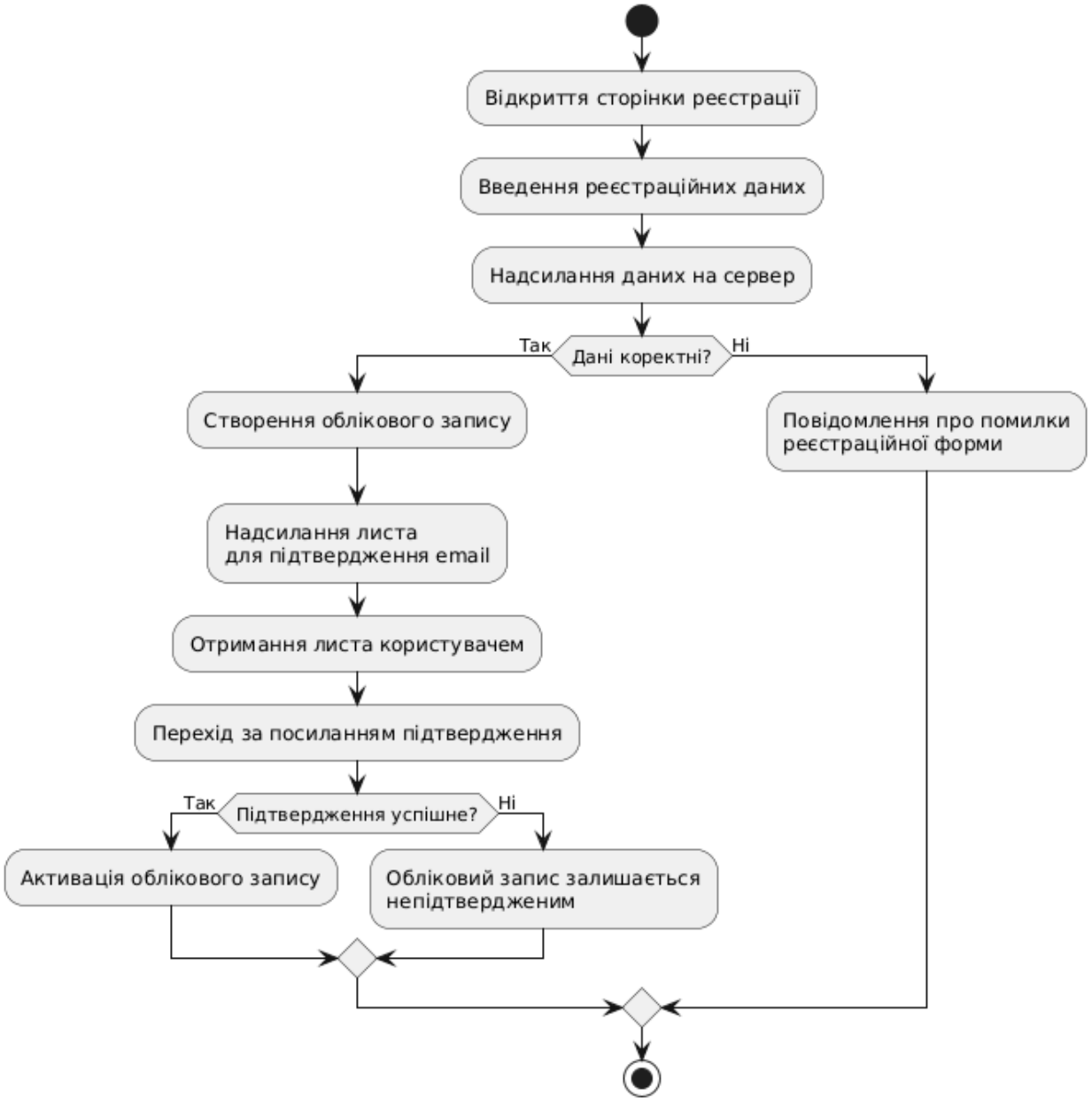


Рис. 2.2 Діаграма активності сценарію реєстрації користувача

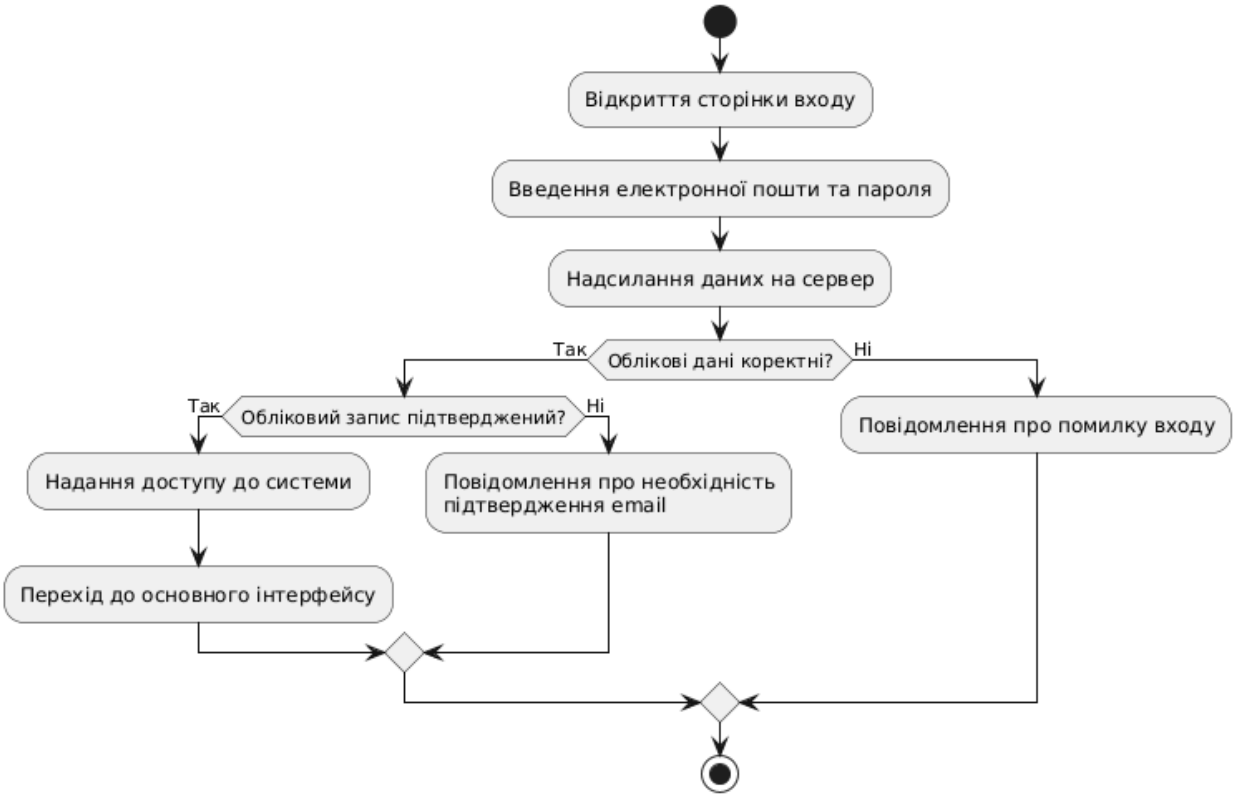


Рис. 2.3 Діаграма активності сценарію автентифікації користувача

Сценарій створення набору даних визначає початковий етап роботи автора з новим ресурсом. Під час цього сценарію користувач переходить до форми створення набору даних, вводить основні обов’язкові для ресурсу метадані та зберігає запис, який створюється у статусі чернетки.

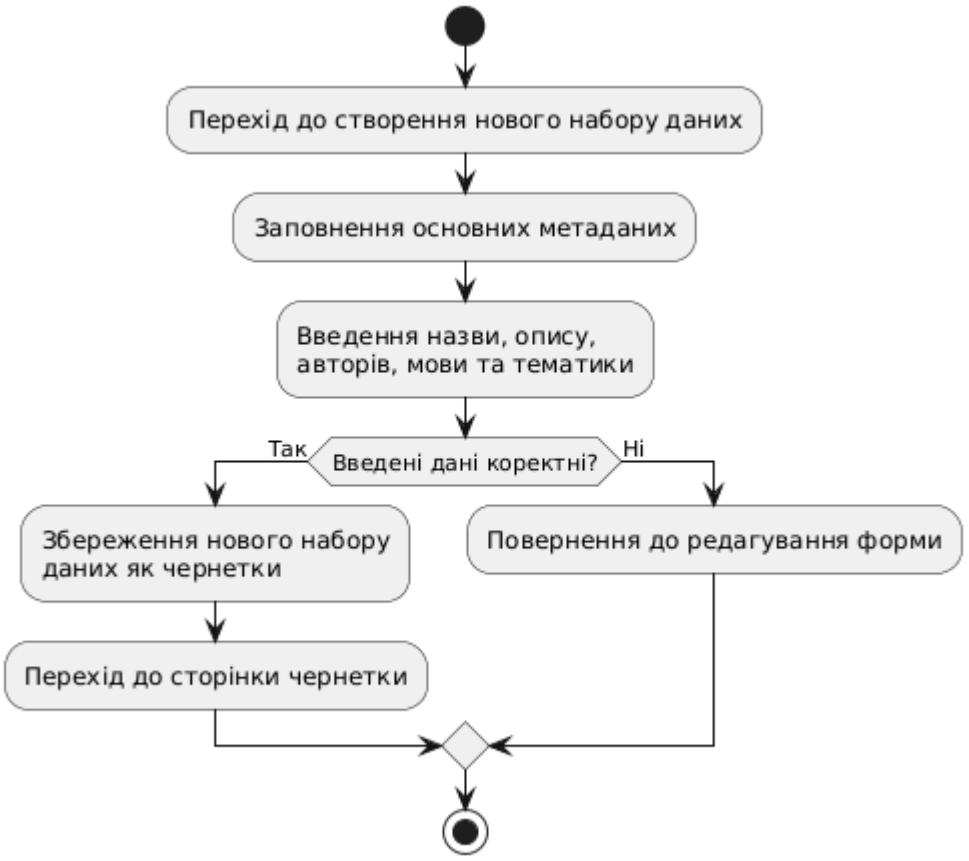


Рис. 2.4 Діаграма активності сценарію створення набору даних

Редагування та публікації набору даних є основним робочим процесом автора після створення чернетки. Сценарій включає в себе зміну метаданих, додавання або оновлення файлів, збереження внесених змін та подальшу публікацію набору даних. Ці дії є об’єднаними в один сценарій, оскільки вони є одним послідовним процесом.

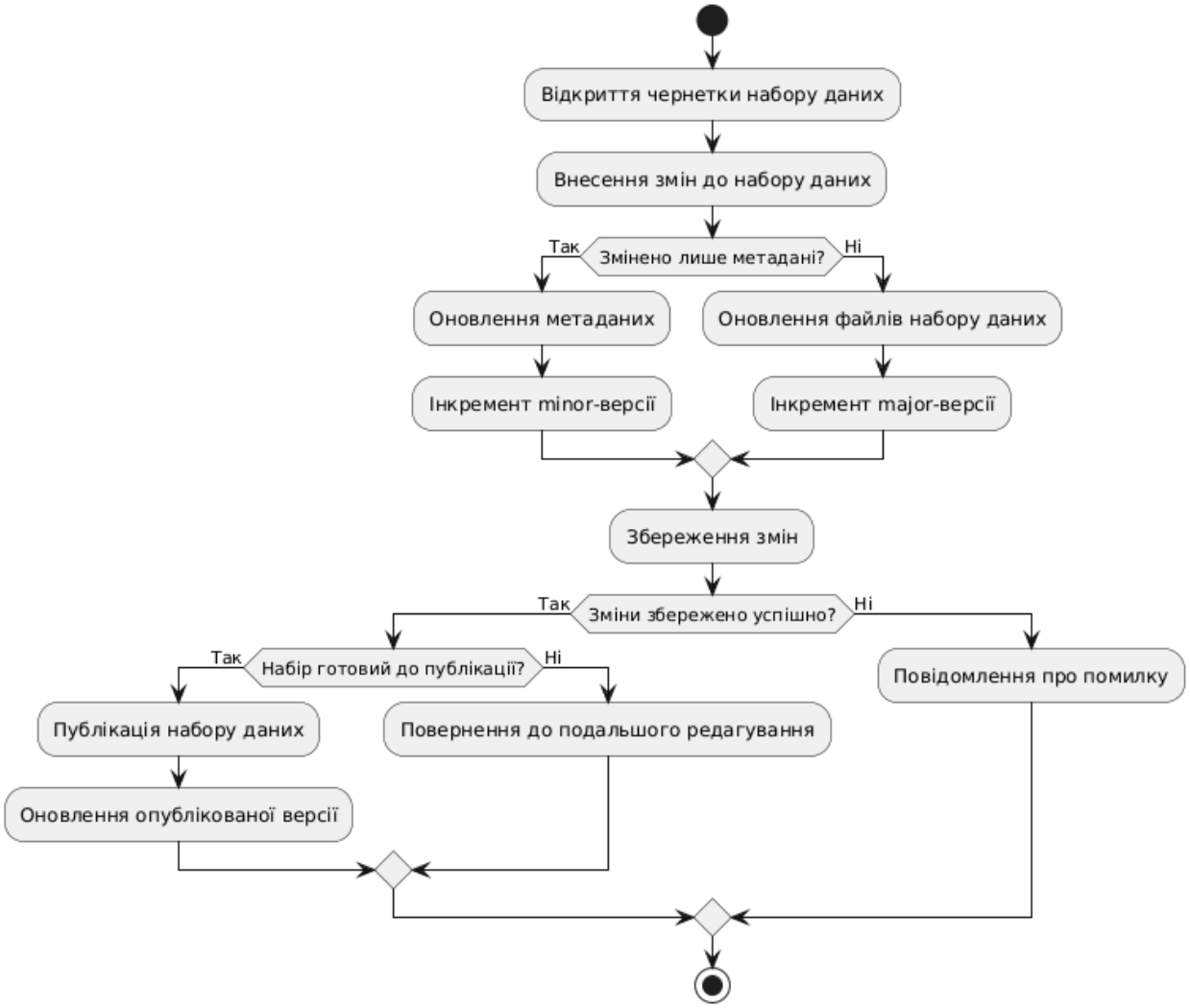


Рис. 2.5 Діаграма активності сценарію редагування та публікації набору даних

Перегляд опублікованих матеріалів іншими користувачами є публічною стороною роботи репозиторію. Користувач, у межах цього процесу, переглядає каталог наборів даних, відкриває сторінку окремого набору, ознайомлюється з його метаданими та, за умови відповідного типу доступу, завантажує файли. Цей сценарій показує використання опублікованих ресурсів іншими учасниками системи.

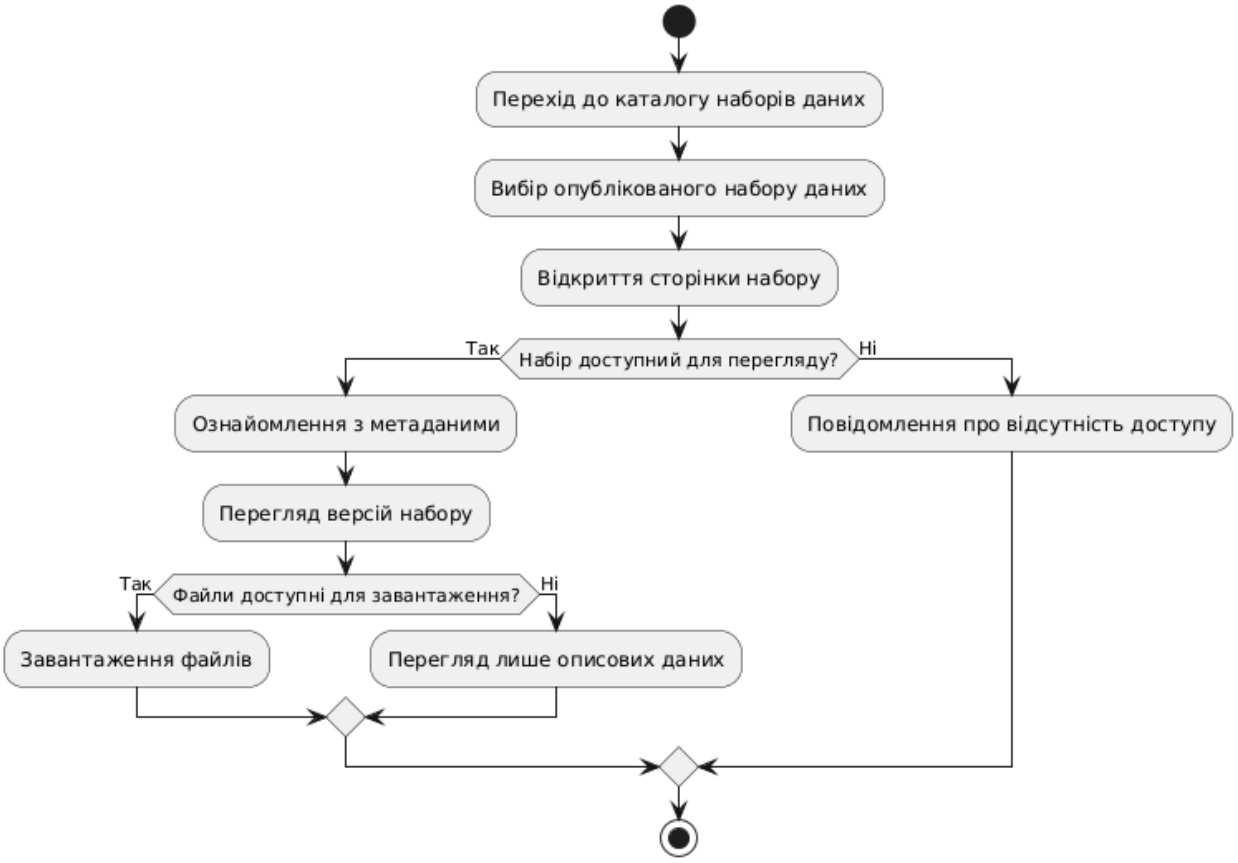


Рис. 2.6 Діаграма активності сценарію перегляду опублікованих матеріалів іншими користувачами

Наведені сценарії дозволяють узагальнити самі основні способи взаємодії користувачів із застосунком та явно відобразити логіку його роботи. Загальна діаграма прецедентів визначає місце кожного сценарію, а окремі діаграми активності цих сценаріїв деталізують послідовність дій окремих процесів. Це створює основу для подальшого проєктування архітектури системи, структури даних і користувацького інтерфейсу.

2.3 Проєктування архітектури застосунку

2.3.1 Загальна архітектура

З перспективи архітектури застосунок побудований як клієнт-серверний веб застосунок, у якому відповідальність чітко розділена між клієнтською та серверною частинами (Backend та Frontend).

Клієнтська частина є односторінковим застосунком, що відповідає за взаємодію з користувачем. Її відповідальностями є:

- навігація між сторінками;
- введення та попередня перевірка даних;
- відображення каталогів наборів даних;
- відображення форм створення та редагування чернеткових ресурсів;
- відображення сторінок профілю;
- відображення історії версій і станів завантаження.

Вона не містить основних бізнес-правил системи та лише забезпечує зручний спосіб їх використання через графічний інтерфейс.

Серверна частина є центральним елементом системи, оскільки саме вона відповідає за:

- обробку запитів;
- автентифікацію користувачів;
- керування життєвим циклом наборів даних;
- перевірку прав доступу;
- версіонування;
- збереження метаданих;
- роботу з файловим сховищем;
- експорт метаданих у стандартизованих форматах.

Backend також виступає єдиним джерелом істини щодо стану користувачів, наборів даних, чернеток, опублікованих версій і пов’язаних із ними файлів.

Взаємодія між frontend і backend відбувається через REST API [27]. Клієнтський застосунок надсилає HTTP-запити до серверної частини для виконання таких операцій, як:

- реєстрація та вхід користувача;
- створення набору даних;
- завантаження файлів;
- публікація чернеткових версій;
- перегляд конкретних версій;
- отримання списку опублікованих наборів;
- редагування метаданих;
- експорт метаданих.

У відповідь сервер повертає структуровані дані, які frontend перетворює у відповідні стани інтерфейсу.

Загальна архітектура застосунку може бути представлена трьома логічними рівнями:

1. Presentation рівень – клієнтський Angular SPA, який відповідає за інтерфейс, маршрутизацію, локальні стани сторінок і виклики API.
2. Application рівень – серверна частина, яка реалізує сценарії, бізнес-правила, авторизацію та обробку запитів.
3. Infrastructure рівень – база даних, об’єктне сховище, механізм сесій, поштовий сервіс та інші зовнішні технічні залежності.

Такий поділ архітектури дозволяє зберегти слабку зв’язаність між частинами системи та розділити їх відповідальності. Frontend може змінювати спосіб представлення інформації без зміни бізнес-логіки, а backend може змінювати внутрішню реалізацію сховища, безпеки або обробки файлів без необхідності переписувати клієнтський інтерфейс. Основним контрактом між ними залишається REST API.

2.3.2 Проєктування серверної архітектури

Backend спроектований як модульний серверний застосунок з використанням принципів Domain-Driven Design [28] та гексагональної архітектури [29]. Таке рішення було обране тому, що система має доволі виражену предметну область, яка включає в себе користувачів, набори даних, чернетки, версії, файли, метадані, правила публікації, обмеження доступу та інші. Ці аспекти та правила, які відносяться до них, не повинні залежати від конкретного веб фреймворку, бібліотек, бази даних чи будь-яких інших технічних залежностей.

Головна ідея гексагональної архітектури полягає в тому, що бізнес-логіка застосунку розміщується в центрі архітектури, а зовнішні технології підключаються до неї через порти та адаптери [28]. Це означає, що внутрішні рівні повинні не залежати напряму від REST-контролерів, JPA, S3-сховища чи інших інфраструктурних механізмів. Вони працюють з абстрактними інтерфейсами – портами, а конкретні технічні реалізації – адаптери – розміщуються в інфраструктурному шарі.

Гексагональна архітектура поділяється на вхідну та вихідну сторони. Вхідна сторона, або driver side, охоплює механізми, які ініціюють виконання сценаріїв. У цьому застосунку до неї належать REST API та контролери, через які frontend звертається до backend. Вихідна сторона, або driven side, охоплює залежності, до яких застосунок звертається під час виконання сценаріїв. До неї належать база даних, файлове сховище, поштовий сервіс, механізми безпеки, обробки файлів та інші інфраструктурні компоненти. Виходячи з цього можна сказати, що адаптери запускають роботу системи, а вихідні адаптери забезпечують доступ до зовнішніх ресурсів, необхідних для виконання бізнес-операцій.

У межах цього підходу сама серверна частина також поділяється на модулі Domain, Application, Infrastructure та Bootstrap. Domain містить бізнес-модель і правила предметної області. Application описує прикладні сценарії та порти взаємодії. Infrastructure імплементує адаптери для REST API, бази даних, файлового сховища, безпеки та інших зовнішніх механізмів. Bootstrap відповідає за запуск, конфігурацію та складання застосунку в єдиний виконуваний артефакт.

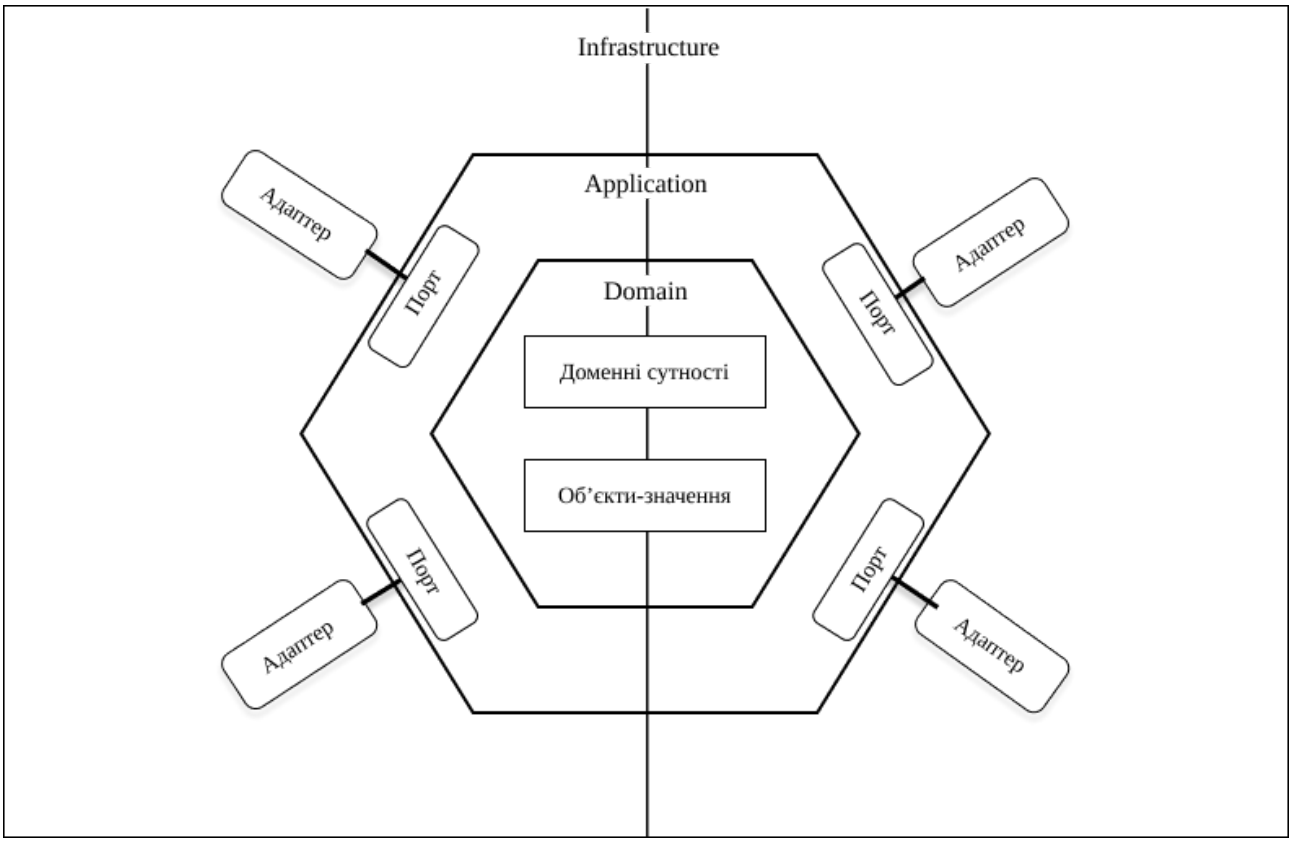


Рис. 2.7 Діаграма гексагональної архітектури

Domain рівень

Domain рівень є внутрішнім ядром системи. Він містить основні бізнес-сутності та правила їхньої поведінки. До ключових доменних сутностей належать:

- користувач;
- набір даних;
- версія набору даних;
- метадані набору даних;
- файл набору даних;
- об’єкт сховища, тощо.

На цьому рівні визначаються правила валідності та поведінки ключових об’єктів системи. Доменний рівень описує, коли користувач може бути створений, коли токен підтвердження є дійсним, за яких умов можна редагувати чернетку, як змінюється номер версії та як публікація відрізняє зміни метаданих від змін файлового складу. Особливо важливим є те, що логіка версіонування належить

саме домену, а не контролерам чи базі даних. Також, наприклад, зміна лише метаданих може формувати мінорну версію, тоді як зміна файлів - мажорну. Це є предметним правилом, тому воно має бути ізольоване від інфраструктурних деталей.

Application рівень

Application рівень відповідає за сценарії використання системи, які в межах застосунку мають назву “Use Cases”. Він координує доменні об’єкти, викликає необхідні порти та визначає послідовність виконання операцій. На цьому рівні розміщені такі сценарії:

- реєстрація користувача;
- підтвердження електронної пошти;
- вхід і вихід із системи;
- створення набору даних;
- редагування метаданих;
- завантаження файлів;
- публікація чернеткових версій;
- видалення чернеткових версій;
- отримання опублікованих наборів;
- отримання історії версій;
- експорт метаданих;
- керування профілем користувача.

У гексагональній архітектурі даний рівень взаємодіє з іншими частинами системи через інтерфейси, які називаються “портами”. Порт є архітектурним контрактом, який описує потрібну дію без прив’язки до конкретної технології. Це дозволяє use case залишатися описом сценарію, а не набором деталей роботи з зовнішніми елементами застосунку. Порти, згідно архітектури, поділяються на вхідні та вихідні [28].

Вхідні порти визначають операції, які backend надає назовні. До них належать сценарії створення набору даних, редагування метаданих, завантаження файлів, публікації чернеткових версій, отримання профілю користувача або

експорту метаданих. REST-контролери в цьому випадку виступають вхідними адаптерами. Вони приймають HTTP-запити, перетворюють їх у команди прикладного рівня та передають виконання відповідному use case.

Вихідні порти визначають залежності, які потрібні прикладному рівню для виконання сценаріїв. Через них прикладні сервіси звертаються до сховища даних, файлового сховища, механізму перевірки файлів, хешування паролів, публікації подій або обробки зображень. При цьому сервіс не залежить від PostgreSQL, S3, SMTP чи інших конкретних технологій. Ці технології підключаються через інфраструктурні адаптери.

Завдяки цьому backend залишається гнучким до змін. Наприклад, якщо виникне потреба замінити S3-сумісне сховище на інше рішення, основна бізнес-логіка не повинна змінюватися. Достатньо реалізувати новий адаптер для відповідного вихідного порту.

Infrastructure рівень

Інфраструктурний рівень містить адаптери, які з’єднують бізнес-логіку з зовнішнім середовищем. До нього належать REST-контролери, конфігурація безпеки, адаптери бази даних, адаптер об’єктного сховища, адаптер поштового сервісу, обробка винятків, планові задачі очищення, механізми роботи із зображеннями та файлами.

У гексагональній архітектурі REST-контролери розглядаються як вхідні адаптери. Вони приймають HTTP-запити, опрацьовують отримані дані та викликають відповідні сценарії. Водночас JPA-репозиторії, S3-сховище, поштовий сервіс або Redis є вихідними адаптерами, оскільки вони реалізують залежності, потрібні прикладному рівню.

Окреме значення має розділення DTO між інфраструктурним і прикладним шарами. DTO рівня API описують форму HTTP-запитів і відповідей та працюють із даними в тому вигляді, у якому вони надходять від клієнта. На цьому етапі дані ще не є частиною доменної моделі, можуть бути неповними або некоректними та проходять валідацію на рівні API.

Для подальшої обробки ці дані мають бути приведені до форми, зрозумілої прикладному та доменному рівням. Зокрема, частина значень перетворюється на доменні об’єкти-значення, які вже мають власні правила валідності. Тому використовувати “value-objects” [28] безпосередньо в DTO запитів недоцільно. Завдяки цьому API-контракт, прикладні сценарії та доменна модель залишаються розділеними.

Bootstrap рівень

Bootstrap – це композиційний шар, який збирає застосунок у виконуваний серверний продукт та є його вхідною точкою. Він відповідає за запуск, конфігурацію середовища, міграції бази даних, шаблони повідомлень і пакування. Цей рівень не містить бізнес-логіки. Його завдання – зв’язати доменний, прикладний та інфраструктурний рівні в одну працездатну систему.

Переваги гексагональної архітектури для застосунку

Для цього застосунку гексагональна архітектура є особливо доцільною через наявність зовнішніх залежностей, які потенційно можуть змінюватися протягом життєвого циклу системи. Наприклад, для збереження файлів може використовуватися S3-сумісне сховище, але на різних етапах розвитку проєкту може виникнути потреба перейти на локальне сховище, дешевший S3-провайдер, хмарне сховище іншого сервісу або окреме корпоративне рішення. Якщо б бізнес-логіка напряму залежала від конкретного S3 SDK, така зміна вимагала б переписування дуже значної частини застосунку. Завдяки порту об’єктного сховища заміна постачальника зводиться переважно до створення або зміни відповідного адаптера.

Аналогічна перевага стосується баз даних, поштового сервісу, механізму обробки зображень, перевірки файлів та сесій автентифікації. Ядро системи має знати лише те, що йому потрібно зберегти об’єкт, надіслати подію, перевірити файл або закодувати пароль. А те, як саме це буде виконано є відповідальністю інфраструктури.

У результаті, серверний застосунок проєктується як система з чітким розподілом відповідальності:

- domain зберігає бізнес-правила та перевіряє інваріанти;
- application описує сценарії використання і порти;
- infrastructure реалізує взаємодію з HTTP, базою даних, сховищем, безпекою та зовнішніми сервісами через адаптери;
- bootstrap збирає застосунок у виконуваний продукт.

Такий підхід робить backend більш придатним до розвитку, тестування та заміни технічних компонентів без порушення основної бізнес-логіки.

2.3.3 Проєктування клієнтської архітектури

Frontend був спроєктований як односторінковий застосунок на основі Angular [30]. Його головне завдання – це надати користувачу зручний інтерфейс для роботи з репозиторієм відкритих даних.

Клієнтська частина не дублює бізнес-логіку backend. Натомість вона виконує роль презентаційного рівня: збирає введені дані, виконує базову клієнтську валідацію, відображає стани сторінок і передає запити до REST API. Остаточні рішення щодо прав доступу, стану набору даних, публікації, версіонування та доступності файлів залишаються відповідальністю серверного застосунку.

Архітектура frontend поділяється на кілька логічних частин.

Глобальна оболонка та маршрутизація

Застосунок має глобальну оболонку, яка містить основну навігацію, заголовок сторінки, посилання на створення набору даних, кнопку профілю та область для відображення поточного маршруту. Маршрутизація визначає основні сторінки системи:

- головна сторінка;
- реєстрація;
- вхід;
- підтвердження електронної пошти;
- каталог наборів даних;
- інструкція користувача;

- створення нового набору;
- редактор чернетки;
- сторінка опублікованого набору;
- сторінка конкретної версії;
- профіль користувача.

Захищені маршрути мають перевіряти наявність активної сесії. Якщо користувач не автентифікований, його потрібно перенаправляти на сторінку входу зі збереженням адреси, на яку він намагався перейти. Це дозволяє після входу повернути користувача до потрібного сценарію.

API-шар

Окремим елементом клієнту є API-шар, який централізує взаємодію з серверним застосунком. Він відповідає за формування HTTP-запитів, передавання cookie-сесії, встановлення заголовків для JSON-запитів, коректну роботу з multipart-завантаженнями [31] файлів, обробку помилок і перетворення відповідей backend у зручні для інтерфейсу типи.

Такий шар потрібен для того, щоб компоненти сторінок не працювали напряму з низькорівневими HTTP-викликами. Компонент має викликати функцію на кшталт створення набору даних, завантаження файлів або отримання профілю, а деталі HTTP-запиту мають залишатися всередині API-модуля.

Оскільки frontend взаємодіє з backend через REST API, важливо спроєктувати клієнтські типи відповідно до серверного контракту. До таких типів належать:

- короткий опис набору даних для каталогу;
- метадані чернеткової версії;
- детальна інформація про набір даних;
- інформація про версію;
- опис файлу;
- поточний користувач;
- публічний профіль користувача.

Клієнтський застосунок також має бути стійким до незначних змін у форматі відповіді серверу, тому доцільно виділити окремі mapper-функції, які перетворюють серверні DTO у внутрішні клієнтські моделі. Таким чином, API-шар відокремлює компоненти інтерфейсу від деталей взаємодії з backend, вони працюють із підготовленими клієнтськими моделями, а HTTP-запити, мапінг DTO та обробка помилок залишаються всередині API-шару.

Стан автентифікації

Оскільки система використовує session-cookie підхід, frontend не повинен зберігати токени доступу у local storage [32]. Авторитетним джерелом стану входу залишається backend-сесія. Під час запуску застосунок має виконувати запит про поточного користувача до серверного застосунку, щоб відновити або скинути локальний стан сесії.

На рівні клієнту достатньо зберігати мінімальний стан:

- поточного автентифікованого користувача;
- профіль користувача для відображення в інтерфейсі;
- факт завантаження або відсутності сесії.

Цей підхід дозволяє спростити клієнтську архітектуру та зменшує ризик розбіжності між реальним станом автентифікації серверного застосунку і тим, що показує інтерфейс.

Компоненти сторінок і локальний стан

Більшість станів клієнтського застосунку достатньо тримати на рівні конкретних сторінок або компонентів. Наприклад, сторінка каталогу має власний стан завантаження, помилки, порожнього результату та поточної сторінки пагінації. Хоча деякі компоненти мають значно складніший стан, як наприклад редактор наборів даних, для такого застосунку немає потреби вводити важкий глобальний state management, якщо більшість станів локальні. Глобальним має залишатися лише те, що справді використовується між різними сторінками, у нашому випадку це здебільшого стан автентифікації та профіль користувача.

Взаємодія з REST API

Frontend взаємодіє з backend через REST API, який охоплює основні сценарії роботи застосунку. Окремо варто розглянути такі сценарії, як наприклад завантаження файлів.

У редакторі чернеткової версії вибрані користувачем файли спочатку додаються лише до локального стану і позначаються як підготовлені до завантаження. На цьому етапі клієнтська частина може перевірити їхній розмір і розширення, але серверний стан ще не змінюється. Фактичне передавання файлів відбувається лише під час збереження версії через multipart-запит до backend. Після цього сервер перевіряє файл, зберігає його у файловому сховищі та повертає дані про збережений об’єкт, які frontend відображає у списку файлів.

Такий підхід дозволяє відділити локальну підготовку змін в інтерфейсі від їхнього остаточного збереження на сервері та забезпечити надійніший контроль над станом об’єктів, уникаючи помилки під час їх збереження чи видалення. Це особливо важливо для таких сценаріїв, як редактор чернеток, де користувач може змінювати метадані й файловий склад до моменту явного збереження.

Валідація та стани інтерфейсу

Frontend має виконувати попередню валідацію там, де це покращує користувацький досвід, як наприклад в:

- обов’язковості полів;
- форматі email;
- мінімальній довжині пароля;
- наявності автора, назви, опису, мови, тематики;
- підтримуваних форматів файлів;
- максимальних розмірів файлу.

Водночас ця перевірка не замінює серверну валідацію. Backend залишається остаточним контролером коректності.

Кожна сторінка, яка залежить від API, має підтримувати такі явні стани:

- завантаження;
- успішне відображення;
- порожній результат;

- помилка;
- відсутність ресурсу;
- виконання дії.

Це важливо для застосунку з асинхронною взаємодією, оскільки користувач повинен розуміти, чи дані ще завантажуються або чи сталася помилка та чи ресурс справді відсутній.

Підсумовуючи, клієнтський застосунок проєктується як легкий, модульний Angular SPA, який цілком доповнює серверний застосунок. Backend зберігає бізнес-логіку та гарантує цілісність системи, а frontend забезпечує зрозумілу й контрольовану взаємодію користувача з цією логікою через REST API. Такий поділ формує цілісну архітектуру застосунку, де клієнтська й серверна частини мають окремі зони відповідальності, але працюють як єдина система.

2.3.4 Проєктування структури бази даних та моделі зберігання даних

Проєктування бази даних у застосунку виконується з урахуванням специфіки предметної області. Оскільки система підтримує повний життєвий цикл роботи з наборами даних та пов’язаними з ними сутностями, модель зберігання повинна забезпечувати цілісність, узгодженість і достатню гнучкість для подальшого розширення. У межах проєкту для цього використано комбінований підхід, за якого структуровані доменні дані зберігаються у реляційній базі даних, великі бінарні об’єкти – у зовнішньому файловому сховищі, а сесійні дані автентифікації – в окремому швидкодійному сховищі.

Основним сховищем системи є PostgreSQL, у якому зберігаються користувачі, набори даних та їх версії та файлові зв’язки. Використання саме реляційної СУБД зумовлене потребою у підтримці зв’язків між сутностями, унікальних обмежень, транзакційної цілісності та надійного керування зв’язками між сутностями [17]. Проте застосунок працює не лише з табличними даними, реальні файли наборів даних та аватари користувачів зберігаються в S3-сумісному сховищі об’єктів, а стан HTTP-сесій підтримується через Redis. Таким чином,

модель зберігання даних є багатокомпонентною та відповідає реальним функціональним потребам застосунку.

У логічній структурі бази даних можна виділити три основні блоки:

1. Перший блок пов’язаний з користувачами та автентифікацією і включає таблиці *users* та *email_verification_tokens*.
2. Другий блок описує набори даних і їх версії та містить таблиці *datasets*, *dataset_versions* і *dataset_version_files*.
3. Третій блок пов’язаний із файловим зберіганням і представлений таблицею *storage_objects*, яка використовується як зв’язок між реляційною моделлю та фізичним зберіганням файлів у зовнішньому сховищі.

Для явного подання взаємозв’язків між основними сутностями реляційної моделі бази даних доцільно використати ER-діаграму. Вона відображає ключові таблиці системи, їх основні атрибути і зв’язки між користувачами, наборами даних, версіями та файловими об’єктами. Особливістю моделі є те, що набір даних розглядається як коренева сутність, пов’язана з версіями, а файли представлені окремо як фізичні об’єкти зберігання та як елементи конкретних версій набору даних.

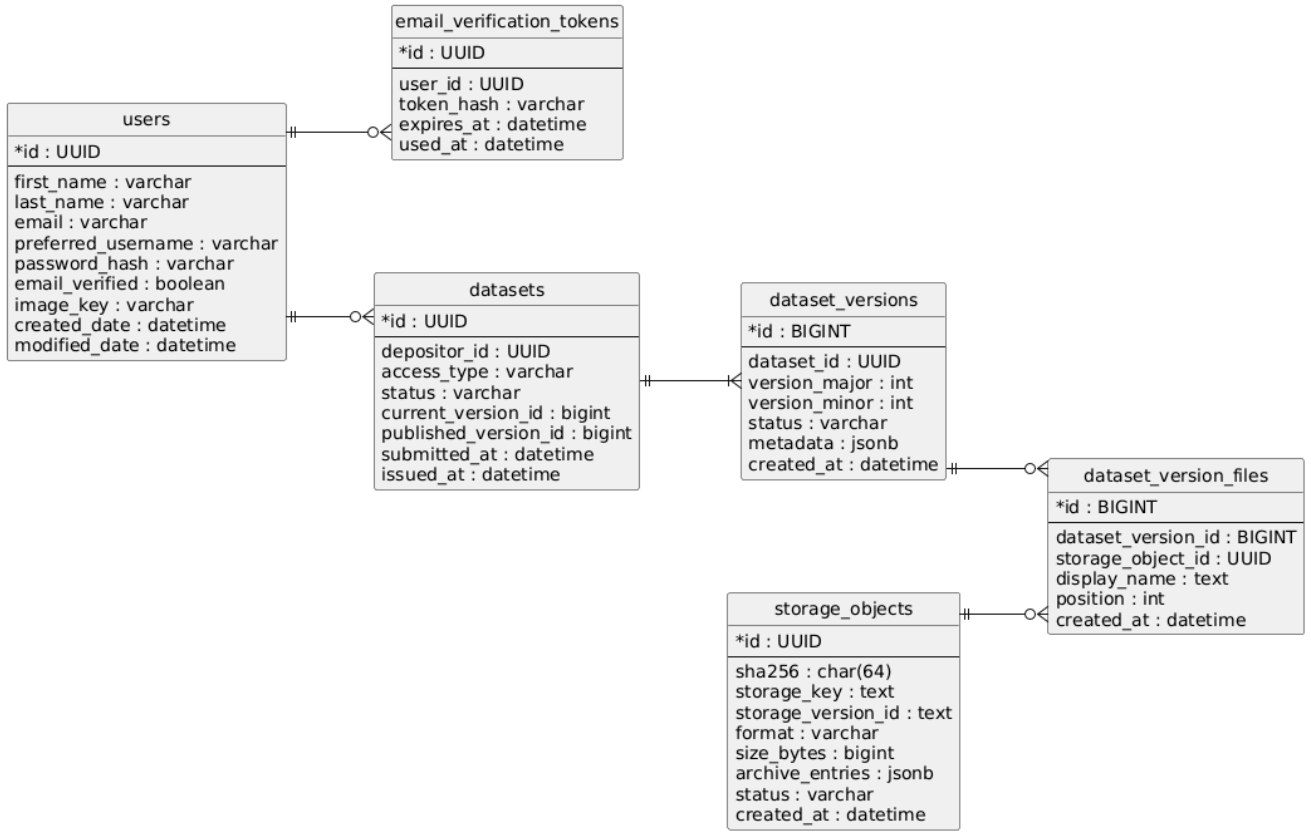


Рис. 2.8 ER-діаграма структури бази даних застосунку

Показані на діаграмі зв’язки відображають основну логіку зберігання даних у системі. Один користувач може бути власником багатьох наборів даних, кожен з яких має одну або більше версій. В свою чергу, кожна версія може містити набір файлів, пов’язаних із фізичними файловими об’єктами. Окремо на діаграмі також відображено зв’язок між користувачем і токенами підтвердження електронної пошти, а також спеціальні посилання з таблиці *datasets* на поточну й опубліковану версії набору даних.

Центральною сутністю моделі є таблиця *datasets*, яка відображає набір даних як логічний об’єкт. У ній зберігається тільки загальний стан ресурсу, власник набору, тип доступу, посилання на поточну версію та, за наявності, на останню опубліковану версію. Такий підхід дозволяє розділити рівень самого набору даних і рівень його окремих версій. Детальний вміст кожного набору зберігається саме в версії, та не в таблиці *datasets*, а в таблиці *dataset_versions*, де фіксуються номер версії, її статус, дата створення та метадані. Такий підхід до

зберігання дозволяє імплементувати важливу для предметної області модель, у якій опублікована версія не може бути змінена, а нові зміни формують чернетковий стан.

Також, важливою особливістю є спосіб зберігання метаданих набору даних. У межах даного проєктування вони розміщуються у полі *metadata* таблиці *dataset_versions* у форматі jsonb [33]. До складу цього об'єкта входять безпосередньо поля, які є метаданими набору даних. Використання типу *jsonb* для зберігання метаданих є доцільним, оскільки в межах репозиторіїв відкритих даних метадані становлять окрему складну частину предметної області. Це пов'язано з існуванням значної кількості стандартів опису даних, які можуть відрізнятися набором полів, ступенем деталізації та внутрішньою структурою. Хоча в поточній реалізації проєкт обмежується заздалегідь визначеним набором атрибутів, сама система розробляється з урахуванням подальшого розвитку, у межах якого може виникнути потреба підтримки інших схем метаданих або розширення вже наявної. Збереження метаданих у форматі *jsonb* дає змогу уникнути надмірної жорсткості реляційної схеми, спрощує адаптацію моделі до нових стандартів і зменшує складність майбутніх міграцій. Крім того, такий підхід дозволяє не прив'язувати структуру таблиці до однієї конкретної схеми метаданих, що є особливо важливим з урахуванням можливої підтримки кількох концептуально відмінних підходів до опису наборів даних.

Файлова модель системи побудована у два рівні. Таблиця *storage_objects* описує фізичні файли, що реально розміщуються у зовнішньому файловому сховищі, і містить їх контрольну суму, ключ об'єкта, формат, розмір, службовий статус та, за потреби, опис вмісту архівів. Водночас таблиця *dataset_version_files* зв'язує конкретну версію набору даних із конкретним файловим об'єктом. Саме в цій таблиці фіксуються атрибути, що мають значення у межах конкретної версії, зокрема відображувана назва файлу та його позиція у списку. Це дозволяє повторно використовувати один і той самий фізичний файл у різних версіях, не дублюючи його у сховищі, якщо на це немає необхідності.

Сутність *users* використовується для зберігання основних даних користувача, облікових реквізитів і стану підтвердження електронної пошти. Окремо від неї існує таблиця *email_verification_tokens*, яка обслуговує процес верифікації пошти. При цьому в базі даних зберігається не сирий токен, а лише його хеш, що підвищує безпеку системи.

З погляду бізнес-логіки особливо важливим є зв’язок структури бази даних із механізмом версіонування. Початковий набір даних створюється як чернеткова версія 1.0. Після публікації ця версія стає стабільною опублікованою версією. Подальше редагування вже опублікованого набору не змінює існуючий запис, а створює новий чернетковий стан. Якщо змінюються лише метадані, застосунок формує нову мінорну версію, а якщо змінюється файловий склад набору – нову мажорну версію. Таким чином таблиці *datasets*, *dataset_versions* і *dataset_version_files* відображають також і саму логіку життєвого циклу набору даних у системі.

Окремої уваги потребує модель фізичного зберігання файлів. У межах проєкту реальні файли наборів даних не зберігаються безпосередньо в PostgreSQL, оскільки це ускладнило б масштабування, резервне копіювання та роботу з великими обсягами двійкових даних. База даних містить лише описові записи про файлові об’єкти, а самі файли зберігаються в сховищі об’єктів за детермінованими ключами, що враховують ідентифікатор користувача, набору даних і самого файлового об’єкта. Це дає змогу ефективно працювати з великими файлами, підтримувати повторне використання даних у межах одного набору та зберігати чіткий зв’язок між логічною та фізичною моделлю ресурсу.

Під час проєктування структури бази даних враховано вимоги до цілісності даних [34]. Частина обмежень забезпечується на рівні самої схеми за допомогою первинних і зовнішніх ключів, унікальних обмежень та зв’язків між таблицями. Інша частина реалізується на рівні доменної логіки, зокрема правила наявності обов’язкових метаданих, коректності мовних кодів, унікальності позицій файлів у межах версії, неможливості змінювати опубліковану версію напряму та коректного переходу між станами DRAFT і PUBLISHED.

Узагальнюючи, у застосунку використано багаторівневу модель зберігання даних, у якій реляційна база даних PostgreSQL відповідає за транзакційні сутності та зв’язки між ними, сховище об’єктів S3 – за фізичне зберігання файлів, а Redis – за сесійний стан автентифікації. Така структура є підтримує версіонування наборів даних, контроль доступу та повторне використання файлових об’єктів, а також створює достатню основу для подальшого розширення платформи.

3 ІМПЛЕМЕНТАЦІЯ ЗАСТОСУНКУ

3.1 Технології та умови імплементації

Імплементація застосунку виконувалася в середовищі, яке поєднувало локальну розробку та контейнеризований запуск інфраструктурних залежностей. Основною операційною системою для розробки були macOS Sequoia 15.6 та Linux Debian 13.4.0.

Серверна частина реалізована на Java з використанням Spring Boot і Maven, клієнтська частина як Angular SPA, а допоміжні сервіси запускалися через Docker Compose.

Для збереження даних застосунок використовує PostgreSQL, для сесій – Redis, а для файлових об’єктів – AWS S3. Схема бази даних керується через Liquibase міграції, що дозволяє відтворювати структуру БД у різних середовищах. Backend під час розробки запускався локально через Docker, тоді як frontend запускався через CLI інструмент Angular – ng serve. Для локальних backend-налаштувань використовувався окремий профіль application-dev.yaml.

Таблиця 3.1

Технічні засоби реалізації застосунку

Компонент	Використана технологія / версія	Призначення
ОС розробки №1	macOS Sequoia 15.6	Середовище розробки застосунку
ОС розробки №2	Linux Debian 13.4.0	Середовище розробки застосунку
Інтегроване середовище розробки №1	IntelliJ IDEA 2026.1	Розробка серверної частини на Java/Spring Boot
Інтегроване середовище розробки №1	WebStorm 2026.1	Розробка клієнтської частини на Angular
Backend-мова	Java 25	Реалізація серверної частини

Продовження таблиці 3.1

Backend-фреймворк	Spring Boot 4.0.3	Побудова серверного застосунку
Система збирання	Maven 3.9.14	Збирання багатомодульного backend проєкту
Контейнеризація	Docker 29.4.0	Запуск інфраструктурних залежностей
Оркестрація локальних сервісів	Docker Compose	Узгоджений запуск PostgreSQL і Redis
Основна база даних	PostgreSQL 18	Збереження користувачів, наборів даних, версій, метаданих і файлових записів
Сховище сесій	Redis 8.6	Збереження session-cookie стану
Файлове сховище	AWS S3	Збереження файлів наборів даних і зображень профілю
Міграції БД	Liquibase 4.32.0	Версіоноване керування схемою бази даних
Frontend-фреймворк	Angular 21.2.x	Реалізація SPA-клієнта
Frontend-мова	TypeScript 5.9.2	Реалізація клієнтської логіки
Менеджер пакетів	npm 11.12.1	Керування залежностями frontend
Тестування frontend	Vitest 4.0.8	Запуск frontend-тестів
Реактивні засоби frontend	RxJS 7.8.x	Робота з асинхронними потоками та Angular-залежностями

3.2 Імплементація серверної частини застосунку

Серверна частина застосунку реалізована як багатомодульний Maven [35] проєкт на основі Spring Boot. Її структура побудована відповідно до принципів гексагональної архітектури та предметно-орієнтованого проєктування [28][29]. Основна ідея такої реалізації полягає в тому, що бізнес-логіка не повинна залежати від HTTP-контролерів, ORM [36], бази даних, файлового сховища або інших інфраструктурних механізмів. Тому серверна частина поділена на окремі модулі з різними рівнями відповідальності.

До складу backend-проєкту входять чотири основні модулі:

- bootstrap;
- domain;
- application;
- infrastructure;

Хоча з погляду залежностей внутрішнім ядром системи є модуль domain, практична точка входу до серверного застосунку розміщена саме в модулі bootstrap. Цей модуль є стартовим для запуску системи, оскільки він підключає решту модулів, містить основні конфігураційні ресурси та формує виконуваний Spring Boot застосунок. Саме через bootstrap серверна частина збирається в один виконуваний артефакт, тоді як інші модулі виконують спеціалізовані ролі в межах архітектури.

Загальний напрям залежностей у backend можна подати як наведено в рисунку 3.1.

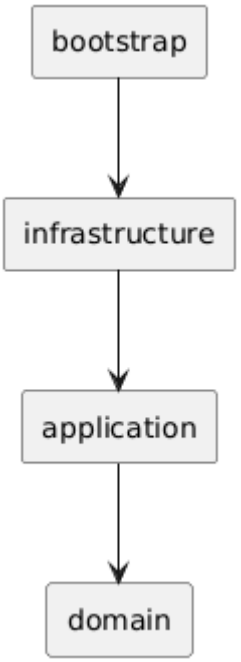


Рис. 3.1 Ієрархія залежностей серверних модулів

Такий напрям означає, що зовнішні шари можуть залежати від внутрішніх, але не навпаки. Доменний шар не знає про інфраструктуру, база даних не впливає

на структуру бізнес-правил та REST API лише викликає прикладні сценарії, не реалізуючи їх самостійно.

Організація Maven-модулів

Перед розглядом окремих серверних компонентів варто розглянути організацію Maven модулів, оскільки саме вони визначають фізичну межу між архітектурними шарами застосунку. У багатомодульному проєкті поділ на bootstrap, domain, application та infrastructure є механізмом контролю залежностей між частинами системи. Через *pom.xml* кожного модуля задається, які бібліотеки та внутрішні компоненти можуть використовуватися на відповідному рівні. Завдяки цьому архітектурні правила, описані під час проєктування, закріплюються на рівні збирання проєкту.

Кореневий *pom.xml* виконує роль агрегатора всього backend-проєкту. У ньому визначено загальну версію проєкту, базовий parent від Spring Boot, версію Java, перелік модулів, а також спільні тестові залежності. Тобто кореневий *pom.xml* не реалізує окрему частину застосунку, а задає спільну конфігурацію збирання для всіх backend-модулів.

Важливою особливістю є те, що в кореневому *pom.xml* Spring Boot Maven Plugin був налаштований із пропуском виконуваного пакування. Це логічно, оскільки root-модуль має тип pom і не є самостійним застосунком. Реальний виконуваний JAR формується не на цьому рівні, а в модулі bootstrap.

Модуль bootstrap відрізняється від інших тим, що саме він підключає всі основні частини backend: domain, application та infrastructure. У його *pom.xml* також підключено необхідні Spring Boot залежності для запуску застосунку, зокрема Web, JPA, Liquibase [37] та Autoconfigure. Для цього модуля Spring Boot Maven Plugin увімкнений без пропуску, тому саме bootstrap є фінальним модулем складання та запуску серверної частини.

Модуль domain має найбільш обмежений *pom.xml*. Він не підключає Spring, JPA, web-залежності або інші технічні бібліотеки. У ньому використано Maven Enforcer Plugin [38], який забороняє сторонні залежності, окрім Lombok [39] і

тестових бібліотек. Це зроблено для того, щоб доменна модель залишалася незалежною від фреймворків та інфраструктури.

Модуль application залежить від domain, але також не підключає інфраструктурні технології. Його *pom.xml* дозволяє залежність від доменного модуля та службові залежності, але не прив'язує прикладний шар до Spring Web, JPA, Redis або S3. Це відповідає ролі application-рівня, за якою він описує сценарії використання та порти, але не визначає конкретні технічні способи їх виконання.

Модуль infrastructure, навпаки, має найбільшу кількість технологічних залежностей. У його *pom.xml* підключено Spring Web MVC, Spring Security, Spring Data JPA, Liquibase, Redis sessions, S3 SDK [40], FreeMarker [41], поштовий стартер, validation, PostgreSQL driver і тестові інструменти. Саме infrastructure-рівень реалізує адаптери до зовнішніх технологій, таких як REST API, бази даних, сховища об'єктів, email-сервісу, сесій і механізмів безпеки. Цей модуль залежить від application, тому що реалізує визначені там порти.

Таким чином, різниця між *pom.xml* окремих модулів відображає їхню архітектурну роль:

- root pom.xml – спільна конфігурація, модулі, тестові залежності;
- bootstrap/pom.xml – точка запуску, складання виконуваного Spring Boot JAR;
- domain/pom.xml – ізольована предметна модель без інфраструктурних залежностей;
- application/pom.xml – use case-и та порти, залежність лише від domain;
- infrastructure/pom.xml – технічні адаптери й зовнішні інтеграції.

Імплементація модуля bootstrap

Реалізація серверної частини починається з модуля bootstrap, саме він відповідає за запуск усього застосунку. Цей модуль не містить основної бізнес-логіки, але виконує важливу інтеграційну роль, тому що збирає всі інші модулі в єдину програму, підключає конфігураційні ресурси та визначає, як саме backend запускається як Spring Boot застосунок.

У цьому модулі також логічно розміщуються ресурси, пов’язані з конфігурацією середовища, міграціями бази даних, шаблонами повідомлень і налаштуваннями запуску. Таким чином, bootstrap є основою виконання backend-рівня. Його призначення – зв’язати доменну, прикладну й інфраструктурну частини в один працездатний застосунок.

Імплементація модуля domain

Модуль domain є ядром бізнес-логіки системи. У ньому реалізовано такі основні предметні поняття, як користувач, набір даних, версія набору даних, метадані, файл набору даних, об’єкт сховища та токен підтвердження електронної пошти.

Центральним агрегатом є Dataset, який моделює набір даних як ресурс із підтримкою чернеток, публікації та версіонування. У межах цього агрегату реалізуються ключові правила:

- створення нового набору даних у статусі чернетки;
- формування початкової версії;
- заборона зміни вже опублікованої версії;
- створення нової чернетки після редагування опублікованого набору;
- розрізнення зміни метаданих і зміни файлів;
- підвищення номера версії залежно від типу змін;
- публікація поточної чернетки.

Окремо реалізовано модель DatasetVersion, яка представляє конкретну версію набору даних. Вона містить номер версії, статус, метадані та перелік файлів. Саме на рівні версії важливо забезпечити незмінність опублікованого стану й контроль допустимості редагування.

Метадані набору даних представлені окремою моделлю, яка включає назву, опис, авторів, мову, тематики, тип ресурсу, видавця та рік публікації. Такий підхід дозволяє розглядати метадані як цілісний доменний об’єкт.

Піддомен користувачів представлений агрегатом User і сутністю EmailVerificationToken. User містить дані профілю, email, хеш пароля та ознаку підтвердження email. EmailVerificationToken інкапсулює такі правила

використання токена, як строк дії, одноразовість і зв’язок із користувачем. Завдяки цьому логіка підтвердження email залишається частиною предметної області.

Імплементація модуля application

Модуль application реалізує сценарії використання системи. Якщо domain відповідає на питання, які бізнес-правила існують у системі, то application визначає, як ці правила застосовуються в конкретних операціях користувача.

У цьому модулі реалізовано сервіси для таких сценаріїв:

- реєстрація користувача;
- підтвердження електронної пошти;
- створення набору даних;
- завантаження файлів;
- оновлення метаданих;
- публікація набору даних;
- видалення чернеткової версії;
- отримання опублікованих наборів;
- отримання конкретної версії;
- експорт метаданих;
- робота з профілем користувача.

Application-рівень не працює напряму з HTTP-запитами, JPA-сутностями або S3 API. Замість цього він використовує порти. Наприклад, для збереження користувача використовується порт репозиторію користувачів, для хешування пароля – порт password encoder, для файлового сховища – відповідний storage port, для публікації подій – event publisher port.

Такий підхід важливий для підтримки гексагональної архітектури. Прикладний сервіс описує сам сценарій, але не знає, чи зберігаються дані через JPA, чи файл потрапляє в S3, чи email надсилається через SMTP. Ці деталі реалізуються пізніше на infrastructure-рівні.

Особливо важливими для застосунку є сценарії роботи з наборами даних. Наприклад, під час створення датасету application-сервіс приймає команду від зовнішнього шару, формує доменні об’єкти авторів і метаданих, створює агрегат

Dataset, перевіряє обмеження та передає агрегат на збереження. Під час завантаження файлів application-рівень координує перевірку прав власника, обчислення контрольної суми, перевірку формату, пошук дубліката, створення storage object і прив’язку файлу до поточної версії.Імплементація модуля infrastructure

Модуль infrastructure реалізує зв’язок серверної частини із зовнішнім середовищем. У ньому розміщені компоненти, які безпосередньо використовують Spring, HTTP, JPA, PostgreSQL, Redis, S3-сумісне сховище, email-сервіс та інші технічні засоби.

REST-контролери цього модуля приймають HTTP-запити від клієнтської частини, перетворюють їх у команди application-рівня та повертають відповіді у форматі DTO. Контролери не повинні містити складну бізнес-логіку. Їхня роль – бути вхідним адаптером, який передає запит до відповідного use case.

JPA-адаптери відповідають за збереження та читання даних із PostgreSQL. При цьому доменні агрегати не зберігаються напряму як JPA-сутності. Між доменною моделлю та фізичною схемою бази даних існує окремий інфраструктурний шар перетворення. Це дає змогу зберігати незалежність доменної моделі від ORM.

Окремі адаптери реалізують роботу з файловим сховищем. Вони відповідають за завантаження файлів, збереження ключів об’єктів, взаємодію із S3-сумісним API та видалення непотрібних файлових ресурсів. У бізнес-логіці при цьому використовується не конкретний S3-клієнт, а абстрактний порт, визначений у прикладному шарі.

Також в infrastructure-рівні реалізовано механізми безпеки. Spring Security використовується для автентифікації користувача, контролю сесії та захисту приватних сценаріїв. Redis застосовується для збереження сесій, а PostgreSQL – для збереження користувачів, токенів, наборів даних, версій і файлових записів.

Таким чином, infrastructure є найбільш технологічно насиченим модулем, але його залежність спрямована до application-рівня. Це означає, що

інфраструктура не диктує бізнес-логіку, а лише забезпечує її виконання у конкретному технічному середовищі.

Загальна послідовність обробки запиту

Роботу серверної частини можна описати через типову послідовність виконання запиту. Спочатку клієнтська частина надсилає HTTP-запит до REST API. Запит потрапляє в контролер infrastructure-рівня. Контролер виконує первинне перетворення вхідних даних у команду або параметри виклику та передає їх у відповідний application-сервіс.

Application-сервіс координує виконання сценарію: перевіряє користувача, викликає доменні методи, звертається до портів збереження, файлового сховища або інших залежностей. Доменний шар при цьому відповідає за правила допустимості зміни стану. Якщо операція потребує збереження, application-сервіс звертається до відповідного порту, а infrastructure-адаптер реалізує цей порт через JPA, S3, Redis або інший технічний механізм.

Після завершення операції результат повертається назад у контролер, де перетворюється у відповідь для клієнтської частини. Така послідовність дозволяє уникнути змішування HTTP-логіки, бізнес-правил і деталей збереження даних.

У спрощеному вигляді цей процес можна подати так, як наведено в рисунку 3.2.

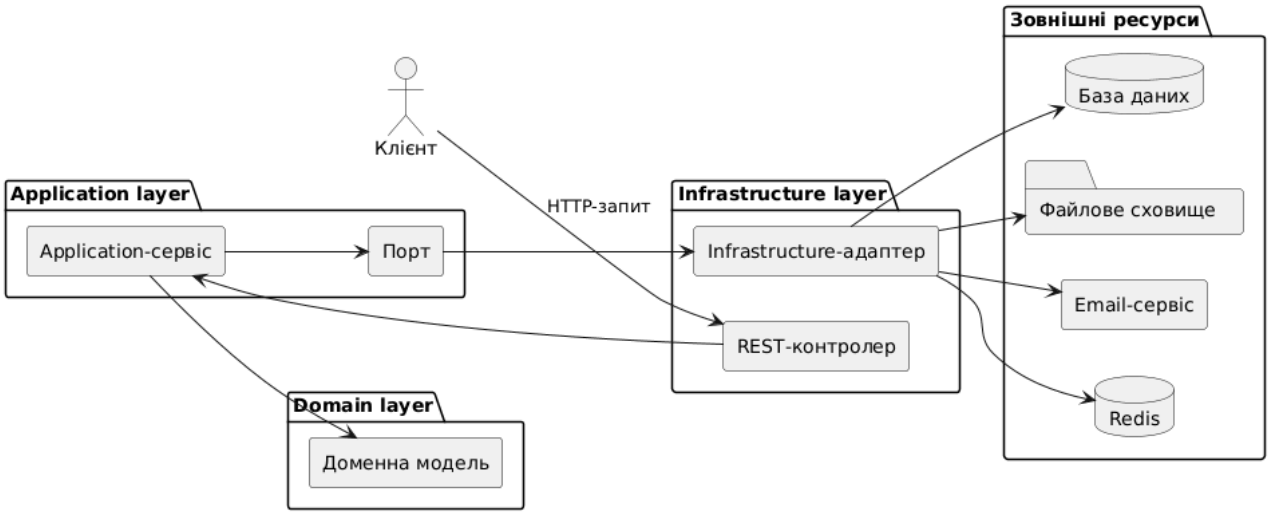


Рис. 3.2 Послідовність обробки запиту в серверній частині застосунку

3.3 Реалізація клієнтської частини застосунку

Клієнтська частина застосунку була реалізована як окремий односторінковий застосунок на Angular. Її імплементація виходила з того, що основна логіка предметної області вже зосереджена на backend-рівні, тому frontend не повинен повторно реалізовувати правила публікації, версіонування, перевірки доступу чи збереження наборів даних. Його роль полягає в тому, щоб надати користувачу послідовний інтерфейс для виконання цих дій і правильно передавати запити до серверної частини.

З цієї причини клієнтська частина була побудована навколо кількох основних механізмів: маршрутизації, компонентів сторінок, сервісу сесії та окремого API-шару. Така структура дозволила розділити логіку за призначенням. Компоненти відповідають за відображення сторінок і локальний стан, маршрути визначають доступні переходи між сценаріями, сервіс сесії зберігає інформацію про поточного користувача, а API-шар ізолює сторінки від деталей HTTP-взаємодії.

Початкова ініціалізація застосунку виконується в кореневому компоненті. Саме він формує спільну оболонку інтерфейсу, у якій розміщуються постійна навігація та область для активної сторінки. Для цього використовується стандартний механізм Angular Router – router-outlet [42], через який у загальний каркас підставляється компонент поточного маршруту. Завдяки цьому сторінки зосереджуються лише на власному сценарії.

Імплементація сервісу сесії

Окрім відображення каркасу, кореневий компонент виконує ще одну важливу дію – намагається відновити сесію користувача після відкриття застосунку. Це зроблено тому, що автентифікація побудована не на збереженні токена у браузері, а на session-cookie, стан якої контролюється серверною частиною. Після запуску frontend звертається до backend із запитом поточного користувача. Якщо сервер підтверджує активну сесію, дані користувача записуються у клієнтський сервіс сесії, а потім додатково завантажується профіль.

Якщо сервер повертає помилку авторизації, локальний стан очищується. Таким чином, frontend не вважає власний стан автентифікації остаточним, а синхронізує його з backend.

Імплементація маршрутизації

Маршрутизація була імплементована так, щоб відобразити основні сценарії роботи із застосунком. Частина маршрутів є публічною і доступна без входу в систему. До таких сторінок належать головна сторінка, каталог опублікованих наборів даних, перегляд окремого набору та перегляд його версій. Інша частина таких маршрутів, як створення нового набору даних, редагування чернетки та сторінка профілю користувача потребує автентифікації. Для таких маршрутів використовується guard [43], який перевіряє, чи є користувач автентифікованим. Якщо сесія відсутня, користувача перенаправляє на сторінку входу, а початковий маршрут зберігається для повернення після успішної автентифікації.

Імплементація API-шару

Взаємодія з backend була винесена в окремий API-шар. Це рішення дозволило не розміщувати низькорівневі HTTP-виклики безпосередньо в компонентах сторінок. Центральний API-wrapper відповідає за формування JSON-запитів, передавання cookie сесії, обробку multipart-запитів для файлів і перетворення помилок сервера у типізований ApiError. Завдяки цьому компоненти працюють з прикладними функціями, які відповідають сценаріям системи: створення набору даних, отримання каталогу, завантаження файлу, публікація чернетки або отримання профілю.

Імплементація компонентів сторінок

Публічна частина інтерфейсу була реалізована як шлях від загального перегляду до конкретного ресурсу. Користувач може перейти з головної сторінки до каталогу, переглянути список опублікованих наборів даних, відкрити окремий набір і, за потреби, переглянути конкретну версію. Каталог отримує дані із серверної частини й відображає їх у вигляді карток. При цьому, окрім успішного завантаження інтерфейс обробляє й проміжні стани, як наприклад очікування відповіді, відсутність опублікованих наборів або помилку отримання даних. Це

було необхідно, щоб сторінка залишалася зрозумілою для користувача незалежно від стану запиту.

Сторінка набору даних реалізує перегляд повної інформації про ресурс. Вона відображає метадані, авторів, тематики, тип доступу, список файлів і доступні версії. Якщо набір відкриває його власник, frontend може показати додаткові дії, наприклад перехід до редагування. Однак така поведінка є лише адаптацією. Остаточну перевірку прав доступу все одно виконує backend, тому приховування або відображення кнопок у frontend не використовується як єдиний механізм безпеки.

Окремо був реалізований перегляд версій набору даних. Якщо користувач відкриває маршрут конкретної версії, frontend звертається до backend за даними вибраної версії. Це дозволяє відокремити перегляд актуального ресурсу від перегляду його історії. Такий підхід узгоджується з backend-моделлю, у якій опубліковані версії є стабільними і не змінюються після публікації.

Сценарій створення набору даних реалізований як перехід від початкової форми до редактора чернетки. На сторінці створення користувач вводить основні метадані майбутнього ресурсу. Перед надсиланням запиту frontend виконує попередню перевірку форми та потрібна для того, щоб швидко показати типові помилки введення й не надсилати очевидно некоректні дані на сервер. При цьому така перевірка не замінює backend-валідацію, оскільки саме серверна частина остаточно визначає допустимість створення ресурсу.

Після успішного створення backend повертає ідентифікатор нового набору даних. Frontend використовує його для переходу до сторінки редагування чернетки. Це відображає життєвий цикл ресурсу в системі, в якому створений набір не одразу стає опублікованим, а спочатку існує як draft. У цьому стані користувач може доповнити метадані, додати файли, зберегти зміни й лише після цього опублікувати ресурс.

Редактор чернетки став основним інтерфейсом автора. Його імплементація складніша за звичайну форму, оскільки він має працювати одночасно з метаданими, файлами, поточним станом чернетки, помилками, станами

завантаження та діями публікації. У цьому компоненті локально зберігається стан форми та список файлів, які користувач підготував до додавання. Це дозволяє користувачу спочатку сформувати набір змін в інтерфейсі, а потім явно зберегти їх.

Робота з файлами була реалізована окремо від роботи з метаданими. Метадані передаються до backend у форматі JSON, тоді як файли завантажуються через multipart-запити. Тому під час збереження чернетки frontend спочатку синхронізує структуровані дані, а потім окремо передає файли. Після завершення цих операцій стан чернетки оновлюється відповідно до відповіді backend. Така послідовність дозволила не змішувати різні типи даних і коректно відображати результат збереження в інтерфейсі.

Публікація чернетки також ініціюється з редактора, але її фактичне виконання залишається відповідальністю backend. Frontend може тимчасово блокувати дію, якщо триває інший запит, є незбережені зміни або ресурс ще не готовий до публікації з погляду інтерфейсу. Але ці перевірки мають допоміжний характер. Серверна частина повторно перевіряє права користувача, стан чернетки та можливість виконання публікації.

Профіль користувача був реалізований як окрема сторінка, яка поєднує інформацію про користувача та пов’язані з ним набори даних. Дані профілю також використовуються в загальній оболонці застосунку після відновлення сесії, щоб інтерфейс міг відображати актуальний стан користувача. Для поточного користувача ця сторінка також використовується як місце, з якого можна завершити сесію.

Загалом робота клієнтської частини будується як послідовність дій від інтерфейсу до backend і назад. Користувач виконує дію на сторінці, компонент оновлює локальний стан і, якщо потрібно, виконує попередню перевірку. Далі API-шар надсилає запит до серверної частини. Після отримання відповіді компонент оновлює інтерфейс або показує помилку. У цій схемі frontend відповідає за зручність і послідовність взаємодії, а backend – за остаточне виконання правил системи.

Ілюстрація цієї взаємодії наведена на рисунку 3.3.

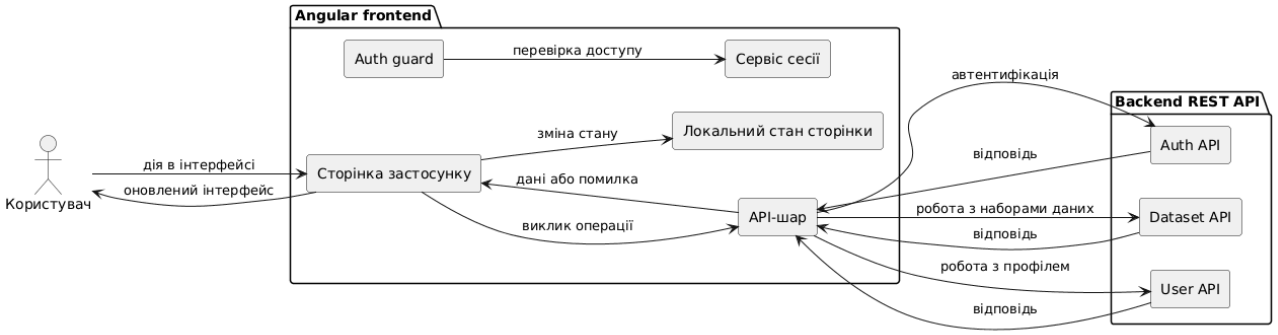


Рис. 3.3 Загальна схема роботи клієнтської частини застосунку

Підсумовуючи, клієнтська частина була імплементована як окремий шар, який послідовно проводить користувача через основні сценарії роботи з репозиторієм відкритих даних. Вона не дублює backend-логіку, а організовує взаємодію з нею через маршрути, компоненти, сервіс сесії та централізований API-шар.

3.4 Імплементація збереження даних і файлової інфраструктури

У застосунку була імплементована комбінована модель зберігання. Це пов’язано з тим, що система працює з різними типами даних, які мають різні вимоги до збереження та обробки. Структуровані дані користувачів, наборів даних, версій і зв’язків між ними потребують транзакційності та цілісності. Метадані наборів даних мають вкладену структуру і можуть еволюціонувати. Файли наборів даних є бінарними об’єктами, які нераціонально зберігати безпосередньо в реляційній базі. Окремо існує сесійний стан автентифікованих користувачів, який має коротший життєвий цикл і не є частиною доменної історії ресурсу.

Тому збереження даних було розділено між трьома основними підсистемами. PostgreSQL використовується як основне транзакційне сховище бізнес-сутностей. S3-сумісне сховище об’єктів відповідає за фактичне збереження

файлів наборів даних. Redis використовується для збереження HTTP-сесій, що підтримуються Spring Session. Такий підхід відповідає polyglot persistence [44], де кожен тип даних зберігається у середовищі, яке найкраще відповідає його призначенню.

Основна схема бази даних створюється через Liquibase-міграції [36]. Це дозволяє фіксувати зміни структури БД у коді проєкту та відтворювати однакову схему в різних середовищах. У реляційній моделі були реалізовані таблиці для користувачів, токенів підтвердження email, наборів даних, версій, зв’язків між версіями та файлами, а також таблиця фізичних файлових об’єктів.

PostgreSQL у цій реалізації використовується не лише як сховище таблиць. Його роль ширша, оскільки саме на рівні БД забезпечуються первинні та зовнішні ключі, унікальні обмеження, транзакційність і частина механізмів конкурентного доступу. Наприклад, UUID [45] використовується для ідентифікаторів користувачів, наборів даних і файлових об’єктів, а jsonb - для збереження складних метаданих версій. Також у системі використовується advisory lock [46] під час завантаження файлів, щоб уникнути конкурентного створення однакових файлових об’єктів.

Важливо, що доменні агрегати не зберігаються у базі даних напряму в тому самому вигляді, у якому вони існують у доменному шарі. Між доменною моделлю та фізичною схемою існує інфраструктурний шар JPA-сутностей, репозиторіїв і проєкцій. Завдяки цьому бізнес-логіка не прив’язується до конкретної структури таблиць, а persistence-рівень може адаптувати доменну модель до реляційного представлення.

Центральною частиною реляційної моделі є збереження наборів даних і їх версій. Таблиця datasets представляє набір даних як логічний ресурс, у якій зберігається власник, тип доступу, загальний статус, дата створення, дата публікації, а також посилання на поточну й останню опубліковану версію. Такий підхід дозволив розділити сам ресурс і його конкретні версії.

Детальний вміст набору даних зберігається в таблиці dataset_versions. Саме вона містить номер версії, статус версії, дату створення та метадані. Це важливо

для реалізації незмінності опублікованих версій. Після публікації версія не редагується напряму. Якщо користувач вносить зміни, система створює нову чернеткову версію, тоді як попередня опублікована версія залишається стабільною.

Метадані версії зберігаються в полі metadata у форматі jsonb. Це рішення було обране тому, що метадані набору даних мають вкладену структуру. Вони містять авторів, назву, опис, мову, тематики, тип ресурсу, видавця та рік публікації. Повна нормалізація цих даних призвела б до появи великої кількості допоміжних таблиць, що ускладнило б модель без суттєвої користі для основних сценаріїв. Збереження у jsonb дозволяє читати метадані як цілісний документ і водночас залишає можливість змінювати їхню структуру в майбутньому.

Зв’язок між версіями та файлами реалізований через таблицю dataset_version_files. Вона фіксує, які файлові об’єкти входять до конкретної версії набору даних, завдяки чому різні версії можуть мати різні набори файлів, а один фізичний об’єкт може повторно використовуватися, якщо він уже був завантажений у межах відповідного набору даних.

Файли наборів даних не зберігаються безпосередньо в PostgreSQL. У базі зберігаються лише записи про файлові об’єкти, тоді як сам бінарний вміст розміщується в S3-сумісному сховищі об’єктів. Такий підхід дозволяє не перевантажувати реляційну базу великими файлами та водночас зберігати чіткий зв’язок між логічною моделлю набору даних і фізичним розміщенням файлу.

Таблиця storage_objects описує фізичний файловий об’єкт. У ній зберігаються контрольна сума SHA-256 [47], ключ об’єкта в сховищі об’єктів, версія об’єкта, формат файлу, розмір, статус і, за потреби, інформація про вміст ZIP-архіву. Таким чином, база даних зберігає тільки його стабільний опис і адресу в зовнішньому сховищі.

Під час завантаження файлу backend обробляє його як окремий об’єкт. Файл перевіряється за розміром, для нього обчислюється SHA-256, визначається формат, після чого система перевіряє, чи вже існує такий об’єкт у межах директорії конкретного набору даних. Якщо файл уже був завантажений раніше,

повторне фізичне збереження не виконується, а існуючий storage_object повторно прив’язується до версії. Якщо дубліката немає, файл завантажується в сховище об’єктів, а в PostgreSQL створюється відповідний запис.

Ключі об’єктів у файловому сховищі формуються так, щоб відображати належність файлу до користувача та конкретного набору даних. Завдяки цьому файлове сховище має зрозумілу ієрархію, а дедуплікація виконується локально в межах набору даних. Це важливо, оскільки однаковий файл у різних наборах даних може мати різний контекст і не повинен обов’язково вважатися одним глобальним ресурсом.

Окремо була реалізована підтримка сесій через Redis. Оскільки автентифікація в системі побудована на session-cookie підході, стан автентифікованого користувача не зберігається на frontend і не передається у вигляді JWT-токена. Замість цього, браузер передає cookie, а backend використовує сесію для відновлення контексту користувача.

Redis у цій схемі використовується як спеціалізоване сховище короткоживучого сесійного стану. Це дозволило не змішувати доменні дані з технічним станом автентифікації. PostgreSQL зберігає користувачів, набори даних і версії, сховище об’єктів зберігає файли, а Redis відповідає за активні сесії. Такий поділ робить модель зберігання набагато зрозумілішою і краще відповідає різному життєвому циклу цих даних.

Ключовою особливістю імплементації є те, що реляційна модель і файлове сховище працюють разом, та не дублюють одне одного. PostgreSQL відповідає за зв’язки, історію версій, права власності, метадані та опис файлових об’єктів. Сховище об’єктів відповідає за збереження фактичного бінарного вмісту. Зв’язком між цими двома частинами є storage_key, який зберігається в таблиці storage_objects і вказує на реальний об’єкт у зовнішньому сховищі.

Під час видалення чернеткових версій система також має враховувати, чи використовується файловий об’єкт іншими версіями. Якщо на storage_object ще існують посилання, фізичний файл не повинен бути видалений. Якщо ж об’єкт більше не використовується, його можна прибрати і з сховища, і з бази даних.

Такий підхід запобігає як втраті файлів, що ще потрібні іншим версіям, так і накопиченню зайвих об’єктів.

Цілісність даних забезпечується на кількох рівнях. Частина правил реалізована через схему PostgreSQL завдяки первинним ключам, зовнішнім ключам та унікальним обмеженням. Інша частина залишається в доменній логіці, зокрема відповідаюча за незмінність опублікованих версій, унікальність позицій файлів у межах версії, заборону повторного додавання одного й того самого файлового об’єкта до однієї версії та правила переходу між станами DRAFT і PUBLISHED.

Загальну взаємодію підсистем зберігання можна подати у вигляді діаграми наведеної на рисунку 3.4.

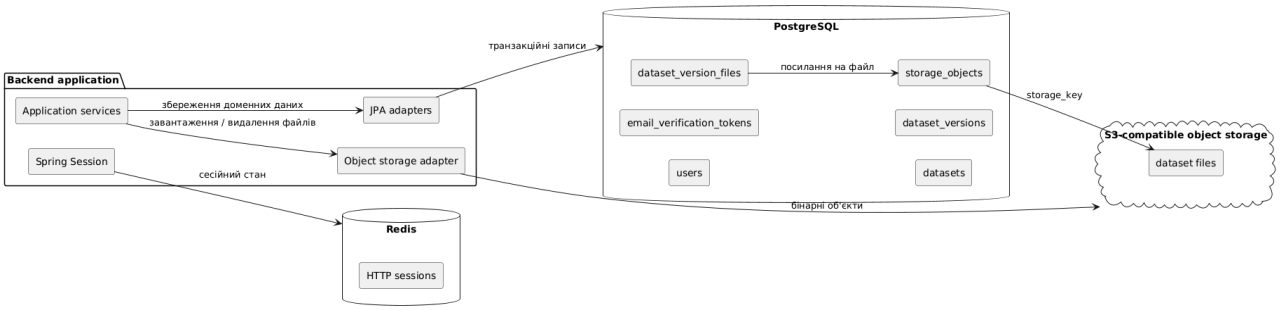


Рис. 3.4 Загальна схема реалізації збереження даних і файлової інфраструктури

Отже, збереження даних у застосунку було реалізовано як поєднання PostgreSQL, S3-сумісного сховища об’єктів та Redis. PostgreSQL забезпечує транзакційну основу й історію версій, сховище об’єктів відповідає за фактичні файли, а Redis підтримує сесійний стан користувачів. Такий поділ дозволяє підтримати основні вимоги репозиторію відкритих даних, якими є незмінність опублікованих версій, збереження складних метаданих, роботу з великими файлами та контрольований доступ до ресурсів.

4 ТЕСТУВАННЯ ТА РЕЗУЛЬТАТИ

4.1 Тестування серверної частини застосунку

Тестування серверної частини було реалізоване відповідно до шарів backend-архітектури. Оскільки застосунок побудований за принципами гексагональної архітектури, перевірки не обмежувалися одним рівнем. Окремо тестувалися доменні правила, application-сервіси, REST-контролери, інфраструктурні адаптери та persistence-рівень. Такий підхід дозволив перевірити поведінку системи на межах між шарами.

Таблиця 4.1

Основні тест-кейси серверної частини застосунку

Тип тестування	Тип тестування	Тип тестування
Domain-тестування	Зміна лише метаданих опублікованого набору даних	Створюється minor draft-версія
Domain-тестування	Зміна файлового складу опублікованого набору даних	Створюється major draft-версія
Application-тестування	Реєстрація користувача з коректними даними	Пароль хешується, користувач зберігається, подія реєстрації публікується
Application-тестування	Завантаження файлу, який уже існує в межах dataset	Повторно використовується наявний StorageObject
REST-тестування	Надсилання коректного запиту на реєстрацію	API повертає 201 Created
REST-тестування	Запит із некоректними даними	API повертає помилку валідації

Domain тестування

На доменному рівні перевірялися правила предметної області, які не повинні залежати від Spring, бази даних або HTTP-рівня. Такі тести виконуються швидко, оскільки не підіймають Spring-контекст і працюють лише з доменними об’єктами.

Основним прикладом є тестування агрегату Dataset, де перевіряється логіка версіонування. Для застосунку це критично, оскільки зміна метаданих і зміна файлів мають різний вплив на номер наступної версії. Зміна лише метаданих повинна створювати minor draft, а зміна файлів – major draft.

Лістинг 4.1. Приклад domain-тесту логіки версіонування набору даних

```
@Test
void updateMetadata_shouldCreateMinorDraftFromPublishedVersion() {
    Dataset dataset = publishedDataset();
    dataset.updateMetadata(dataset.getCurrentMetadata().patch(
            null, "Edited title", null, null, null
    ));
    assertThat(dataset.getCurrentVersionNumber())
            .isEqualTo(VersionNumber.of((byte) 1, (byte) 1));
}
```

Також на цьому рівні перевірялися правила створення користувача, підтвердження email, заборона порожніх обов’язкових полів і поведінка токена підтвердження електронної пошти.

Тестування application-сервісів

Application-рівень тестувався як рівень прикладних сценаріїв. На цьому рівні сервіси координують доменні об’єкти та взаємодіють із зовнішніми залежностями через порти. У тестах ці порти підміняються моками, тому можна перевірити поведінку use case без реального доступу до бази даних, файлового сховища або поштового сервісу.

Наприклад, у сценарії реєстрації користувача важливо перевірити, що пароль кодується до збереження користувача, а в репозиторій не потрапляє сире значення пароля.

Лістинг 4.2 Приклад тесту application-сервісу реєстрації користувача

```
@Test
void execute_shouldEncodePassword_beforePersisting() {
    when(passwordEncoder.encode(anyString())).thenReturn("hashedPassword");

    service.execute(validCommand);

    verify(passwordEncoder).encode("johndoe123");
    verify(userRepository).create(argThat(u ->
            u.getPasswordHash().equals("hashedPassword")));
}
```

Також на application-рівні тестувалися сценарії створення та оновлення dataset, публікація, завантаження файлів, дедуплікація StorageObject, завантаження файлів із перевіркою доступу, експорт Dublin Core і DataCite XML [48], а також оновлення зображення профілю користувача. Найважливішими тут є тести, які перевіряють повні прикладні сценарії, як наприклад хто виконує дію, який ресурс змінюється, які права має користувач і який результат має бути отриманий.

Тестування REST-контролерів

REST-рівень перевірявся через MockMvc [49] та web-slice тести [50]. У таких тестах підіймається лише web-рівень, а application-сервіси замінюються моками. Це дозволяє перевірити зовнішній HTTP-контракт без запуску всієї системи.

На цьому рівні перевірялися маршрути, HTTP-статуси, структура JSON-відповідей, валідація вхідних DTO, multipart-запити та заголовки для завантаження файлів або XML-документів. Наприклад, endpoint реєстрації повинен повертати 201 Created, якщо запит є коректним.

Такі тести не дублюють application-тести. Їхня задача полягає в перевірці того, що зовнішній API поводиться очікувано: приймає правильні запити, відхиляє некоректні дані та повертає відповідні HTTP-коди.

Лістинг 4.3 Приклад REST-тесту endpoint-у реєстрації користувача

```
@Test
void signup_shouldReturn201_whenRequestIsValid() throws Exception {
    mockMvc.perform(post("/api/v1/auth/signup")
            .contentType(MediaType.APPLICATION_JSON)
            .content(objectMapper.writeValueAsString(body)))
            .andExpect(status().isCreated());
}
```

Тестування інфраструктурних адаптерів

Окрема група тестів була спрямована на інфраструктурні адаптери. Вони перевіряли логіку, яка працює з файлами, object storage, обробкою зображень, подіями та scheduler-механізмами [51].

Наприклад, для файлового інспектора важливо, щоб формат файлу визначався не лише за розширенням, а й за фактичним вмістом. Це зменшує ризик прийняття некоректного або замаскованого файлу.

Також тестувалися S3-адаптер, створення storage-каталогів після реєстрації користувача, генерація verification email, очищення прострочених токенів підтвердження email та обробка аватарів користувача.

Лістинг 4.4 Приклад тесту інфраструктурного адаптера інспекції файлів

```
@Test
void inspect_shouldPreferContentOverExtension() throws Exception {
    Path file = tempDir.resolve("data.csv");
    Files.writeString(file, "{\"id\":1}", StandardCharsets.UTF_8);

    var result = inspector.inspect(file, file.getFileName().toString());

    assertThat(result.format()).isEqualTo(DatasetFileFormat.JSON);
}
```

Інтеграційне тестування persistence-рівня

Найбільш наближеними до реального середовища були persistence-тести. Вони запускалися через анотацію DataJpaTest із реальною PostgreSQL, піднятою за допомогою Testcontainers [52]. Це дозволило перевірити не лише Java-логіку, а й

відповідність між доменною моделлю, JPA-мапінгом і поведінкою реальної бази даних.

Прикладом такого тесту є перевірка ліміту кількості чернеток користувача.

Лістинг 4.5 Приклад інтеграційного persistence-тесту ліміту чернеток

```
@Test
void shouldReturnTrueWhenDraftLimitExceeded() {
    userRepositoryJpa.save(depositor);
    datasetRepositoryJpa.saveAll(List.of(
            DatasetEntityStub.draft(depositor),
            DatasetEntityStub.draft(depositor),
            DatasetEntityStub.draft(depositor)
    ));

    boolean result = datasetRepositoryAdapter
            .isDraftLimitExceeded(new UserId(depositor.getId()));

    assertThat(result).isTrue();
}
```

Такі тести важливі, оскільки частина помилок може проявитися лише на рівні реальної бази даних. Частими випадками є неправильні зв’язки між сутностями, помилки каскадного збереження, некоректне сортування, проблеми з версіями або збереженням файлових прив’язок.

4.2 Тестування клієнтської частини застосунку

Клієнтська частина тестувалася за допомогою Angular TestBed [53], Vitest [54] і JSDOM [55]. Така конфігурація дозволяє запускати тести через Angular CLI, але виконувати їх у середовищі Vitest без відкриття реального браузера. JSDOM при цьому симулює DOM-структуру, тому можна перевіряти компоненти, маршрутизацію, поведінку інтерфейсу та роботу браузерних API.

У frontend-проєкті реалізовано 7 test suite-файлів і 47 автоматизованих тестів. Вони охоплюють три основні напрями: поведінку застосунку як цілісного інтерфейсу, API-адаптери та правила валідації даних. Окремо перевірялися маршрути, guard-логіка, відновлення сесії, сторінки авторизації, каталог dataset,

сторінка деталей, редактор чернетки, staged upload файлів і мапінг DTO між backend і frontend.

Таблиця 4.2

Основні тест-кейси клієнтської частини застосунку

Тип тестування	Тест-кейс	Очікуваний результат
Компонентне тестування	Відкриття маршруту /signup	Відображається форма реєстрації
Компонентне тестування	Відкриття сторінки каталогу	Відображаються опубліковані набори даних
Guard-тестування	Неавтентифікований користувач відкриває захищений маршрут	Виконується перенаправлення на сторінку входу
Тестування сесії	Активна сесія підтверджується backend	Дані користувача зберігаються в session-сервісі
API-тестування	Виконання запиту входу користувача	Надсилається POST-запит із credentials: include
API-тестування	Backend повертає помилку	Frontend формує типізований ApiError
Валідація форми	Некоректний ідентифікатор автора	Відображається повідомлення про помилку
Сценарне тестування	Вибір файлу в редакторі чернетки	Файл потрапляє в pending-стан і завантажується лише після Save

Компонентне тестування та маршрутизація

Найбільша частина перевірок зосереджена навколо кореневого компонента App і маршрутизації. У таких тестах створюється Angular TestBed, підключається реальний route table, відкривається потрібний маршрут і перевіряється DOM, який був сформований компонентом.

Лістинг 4.6 Приклад компонентного тесту маршруту реєстрації

```
it('should render the signup page route', async () => {
  const fixture = TestBed.createComponent(App);
  const router = TestBed.inject(Router);

  await router.navigateByUrl('/signup');
```

```
  fixture.detectChanges();
  await fixture.whenStable();

  expect(fixture.nativeElement.querySelector('form.signup-form')).toBeTruthy();
});
```

Такі тести перевіряють поведінку клієнту як застосунку: чи правильно працює маршрут, чи створюється потрібний компонент і чи з’являються очікувані елементи інтерфейсу.

Тестування API-шару

API-шар тестувався через мокування глобального fetch. Це дозволило перевіряти запити без запуску реального backend. У таких тестах перевіряється URL, HTTP-метод, заголовки, тіло запиту та реакція frontend на успішні або помилкові відповіді.

Лістинг 4.7 Приклад тесту API-шару автентифікації

```
it('posts signin through the session-cookie API wrapper', async () => {
  const fetchSpy = vi.spyOn(globalThis, 'fetch').mockResolvedValue(
    new Response(JSON.stringify({ id: 'user-1', email: 'ada@example.com' }), { status: 200
}),
  );

  await signin({ email: 'ada@example.com', password: 'password123' });

  expect(fetchSpy).toHaveBeenCalledWith('/api/v1/auth/signin', {
    method: 'POST',
    credentials: 'include',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ email: 'ada@example.com', password: 'password123' }),
  });
});
```

Цей тип тестування важливий для стабільності інтеграції між frontend і backend. Навіть якщо сервер не запускається під час frontend-тестів, можна перевірити, що клієнтська частина формує запити у правильному форматі.

Тестування сценаріїв інтерфейсу

Окремо перевірявся staged upload файлів у редакторі чернетки. Цей сценарій важливий, оскільки файл не має завантажуватися одразу після вибору користувачем. Спочатку він потрапляє до локального стану сторінки, а реальне завантаження виконується лише після збереження чернетки.

Цей тест перевіряє правильну послідовність таких дій, як вибір файлу, локальне очікування, збереження та фактичне завантаження.

Лістинг 4.8 Приклад сценарного тесту staged upload файлів

```
it('stages selected files and uploads them only when the draft is saved', async () => {
  input.dispatchEvent(new Event('change', { bubbles: true }));
  fixture.detectChanges();

  expect(hasFetchCall(fetchSpy, `/api/v1/dataset/${datasetId}/files/uploads`,
'POST')).toBe(false);

  compiled.querySelector<HTMLButtonElement>('.save-button')?.click();
  fixture.detectChanges();

  expect(hasFetchCall(fetchSpy, `/api/v1/dataset/${datasetId}/files/uploads`,
'POST')).toBe(true);
});
```

4.3 Результати тестування

За результатами тестування серверної частини було зафіксовано 123 автоматизовані тести. Вони розподілені між модулями backend таким чином:

- 13 тестів у модулі domain;
- 55 тестів у модулі application;
- 55 тестів у модулі infrastructure.

За останнім прогоном усі тести завершилися успішно, без failures, errors і skipped-тестів.

Клієнтська частина містить 7 test suite-файлів і 47 автоматизованих тестів. Вони перевіряють маршрутизацію, auth guard, відновлення сесії, API-запити, мапінг DTO, створення і редагування dataset, staged upload файлів, сторінку

деталей dataset і роботу профілю користувача. Фактичний запуск frontend-тестів також завершився успішно та усі 47 тестів пройшли без помилок.

Загальний результат автоматизованого тестування подано у вигляді таблиці.

Таблиця 4.3

Результати автоматизованого тестування застосунку

Частина застосунку	Кількість test suites / груп	Кількість тестів	Результат
Backend, модуль domain	3 основні групи	13	Успішно
Backend, модуль application	Набір service-тестів	55	Успішно
Backend, модуль infrastructure	Web, adapter і persistence-тести	55	Успішно
Frontend	7 test suites	47	Успішно
Усього		170	Успішно

Отримані результати показують, що тестування охоплює основні критичні частини системи. На backend-рівні перевірено доменні правила, application use case, REST-контракти, файлові адаптери, подієву логіку, scheduler-механізми та persistence-рівень із реальною PostgreSQL через Testcontainers. На frontend-рівні перевірено маршрутизацію, роботу сторінок, API-взаємодію, обробку помилок і сценарії роботи з наборами даних.

Узагальнюючи, результати тестування підтверджують працездатність основних функціональних сценаріїв застосунку. Backend-тести забезпечують контроль бізнес-логіки, правил доступу, версіонування та збереження даних, а frontend-тести підтверджують коректну роботу інтерфейсу, маршрутизації та взаємодії з API. Загалом було виконано 170 автоматизованих тестів, усі з яких завершилися успішно.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було розроблено веб застосунок репозиторію відкритих даних, орієнтований на спрощення основних процесів роботи з наборами даних. На основі аналізу предметної галузі та порівняння таких платформ, як Harvard Dataverse, Zenodo та Kaggle, було визначено, що універсальні репозиторії часто мають надлишковий функціонал, який ускладнює базові сценарії публікації та використання даних. Тому в межах роботи було обрано підхід свідомого обмеження функціональності до таких ключових процесів, як створення, опис, збереження, версіонування, публікація та перегляд наборів даних.

У результаті роботи було реалізовано повноцінну клієнт-серверну платформу, яка забезпечує такі ключові можливості:

1. створення, редагування та публікацію наборів даних;
2. збереження файлів і стандартизованих метаданих;
3. підтримку версіонування опублікованих ресурсів;
4. розмежування доступу між публічними користувачами, авторами та власниками наборів даних;
5. експорт метаданих у форматах Dublin Core та DataCite.

Серверна частина побудована на Java та Spring Boot як багатомодульний застосунок із використанням принципів гексагональної архітектури та предметно-орієнтованого проєктування. Клієнтська частина була реалізована як Angular SPA, що забезпечує користувацький інтерфейс для реєстрації, автентифікації, створення чернеток, редагування метаданих, завантаження файлів, публікації наборів даних, перегляду каталогу, історії версій і профілів користувачів.

Окремим результатом роботи стало використання комбінованої моделі зберігання. PostgreSQL використовується як основне транзакційне сховище для користувачів, наборів даних, версій, метаданих і файлових записів. Redis

застосовується для збереження сесій, а AWS S3 використовується для розміщення бінарних файлових об’єктів. Також було реалізовано підтримку версіонування, за якої зміни метаданих і зміни файлів мають різний вплив на номер версії, а вже опубліковані версії залишаються незмінними.

Розроблений застосунок було перевірено за допомогою автоматизованого тестування. Для серверної частини виконано тести доменного рівня, application-сервісів, REST-контролерів, інфраструктурних адаптерів і persistence-рівня з використанням реальної PostgreSQL через Testcontainers. Для клієнтської частини проведено тести маршрутизації, API-шару, guard-логіки, відновлення сесії, створення та редагування наборів даних і staged upload файлів. Загалом було виконано 170 автоматизованих тестів, усі з яких завершилися успішно, що підтверджує коректність основних функціональних сценаріїв і стабільність ключових частин системи.

Виходячи з вищенаведеного, поставлену мету роботи було досягнуто. Було створено спеціалізовану платформу відкритого репозиторію, яка зосереджується на основних сценаріях управління та публікації наборів даних і не перевантажує користувача надлишковими можливостями. Розроблене рішення може використовуватися невеликими організаціями, дослідницькими групами або освітніми установами як основа для збереження та поширення відкритих наборів даних. У подальшому систему можна розвивати шляхом додавання пошуку й фільтрації, повноцінної підтримки DOI, розширення адміністративних функцій, впровадження end-to-end тестування [56], інтеграції з іншими науковими системами та розгортання в хмарному середовищі.

Додатково, робота пройшла апробацію. За методикою її розробки та результатами було опубліковано наступні тези доповідей:

1. Лисенко В.І. Дизайн веб-застосунку з використанням гексагональної архітектури/Герцюк М.М., Лисенко В.І.// VII Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез 23.04.2026, м. Київ.

2. Лисенко В.І. Дизайн веб-застосунку з використанням гексагональної архітектури/Герцюк М.М., Лисенко В.І.// Всеукраїнська науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез 03.05.2026, м. Київ.

ПЕРЕЛІК ПОСИЛАНЬ

1. Understanding Corporate Data Sharing Decisions: Practices, Challenges, and Opportunities for Sharing Corporate Data with Researchers. Future of Privacy Forum. URL: https://fpf.org/resource/understanding-corporate-data-sharing-decisions-practices-challenges-and-opportunities-for-sharing-corporate-data-with-researchers/ (дата звернення: 14.05.2026).

2. Open Data: A History. U.S. Government's Open Data. URL: https://data.gov/blog/open-data-history/ (дата звернення: 14.05.2026).

3. Chignard S. A Brief History of Open Data. ParisTech Review. URL: https://www.paristechreview.com/2013/03/29/brief-history-open-data/ (дата звернення: 14.05.2026).

4. Open Data’s Impact: Key Findings. Open Data’s Impact. URL: https://odimpact.org/key-findings.html (дата звернення: 14.05.2026).

5. Crosas M. The Dataverse Network: An Open-Source Application for Sharing, Discovering and Preserving Data. D-Lib Magazine. URL: https://www.dlib.org/dlib/january11/crosas/01crosas.html (дата звернення: 14.05.2026).

6. Dataset + File Management. Dataverse.org. URL: https://guides.dataverse.org/en/latest/user/dataset-management.html (дата звернення: 14.05.2026).

7. Zenodo. OpenScience at CERN. URL: https://openscience.cern/zenodo (дата звернення: 14.05.2026).

8. Sharing Your Data and Software on Zenodo. Zenodo. URL: https://zenodo.org/records/802100 (дата звернення: 14.05.2026).

9. FAQ: Versioning. Zenodo. URL: https://zenodo.org/help/versioning (дата звернення: 14.05.2026).

10. Zenodo FAQ: GitHub Integration. Zenodo Support. URL: https://support.zenodo.org/help/en-gb/24-github-integration (дата звернення: 14.05.2026).

11. A Beginner's Guide to Kaggle. MakeUseOf. URL: https://www.makeuseof.com/beginners-guide-to-kaggle/ (дата звернення: 14.05.2026).

12. Notebooks Documentation. Kaggle. URL: https://www.kaggle.com/docs/notebooks (дата звернення: 14.05.2026).

13. Toward Enhanced Reusability: A Comparative Analysis of Metadata for Machine Learning Objects and Their Characteristics in Generalist and Specialist Repositories. Journal of eScience Librarianship. URL: https://publishing.escholarship.umassmed.edu/jeslib/article/id/685/ (дата звернення: 14.05.2026).

14. Kindling M., Strecker D. Data Quality Assurance at Research Data Repositories. Data Science Journal. URL: https://datascience.codata.org/articles/dsj-2022-018 (дата звернення: 14.05.2026).

15. A Review on Java Programming Language. ResearchGate. URL: https://www.researchgate.net/publication/371166744_A_Review_on_Java_Programming_Language (дата звернення: 14.05.2026).

16. What Is Java Spring Boot? Microsoft Azure. URL: https://azure.microsoft.com/en-ca/resources/cloud-computing-dictionary/what-is-java-spring-boot (дата звернення: 14.05.2026).

17. PostgreSQL: About. PostgreSQL. URL: https://www.postgresql.org/about/ (дата звернення: 14.05.2026).

18. Session Management. Redis. URL: https://redis.io/solutions/session-management/ (дата звернення: 14.05.2026).

19. What Is Amazon S3? Amazon Web Services Documentation. URL: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html (дата звернення: 14.05.2026).

20. What Is Amazon SES? Amazon Web Services Documentation. URL: https://docs.aws.amazon.com/ses/latest/dg/Welcome.html (дата звернення: 14.05.2026).

21. Angular Single Page Applications: What Are the Benefits? Angular University. URL: https://blog.angular-university.io/why-a-single-page-application-what-are-the-benefits-what-is-a-spa/ (дата звернення: 14.05.2026).

22. Why Use Angular: Top Benefits of This Powerful Framework. Angular Minds. URL: https://www.angularminds.com/blog/why-use-angular (дата звернення: 14.05.2026).

23. Mission & Vision. DataCite. URL: https://datacite.org/mission/ (дата звернення: 14.05.2026).

24. Overview: DataCite Metadata Schema 4.5. DataCite Metadata Schema Documentation. URL: https://datacite-metadata-schema.readthedocs.io/en/4.5/properties/overview/ (дата звернення: 14.05.2026).

25. About DCMI. Dublin Core Metadata Initiative. URL: https://www.dublincore.org/about/ (дата звернення: 14.05.2026).

26. The ISO 639 Language Code. Infoterm. URL: http://www.infoterm.info/standardization/iso_639_1_2002.php (дата звернення: 14.05.2026).

27. What Is a REST API? Red Hat. URL: https://www.redhat.com/en/topics/api/what-is-a-rest-api (дата звернення: 14.05.2026).

28. Fowler M. Domain-Driven Design. Martin Fowler. URL: https://martinfowler.com/bliki/DomainDrivenDesign.html (дата звернення: 14.05.2026).

29. Cockburn A. Hexagonal Architecture. Alistair Cockburn. URL: https://alistair.cockburn.us/hexagonal-architecture (дата звернення: 14.05.2026).

30. What Is Angular? Angular. URL: https://angular.dev/overview (дата звернення: 14.05.2026).

31. Masinter L. RFC 7578: Returning Values from Forms: multipart/form-data. IETF Datatracker. URL: https://datatracker.ietf.org/doc/html/rfc7578 (дата звернення: 14.05.2026).

32. Session Management Cheat Sheet. OWASP Cheat Sheet Series. URL: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html (дата звернення: 14.05.2026).

33. JSON Types. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/current/datatype-json.html (дата звернення: 14.05.2026).

34. What Is Data Integrity? IBM. URL: https://www.ibm.com/think/topics/data-integrity (дата звернення: 14.05.2026).

35. Maven: Introduction. Apache Maven. URL: https://maven.apache.org/what-is-maven.html (дата звернення: 14.05.2026).

36. What Is Object-Relational Mapping? Amazon Web Services. URL: https://aws.amazon.com/what-is/object-relational-mapping/ (дата звернення: 14.05.2026).

37. Intro to Liquibase. Liquibase Documentation. URL: https://docs.liquibase.com/secure/implementation-guide-5-1/intro-to-liquibase (дата звернення: 14.05.2026).

38. Apache Maven Enforcer Plugin: Introduction. Apache Maven. URL: https://maven.apache.org/enforcer/maven-enforcer-plugin/ (дата звернення: 14.05.2026).

39. Project Lombok. Lombok. URL: https://projectlombok.org/ (дата звернення: 14.05.2026).

40. What Is SDK? Amazon Web Services. URL: https://aws.amazon.com/what-is/sdk/ (дата звернення: 14.05.2026).

41. What Is Apache FreeMarker? Apache FreeMarker. URL: https://freemarker.apache.org/index.html (дата звернення: 14.05.2026).

42. RouterOutlet. Angular API Reference. URL: https://angular.dev/api/router/RouterOutlet (дата звернення: 14.05.2026).

43. Control Route Access with Guards. Angular. URL: https://angular.dev/guide/routing/route-guards (дата звернення: 14.05.2026).

44. Fowler M. Polyglot Persistence. Martin Fowler. URL: https://martinfowler.com/bliki/PolyglotPersistence.html (дата звернення: 14.05.2026).

45. UUID. MDN Web Docs Glossary. URL: https://developer.mozilla.org/en-US/docs/Glossary/UUID (дата звернення: 14.05.2026).

46. Explicit Locking. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/current/explicit-locking.html (дата звернення: 14.05.2026).

47. SHA-256 Algorithm: Characteristics, Steps, and Applications. Simplilearn. URL: https://www.simplilearn.com/tutorials/cyber-security-tutorial/sha-256-algorithm (дата звернення: 14.05.2026).

48. XML Metadata. Nintex Documentation. URL: https://help.nintex.com/en-US/k2blackpearl/userguide/4.6.11/XML_Metadata.html (дата звернення: 14.05.2026).

49. MockMvc. Spring Framework Documentation. URL: https://docs.spring.io/spring-framework/reference/testing/mockmvc.html (дата звернення: 14.05.2026).

50. Test Slices. Spring Boot Documentation. URL: https://docs.spring.io/spring-boot/appendix/test-auto-configuration/slices.html (дата звернення: 14.05.2026).

51. Task Execution and Scheduling. Spring Framework Documentation. URL: https://docs.spring.io/spring-framework/reference/integration/scheduling.html (дата звернення: 14.05.2026).

52. Testcontainers. Spring Boot Documentation. URL: https://docs.spring.io/spring-boot/reference/testing/testcontainers.html (дата звернення: 14.05.2026).

53. TestBed. Angular API Reference. URL: https://angular.dev/api/core/testing/TestBed (дата звернення: 14.05.2026).

54. Getting Started. Vitest Guide. URL: https://vitest.dev/guide/ (дата звернення: 14.05.2026).

55. JavaScript Document Object Model (JSDOM). U.S. Department of Veterans Affairs. URL: https://www.oit.va.gov/Services/TRM/ToolPage.aspx?tid=11176 (дата звернення: 14.05.2026).

56. What Is End-to-End (E2E) Testing? IBM. URL: https://www.ibm.com/think/topics/end-to-end-testing (дата звернення: 14.05.2026).

57. Лисенко В.І. Дизайн веб-застосунку з використанням гексагональної архітектури/Герцюк М.М., Лисенко В.І.// VII Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез 23.04.2026, м. Київ. ст. 20-23.

58. Лисенко В.І. Дизайн веб-застосунку з використанням гексагональної архітектури/Герцюк М.М., Лисенко В.І.// Всеукраїнська науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез 03.05.2026, м. Київ (подано до друку)

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

Цифрова платформа відкритого репозиторію для управління та публікації відкритих наборів даних

Виконав студент 4 курсу
групи ТЦР-44
Лисенко Владислав Ігорович

Керівник роботи
Д-р філософії, доцент кафедри ТЦР Герцюк Микола Модестович

Київ - 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: Вдосконалити процес публікації та управління відкритими наборами даних шляхом розроблення платформи репозиторію відкритих даних, що обмежує функціонал до ключових можливостей, фокусується саме на наборах даних і базується на гнучкій архітектурі

Об'єкт дослідження: Процеси управління, зберігання та публікації відкритих наборів даних у межах спеціалізованих інформаційних систем. Основна увага приділяється організації доступу до даних, їх структуризації, версіонування, опису метаданих та можливості повторного використання

Предмет дослідження: Цифрова платформа відкритого репозиторію, призначена для управління та публікації відкритих наборів даних

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Дослідити предметну галузь та проаналізувати складності дизайну подібних систем.
2. Обрати набір технологій для реалізації
3. Спроєктувати цифрову платформу.
4. Реалізувати серверну та клієнтську частини платформи для управління та публікації відкритих наборів даних.
5. Протестувати обидві частини застосунку.

3

АНАЛІЗ АНАЛОГІВ

Критерій	Harvard Dataverse	Zenodo	Kaggle	OpenData
Орієнтація платформи	Спеціалізація на наукових даних, але з розширеним функціоналом	Універсальна платформа (дані, статті, ПЗ тощо)	Орієнтація на аналіз даних і машинне навчання	Чітка спеціалізація тільки на наборах даних
Стандарти метаданих	Високий рівень стандартизації (DataCite та галузеві метадані)	Узагальнена підтримка, яка включає основні стандарти	Обмежена, не формалізована (Schema.org, Croissant)	Фокус на основні DataCite та Dublin Core
Постійні ідентифікатори (DOI)	Підтримуються	Підтримуються	Частково (не основний фокус)	Підтримуються
Версіонування	Розвинене, але може бути складним	Підтримується	Обмежене/неявне	Спрощене та прозоре
Складність використання	Середня/висока через гнучкість	Середня	Низька	Низька за рахунок обмеженого функціоналу
Управління доступом	Гнучке, багаторівневе	Гнучке	Обмежене	Чітка і проста модель доступу без надлишкової складності

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

- реєстрація, підтвердження email та автентифікація користувача;
- створення, редагування та публікація наборів даних;
- додавання файлів і заповнення основних метаданих;
- підтримка версіонування наборів даних;
- перегляд каталогу, сторінки набору даних та історії версій;
- експорт метаданих у форматах Dublin Core і DataCite.

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ (прод.)

Нефункціональні вимоги:

- захист облікових даних і перевірка прав доступу;
- незмінність опублікованих версій та збереження історії змін;
- валідація метаданих і файлів на клієнті та сервері;
- обробка помилок автентифікації, конфліктів і неіснуючих ресурсів;
- простий і зрозумілий інтерфейс користувача.

ПРОГРАМНІ ЗАСОБИ ІМПЛЕМЕНТАЦІЇ

Мови програмування: Java, TypeScript

Фреймворки: Spring Boot, Angular

Бази даних: PostgreSQL, Redis

Хмарні сервіси: AWS S3, AWS SES

Середовища розробки: IntelliJ IDEA, WebStorm

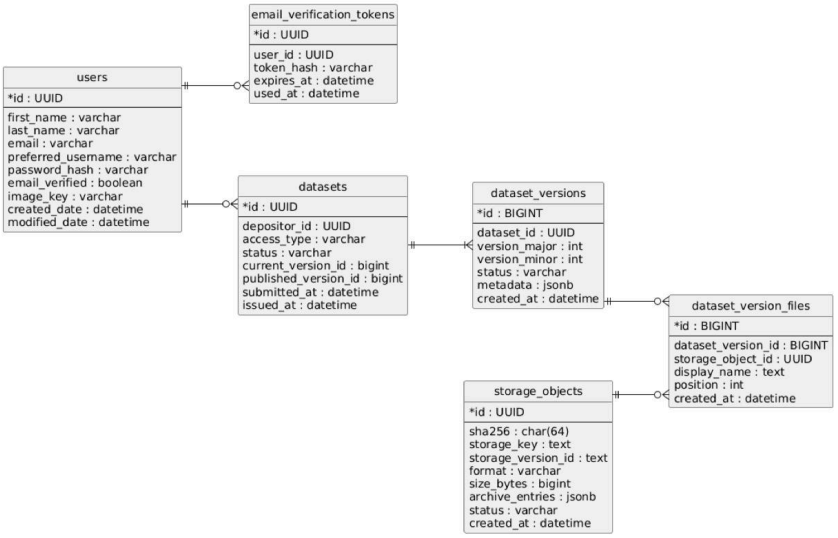
Інструменти розробки: Liquibase, Git

7

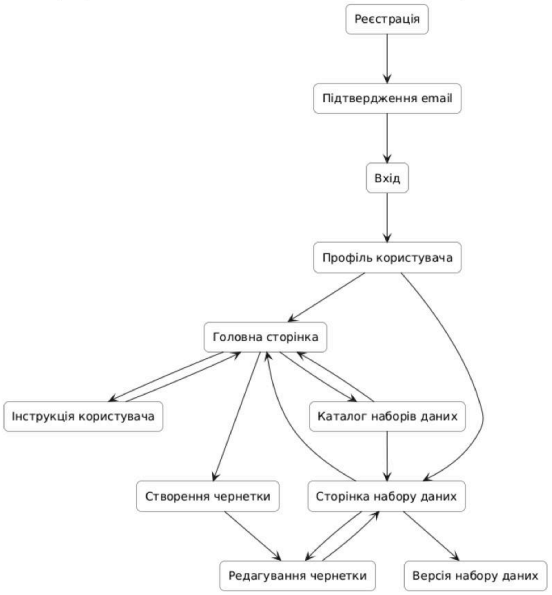
ДІАГРАМА ПРЕЦЕДЕНТІВ

8

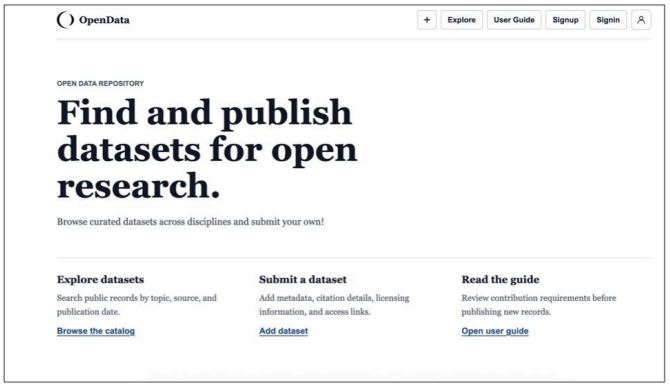
ER-ДІАГРАМА ЗВ’ЯЗКІВ

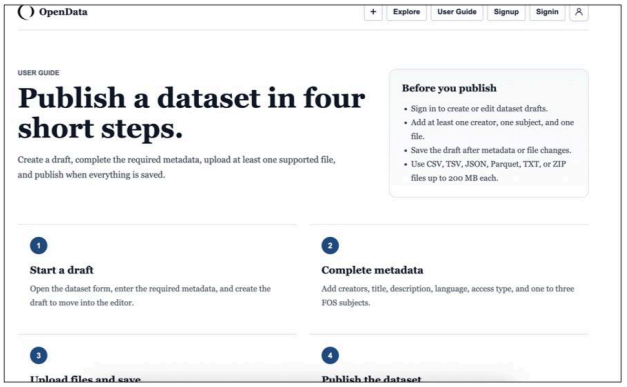


ДІАГРАМА НАВІГАЦІЇ



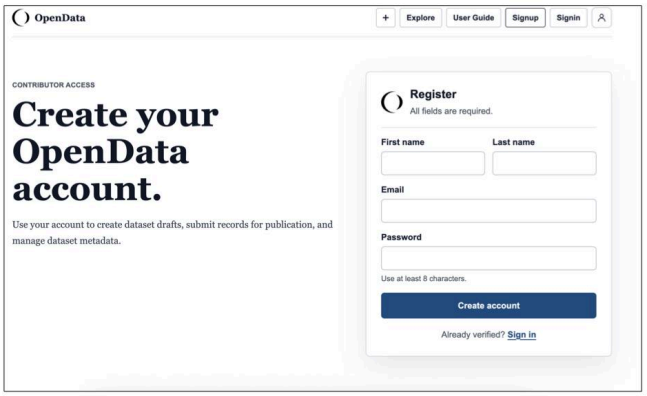
ЕКРАННІ ФОРМИ

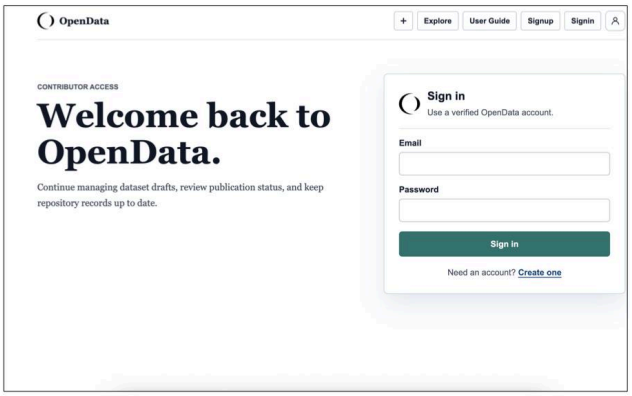




11

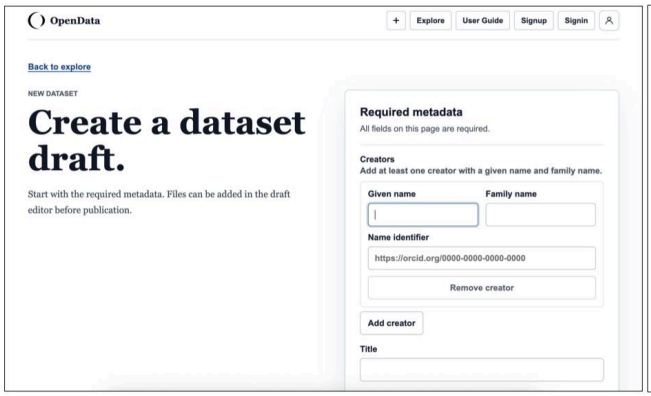
ЕКРАННІ ФОРМИ

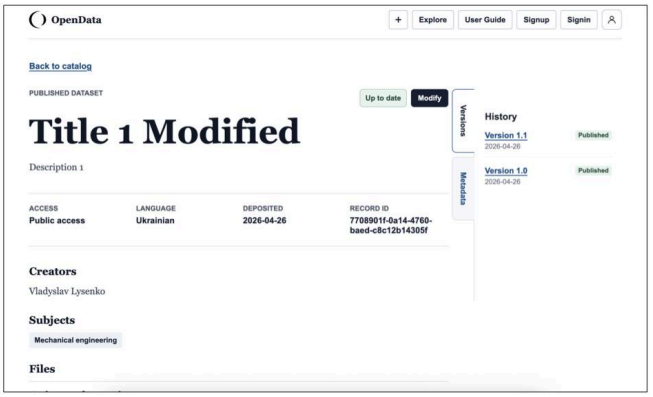




12

ЕКРАННІ ФОРМИ





13

ВІДЕО ПРЕЗЕНТАЦІЯ ЗАСТОСУНКУ

14

ТЕСТУВАННЯ

Проведено автоматизоване тестування backend і frontend частин.
Перевірено основні сценарії роботи з користувачами, наборами даних, версіями, файлами та API.
Загалом виконано 170 тестів, усі завершилися успішно.

Частина	Що перевірялося	Кількість	Результат
Backend	Domain, application services, REST controllers, infrastructure, persistence	123	Успішно
Frontend	Routing, API layer, guards, session restore, dataset workflows	47	Успішно
Загалом		170	Успішно

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Лисенко В.І. Дизайн веб-застосунку з використанням гексагональної архітектури VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез 23.04.2026, м. Київ.

2. Лисенко В.І. Дизайн веб-застосунку з використанням гексагональної архітектури Всеукраїнської науково-технічної конференції «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез 03.05.2026, м. Київ.

ВИСНОВКИ

Поставлену мету роботи було досягнуто: розроблено платформу репозиторію відкритих даних, функціонал якої обмежено до публікації та управління саме відкритими наборами даних.

У результаті виконання роботи можна сформулювати такі висновки:

1. Проаналізовано предметну галузь та основні складності управління відкритими наборами даних.
2. Обґрунтовано потребу у платформі з ключовими функціями для публікації та повторного використання даних.
3. Спроєктовано веб-застосунок для створення, опису, версіонування та перегляду наборів даних.
4. Перевірено основну бізнес-логіку системи та підтверджено працездатність реалізованого рішення.

ДЯКУЮ ЗА УВАГУ!

ДОДАТОК Б. ЛІСТИНГИ КОДУ ПРОГРАМНИХ МОДУЛІВ

Фрагменти серверної частини застосунку

1. Доменне версіонування в агрегаті Dataset

```
public void replaceFiles(List<DatasetFile> files) {
   if (getCurrentVersion().hasSameFiles(files)) {
      return;
   }

   DatasetVersion draft = findDraftVersion()
         .map(this::promoteDraftForFileChanges)
         .orElseGet(() -> createNewDraftVersion(true));
   draft.replaceFiles(files);
}

private DatasetVersion createNewDraftVersion(boolean major) {
   DatasetVersion published = getLatestPublishedVersion();

   VersionNumber version = major
         ? published.getVersion().incrementMajor()
         : published.getVersion().incrementMinor();

   DatasetVersion draft = published.copyAsDraft(version);

   versions.add(draft);
   this.currentVersion = version;
   return draft;
}

public void publish() {
   DatasetVersion draft = getDraftVersion();

   draft.publish();
   if (this.status == ResourceStatus.DRAFT) this.status =
ResourceStatus.PUBLISHED;

   this.publishedVersion = draft.getVersion();
   this.currentVersion = draft.getVersion();
   this.issuedAt = Instant.now();
}
```

2. Створення набору даних у CreateDatasetService

```
@Override
public CreateDatasetResult execute(UserId depositorId,
CreateDatasetCommand request) {
   if (datasetRepository.isDraftLimitExceeded(depositorId)) throw
LimitExceededException.ofDrafts();

   List<Creator> creators = request.creators().stream()
         .map(CreatorCommand::toDomain)
         .toList();

   DatasetAccessType accessType =
DatasetAccessType.valueOf(request.accessType());

   DatasetMetadata metadata = DatasetMetadata.initiate(
         creators,
         request.title(),
         request.description(),
         Language.from(request.language()),
         request.subjects()
   );

   Dataset dataset = Dataset.create(depositorId, accessType, metadata);
   List<DatasetFile> files = fileResolver.resolve(request.files());
   dataset.replaceFiles(files);

   datasetRepository.create(dataset);

   return new CreateDatasetResult(dataset.getId().value());
}
```

3. REST-адаптер у DatasetController

```
@PostMapping
public ResponseEntity<CreateDatasetResponse>
create(@AuthenticationPrincipal OpenDataUserDetails principal,
                                    @Valid @RequestBody
CreateDatasetRequest request) {

   if (principal == null) return ResponseEntity.status(401).build();
   UserId principalId = UserId.fromUUID(principal.getId());

   List<CreatorCommand> creators = request.creators().stream()
         .map(DatasetController::toCreatorCommand)
         .toList();

   var result = createUseCase.execute(principalId,
toCreateDatasetCommand(request, creators));
   return new ResponseEntity<>(new
CreateDatasetResponse(result.id()), HttpStatus.CREATED);
}

private static CreateDatasetCommand
toCreateDatasetCommand(CreateDatasetRequest request,
                                    List<CreatorCommand> creators)
{
   return new CreateDatasetCommand(
         creators,
         request.title(),
         request.description(),
         request.language(),
```

```
        request.accessType(),
        request.subjects(),
        toDatasetFileCommands(request.files())
    );
}
```

4. JPA-мапінг версії набору даних

```
@JdbcTypeCode(SqlTypes.JSON)
@Convert(converter = MetadataConverter.class)
@Column(name = "metadata", columnDefinition = "jsonb")
private DatasetMetadata metadata;

@OneToMany(mappedBy = "datasetVersion", cascade =
CascadeType.ALL, orphanRemoval = true)
@OrderBy("position ASC")
private List<DatasetVersionFileEntity> files = new ArrayList<>();
```

Фрагменти клієнтської частини застосунку

1. Маршрутизація клієнтського Angular-застосунку

```
import { Routes } from '@angular/router';
import { requireAuthGuard } from './auth/auth.guard';
import { DatasetDetailPage } from './dataset-detail/dataset-detail';
import { DatasetCreatePage } from './datasets/dataset-create';
import { DatasetDraftEditorPage } from
'./datasets/dataset-draft-editor';
import { ExplorePage } from './explore/explore';
import { HomePage } from './home/home';
import { SigninPage } from './signin/signin';
import { SignupPage } from './signup/signup';
import { UserGuidePage } from './user-guide/user-guide';
import { UserProfilePage } from './user-profile/user-profile';
import { VerifyEmailPage } from './verify-email/verify-email';

export const routes: Routes = [
  {
    path: '',
    component: HomePage,
    title: 'OpenData',
  },
  {
    path: 'signup',
    component: SignupPage,
    title: 'Signup | OpenData',
  },
  {
    path: 'signin',
    component: SigninPage,
    title: 'Signin | OpenData',
  },
  {
    path: 'verify-email',
    component: VerifyEmailPage,
    title: 'Verify Email | OpenData',
  },
  {
    path: 'explore',
    component: ExplorePage,
    title: 'Explore | OpenData',
  },
  {
    path: 'user-guide',
    component: UserGuidePage,
    title: 'User Guide | OpenData',
  },
  {
    path: 'datasets/new',
    component: DatasetCreatePage,
    canActivate: [requireAuthGuard],
    title: 'Create Dataset | OpenData',
  },
  {
    path: 'datasets/:id/draft',
    component: DatasetDraftEditorPage,
    canActivate: [requireAuthGuard],
    title: 'Dataset Draft | OpenData',
  },
  {
    path: 'datasets/:id/versions/:versionId',
    component: DatasetDetailPage,
    title: 'Dataset Version | OpenData',
  },
  {
    path: 'datasets/:id',
    component: DatasetDetailPage,
    title: 'Dataset | OpenData',
  },
  {
    path: 'user/:userId',
    component: UserProfilePage,
    canActivate: [requireAuthGuard],
    title: 'User Profile | OpenData',
  },
  {
    path: '**',
    redirectTo: '',
  },
];
```

2. Відновлення сесії в кореневому компоненті

```
import { Component, computed, inject } from '@angular/core';
import { RouterLink, RouterOutlet } from '@angular/router';
import { ApiError } from './api/api-client';
import { fetchCurrentUser } from './auth/auth-api';
```

```
import { AuthSession } from './auth/auth-session';
import { fetchUserProfile } from './users/user-profile-api';

@Component({
  selector: 'app-root',
  imports: [RouterLink, RouterOutlet],
  templateUrl: './app.html',
  styleUrl: './app.css',
})
export class App {
  readonly session = inject(AuthSession);
  readonly profileLink = computed(() => {
    const user = this.session.currentUser();
    return user ? ['/user', user.id] : ['/signin'];
  });

  constructor() {
    void this.restoreSession();
  }

  private async restoreSession(): Promise<void> {
    if (this.session.currentUser()) {
      return;
    }

    try {
      const user = await fetchCurrentUser();
      this.session.setUser(user);

      try {
        this.session.setProfile(await fetchUserProfile(user.id));
      } catch {
        // this auth check is authoritative, profile hydration only improves
header avatar
      }
    } catch (error) {
      if (error instanceof ApiError && error.status === 401) {
        this.session.clear();
      }
    }
  }
}
```

3. Централізована обробка HTTP-запитів в API-шарі

```
export type ApiMethod = 'GET' | 'POST' | 'PATCH' | 'DELETE';

export type ApiErrorBody = {
  timestamp?: string;
  message?: string;
  path?: string;
  error?: string;
  statuscode?: number;
};

export type ApiRequestOptions = {
  method: ApiMethod;
  body?: unknown;
};

export type ApiFormDataRequestOptions = {
  method: Extract<ApiMethod, 'POST' | 'PATCH'>;
  body: FormData;
};

export class ApiError extends Error {
  constructor(
    readonly status: number,
    message: string,
    readonly body: ApiErrorBody | null,
  ) {
    super(message);
    this.name = 'ApiError';
  }
}

export async function apiFetch(
  path: `/api/v1/${string}`,
  options: ApiRequestOptions,
): Promise<Response> {
  const hasJsonBody = options.body !== undefined;
  const response = await fetch(path, {
    method: options.method,
    credentials: 'include',
    headers: hasJsonBody ? { 'Content-Type': 'application/json' } :
undefined,
    body: hasJsonBody ? JSON.stringify(options.body) : undefined,
  });

  if (!response.ok) {
    const errorBody = await readApiErrorBody(response);
    throw new ApiError(response.status, errorBody?.message ??
response.statusText, errorBody);
  }

  return response;
}

export async function apiFormDataFetch(
  path: `/api/v1/${string}`,
  options: ApiFormDataRequestOptions,
): Promise<Response> {
  const response = await fetch(path, {
    method: options.method,
    credentials: 'include',
    body: options.body,
```

```
  });

  if (!response.ok) {
    const errorBody = await readApiErrorBody(response);
    throw new ApiError(response.status, errorBody?.message ??
response.statusText, errorBody);
  }

  return response;
}

async function readApiErrorBody(response: Response):
Promise<ApiErrorBody | null> {
  const contentType = response.headers.get('content-type') ?? '';
  if (!contentType.includes('application/json')) {
    return null;
  }

  const body: unknown = await response.json().catch(() => null);
  if (!isApiErrorBody(body)) {
    return null;
  }

  return body;
}

function isApiErrorBody(body: unknown): body is ApiErrorBody {
  if (body === null || typeof body !== 'object') {
    return false;
  }

  const candidate = body as Record<string, unknown>;
  return (
    optionalString(candidate['timestamp']) &&
    optionalString(candidate['message']) &&
    optionalString(candidate['path']) &&
    optionalString(candidate['error']) &&
    optionalNumber(candidate['statuscode'])
  );
}

function optionalString(value: unknown): boolean {
  return value === undefined || typeof value === 'string';
}

function optionalNumber(value: unknown): boolean {
  return value === undefined || typeof value === 'number';
}
```