Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук

(повна назва)

Кафедра програмної інженерії

(повна назва)

**КВАЛІФІКАЦІЙНА РОБОТА**

**Пояснювальна записка**

рівень вищої освіти перший (бакалаврський)

Програмна система для реалізації параметричного пошуку партнерів для домашніх улюбленців з метою розведення породистих тварин. Контроль якості програмної системи

(тема)

Виконав:

здобувач 4 року навчання

групи ПЗПІ-21-11

Владислав ОМЕЛЬЧЕНКО

(Власне ім’я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного\_ \_\_

забезпечення

(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Програмна інженерія

(повна назва освітньої програми)

Керівник ст.викл. кафедри ПІ Олександр САМАНЦОВ

(посада, Власне ім’я, ПРІЗВИЩЕ)

Допускається до захисту

Зав. кафедри \_\_\_\_\_\_\_\_\_\_\_ Кирило СМЕЛЯКОВ

(підпис) (Власне ім’я, ПРІЗВИЩЕ)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук

Кафедра програмної інженерії

Рівень вищої освіти перший (бакалаврський)

Спеціальність 121 – Інженерія програмного забезпечення

Тип програми Освітньо-професійна

Освітня програма Програмна Інженерія

(шифр і назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри \_\_\_\_\_\_\_\_\_

(підпис)

« » 2025 р.

**ЗАВДАННЯ**

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Омельченку Владиславу Віталійовичу

(прізвище, ім’я, по батькові)

1. Тема роботи Програмна система для реалізації параметричного пошуку партнерів для домашніх улюбленців з метою розведення породистих тварин. Контроль якості програмної системи

Затверджена наказом по університету від 19.05.2025р. № 397 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 06.06.2025 \_

3. Вихідні дані до роботи Розробити систему тестування клієнтської частини вебзастосунку для параметричного підбору партнерів серед домашніх тварин. Реалізацію виконати з використанням JavaScript, бібліотеки React і фреймворків Playwright, Jest і React Testing Library для модульного, інтеграційного та e2e-тестування з інтеграцією REST API та WebSocket.

4. Перелік питань, що потрібно опрацювати в роботі

Вступ, аналіз предметної області, виявлення та вирішення проблем, постановка задачі, формування вимог до програмної системи, архітектура та проєктування, опис прийнятих програмних рішень, реалізація основних компонентів, висновки.

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| № | Назва етапів роботи | Термін виконання етапів роботи | Примітка |
| 1 | Аналіз предметної галузі | 20.05.2025 | *виконано* |
| 2 | Створення специфікації ПЗ | 22.05.2025 | *виконано* |
| 3 | Проектування ПЗ | 28.05.2025 | *виконано* |
| 4 | Розробка ПЗ | 30.05.2025 | *виконано* |
| 5 | Тестування ПЗ | 06.06.2025 | *виконано* |
| 6 | Оформлення пояснювальної записки | 07.06.2025 | *виконано* |
| 7 | Підготовка презентації та доповіді | 08.06.2025 | *виконано* |
| 8 | Попередній захист | 09.06.2025 | *виконано* |
| 9 | Нормоконтроль, рецензування | 11.06.2025 | *виконано* |
| 10 | Здача роботи у електронний архів | 11.06.2025 | *виконано* |
| 11 | Допуск до захисту у зав. кафедри | 12.06.2025 | *виконано* |

Дата видачі завдання «19» «травня» 2025р.

Здобувач \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис)

Керівник роботи ст.викл. кафедри ПІ Олександр САМАНЦОВ

(підпис) (посада, Власне ім’я, ПРІЗВИЩЕ)

**РЕФЕРАТ**

Пояснювальна записка до кваліфікаційної роботи бакалавра, 52 стор., 4 рис., 1 табл., 13 джерел, 3 додатки.

РОЗВЕДЕННЯ, ТВАРИНИ, ТЕСТУВАННЯ, JEST, PLAYWRIGHT, PYTHON, REACT, SPA, QA

У роботі розглянуто забезпечення якості клієнтської частини інформаційної системи для параметричного підбору партнерів серед домашніх тварин. Проєкт реалізовано як односторінковий вебзастосунок (SPA) на базі бібліотеки React із використанням Redux Toolkit, i18next та CSS. Основний функціонал охоплює створення анкет тварин, пошук, чат, симпатії, відгуки та підтримку багатомовності.

Особливу увагу приділено впровадженню комплексної стратегії тестування, що включає unit-, інтеграційні та end-to-end перевірки з використанням Jest, React Testing Library і Playwright. Розроблено систему мок-серверів для емуляції API-відповідей, протестовано сценарії з втратою з’єднання, помилками авторизації, багатомовністю та edge-case ситуаціями взаємодії.

Результатом є повністю протестований SPA-застосунок з централізованим контролем якості, готовий до CI/CD-інтеграції та подальшого масштабування.

**ABSTRACT**

Explanatory note to the bachelor's qualification project, 52 pages, 4 figures, 1 table, 13 sources, 3 additions.

BREEDING, ANIMALS, TESTING, JEST, PLAYWRIGHT, PYTHON, REACT, SPA, QA

This thesis addresses the quality assurance of the client-side of a web application for parameter-based partner matching among domestic animals. The project is implemented as a single-page application (SPA) based on the React library with Redux Toolkit, i18next and CSS. Key features include pet profiles, parametric search, real-time chat, likes, reviews, and multilingual support.

The core focus is on developing a comprehensive testing strategy, integrating unit, integration, and end-to-end testing with Jest, React Testing Library, and Playwright. A mock server system is implemented for API emulation, and scenarios such as connection loss, authorization errors, localization, and edge-case handling are tested.

The result is a fully tested SPA with centralized quality control, ready for CI/CD integration and future scalability.

**ЗМІСТ**

[Перелік скорочень 7](#_Toc200824812)

[Вступ 8](#_Toc200824813)

[1 Аналіз предметної галузі 11](#_Toc200824814)

[1.1 Аналіз предметної області 11](#_Toc200824815)

[1.2 Виявлення та вирішення проблем 13](#_Toc200824816)

[1.3 Постановка задачі 15](#_Toc200824817)

[2 Аналіз об’єкта дослідження 18](#_Toc200824818)

[3 Архітектура та проєктування 21](#_Toc200824819)

[3.1 UML-проєктування ПЗ 21](#_Toc200824820)

[3.2 Архітектура ПЗ 25](#_Toc200824821)

[3.3 Проєктування структури зберігання даних 27](#_Toc200824822)

[3.4 Приклади найцікавіших алгоритмів та методів 29](#_Toc200824823)

[3.5 Створення UI/UX або іншого дизайну системи 32](#_Toc200824824)

[4 Тестування програмної системи 34](#_Toc200824825)

[4.1 Підхід до тестування SPA-застосунку 34](#_Toc200824826)

[4.2 Unit- і інтеграційне тестування з Jest та RTL 36](#_Toc200824827)

[4.3 Поведінкове тестування та підсумкові результати 38](#_Toc200824828)

[Висновки 41](#_Toc200824829)

[Перелік джерел посилання 43](#_Toc200824830)

[Додаток А 44](#_Toc200824831)

[Додаток Б 45](#_Toc200824832)

[Додаток в 46](#_Toc200824833)

# **Перелік скорочень**

SPA – Single-Page Application

QA – Quality Assurance

API – Application Programming Interface

CI – Continuous Integration

CD – Continuous Delivery

MIME – Multipurpose Internet Mail Extensions

QA – Quality Assurance

CSRF – Cross-Site Request Forgery

RTL – React Testing Library

# **Вступ**

У сучасному середовищі веброзробки якість програмного забезпечення визначає не лише його працездатність, а й довіру користувачів, здатність до масштабування, передбачуваність поведінки та рівень підтримки в довгостроковій перспективі. Забезпечення якості (QA) у цьому контексті вже давно вийшло за межі ручного тестування й охоплює широкий спектр методів: автоматизацію, перевірку відповідності інтерфейсу, контроль граничних сценаріїв, аналіз помилок API, інтеграцію з CI/CD тощо. Особливої актуальності набуває комплексне тестування односторінкових застосунків (SPA), які не просто відображають дані, а формують динамічну логіку взаємодії, стану, умовного рендерингу й багатомовності.

Об’єктом кваліфікаційної роботи є інформаційна система PetBreedingApp – вебзастосунок для параметричного підбору партнерів серед домашніх тварин. У межах реалізації користувачі мають змогу створювати анкети тварин, фільтрувати їх за характеристиками, додавати до обраного, ставити симпатії, обмінюватися повідомленнями, залишати відгуки, переглядати рейтинги, а також керувати фото у вигляді галереї. Уся взаємодія реалізована через SPA-інтерфейс на React із централізованим керуванням станом (Redux Toolkit), WebSocket-чатом, багатомовною підтримкою (i18next) та адаптивною версткою (Tailwind CSS). Бекенд функціонує на основі Django з асинхронною WebSocket-підсистемою та REST API. За зберігання даних відповідає MongoDB [1].

Складність і варіативність користувацьких сценаріїв у проєкті передбачає підвищені вимоги до QA-компоненти. Основна мета роботи – впровадження системної стратегії забезпечення якості на рівні фронтенду, з орієнтацією на стабільність у граничних випадках, коректне відображення станів, підтримку мовного оточення, умовного рендерингу та асинхронної поведінки. Особлива увага приділяється тестуванню життєвих циклів компонентів (форма реєстрації, логін, створення анкети, надсилання повідомлень), обробці помилок авторизації, перевірці валідації форм, реакції на втрату сесії, правильному відображенню toast-повідомлень та скелетонів, а також цілісності навігації в умовах різних ролей.

Автоматизоване тестування є ключовою складовою підходу. Тестування компонентів здійснюється за допомогою бібліотек Jest і React Testing Library [2], які дозволяють реалізувати як unit, так і інтеграційні тести. Поведінкове тестування реалізовано через Playwright – фреймворк для e2e-тестів у headless-браузерах [3]. Він дозволяє емуляцію реальних сценаріїв користувача: від авторизації до взаємодії з чатами. Також розроблено набір мок-серверів для імітації відповідей API з різними статусами (200, 401, 403, 500), що дозволяє тестувати поведінку системи при помилках мережі, тайм-аутах або втраті з’єднання.

Система має багатокомпонентну структуру, що додатково ускладнює перевірку цілісності. Наприклад, компонент MatchSwiper реалізує Tinder-подібну логіку «свайпів» між анкетами, що потребує перевірки не лише на рівні правильного відображення, а й на рівні зміни стейту, збереження історії «свайпів», генерації взаємної симпатії й подальшого переходу до чату. Галерея фото також складається з кількох залежних компонентів – PhotoGallery, PhotoPreviewGrid, PhotoViewer – і потребує перевірки логіки завантаження, видалення, ререндерингу з кешем, валідації формату файлів та відображення на різних пристроях.

Локалізація є ще одним критичним аспектом. Уся система підтримує динамічне перемикання мов у реальному часі без перезавантаження сторінки. Це означає, що кожен компонент має коректно оновити свій текстовий контент після зміни контексту локалізації, зберігаючи при цьому стан (наприклад, уже заповнену форму або відкриту вкладку). Тестування локалізації виконується в ізольованих середовищах із перевіркою наявності всіх ключів у словниках, коректного перемикання та відсутності візуальних зсувів у верстці через різну довжину тексту в різних мовах.

Важливо також забезпечити підтримку безперервної інтеграції: тести мають запускатись автоматично при кожному коміті в основну гілку. Для цього в проєкті передбачена інтеграція автоматизованих тестів у CI-пайплайн [4]. Усі тести проходять у headless-режимі через Playwright CLI, із логуванням результатів, таймінгів, статусів та візуальних скріншотів у разі помилки. Це дозволяє не лише виявляти регресії, а й підтримувати сталість поведінки під час розширення функціоналу.

Таким чином, у межах кваліфікаційної роботи було поставлено завдання – розробити та впровадити повноцінну QA-стратегію для фронтенд-частини складного SPA-застосунку з урахуванням усіх особливостей: асинхронності, мультимовності, багаторівневої логіки доступу, валідації, адаптивності та підтримки WebSocket. Робота охоплює аналіз потенційних ризиків, формалізацію вимог до тестованості, побудову карти покриття, створення стабільних тест-кейсів, макетування edge-case сценаріїв і забезпечення інтеграції в процеси розробки на всіх етапах життєвого циклу системи.

# **1 Аналіз предметної галузі**

1.1 Аналіз предметної області

Платформи для пошуку партнерів у розведенні домашніх тварин займають проміжну нішу між класичними сервісами оголошень і соціальними мережами з розширеним функціоналом взаємодії. Їхня відмінність полягає у високій структурованості даних: кожна тварина представлена як унікальна анкета з полями, які мають значення для біологічної сумісності – порода, стать, вік, колір шерсті, активність профілю тощо. Крім цього, власники прагнуть бачити не лише самі анкети, а й мати змогу оцінити профіль іншого користувача, залишити або переглянути відгуки, поспілкуватись у реальному часі та отримати відповідь без затримок. Усе це потребує інтегрованого підходу до побудови інтерфейсу з одночасною підтримкою складної бізнес-логіки, адаптивності та динамічних сценаріїв поведінки.

Система PetBreedingApp реалізує повний цикл взаємодії між зареєстрованими користувачами: реєстрація, авторизація, створення анкети тварини, фільтрація за заданими параметрами, відмітка симпатії, чат при взаємному збігу, залишення та перегляд відгуків, керування фото та профілем. Усі ці дії здійснюються у межах єдиного клієнтського застосунку, побудованого як SPA. Основна взаємодія відбувається через REST API, а реальний час забезпечується WebSocket-комунікацією.

З погляду QA-фахівця, система є високонавантаженою як на рівні логіки переходів, так і з точки зору паралельних станів компонентів. Наприклад, одна й та сама форма (PetEditor) може знаходитись у кількох режимах: створення нової анкети, редагування існуючої, відображення помилок API, блокування при повторному сабміті, показ скелетону тощо. Інший приклад – MatchSwiper, де потрібна перевірка не лише логіки swipe-вибору, а й обробки симпатії, зміни поточної картки, відображення toast-сповіщення у разі взаємного збігу та відкриття модального вікна.

Також варто врахувати, що усі користувачі системи розподіляються на ролі. Незареєстровані мають доступ лише до сторінок входу й реєстрації. Авторизовані користувачі мають змогу створювати анкети, ставити лайки, вести чат, а також бачити чужі профілі. Адміністратори мають доступ до модерації анкет і відгуків. Усі компоненти повинні поводитись інакше залежно від контексту авторизації. Це означає, що кожен тестовий сценарій має бути протестований не лише із «чистого профілю», а й у контексті прав доступу: наприклад, чужу анкету не можна редагувати, чужий відгук – видалити, а вхід до чату можливий лише при наявності симпатії [5].

Окрему категорію складності формує інтеграція з локалізацією. Усі тексти винесені до словників через i18n, а перемикання мови не скидає стан компонента [6]. Тобто перевірка поведінки застосунку має включати сценарії, коли користувач змінює мову на середньому етапі заповнення форми, під час відкритого toast або при активному WebSocket-з’єднанні. У разі помилки ключа або недоступності перекладу мають відображатися fallback-підписи, які також перевіряються через тестування.

Також значну увагу слід приділити взаємодії з мультимедіа. Компоненти PhotoGallery та PhotoViewer підтримують додавання, перегляд, видалення й перемикання фото, включно з попереднім завантаженням, кешем, адаптацією до ширини екрану й обробкою збоїв при завантаженні. Крім того, у разі надсилання файлу з неприпустимим MIME-типом має спрацьовувати клієнтська валідація, яка блокує подальшу дію. Перевірка цих сценаріїв неможлива без використання e2e-тестів у headless-браузері.

Реалізація обміну повідомленнями також вимагає особливої уваги. Компонент ChatRoom взаємодіє з WebSocket-з’єднанням через Redux-слайси та локальний стан [7]. При цьому потрібно перевірити як вхідні повідомлення, так і реакцію на недоступність з'єднання, обрив каналу, втрату сесії або відправлення повідомлення в розірваний канал. Усі ці ситуації мають бути змодельовані й протестовані із симульованими відповідями.

Таким чином, предметна область проєкту є прикладом високореактивної системи з багатокомпонентною взаємодією, динамічними змінами стану, мультимовною адаптацією, WebSocket-комунікацією та мультимедійним контентом. З погляду тестування, це вимагає розробки комплексного підходу до покриття, валідації сценаріїв, контролю поведінкових переходів, перевірки рендерингу в умовах нестабільного API, а також систематичного модульного й end-to-end тестування із залученням реального браузерного середовища.

1.2 Виявлення та вирішення проблем

Інформаційні системи, орієнтовані на соціальну взаємодію користувачів та обмін структурованими даними, зокрема в контексті підбору партнерів для домашніх тварин, мають низку типових функціональних і нефункціональних проблем. Виявлення таких проблем ще на етапі проєктування дозволяє сформувати адекватні вимоги до системи, знизити кількість критичних помилок у процесі розробки та спростити подальше тестування. Забезпечення якості в подібних системах потребує передбачення не лише очевидних ризиків, але й складних випадків взаємодії між користувачами, формами введення, асинхронними подіями та обмеженнями доступу.

Однією з головних проблем предметної області є відсутність перевірки достовірності введених даних у формах, які мають критичне значення – наприклад, анкета тварини або реєстрація користувача. Якщо система дозволяє ввести некоректні або частково заповнені дані, це призводить до накопичення помилкової інформації, яка знижує ефективність подальшого пошуку. Тому вже на етапі технічного проєктування слід передбачити багаторівневу валідацію: як на клієнтському рівні (відображення помилок у режимі реального часу), так і на серверному (перевірка унікальності, коректності типів і структури даних).

Ще однією характерною проблемою є порушення правил доступу. У випадках, коли користувач має змогу переглядати або редагувати дані, які йому не належать, виникає загроза як безпеці, так і довірі до системи. У подібних застосунках важливо забезпечити чітке розмежування між правами власника анкети, стороннього користувача та адміністратора. Всі компоненти повинні динамічно адаптуватись до ролі поточного користувача та не допускати несанкціонованих дій навіть при спробі прямого доступу через URL або інші засоби обхідної навігації.

Особливе значення в предметній області мають мультимедійні файли, зокрема фото тварин. Системи, які дозволяють завантаження файлів, стикаються з проблемами перевантаження сховища, спроб завантаження недопустимих типів, дублювання, а також з питаннями адаптивного відображення зображень на різних пристроях. Відсутність обробки edge-case сценаріїв, як-от надсилання пошкоджених або надмірно великих файлів, може спричинити збій у роботі клієнта. Для таких ситуацій необхідно заздалегідь проєктувати механізми захисту, кешування та fallback-логіку.

Високий рівень інтерактивності, характерний для SPA, створює додаткові труднощі в перевірці послідовності дій. Наприклад, одночасна взаємодія з формою та переходом між сторінками, відкриття кількох модальних вікон або швидка зміна стану інтерфейсу можуть призводити до конфліктів. У таких умовах важливо передбачити централізоване керування станом, яке унеможливить зберігання неактуального контенту або накладання кількох станів компонента.

Ще однією проблемою є динамічний контент, залежний від взаємодії користувачів: наприклад, відгуки або повідомлення в чаті. У подібних розподілених сценаріях система повинна забезпечити цілісність інформації при оновленні в реальному часі, з урахуванням можливих затримок, обриву з'єднання або некоректного завершення сесії. Необхідно заздалегідь формалізувати очікувану поведінку в кожному з цих випадків та підготувати інтерфейс до безпечного відновлення або сповіщення користувача про помилку.

У проєктах, що передбачають багатомовність, окрема категорія проблем стосується локалізації. Відсутність ключів перекладу, некоректне відображення тексту, зсуви у верстці через надмірно довгі або короткі підписи – усе це негативно впливає на користувацький досвід. У багатьох системах перемикання мови виконується не повністю або призводить до перезавантаження сторінки, що в умовах SPA є неприйнятним. Важливо на етапі моделювання врахувати використання бібліотек із підтримкою динамічного перемикання мов без втрати поточного стану інтерфейсу.

Проблеми з асинхронною логікою також є типовими для подібних систем. Наприклад, подвійний сабміт форм, запити, що не були скасовані після переходу на іншу сторінку, або конфлікти між паралельними запитами до API. Усі ці ситуації мають бути змодельовані на ранньому етапі та враховані при розробці структури компонентів, системи навігації та обробки подій.

Нарешті, варто зазначити, що чимало проблем пов’язані не з окремими компонентами, а з їх взаємодією. Це вимагає наявності у системі таких властивостей, як відмовостійкість, спостережуваність, передбачуваність поведінки при edge-case подіях, що особливо важливо в умовах високої реактивності й великої кількості користувачів, які взаємодіють одночасно.

Таким чином, аналіз предметної області показує, що потенційні проблеми охоплюють перевірку валідності даних, підтримку ролей, безпечну роботу з файлами, локалізацію, узгодженість станів компонентів, обробку асинхронних сценаріїв і забезпечення цілісності при реальній взаємодії між користувачами. З огляду на це, система має бути з самого початку спроєктована з орієнтацією на тестованість, прозору структуру, можливість емуляції граничних сценаріїв і підтримку повного циклу забезпечення якості.

1.3 Постановка задачі

Ураховуючи загальні властивості предметної області, функціональну структуру системи PetBreedingApp, а також типові ризики та проблеми, які виникають при реалізації SPA-додатків із високим рівнем інтерактивності, у цій кваліфікаційній роботі ставиться задача впровадження комплексного підходу до забезпечення якості фронтенд-частини застосунку. Оскільки проект реалізовано командою з розподілом ролей, у межах цієї частини роботи основним об’єктом є саме механізми тестування, валідації та підтримки стійкості користувацького інтерфейсу в умовах реального використання системи.

Система має багатофункціональну структуру, що охоплює такі основні компоненти: модулі авторизації та реєстрації користувачів, редагування профілю, створення та перегляд анкет тварин, фільтрацію за параметрами, систему симпатій, чат між користувачами, модулі керування фото та відгуками, рейтинг, а також інтерфейс локалізації. Уся взаємодія реалізована на основі клієнтського SPA, що створює особливі вимоги до стабільності рендерингу, цілісності стану, коректного відображення повідомлень, обробки помилок API, поведінки при обриві з’єднання та перевірки граничних сценаріїв.

Метою цієї роботи є розробка та реалізація стратегії забезпечення якості, яка охоплює:

* формування тестової моделі компонентів системи;
* виявлення сценаріїв, що підлягають обов’язковому контролю;
* створення автоматизованих unit- і інтеграційних тестів за допомогою Jest і React Testing Library;
* реалізацію e2e-тестування користувацьких сценаріїв за допомогою Playwright;
* перевірку стабільності компонентів у реактивному середовищі;
* забезпечення відповідності інтерфейсу мовному оточенню;
* оцінку рівня покриття критичних функцій тестами;
* побудову карти покриття та дерева рішень для контролю поведінкових відхилень.

Додатково важливим завданням є контроль збереження функціональності при оновленні та розширенні системи, у тому числі перевірка на наявність регресій, дублювання коду, конфліктів у роботі модальних вікон, перехресних змін у Redux-слайсах та некоректного ререндерингу. Реалізовані тести повинні не лише перевіряти конкретні компоненти, а й моделювати взаємодію між ними, поведінку в edge-case ситуаціях, зміну контексту (авторизація, мова, стан з'єднання) та відновлення після помилок.

Таким чином, основна задача цієї роботи полягає в тому, щоб сформувати цілісну й ефективну архітектуру тестування, яка дозволяє:

* гарантувати якість роботи SPA-застосунку при розширенні його функціоналу;
* забезпечити контроль над інтерфейсною поведінкою в умовах складної взаємодії між компонентами;
* автоматизувати процеси перевірки якості;
* створити систему виявлення помилок, яку можна масштабувати.

Результатом виконання поставленого завдання має стати стабільна, контрольована та протестована система з боку користувача (frontend), включно з усіма сценаріями взаємодії із серверною частиною, готова до використання в реальному середовищ.

# **2 Аналіз об’єкта дослідження**

Об’єктом дослідження є інформаційна система PetBreedingApp – односторінковий вебзастосунок для підбору партнерів серед домашніх тварин. Система реалізована у вигляді клієнт-серверної архітектури, де клієнтська частина побудована на основі бібліотеки React [8], а серверна – на Django з підтримкою WebSocket. Уся взаємодія між користувачем і даними відбувається через клієнтський SPA, що працює з API і сокетами в асинхронному режимі.

Користувачі системи мають змогу створити обліковий запис, додати анкету тварини, переглядати анкети інших тварин, залишати симпатії, обмінюватися повідомленнями в реальному часі, залишати відгуки, переглядати профілі власників і додавати або видаляти фото в анкетах. Окрему увагу приділено параметричному фільтруванню анкет (MatchSwiper), підтримці рейтингів, системі локалізації та реактивному відображенню toast-повідомлень, скелетонів, сповіщень про помилки та переходів.

Клієнтська частина реалізована як типова структура React-проєкту з поділом на функціональні модулі: auth, profile, pets, chat, reviews, gallery, matching, localization. Кожен модуль містить компоненти, slice-и Redux Toolkit, стилі та окремі сервіси для взаємодії з API. Взаємодія між модулями відбувається через маршрутизацію (React Router), глобальний стан (Redux), а також локальний контекст (наприклад, для обробки авторизації).

Система має складну модель авторизації. Незареєстровані користувачі мають доступ лише до сторінок входу та реєстрації. Після входу користувач отримує доступ до свого профілю, може створити анкету, взаємодіяти з іншими користувачами, залишати відгуки та вести чат. Авторизація реалізована без JWT – автентифікація підтримується через механізми CSRF-захисту Django з cookie-сесіями. Усі запити до захищених маршрутів перевіряються на наявність відповідних токенів, а при їх втраті користувач отримує помилку 403. Це створює особливі умови для тестування сценаріїв відмови або несанкціонованого доступу.

Компонент MatchSwiper реалізує механіку типу «свайп» – перелік потенційних партнерів формується на основі фільтрів за видом, породою, статтю, віком і типом взаємодії. При кожному «свайпі» інформація змінюється на клієнті та синхронізується з сервером. При взаємній симпатії система автоматично створює чат і відображає модальне вікно з варіантами подальших дій. Така поведінка вимагає наявності стабільного внутрішнього стану, коректної роботи редукс-слайсів і перевірки цілісності при оновленні в реальному часі.

Чат реалізовано через WebSocket-протокол на базі Django Channels. Компонент ChatRoom підключається до відповідного чату, надсилає повідомлення через сокети та відображає отримані повідомлення з live-оновленням. Поведінка залежить від авторизації, права доступу до конкретного чату, а також стійкості до розриву з’єднання. Це створює необхідність у тестуванні WebSocket-сценаріїв, відновлення з’єднання, перевірки відображення нових повідомлень і обробки помилок.

Компонент PhotoGallery забезпечує додавання, перегляд, видалення та перемикання фото, а також взаємодіє з API-запитами для оновлення фотографій на сервері. Важливо, що компонент повинен коректно працювати при швидкому додаванні кількох зображень, обмеженні формату файлів, перевірці ваги, обробці помилок API та коректному відображенні стану (loading, success, error). Для цього використовуються як клієнтські валідації, так і перевірки відповіді API.

Підтримка багатомовності реалізована через i18next. Усі тексти виведені в окремі словники, перемикання мови відбувається динамічно, без перезавантаження сторінки. Кожен компонент підключено до контексту локалізації, що створює потребу перевірки коректної реакції UI на зміну мови в будь-який момент часу – у тому числі в середині заповненої форми або при наявності відкритих модальних вікон.

Важливою особливістю системи є її реактивність – відображення компонентів, повідомлень, блоків і форм прив’язано до змін у стані, відповіді сервера та параметрів користувача. У таких умовах навіть незначні помилки в логіці ререндерингу можуть призвести до непередбачуваної поведінки, що робить QA-опрацювання критично необхідним для стабільної роботи системи.

Таким чином, об'єкт дослідження являє собою масштабований, інтерактивний SPA-застосунок, побудований за сучасними принципами розділення логіки, асинхронної взаємодії, централізованого керування станом і клієнтської відповідальності за поведінку в реальному часі. Уся архітектура налаштована на активну взаємодію з серверною частиною, і тому забезпечення якості вимагає глибокої перевірки не лише UI, а й усіх шляхів обміну даними та граничних сценаріїв поведінки.

# **3 Архітектура та проєктування**

3.1 UML-проєктування ПЗ

UML-діаграми традиційно використовуються в процесі моделювання програмного забезпечення для формального опису логіки, структури, поведінки та взаємодії об’єктів у межах системи. У рамках цієї кваліфікаційної роботи UML-підхід був адаптований для вирішення завдань, пов’язаних не з проєктуванням системи в класичному сенсі, а з побудовою архітектури її тестування. Тобто діаграми не лише відображають загальну структуру, а й слугують інструментом для формування повноцінного покриття QA-сценаріями, оцінки складності компонентів, виявлення критичних шляхів і побудови прогнозованої моделі поведінки інтерфейсу.

Першим кроком стало формування діаграми послідовності для найтиповішого і водночас критичного сценарію – авторизації користувача. В умовах, коли застосунок є повністю клієнтським, із авторизацією на боці сервера через CSRF-сесію, життєвий цикл авторизації стає чутливим до втрати сесії, недійсного токена, збоїв на рівні відповіді API, а також некоректного рендерингу після входу. Діаграма відображає взаємодію між компонентом форми входу, авторизаційним контекстом, маршрутизатором, редуксом і серверним API, показуючи часову прив’язку між подіями, очікуванням відповіді, сабмітом і перенаправленням. З погляду тестування, ця діаграма дозволяє виявити, на яких етапах можуть виникати розриви логіки, в якому місці слід мокати серверну відповідь, як перевірити редірект після успішного входу, і де саме перевірити наявність повідомлення про помилку у разі невдалого логіну. Виконана діаграма розташована на рисунку 3.1:

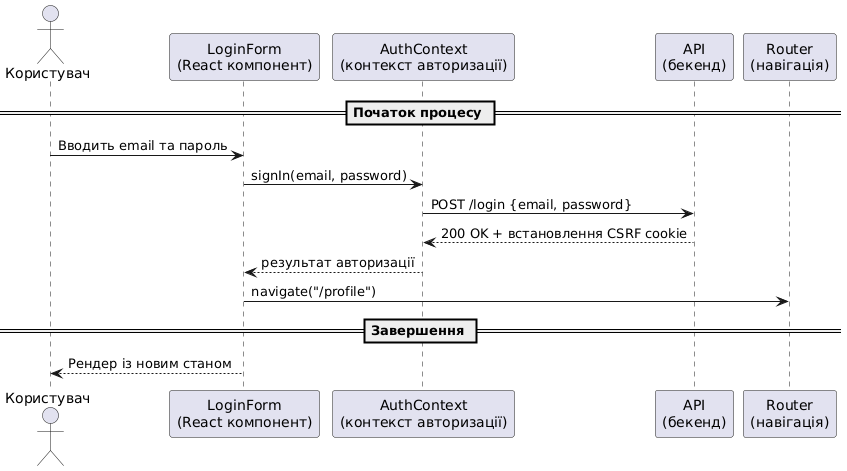


Рисунок 3.1 – Діаграма послідовностей (розроблена самостійно)

Для відображення поведінкової логіки компонентів було сформовано діаграму активностей. Вона стосується одного з найважливіших сценаріїв – створення анкети тварини. Компонент PetEditor, з погляду користувача, має вигляд звичайної форми, але в реальності включає низку внутрішніх перевірок, валідацію полів, обробку локального стану, запити до API і відображення toast-повідомлень. Діаграма дозволила описати повний шлях користувача від відкриття форми до сабміту, з гілками поведінки на випадок помилкових або неповних даних, некоректних файлів фото, недоступності сервера, повторного сабміту тощо. Цей сценарій є особливо показовим для тестування edge-case ситуацій: наприклад, натискання кнопки відправки кілька разів, зникнення з’єднання під час запиту, помилка типу 500 на сервері, або спроба відправки без CSRF. Діаграма активностей дає змогу візуально описати кожен з цих варіантів, а також допомагає автоматизувати їх перевірку через e2e-фреймворк Playwright або модульні тести. Повністю виконана діаграма розташована на рисунку 3.2:

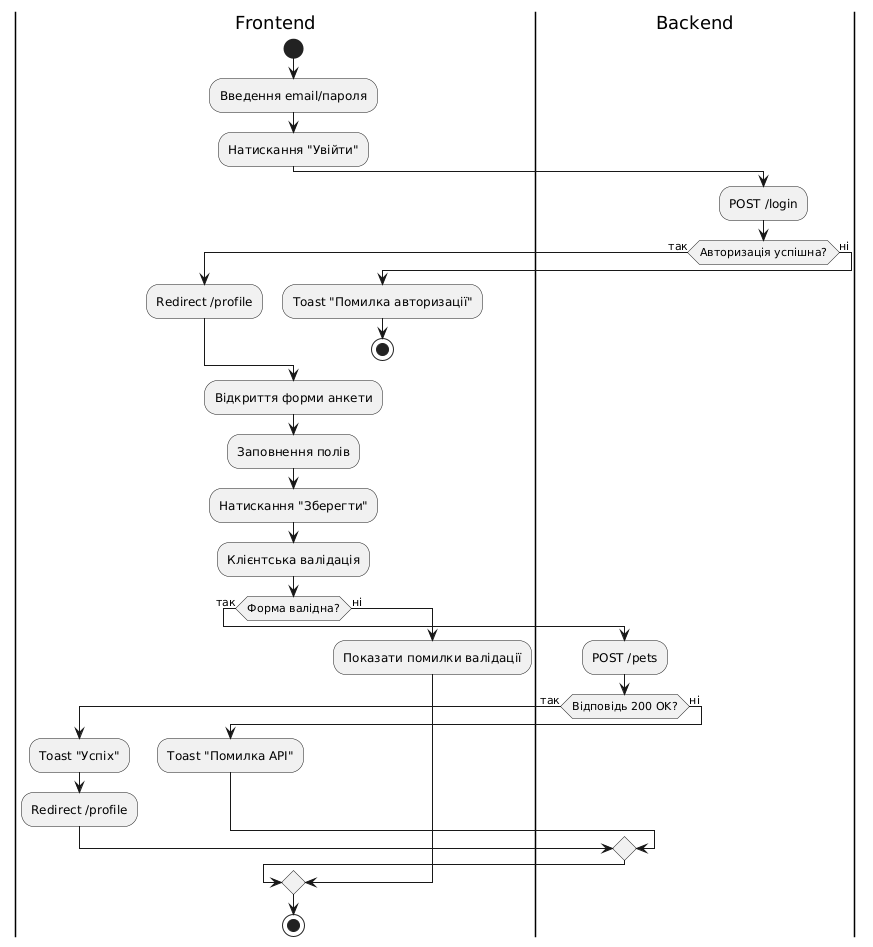


Рисунок 3.2 – Діаграма активностей (розроблена самостійно)

Для моделювання типових негативних сценаріїв було побудовано decision tree, яке ілюструє обробку введення при реєстрації користувача. Реєстрація – це сценарій з високою кількістю гілок: порожні поля, занадто короткий пароль, невірний формат email, наявність уже зареєстрованого email, невалідні символи у логіні тощо. Кожна така гілка є потенційним джерелом помилки або несподіваної поведінки. Decision tree дає змогу сформулювати повну карту всіх шляхів користувача і візуалізувати, які з них завершуються успіхом, а які – блокуються з відповідним повідомленням. Це дозволяє сформувати повний набір негативних e2e-тестів, автоматизованих через сценарії типу: «введено неправильний email → натиснуто кнопку → очікується toast з помилкою». Без такого дерева легко випустити з уваги важливі гілки, які не покриваються поточними тестами. Ця діаграма розташована у додатку А на рисунку А.1.

Карта покриття (coverage map) є ще одним інструментом візуального контролю тестової архітектури. У класичній розробці вона може асоціюватися з покриттям рядків коду, однак у цій роботі мапа побудована як логічна сітка між компонентами системи та типами тестів. По вертикалі розміщено ключові функціональні блоки: авторизація, профіль, анкети, фільтр, симпатії, чат, галерея фото, відгуки, локалізація. По горизонталі – рівні перевірки: unit-тести, інтеграційні тести, e2e-тести, ручне UX-тестування, accessibility, перевірка локалізації. Це дозволяє зафіксувати не лише наявність або відсутність тестів, а й розподіл відповідальності, виявити компоненти з високою складністю, але недостатнім покриттям, а також компоненти, що протестовані тільки вручну. Такий підхід дає змогу не тільки оцінити поточний стан QA, але й спрогнозувати план його розвитку. Ця діаграма розташована на рисунку 3.3:

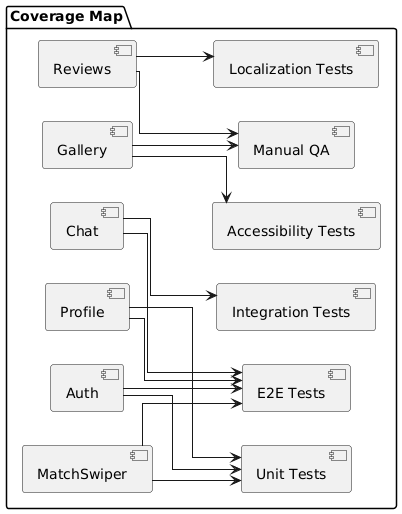


Рисунок 3.3 – Карта покриття (розроблена самостійно)

Усі зазначені діаграми не лише виконують функцію документування, а й безпосередньо впливають на реалізацію стратегії тестування. Вони дають змогу зменшити кількість неочікуваних гілок, формалізувати поведінку компонентів, зафіксувати залежності від API, і наочно представити критичні місця, які вимагають посиленого контролю. Без них тестування було б інтуїтивним, безсистемним і не масштабованим. У межах цієї роботи UML-підхід було адаптовано саме до потреб забезпечення якості, а не класичної розробки – що демонструє універсальність і ефективність такого підходу для SPA-застосунків із розширеною логікою клієнтської взаємодії.

Архітектура системи забезпечення якості в межах цього проєкту охоплює побудову цілісної структури тестування клієнтської частини PetBreedingApp на різних рівнях контролю. З огляду на складну взаємодію між користувачем, асинхронними API-запитами, маршрутизацією, внутрішнім станом Redux і WebSocket-підключеннями, до системи тестування було висунуто підвищені вимоги щодо охоплення сценаріїв, точності поведінкових перевірок, незалежності від бекенду та масштабованості.

3.2 Архітектура ПЗ

Уся система тестування побудована навколо концепції тестової піраміди. На її основі наймасовішими є unit-тести, які перевіряють поведінку окремих компонентів, slice-ів, утиліт і валідаційних функцій. Для цього використано бібліотеку Jest як базовий фреймворк для виконання тестів, у поєднанні з React Testing Library (RTL) для рендерингу компонентів, взаємодії з DOM і перевірки виводу. RTL дозволяє перевіряти поведінку інтерфейсу з позиції користувача: наприклад, чи з’явилось повідомлення про помилку, чи відображається відповідне поле, чи ререндериться компонент після зміни значення. Це дозволяє уникнути надмірної прив’язки до внутрішньої реалізації та підвищує стійкість тестів до рефакторингу.

Інтеграційні тести займають наступний рівень – вони перевіряють взаємодію між компонентами, наприклад, зв’язок між формою і Redux, коректне оновлення локального та глобального стану після події, або виведення toast-повідомлення після завершення асинхронної дії. Тут використовується мокінг API-викликів із заміною fetch або axios на підставні функції, що повертають контрольовану відповідь. У такому режимі можна симулювати як успішні сценарії, так і помилки: від time-out до 403 Forbidden. Структура тестів розміщена всередині кожного функціонального модуля (наприклад, features/pets/\_\_tests\_\_/PetEditor.test.jsx), що дозволяє підтримувати локалізовану відповідальність і зменшує зв'язність тестової бази.

Найвищий рівень перевірки представлений e2e-тестами, реалізованими у Playwright. Ці тести імітують повноцінну поведінку користувача у браузері: введення даних, переходи між сторінками, взаємодію з інтерфейсом, відкриття модальних вікон, «свайпи», натискання, очікування на відповіді API, симпатії та чати. Завдяки Playwright вдається тестувати навіть складні реактивні сценарії, наприклад, зміна інтерфейсу після отримання відповіді через WebSocket, відновлення після втрати сесії, динамічне перемикання мови без перезавантаження сторінки або повторний сабміт форми після помилки.

Архітектурно вся система тестування організована за принципом модульності. Кожен функціональний блок проєкту (auth, profile, pets, chat, reviews тощо) містить власну структуру тестів, відповідні моки, фікстури, утиліти та сценарії. Для Playwright винесено окрему структуру тестів (/e2e/) з розділенням за сценаріями (login, registration, matching, chat). Усі тести мають централізовану конфігурацію, підтримують паралельний запуск, ізоляцію стану, автоматичну підготовку до виконання (наприклад, логін користувача через API перед початком тесту), а також cleanup після завершення.

Особливу увагу приділено стабільності тестів. Оскільки в SPA навіть невелика зміна в ререндерингу або DOM може спричинити помилку тесту, у проекті реалізовано політику tolerant match – перевірка не наявності точного вузла, а правильного тексту або очікуваного патерну. Наприклад, замість пошуку за className або ID тести перевіряють getByRole, getByText, getByLabelText, що відповідає рекомендаціям RTL і підвищує стійкість до стилістичних змін. Також реалізовано обгортки над компонентами для тестування у відповідному контексті – наприклад, withProvider() або withRouter() для тестування компонентів, що залежать від стану або навігації.

Усі тести інтегровані в CI-процес – при кожному пуші в репозиторій запускається автоматичне тестування, результати якого зберігаються в артефактах збірки. Це дозволяє одразу виявляти регресії, контролювати загальний стан системи, а також отримувати статистику покриття. За потреби використовуються локальні запускальники з мок-сервером, що дозволяє перевірити поведінку в ізольованому середовищі.

Враховуючи обсяг системи та кількість сценаріїв, архітектура тестування побудована таким чином, щоб її можна було масштабувати. Додавання нового компонента автоматично включає створення тестів відповідного рівня – unit, інтеграційних і, за необхідності, e2e. Уся система тестування розглядається не як допоміжний модуль, а як невід’ємна частина архітектури PetBreedingApp, що підтримує її стабільність, передбачуваність і контрольованість на всіх етапах використання.

3.3 Проєктування структури зберігання даних

У контексті забезпечення якості проєктування структури зберігання даних стосується не лише бази даних застосунку, а й способу організації тестових сценаріїв, мок-даних, фікстур, контрольних відповідей сервера та шаблонів взаємодії. Від правильності структури зберігання цих ресурсів залежить стабільність і масштабованість тестової бази, можливість багаторазового використання перевірених даних у різних контекстах, а також зменшення залежності тестів від змін у продуктивному середовищі.

У рамках цієї роботи зберігання тестових даних реалізоване як багаторівнева система. На нижчому рівні знаходяться фікстури – статичні об'єкти, які описують базовий стан системи перед запуском тесту. Це можуть бути шаблони анкет, мок-версії користувачів, заповнені профілі, приклади повідомлень у чаті або масиви фотографій. Ці фікстури розміщуються у відповідних папках \_\_mocks\_\_ або \_\_fixtures\_\_ всередині модулів і використовуються як у unit-, так і в інтеграційних тестах. Завдяки цьому забезпечується локалізований контекст кожного тесту: компонент PetEditor використовує власну анкету, ChatRoom – власний список повідомлень тощо.

На наступному рівні відбувається мокування API. У всіх тестах, що залежать від відповіді сервера, замість реального запиту використовується симуляція виклику з контрольованою відповіддю. Для цього застосовується бібліотека, що дозволяє перехоплювати виклики fetch або axios і повертати заздалегідь визначену відповідь. Наприклад, у тестах для авторизації можна повертати як позитивну відповідь зі статусом 200, так і негативну з кодом 403, змінюючи тіло відповіді залежно від сценарію. Усі мок-дані мають окремі константи, що дозволяє зберігати стабільність і узгодженість тестів. Для критичних API реалізовано набори варіантів: успішна відповідь, помилка валідації, помилка сесії, внутрішня помилка сервера.

У рамках e2e-тестування використовується окрема модель даних. Замість моків у тестах Playwright проводиться взаємодія з реальним інтерфейсом, тому зберігаються сценарії взаємодії, попередньо підготовлені дані для форми, очікувані повідомлення та значення, які перевіряються на етапі assert-ів. Наприклад, для тесту реєстрації з використанням невалідного email заздалегідь зберігається значення помилкового формату, очікуване повідомлення про помилку та маршрут редиректу після сабміту. Усі ці значення винесені до окремих конфігураційних об’єктів і дозволяють легко змінити тестові дані без правки логіки тесту.

Особливої уваги потребує зберігання тестових знімків стану інтерфейсу, зокрема для компонентів, що мають складну реактивну поведінку. Наприклад, компонент PhotoGallery, який змінює вміст залежно від кількості фото, використовує знімки DOM для перевірки ререндерингу після кожної дії. Такі снапшоти зберігаються як окремі шаблони в тестовій структурі, що дозволяє порівнювати фактичний стан з очікуваним без втручання у візуальну частину.

Ще одним типом даних є шаблони локалізації. Для перевірки мультимовності всі компоненти протестовано на коректність перемикання мови, відображення перекладу та стабільність при зміні мовного контексту. Для цього створено окремі словники локалізації, які зберігаються у форматі JSON і завантажуються перед запуском тесту. Таким чином, кожен тест виконується з точно визначеною мовною конфігурацією, що дозволяє перевірити, чи немає пропущених ключів або зсуву верстки.

Окрема категорія збережених даних – тестові сценарії. Це не просто функції або скрипти, а формалізовані послідовності дій, які виконуються в рамках тесту. Наприклад, сценарій входу користувача включає введення логіна, пароля, натискання кнопки та очікування перенаправлення. Цей сценарій зберігається як окрема функція, яка може викликатися в різних тестах – як для smoke-тестів, так і для перевірки логіки після авторизації. Такий підхід забезпечує багаторазове використання шаблонів поведінки та спрощує розширення тестової бази.

Уся система зберігання тестових даних організована відповідно до структури проєкту. Кожен модуль має власну папку тестів, моків, фікстур і утиліт, що дозволяє мінімізувати залежність між компонентами та уникнути дублювання. Це підвищує зрозумілість тестів, пришвидшує локалізацію помилок і спрощує оновлення логіки. У випадку масштабування або зміни API структура дозволяє швидко замінити відповідні фікстури без перезапису всієї тестової бази.

Таким чином, проєктування структури зберігання даних у цій роботі полягає у формуванні чітко визначеної, багаторівневої та модульної системи тестових фікстур, шаблонів, мок-даних і конфігураційних сценаріїв, яка підтримує стабільність тестів, знижує вартість їх обслуговування та забезпечує повноцінне тестове покриття системи.

3.4 Приклади найцікавіших алгоритмів та методів

У межах забезпечення якості клієнтської частини PetBreedingApp було реалізовано низку перевірок, які виходять за рамки типових unit- або інтеграційних тестів. Йдеться про сценарії, які вимагають моделювання складної поведінки, симуляції асинхронної взаємодії з сервером, обробки граничних випадків або динамічних змін стану інтерфейсу. Усі приклади наведено з акцентом на практичну реалізацію тестування в умовах високої взаємозалежності компонентів, що є характерною ознакою SPA-додатків.

Одним із найбільш показових сценаріїв є перевірка поведінки компонента MatchSwiper. Цей компонент реалізує механіку покрокового відбору анкет тварин, подібну до взаємодії у форматі «свайпів»: користувач натискає кнопку «подобається» або «не подобається», після чого система оновлює внутрішній стан, надсилає запит до API та, у випадку взаємної симпатії, створює новий чат. В одному з тестів симулюється ланцюгова поведінка: користувач натискає «подобається», сервер повертає інформацію про взаємну симпатію, і компонент виводить модальне вікно з варіантами дій. У рамках цього тесту перевіряється не лише поява вікна, а й коректність навігації після вибору користувачем відповідної опції. Такий сценарій охоплює кілька шарів логіки – Redux, API, повторне рендерення, модальне вікно та роутинг – і демонструє комплексність тестового покриття.

Ще один цікавий приклад – перевірка обробки втрати авторизації. Система використовує CSRF-захист із cookie-сесіями, і якщо сесія стає недійсною (наприклад, після тайм-ауту), сервер повертає 403 Forbidden. У спеціальному тесті симулюється ситуація, коли користувач відкриває компонент профілю або редагування анкети, надсилає запит, але отримує відмову. Очікуваною поведінкою є автоматичне очищення локального стану, виведення toast-повідомлення та перенаправлення на сторінку логіну. Перевірка такої логіки дозволяє гарантувати стабільність у реальних умовах втрати сесії та уникнути “битих” станів інтерфейсу.

Особливу увагу було приділено WebSocket-логіці в чаті. Компонент ChatRoom має пряме підключення до сокета, отримує повідомлення в реальному часі та оновлює список повідомлень без перезавантаження сторінки. У тесті, побудованому за допомогою мок-серверу, симулюється надходження нового повідомлення, перевіряється, що воно з'являється в DOM без додаткових запитів, і що користувач бачить оновлення навіть без активної взаємодії. Також у цьому ж тесті імітується розрив з’єднання та перевіряється поведінка системи – відображення повідомлення про втрату зв'язку, спроба reconnect та відновлення історії.

Ще одним критичним кейсом є взаємодія з галереєю фото. У компоненті PhotoGallery реалізовано логіку попереднього перегляду, додавання та видалення фотографій, а також перевірки формату файлів, розміру та кількості. У тестах мокуються API-виклики, але також перевіряється локальна поведінка: завантаження файлу, оновлення прев’ю, відображення кнопки “Видалити”, та видалення з локального стану ще до відповіді сервера. Додатково тестується реакція інтерфейсу при завантаженні двох однакових фото, при перевищенні ліміту, при спробі завантажити файл невірного формату. Це дозволяє перевірити, що UI не виходить з очікуваного стану навіть у некоректних умовах.

Цікавою особливістю є перевірка мультимовності системи в активному стані. Оскільки застосунок підтримує перемикання мови без перезавантаження сторінки, в одному з тестів перевіряється, що при зміні мови в середині заповненої форми відображення змінюється відповідно до словника, значення введених полів не губляться, а toast-повідомлення після сабміту виводяться вже в новій мові. Це важливо в контексті UX і демонструє увагу до поведінки, яку складно виявити без відповідних перевірок.

Окремі тести стосуються edge-case сценаріїв: натискання кнопки кілька разів поспіль, відправка незаповненої форми, зміна полів під час сабміту, завантаження фото в момент втрати з’єднання, конфліктуючі стани між Redux та локальним useState. Усі ці сценарії не є обов’язковими з погляду базового покриття, але саме вони показують, наскільки стійкою є система до несподіваної поведінки користувача або змін середовища.

Таким чином, реалізовані приклади охоплюють як стандартні перевірки, так і складні моделі поведінки. Вони ілюструють, що тестування в SPA-середовищі – це не лише перевірка окремих компонентів, а й моделювання інтегрованої логіки, взаємодії станів, асинхронності та реактивності. Такі методи не просто знаходять помилки, а гарантують контрольованість усієї системи під час реального використання.

3.5 Створення UI/UX або іншого дизайну системи

У межах забезпечення якості PetBreedingApp одним із ключових аспектів є тестування поведінки інтерфейсу з точки зору користувача, тобто контроль за відповідністю UI/UX-реалізації очікуваній логіці взаємодії. Оскільки проєкт реалізований як SPA з великою кількістю компонентів, які динамічно змінюють стан, відображають toast-повідомлення, реагують на асинхронні події або ререндеряться у відповідь на дії користувача, перевірка UX стала невід’ємною частиною загальної стратегії тестування.

Одним із напрямів перевірки є контроль поведінки інтерфейсу в умовах завантаження. Багато компонентів – зокрема MatchSwiper, PetProfile, GalleryModal – мають skeleton-стани або індикатори очікування. У тестах перевіряється наявність цих індикаторів у момент переходу на сторінку, а також їх зникнення після завантаження даних. Це дозволяє переконатись, що користувач не бачить порожнього або зламаного інтерфейсу, навіть якщо відповідь API затримується. Крім того, для складних компонентів перевіряється, що в skeleton-режимі не доступні інтерфейсні дії – кнопки або поля, які повинні бути недоступними до завершення завантаження.

Велика увага приділена toast-повідомленням – як інструменту зворотного зв'язку. У кожному модулі, що має сабміт або запит до сервера, перевіряється: чи з’являється повідомлення, чи має воно коректний текст (відповідно до поточної мови), чи зникає автоматично або за дією користувача, чи не блокує важливі елементи інтерфейсу. У тестах симулюються як успішні сценарії (профіль оновлено, відгук залишено), так і негативні (помилка авторизації, відсутність з’єднання). Це дозволяє гарантувати, що інтерфейс інформує користувача про результат дії в усіх можливих випадках.

Окремо тестується стан елементів керування – кнопок, полів введення, модальних вікон – у різних UX-ситуаціях. Наприклад, у компоненті PetEditor після сабміту форма повинна блокуватись до завершення відповіді від API, і тільки після підтвердження знову ставати активною. У тестах перевіряється, що користувач не може натиснути «Зберегти» двічі поспіль, що після помилки валідації фокус автоматично переноситься на перше проблемне поле, що кнопки «Скасувати» та «Очистити» працюють навіть під час активного сабміту. Такий рівень перевірки дозволяє імітувати реальну поведінку користувача і виявити неочевидні UX-збої ще до релізу.

Також реалізовано часткову перевірку доступності інтерфейсу. За допомогою RTL та Playwright тестується навігація клавіатурою, наявність доступних міток (aria-label, alt), фокусування полів при завантаженні, збереження фокусу при зміні модальних вікон. Особливо важливо це для компонентів GalleryModal, ChatRoom і ReviewForm, де велика кількість взаємодій може спричинити втрату фокусу або збої в навігації. Також перевіряється, що після появи помилки або toast-повідомлення екран не «стрибає», а фокусовані елементи залишаються доступними.

Оскільки інтерфейс підтримує динамічне перемикання мови, кожен компонент також протестовано на поведінку в момент зміни локалізації. Це включає перевірку оновлення текстів, збереження поточних даних (наприклад, незаповненої форми), коректного відображення повідомлень у новій мові, а також стабільності стилів при зміні довжини текстів. Такий підхід дозволяє гарантувати, що користувач отримає послідовний досвід незалежно від вибору мови або стану системи.

Загалом реалізоване тестування UI/UX не є формальністю, а слугує ключовим механізмом підтримки користувацької довіри до системи. Завдяки перевірці інтерактивності, адаптивності, послідовності, передбачуваності та зворотного зв’язку забезпечується не лише відсутність помилок, а й відчуття стабільної, логічної та завершеної взаємодії. Це дозволяє оцінювати PetBreedingApp не лише як технічно правильний застосунок, а як продукт, орієнтований на повноцінну підтримку якісного досвіду користувача.

# **4 Тестування програмної системи**

4.1 Підхід до тестування SPA-застосунку

Забезпечення якості програмного забезпечення в умовах односторінкових застосунків (SPA) потребує особливого підходу, відмінного від традиційного тестування статичних або серверно-рендерених інтерфейсів. Особливості SPA – це постійна взаємодія користувача з динамічним інтерфейсом без перезавантаження сторінок, активне використання локального стану, асинхронних запитів до API, а також умовне відображення елементів залежно від ролі користувача, наявності даних чи мовного контексту.

У межах кваліфікаційної роботи тестуванню підлягає клієнтська частина вебсистеми PetBreedingApp, яка реалізована на React із використанням Redux Toolkit для централізованого управління станом [9], i18next для багатомовності, Tailwind CSS для адаптивного інтерфейсу [10] та WebSocket-інфраструктури для чатів у реальному часі. Основна мета тестування – перевірка цілісності поведінки кожного компоненту, його здатності до коректної взаємодії з іншими частинами системи та стійкість до граничних сценаріїв.

Підхід до тестування включає багаторівневу стратегію:

* модульне тестування (unit tests) – дозволяє ізольовано перевірити логіку окремих компонентів React, зокрема рендеринг за певних props, виклики подій, відображення валідаційних повідомлень, тощо;
* інтеграційне тестування – охоплює взаємодію між компонентами, роботу з Redux-слайсами, передачу стану через контексти, синхронізацію між формами, модальними вікнами, маршрутами;
* E2E (end-to-end) тестування – реалізується через фреймворк Playwright і дозволяє змоделювати повноцінні сценарії користувача в headless-браузері: авторизація, створення анкети, встановлення симпатії, відкриття чату, перегляд фото, перемикання мови, реакція на помилки API.

Особливу увагу приділено тестуванню реактивного рендерингу в умовах зміни стану, а також поведінці застосунку у випадках нестабільної мережі (штучне створення ситуацій втрати з’єднання, помилок 401/403/500, тайм-аутів).

Кожен компонент проходить перевірку в кількох режимах:

* у стані авторизованого та неавторизованого користувача;
* у контексті різних мовних налаштувань (ua/en);
* із порожніми даними (порожні списки, відсутність фото, неіснуючі анкети);
* під час взаємодії з WebSocket, включно з його розривом або reconnection.

Під час побудови тестової стратегії важливо було також врахувати особливості SPA-навігації без повного перезавантаження сторінки. Зокрема, тести мають перевіряти поведінку при переходах між маршрутизованими розділами (/profile, /pets/:id, /match, /chats/:id) у різних умовах.

Ще одним важливим аспектом є підтримка багатомовності. Тестування включає перевірку того, чи змінюється мовний контент без втрати стану компонента, чи немає зсувів у верстці після зміни мови, і чи відображаються fallback-тексти у випадку відсутніх ключів у словнику.

Для ізоляції зовнішніх залежностей, таких як база даних або сторонній бекенд, використано набір мок-серверів і фіктивних API-відповідей [11]. Це дозволило:

* стабільно запускати тести незалежно від поточного стану сервера;
* відтворювати помилки з боку API або затримки мережі;
* перевіряти поведінку у випадках, які складно змоделювати на продакшн-сервері (наприклад, повна відсутність відповіді).

Усі типи тестів інтегровані в CI/CD пайплайн із запуском у headless-режимі під час кожного коміту в основну гілку. Це дозволяє швидко виявляти регресії, а також зберігати стабільність поведінки застосунку протягом усього життєвого циклу розробки.

Таким чином, застосований підхід забезпечує повноцінне покриття основних сценаріїв взаємодії, мінімізує ймовірність прихованих помилок у продакшн-версії системи та дозволяє підтримувати якість на високому рівні навіть у випадку активного розвитку проєкту.

4.2 Unit- і інтеграційне тестування з Jest та RTL

Значну частину системи забезпечення якості у проєкті становлять модульні та інтеграційні тести, реалізовані за допомогою Jest [12] у поєднанні з React Testing Library. Цей інструментарій дозволяє тестувати поведінку окремих компонентів у максимально наближених до реальності умовах, з урахуванням динамічного рендерингу, подій, валідації та змін стану застосунку.

У рамках тестування було перевірено критично важливі компоненти, зокрема форми логіну й реєстрації, редактор анкет тварин, Tinder-подібний MatchSwiper, а також фото-галерею. Для компонента LoginForm основна увага зосереджувалась на валідації полів та відображенні повідомлень про помилки. Наприклад, тест перевіряє, що у випадку порожнього поля пароля система виводить відповідне попередження:

**test("форма логіну не сабмітиться без пароля", () => {**

**render(<LoginForm />);**

**fireEvent.change(screen.getByLabelText(/Email/i), {**

**target: { value: "user@example.com" }**

**});**

**fireEvent.click(screen.getByRole("button"));**

**expect(screen.getByText(/Пароль обов’язковий/i)).toBeInTheDocument();**

**});**

У RegisterForm реалізовано перевірки на збіг паролів, правильний формат email та блокування кнопки до моменту заповнення всіх полів. Подібні тести гарантують передбачувану поведінку інтерфейсу навіть при неправильному введенні даних або симуляції повільного API.

Форма створення анкети тварини (PetEditor) перевіряється як у режимі створення, так і в режимі редагування. Особливу увагу приділено обробці edge-сценаріїв: подвійний сабміт, відсутність породи, некоректний вік або помилки API. Для перевірки інтеграції з Redux реалізовано тести, які контролюють зміну глобального стану після взаємодії користувача:

**test("створення нової анкети змінює глобальний стан", async () => {**

**renderWithReduxAndRouter(<PetEditor />);**

**fireEvent.change(screen.getByLabelText(/Ім’я тварини/i), {**

**target: { value: "Барсік" }**

**});**

**fireEvent.submit(screen.getByRole("form"));**

**await waitFor(() =>**

**expect(store.getState().pets.list).toContainEqual(**

**expect.objectContaining({ name: "Барсік" })**

**)**

**);**

**});**

У компоненті MatchSwiper перевіряється поведінка при зміні картки, реакція на «свайпи», генерація взаємної симпатії та виклик модального вікна. Оскільки логіка компонента включає взаємодію з WebSocket та обробку toast-сповіщень, важливо протестувати не лише зміну візуального стану, але й логіку переходу до чату в разі збігу.

Компоненти для роботи з фотографіями (PhotoGallery та PhotoViewer) проходять тестування на предмет завантаження, перегляду, видалення зображень, реакції на помилки типу MIME, адаптації до ширини екрану та ререндерингу після змін. У тестах моделюються сценарії з неприпустимим розширенням файлу, а також перевіряється, що після видалення елементи галереї оновлюються без помилок.

Для забезпечення стабільності при зміні контексту застосунку, наприклад мови, усі тести проводяться в умовах обгортки з i18next і мок-словниками. Це дозволяє контролювати, що текстові елементи змінюються відповідно до вибраної локалі, не викликаючи помилок або візуальних зсувів.

Взаємодія з Redux і маршрутизатором також тестується через спеціальні обгортки, що дозволяє проводити перевірку навігації, оновлення глобального стану, реакції компонентів на зміни URL. Такі тести охоплюють ситуації, коли користувач після реєстрації автоматично перекидається до сторінки редагування профілю або коли відкриття певного компонента блокується для неавторизованих користувачів.

Загалом, реалізоване unit- і інтеграційне тестування охоплює не лише логіку рендерингу, а й динаміку станів, обробку помилок, зміну контексту та взаємодію компонентів у повному циклі. Це дозволяє гарантувати цілісність системи навіть при значних змінах у кодовій базі.

4.3 Поведінкове тестування та підсумкові результати

Окрім модульних та інтеграційних тестів, у межах забезпечення якості клієнтської частини було реалізовано поведінкове тестування у вигляді повноцінних end-to-end сценаріїв. Цей підхід дозволяє симулювати поведінку реального користувача у браузері, охоплюючи всі етапи взаємодії – від логіну до перегляду анкети чи початку чату. Для реалізації e2e-тестування було використано фреймворк Playwright, який забезпечує автоматичну перевірку UI шляхом взаємодії з DOM у реальному headless-режимі.

Сценарії охоплюють ключові життєві цикли взаємодії: авторизація, створення нової анкети, встановлення симпатії, відкриття чату, перемикання мови. В кожному випадку перевіряється не лише видимість очікуваних компонентів, але й правильна послідовність переходів, обробка подій, завантаження даних із сервера та відсутність помилок у логах.

Одним із прикладів є тест перевірки механізму симпатій, при якому імітується натискання кнопки вподобання на чужій анкеті. Якщо в системі зафіксована взаємна симпатія, з’являється модальне вікно із повідомленням:

**import { test, expect } from "@playwright/test";**

**test("взаємна симпатія відкриває модальне вікно", async ({ page }) => {**

**await page.goto("/match/123");**

**await page.click("button:has-text('Підходить')");**

**await expect(page.locator(".modal")).toBeVisible();**

**await expect(page.locator(".modal")).toContainText("Взаємна симпатія!");**

**});**

Інший тест перевіряє відкриття чату після кліку на кнопку «Написати повідомлення», при умові наявності активного чату між користувачами. Усі e2e-тести моделюють умови роботи як авторизованого, так і неавторизованого користувача, з урахуванням доступності маршрутів, перенаправлень та реакції на помилки доступу.

Для підтвердження якості реалізованого функціоналу було проведено серію запусків e2e-набору у середовищі без підключення до реального бекенду, з використанням мок-серверів. Це дозволило повторювано симулювати необхідні API-відповіді та досягти стабільних результатів. Покриття основних функціональних сценаріїв e2e-тестами наведено у таблиці 4.1:

Таблиця 4.1 – Результати поведінкового тестування основних сценаріїв (таблиця виконана самостійно)

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Функціональність** | **Тестовий сценарій** | **Статус** |
| 1 | Авторизація користувача | Логін через форму | Пройдено |
| 2 | Створення нової анкети тварини | Заповнення форми, сабміт | Пройдено |
| 3 | Встановлення симпатії | «Свайп» на MatchSwiper | Пройдено |
| 4 | Взаємна симпатія | Відкриття модального вікна | Пройдено |
| 5 | Відкриття чату | Перехід у чат з анкети | Пройдено |
| 6 | Перемикання мови | Зміна мови, оновлення інтерфейсу | Пройдено |
| 7 | Вихід із системи | Клік по «Вийти», очищення сесії | Пройдено |

Усі поведінкові сценарії були успішно виконані без збоїв у різних режимах запуску, що підтверджує стабільність роботи клієнтської частини за умов, наближених до реальної експлуатації. Перевірка ключових взаємодій користувача, зокрема сценаріїв симпатій, переходів між розділами, авторизації та роботи з чатом, дозволила виявити та усунути потенційні помилки, які могли б проявитися лише в реальному середовищі.

Таким чином, поведінкове тестування доповнило загальну стратегію контролю якості, дозволивши перевірити систему на найвищому рівні – у контексті повноцінних користувацьких сценаріїв. У поєднанні з модульним та інтеграційним тестуванням це забезпечує комплексне покриття функціоналу й підвищує впевненість у надійності застосунку.

**Висновки**

У межах виконання кваліфікаційної роботи було проведено комплексне забезпечення якості клієнтської частини вебзастосунку PetBreedingApp, який призначений для параметричного підбору партнерів серед домашніх тварин з метою їх розведення. Основним завданням стало впровадження повноцінної системи тестування функціоналу односторінкового React-застосунку, орієнтованого на взаємодію користувачів через динамічні анкети, систему симпатій та обмін повідомленнями.

Зважаючи на складність клієнтської логіки, динамічну структуру DOM, активне використання глобального стану (через Redux), багатомовність та інтеграцію WebSocket, було обґрунтовано застосування багаторівневого підходу до тестування. Це дозволило забезпечити перевірку як ізольованої поведінки компонентів, так і повноцінних користувацьких сценаріїв у середовищі, максимально наближеному до реального використання.

У ході роботи виконано аналіз вимог до системи та сформовано набір сценаріїв, що охоплюють критичні точки взаємодії з інтерфейсом: авторизацію, керування анкетами тварин, встановлення симпатій, створення чатів, перемикання мови та відображення повідомлень. Для кожного з обраних напрямів було реалізовано відповідну стратегію тестування.

На рівні модульного тестування (unit tests) було перевірено логіку окремих React-компонентів, включно з валідацією форм, викликами подій та реакцією на зміну props. Компоненти LoginForm, RegisterForm, PetForm, та інші були протестовані на коректність рендерингу, поведінку при помилках та дотримання очікуваної логіки відображення.

У рамках інтеграційного тестування було перевірено зв’язки між компонентами, обробку подій, взаємодію з Redux-станом, а також реакцію на симульовані API-виклики. Значна увага приділялася формам та механізму MatchSwiper, який має як внутрішній стан, так і зовнішні залежності – від маршруту, глобального стану, API-відповідей та взаємодії з іншими користувачами.

На рівні поведінкового тестування (end-to-end) реалізовано симуляцію реальної взаємодії користувача з інтерфейсом: перехід на сторінки, клік по елементах, очікування відповіді системи, підтвердження зміни стану. Сценарії охоплюють повний життєвий цикл дій: від реєстрації до створення анкети, встановлення симпатії, відкриття чату, виходу із системи. Для реалізації таких перевірок використано Playwright – сучасний інструмент для тестування у браузері, що дозволяє проводити перевірку як у headless-режимі, так і у візуальному середовищі.

Усі критичні функції було протестовано у стабільному середовищі із використанням мок-серверів, що дало змогу відтворити різні варіанти відповіді системи, включаючи обробку помилок, граничних умов та відсутність мережі. Це дозволило виявити та усунути дефекти, які не були б помітні при статичному тестуванні.

На основі результатів перевірок можна зробити висновок про високу стабільність клієнтської частини застосунку в умовах реального використання. Автоматизоване тестування, реалізоване в межах даної роботи, охоплює ключові компоненти, критичні точки входу, логіку взаємодії та можливі помилки, які можуть виникнути в процесі користування.

Таким чином, мету роботи досягнуто: побудовано ефективну систему контролю якості для клієнтської частини багатокомпонентного вебзастосунку, що дозволяє своєчасно виявляти помилки, гарантувати цілісність користувацьких сценаріїв та підвищити надійність усієї системи в цілому. Застосований підхід до тестування є масштабованим, придатним до повторного використання та сумісним з CI/CD-інтеграцією, що забезпечує ефективність підтримки проєкту в довгостроковій перспективі.

**Перелік джерел посилання**

1. MongoDB. MongoDB Documentation. [Електронний ресурс]: <https://www.mongodb.com/docs/> (дата звернення: 05.06.2025).
2. React Testing Library. [Електронний ресурс]: <https://testing-library.com/docs/react-testing-library/intro/> (дата звернення: 05.06.2025).
3. Playwright. Playwright Documentation. [Електронний ресурс]: <https://playwright.dev/> (дата звернення: 05.06.2025).
4. GitHub Actions. GitHub Docs. [Електронний ресурс]: <https://docs.github.com/en/actions> (дата звернення: 05.06..2025).
5. React Router. React Router Documentation. [Електронний ресурс]: <https://reactrouter.com/en/main/start/overview> (дата звернення: 05.06.2025).
6. i18next. [Електронний ресурс]: <https://www.i18next.com/> (дата звернення: 05.06.2025).
7. WebSocket. MDN Web Docs. [Електронний ресурс]: <https://developer.mozilla.org/en-US/docs/Web/API/WebSocket> (дата звернення: 05.06.2025).
8. React. React Documentation [Електронний ресурс]: <https://react.dev/> (дата звернення: 05.06.2025).
9. Redux Toolkit. Redux Documentation. [Електронний ресурс]: <https://redux-toolkit.js.org/> (дата звернення: 05.06.2025).
10. Tailwind CSS. Tailwind CSS Documentation. [Електронний ресурс]: <https://tailwindcss.com/docs> (дата звернення: 05.06.2025).
11. Mock Service Worker (MSW). [Електронний ресурс]: <https://mswjs.io/docs/> (дата звернення: 05.06.2025).
12. Jest. Jest: Delightful JavaScript Testing. [Електронний ресурс]: <https://jestjs.io/> (дата звернення: 05.06.2025).
13. GitHub-репозиторій з кодом проєкту. [Електронний ресурс]: <https://github.com/physm413/PI_PZPI-21-11_OMELCHENKO_V_V> (дата звернення: 05.06.2025).

**Додаток А**

Decision tree діаграма реєстрації

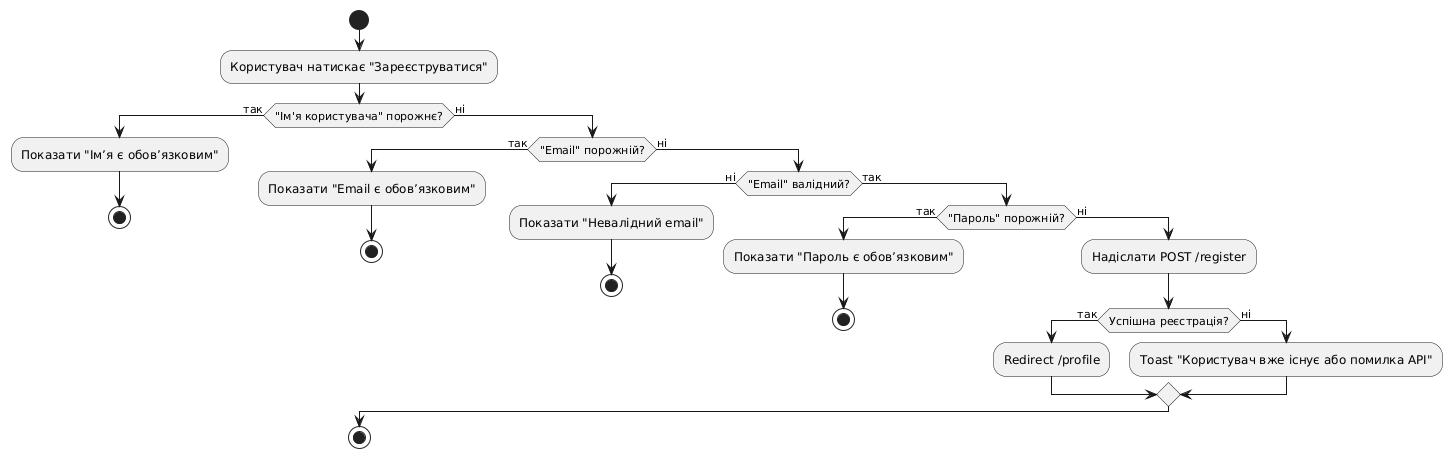
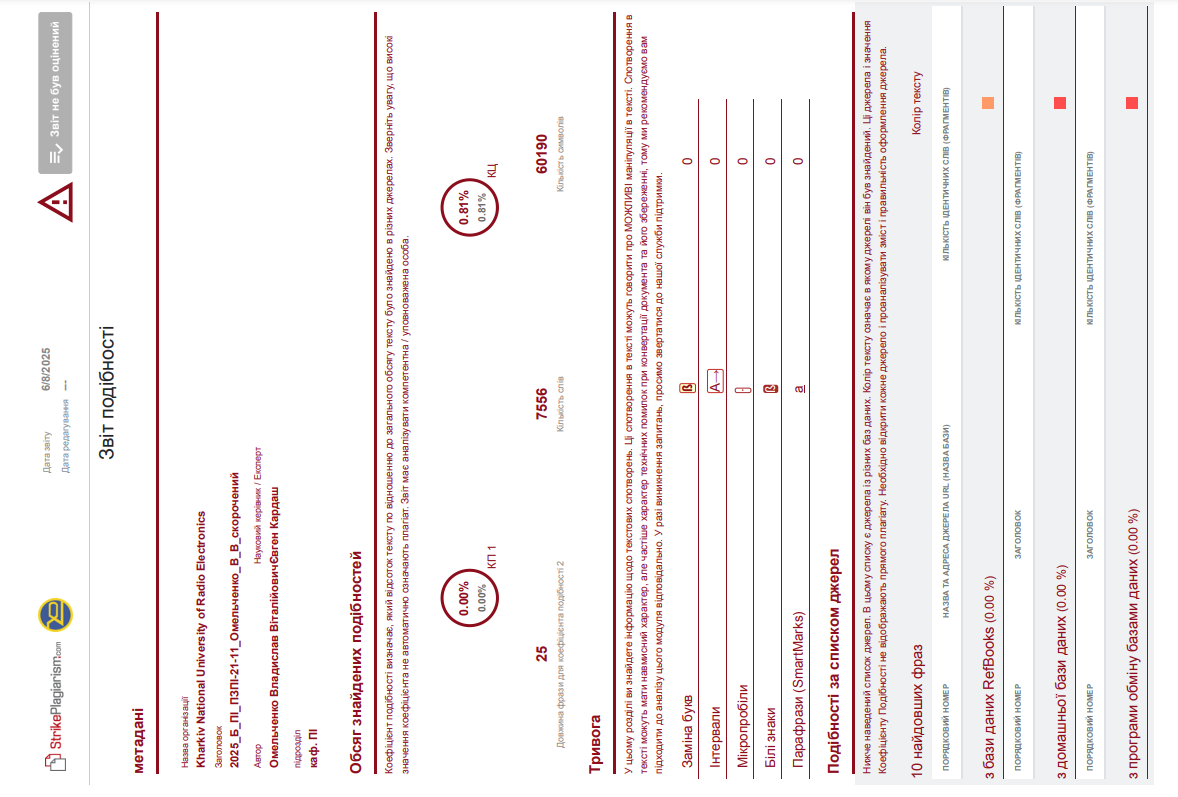
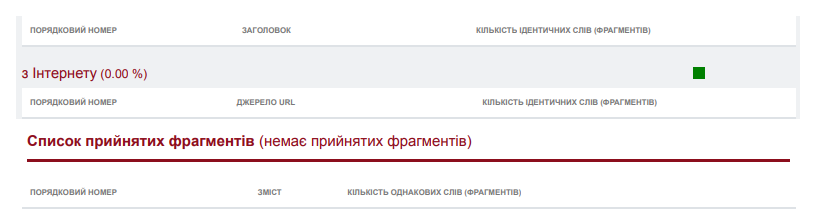


Рисунок А.1 – Decision tree діаграма реєстрації (розроблена самостійно)

**Додаток Б**

Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ





**Додаток в**

Слайди презентації

