МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ

циклова комісія комп’ютерної інженерії та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему
Онлайн-платформи для командної роботи тестувальників із підтримкою чеклістів та звітів

Виконав:
студент групи 2П-22
спеціальності
121 Інженерія програмного
забезпечення
Дмитро КОНОНЧУК
Керівник
Дмитро ПОДОРОШКО

Черкаси 2026

АНОТАЦІЯ

Кваліфікаційна робота присвячена проєктуванню та розробці онлайн-платформи для командної роботи тестувальників із підтримкою чеклістів та звітів (на базі системи QATracker Pro).

Актуальність теми. У сучасному циклі розробки програмного забезпечення (SDLC) забезпечення якості (Quality Assurance) є одним із найважливіших етапів. Зі зростанням складності програмних продуктів та переходом більшості компаній на гнучкі методології розробки (Agile, Scrum), виникає гостра потреба у зручних, швидких та інтегрованих інструментах для управління процесом тестування. Існуючі рішення часто є або занадто складними та дорогими для невеликих команд, або не надають достатнього функціоналу для зручного ведення чеклістів та автоматизованої генерації звітів. Тому створення сучасної, адаптивної та масштабованої вебплатформи для координації QA-інженерів є актуальним завданням.

Мета роботи: розробити онлайн-платформу, яка забезпечить ефективну взаємодію команди тестувальників, дозволить створювати та управляти чеклістами, фіксувати результати тестування та автоматично формувати детальну звітність щодо якості програмного продукту.

Завдання дослідження:

1. Проаналізувати предметну область, процеси забезпечення якості та існуючі системи управління тестуванням.
2. Сформувати функціональні та нефункціональні вимоги до вебплатформи.
3. Спроєктувати архітектуру системи та структуру реляційної бази даних.
4. Розробити серверну (Backend) та клієнтську (Frontend) частини платформи.
5. Провести комплексне тестування розробленого програмного продукту.
6. Оцінити ефективність впровадження системи у реальні процеси розробки.

Результати розробки. У результаті виконання роботи створено повноцінну онлайн-платформу QATracker Pro. Реалізовано архітектуру клієнт-

сервер з використанням сучасного стека технологій: React, TypeScript, Tailwind CSS для frontend-частини, та Node.js, Express.js із базою даних PostgreSQL для backend-частини. Забезпечено безпечну авторизацію за допомогою JWT-токенів. Розроблено модулі управління проєктами, інтерактивного ведення чеклістів, контролю виконання завдань та генерації аналітичних звітів з можливістю їх експорту.

Практична цінність системи. Впровадження платформи дозволяє скоротити час на організацію ручного тестування на 25-30%, мінімізувати втрату даних щодо знайдених дефектів та підвищити загальну прозорість процесу контролю якості для всіх учасників команди розробки. Система може бути використана як у навчальному процесі для підготовки QA-спеціалістів, так і в комерційних IT-компаніях малого та середнього бізнесу.

Ключові слова: тестування програмного забезпечення, чекліст, баг-трекінг, звітність, вебплатформа, командна робота, QA.

ABSTRACT

The qualification work is devoted to the design and development of an online platform for the teamwork of testers with the support of checklists and reports (based on the QATracker Pro system).

Relevance of the topic. In the modern software development life cycle (SDLC), Quality Assurance is one of the most critical stages. With the growing complexity of software products and the transition of most companies to flexible development methodologies (Agile, Scrum), there is an urgent need for convenient, fast, and integrated tools for managing the testing process. Existing solutions are often either too complex and expensive for small teams or lack sufficient functionality for convenient checklist management and automated report generation. Therefore, the creation of a modern, adaptive, and scalable web platform for QA engineer coordination is a highly relevant task.

The goal of the work: to develop an online platform that will ensure effective interaction of the testing team, allow creating and managing checklists, record testing results, and automatically generate detailed reports on software product quality.

Research objectives:

1. Analyze the subject area, quality assurance processes, and existing test management systems.
2. Formulate functional and non-functional requirements for the web platform.
3. Design the system architecture and relational database structure.
4. Develop the server (Backend) and client (Frontend) parts of the platform.
5. Conduct comprehensive testing of the developed software product.
6. Evaluate the efficiency of implementing the system into real development processes.

Development results. As a result of the work, a full-fledged online platform, QATracker Pro, was created. A client-server architecture was implemented using a modern technology stack: React, TypeScript, Tailwind CSS for the frontend, and Node.js, Express.js with a PostgreSQL database for the backend. Secure authorization using JWT tokens was ensured. Modules for project management, interactive checklist maintenance, task execution control, and analytical report generation with export capabilities were developed.

Practical value of the system. The implementation of the platform reduces the time spent organizing manual testing by 25-30%, minimizes data loss regarding found defects, and increases the overall transparency of the quality control process for all development team members. The system can be used both in the educational process for training QA specialists and in commercial IT companies of small and medium-sized businesses.

Keywords: software testing, checklist, bug tracking, reporting system, web platform, team collaboration, quality assurance.

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ

QA – Quality Assurance (Забезпечення якості)

QC – Quality Control (Контроль якості)

UI – User Interface (Інтерфейс користувача)

UX – User Experience (Досвід користувача)

REST – Representational State Transfer (Передача стану представлення)

JWT – JSON Web Token (Вебтокен JSON)

API – Application Programming Interface (Програмний інтерфейс додатку)

CRUD – Create, Read, Update, Delete (Створення, читання, оновлення, видалення)

SQL – Structured Query Language (Мова структурованих запитів)

CI/CD – Continuous Integration / Continuous Deployment (Безперервна інтеграція / Безперервне розгортання)

SDLC – Software Development Life Cycle (Життєвий цикл розробки програмного забезпечення)

DOM – Document Object Model (Об'єктна модель документа)

ERD – Entity-Relationship Diagram (Діаграма "Сутність-Зв'язок")

HTTP – HyperText Transfer Protocol (Протокол передачі гіпертексту)

ЗМІСТ

ВСТУП

Актуальність теми. Сучасна індустрія розробки програмного забезпечення характеризується стрімким зростанням складності продуктів, скороченням термінів випуску та підвищенням вимог до їх якості. За цих умов тестування стає невід'ємною складовою життєвого циклу розробки — воно забезпечує відповідність програмних продуктів функціональним вимогам, виявляє дефекти на ранніх етапах і знижує ризики виходу з ладу критичних систем. За даними дослідницьких агентств, витрати на усунення дефектів[4], виявлених після релізу, у десятки разів перевищують витрати на їх виправлення під час розробки, що підкреслює стратегічну значущість якісного тестування.

Командна взаємодія між тестувальниками є ключовим чинником ефективності процесу контролю якості. У сучасних agile- та DevOps-командах тестувальники працюють паралельно над різними модулями системи, тому необхідно забезпечити узгоджений розподіл завдань, прозорий обмін результатами та консолідований облік знайдених дефектів. Відсутність єдиного середовища для командної роботи призводить до дублювання тест-кейсів, втрати інформації та сповільнення процесу виявлення критичних помилок.

Чеклісти є одним з найефективніших інструментів систематизації тестових сценаріїв. Вони дозволяють стандартизувати перевірки, зменшити кількість пропущених дефектів та спростити процес введення нових членів команди у проєкт. Проте ведення чеклістів вручну у форматі електронних таблиць або текстових документів є трудомістким, схильним до помилок і не забезпечує можливості реального відстеження прогресу. Автоматизація ведення чеклістів у рамках спеціалізованої платформи усуває ці недоліки та підвищує загальну ефективність команди тестувальників.

Не менш актуальною є проблема формування звітів про результати тестування. Ручна підготовка звітів потребує значних часових витрат і може містити суб'єктивні помилки у викладенні даних. Натомість автоматизоване формування

звітів на основі зібраних метрик дозволяє оперативно отримувати об'єктивну картину стану якості продукту, що є критично важливим для прийняття управлінських рішень щодо виходу продукту на ринок. Використання ж розрізнених інструментів — окремих трекерів задач, електронних таблиць та систем обміну повідомленнями — не лише ускладнює управління процесом тестування, а й збільшує ризик втрати даних та знижує прозорість роботи команди.

Ступінь дослідження проблеми[8][9][10][11]. Питання управління тестуванням програмного забезпечення досліджувалося у роботах вітчизняних та зарубіжних науковців, зокрема в контексті методологій гнучкої розробки, стандартів ISO/IEC 29119[1] та підходів до Test Management. Практичне вирішення завдання автоматизації управління тестуванням знайшло відображення у низці комерційних та відкритих програмних продуктів. Зокрема, Jira у поєднанні з плагінами Zephyr або Xray є одним із найпоширеніших рішень, що забезпечує інтеграцію трекінгу дефектів із управлінням тест-кейсами. Платформа TestRail пропонує розвинений інструментарій для організації тестових прогонів та формування звітів, однак її висока вартість і складність налаштування є суттєвим бар'єром для малих команд. Система Qase вирізняється сучасним інтерфейсом та гнучкою моделлю ціноутворення, але обмежені можливості кастомізації чеклістів знижують її ефективність для специфічних проєктів. TestLink є безкоштовним open-source рішенням з давньою історією, проте застарілий інтерфейс та відсутність підтримки сучасних робочих процесів обмежують його застосування. Таким чином, більшість існуючих платформ або є надмірно складними й дорогими для невеликих команд, або не забезпечують у повному обсязі підтримки чеклістів і автоматизованого формування звітів, що зумовлює необхідність розроблення спеціалізованого рішення.

Об'єкт дослідження — процеси організації командної роботи тестувальників під час контролю якості програмного забезпечення.

Предмет дослідження — методи, засоби та технології розробки вебзастосунків для керування чеклістами та звітами тестування.

Мета роботи — розроблення онлайн-платформи для командної роботи тестувальників із підтримкою чеклістів та автоматизованого формування звітів.

Завдання роботи. Для досягнення поставленої мети необхідно вирішити такі завдання:

1) дослідити предметну область управління тестуванням програмного забезпечення;

2) проаналізувати існуючі аналоги та визначити їх переваги й недоліки;

3) сформувати функціональні та нефункціональні вимоги до розроблюваної системи;

4) спроєктувати архітектуру системи;

5) розробити базу даних;

6) реалізувати вебзастосунок;

7) реалізувати систему авторизації;

8) реалізувати модуль чеклістів;

9) реалізувати модуль звітів;

10) провести тестування програмного продукту;

11) оцінити результати впровадження.

Методи дослідження. У роботі використовуються такі методи: аналіз наукових літературних джерел та технічної документації; порівняльний аналіз існуючих програмних продуктів; методи об'єктно-орієнтованого проєктування; UML-моделювання[22] для візуалізації архітектури та поведінки системи; методи тестування програмного забезпечення, зокрема функціональне та інтеграційне тестування.

Практична цінність роботи полягає у тому, що розроблена платформа може бути використана малими та середніми командами тестувальників для організації спільної роботи в рамках єдиного середовища. Запропоноване рішення спрощує процес контролю якості програмного забезпечення, знижує трудомісткість

ведення документації та підвищує прозорість тестового процесу завдяки централізованому зберіганню даних і автоматизованому формуванню звітів.

Очікувані результати. За підсумками виконання кваліфікаційної роботи очікується створення працездатної вебплатформи з реалізованими механізмами командної взаємодії, підтримкою чеклістів, автоматичним формуванням звітів та централізованим зберіганням інформації про тестування. Отриманий програмний продукт може бути впроваджений у реальних командах тестувальників і стати основою для подальшого розвитку у напрямі інтеграції з CI/CD-процесами та засобами автоматизованого тестування.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Характеристика предметної області

Сучасна індустрія розробки програмного забезпечення характеризується постійним зростанням складності продуктів, скороченням термінів виходу на ринок та підвищенням вимог до якості. В умовах конкурентного середовища організації змушені приділяти значну увагу процесам забезпечення якості програмного забезпечення (Quality Assurance, QA). Тестування є невід'ємною складовою цих процесів і відіграє ключову роль у забезпеченні надійності, функціональності та безпеки програмних продуктів.

Предметна область даної кваліфікаційної роботи охоплює процеси організації та управління тестуванням програмного забезпечення в умовах командної роботи. Тестування програмного забезпечення (Software Testing) визначається як процес перевірки програмного продукту з метою виявлення дефектів, підтвердження відповідності функціональних і нефункціональних вимог, а також оцінювання рівня якості перед введенням в експлуатацію.

У контексті сучасних підходів до розробки ПЗ тестування розглядається не як окрема фаза, що слідує за розробкою, а як безперервний процес, інтегрований в усі етапи життєвого циклу. Стандарт ISO/IEC/IEEE 29119 визначає тестування як систематичну діяльність із пошуку та документування дефектів, а також підтвердження відповідності програмного забезпечення встановленим вимогам[1].

Місце тестування у життєвому циклі розробки програмного забезпечення (SDLC) залежить від обраної методології. У класичній каскадній моделі (Waterfall) тестування є окремою фазою після завершення розробки. Проте в

сучасних гнучких методологіях, зокрема Agile, Scrum та Kanban,[5] тестування здійснюється паралельно з розробкою протягом кожної ітерації або спринту[4]. В методології DevOps тестування є безперервним процесом, що підтримується автоматизованими засобами у рамках конвеєрів безперервної інтеграції та безперервного розгортання (CI/CD).

Незалежно від методології розробки, діяльність QA-команди охоплює такі основні напрями: аналіз вимог та їх тестопридатність, розробку стратегії та плану тестування, проєктування тестових сценаріїв і чеклістів, виконання тестів (ручне та автоматизоване), реєстрацію та відстеження дефектів, формування звітів про результати тестування, а також аналіз метрик якості.

У типовій QA-команді виділяють кілька ключових ролей. Manual QA (тестувальник) — це фахівець, який виконує ручне тестування програмного забезпечення відповідно до розроблених тест-кейсів і чеклістів. Тестувальник відповідає за виявлення дефектів, їх документування в системі управління тестуванням та підтвердження виправлень після усунення помилок розробниками.

QA Lead (керівник QA) — фахівець, який поєднує технічні та організаційні функції. Він відповідає за координацію роботи команди тестувальників, розробку стратегії тестування, розподіл завдань між членами команди, контроль якості тестової документації та взаємодію з менеджерами проєктів і розробниками. QA Lead також аналізує метрики якості та готує зведені звіти.

Test Manager (менеджер тестування) — роль стратегічного рівня, яка охоплює планування ресурсів, бюджетування процесів тестування, управління ризиками якості та комунікацію з бізнес-стейкхолдерами. Test Manager відповідає за загальну організацію тестування в рамках кількох проєктів або продуктових ліній.

Administrator (адміністратор системи) — роль, відповідальна за технічне налаштування платформи управління тестуванням, управління обліковими записами користувачів, налаштування прав доступу, підтримку інфраструктури та забезпечення доступності інструментів для команди.

Чеклісти (check-lists) є одним із базових інструментів у практиці ручного тестування. Чекліст являє собою структурований перелік перевірок, які необхідно виконати для підтвердження коректності роботи програмного компоненту або функціональності. На відміну від детальних тест-кейсів, чеклісти містять лаконічні пункти перевірки без детального опису кроків і очікуваних результатів. Це дозволяє пришвидшити виконання регресійного тестування та перевірку нових функціональностей.

Процес роботи з чеклістами включає кілька етапів: створення чекліста (визначення переліку перевірок відповідно до вимог), призначення чекліста тестувальнику, виконання перевірок із фіксацією статусу кожного пункту (пройдено / не пройдено / заблоковано), реєстрацію виявлених дефектів та формування звіту за результатами виконання. Ефективне управління чеклістами потребує їх версіонування, можливості повторного використання в різних проєктах та централізованого зберігання[6].

Звіти (reports) у процесі тестування виконують функцію комунікаційного інструменту між командою тестування та іншими зацікавленими сторонами: менеджерами проєктів, розробниками, бізнес-замовниками. Типовий звіт про тестування містить загальну статистику виконання тестів (кількість пройдених, провалених, заблокованих перевірок), перелік виявлених дефектів із класифікацією за критичністю, покриття вимог тестами, а також рекомендації щодо готовності продукту до випуску.

Типові бізнес-процеси взаємодії тестувальників у команді включають: розподіл чеклістів між виконавцями, паралельне тестування різних модулів системи, узгодження статусів дефектів між тестувальниками і розробниками, проведення командних ревью результатів тестування, а також ескалацію

критичних дефектів до менеджменту. Ефективна командна робота вимагає єдиного простору для зберігання тестової документації та прозорості у відображенні прогресу тестування для всіх учасників процесу.

На практиці значна частина QA-команд досі використовує для управління тестуванням електронні таблиці (Microsoft Excel, Google Sheets) та офісні документи (Microsoft Word, Google Docs). Такий підхід має суттєві обмеження. По-перше, електронні таблиці не забезпечують ефективної підтримки колаборативної роботи в режимі реального часу: одночасне редагування декількома тестувальниками призводить до конфліктів версій і втрати даних. По-друге, відсутність рольової моделі доступу призводить до ризиків несанкціонованого редагування документації. По-третє, ручне формування звітів на основі розрізнених таблиць потребує значних часових витрат і є джерелом помилок.

Проблема централізованого зберігання тестової документації є однією з ключових у великих командах. При використанні файлових сховищ (файловий сервер, хмарне сховище) документи зберігаються без логічного зв'язку між собою: чеклісти не пов'язані з проєктами, результати тестування не прив'язані до конкретних чеклістів, а звіти формуються окремо. Це унеможливлює ефективний аналіз якості та відстеження прогресу в динаміці.

Потреба в автоматизації процесів управління тестуванням є визнаним трендом у сучасній індустрії ПЗ. Спеціалізовані платформи управління тестуванням (Test Management Tools) дозволяють: централізувати зберігання тестової документації, автоматизувати формування звітів, забезпечити ефективний розподіл завдань у команді, налаштувати рольову модель доступу, а також відстежувати прогрес тестування в режимі реального часу. Впровадження таких систем дозволяє скоротити витрати часу на адміністративні завдання та підвищити ефективність QA-процесів на 20–40%.

На основі проведеного аналізу предметної області можна сформулювати такі основні проблеми, які має вирішувати майбутня система:

1. відсутність єдиного централізованого середовища для зберігання та виконання чеклістів тестування;
2. складність організації командної роботи при використанні розрізнених інструментів (таблиць, документів, мессенджерів);
3. відсутність автоматизованого формування звітів за результатами виконання чеклістів;
4. неможливість відстеження прогресу тестування в режимі реального часу у розподілених командах;
5. відсутність рольової моделі доступу до тестової документації;
6. низька прозорість процесів тестування для менеджменту та замовників.

1.2 Аналіз наявних рішень

Ринок інструментів управління тестуванням пропонує широкий спектр продуктів — від інтегрованих платформ управління проєктами до спеціалізованих систем управління тестами. Нижче наведено аналіз п'яти найпоширеніших рішень.

Jira (Atlassian)

Jira — це комплексна платформа управління проєктами та відстеження завдань, розроблена компанією Atlassian. Попри те, що Jira не є спеціалізованим інструментом управління тестуванням, вона широко використовується QA-командами завдяки розвиненій екосистемі плагінів та інтеграцій.

Основні можливості: управління завданнями та спринтами (Scrum/Kanban), відстеження дефектів, гнучка система фільтрів і пошуку,

конфігурація робочих процесів (workflows), дошки проєктів, базова звітність (burndown-charts, velocity-charts), API для інтеграцій.

Переваги Jira: широка функціональність управління проєктами, велика спільнота користувачів, багатий маркетплейс плагінів, підтримка хмарного та серверного розгортання, добре розвинена API та інтеграції з інструментами CI/CD.

Недоліки: Jira не має вбудованої підтримки чеклістів тестування (потребує встановлення окремих плагінів, наприклад Checklist for Jira); складний інтерфейс із крутою кривою навчання; висока вартість для великих команд; звіти з тестування потребують налаштування або підключення спеціалізованих плагінів (Zephyr Scale, Xray).

TestRail (Gurock / Idera)

TestRail — спеціалізована система управління тестуванням, що є одним із лідерів ринку у своєму сегменті. Продукт орієнтований на QA-команди та надає розгалужені можливості для планування, виконання тестів і формування звітності.

Основні можливості: управління тест-кейсами та тест-планами, підтримка тест-раніків, формування детальних звітів про проходження тестів, відстеження прогресу в реальному часі, інтеграція з Jira та іншими інструментами, API для автоматизованого додавання результатів.

Переваги: потужна звітність, зручна організація тест-кейсів у ієрархічну структуру, широкі можливості фільтрації, підтримка масштабних команд, добре налагоджені інтеграції з популярними інструментами.

Недоліки: висока вартість (ліцензування per-user); чеклісти реалізовані обмежено — система орієнтована переважно на детальні тест-кейси; складність налаштування для невеликих команд; відсутність безкоштовного плану.

Qase

Qase — сучасна хмарна платформа управління тестуванням, орієнтована на QA-команди різного розміру. Відзначається зручним інтерфейсом і достатньо повним набором функцій для управління тест-кейсами, чеклістами та звітністю.

Основні можливості: управління тест-кейсами та чеклістами, тест-плани та тест-рани, формування звітів, відстеження дефектів, інтеграції з популярними інструментами (Jira, GitHub, Slack), підтримка API, управління середовищами тестування.

Переваги: зрозумілий і сучасний інтерфейс, наявність чеклістів як окремої сутності, безкоштовний план для невеликих команд, активний розвиток продукту, хмарна архітектура без необхідності налаштування серверної інфраструктури.

Недоліки: функціональність безкоштовного плану обмежена; глибина налаштування звітів поступається TestRail; залежність від хмарної інфраструктури може бути обмеженням для компаній із суворими вимогами до зберігання даних.

Zephyr Scale (SmartBear)

Zephyr Scale — плагін для Jira, що розширює її функціональність до повноцінної системи управління тестуванням. Є частиною екосистеми SmartBear і орієнтований на команди, які вже використовують Jira як основний інструмент.

Основні можливості: управління тест-кейсами безпосередньо в Jira, тест-плани та тест-цикли, звітність у реальному часі (дашборди), відстеження покриття вимог, інтеграція з системами автоматизації тестування.

Переваги: тісна інтеграція з Jira, можливість пов'язувати тест-кейси з вимогами та дефектами в єдиному просторі, розгалужена звітність, підтримка хмарного та серверного варіантів.

Недоліки: є залежним від Jira — не може використовуватись як самостійний інструмент; висока сукупна вартість (Jira + Zephyr Scale); відсутня підтримка чеклістів як окремої сутності; складна початкова конфігурація.

TestLink

TestLink — безкоштовна система управління тестуванням із відкритим вихідним кодом (Open Source). Є одним із найдавніших інструментів у своєму сегменті та широко використовується в командах із обмеженим бюджетом.

Основні можливості: управління тест-кейсами і тест-планами, виконання тест-раніків, базова звітність, управління вимогами, розподіл завдань між тестувальниками, підтримка кількох проєктів.

Узагальнені результати порівняльного аналізу розглянутих рішень наведено в таблиці 1.1.[7],[8],[9],[10],[11]

Переваги: безкоштовне використання, відкритий вихідний код (можливість модифікації), підтримка самостійного розгортання (on-premise), підтримка великої кількості тест-кейсів.

Недоліки: застарілий інтерфейс, що значно поступається сучасним рішенням у зручності використання; відсутність підтримки чеклістів; обмежені можливості командної роботи в режимі реального часу; повільний темп оновлень; складне налаштування та адміністрування; відсутність хмарного варіанту розгортання.

Таблиця 1.1 — Порівняльний аналіз наявних рішень для управління тестуванням

Критерій	Jira	TestRail	Qase	Zephyr Scale	TestLink
Управління проєктами	Так (розширене)	Частково	Так	Так (через Jira)	Так
Чеклісти	Ні (плагіни)	Частково	Так	Ні	Ні
Звіти	Так (базові)	Так (розширені)	Так	Так	Обмежені
Командна робота	Так (широка)	Так	Так	Так	Обмежена
Простота використання	Складна	Середня	Висока	Середня	Низька
Вартість	Платна / безкоштовна (до 10 осіб)	Платна	Платна / безкоштовний план	Платна	Безкоштовна (Open Source)

Проведений аналіз дозволяє зробити такі узагальнені висновки. З-поміж розглянутих рішень TestRail демонструє найкращі показники у сфері звітності та управління тест-кейсами, Jira — у сфері управління проєктами та інтеграцій, Qase — у сфері зручності використання та підтримки чеклістів.

Водночас для більшості систем характерні спільні недоліки: висока вартість або обмеженість безкоштовних планів (Jira, TestRail, Zephyr Scale); чеклісти або не підтримуються зовсім (Zephyr Scale, TestLink), або реалізовані як другорядна функція (TestRail); складність початкового налаштування та висока крива навчання для нових користувачів; відсутність інтуїтивно зрозумілого робочого процесу виконання чеклістів у командному режимі.

Таким чином, виявлено практичну відсутність доступних рішень, які б органічно поєднували зручну роботу з чеклістами, командну взаємодію та автоматизовану звітність у рамках єдиної платформи з простим інтерфейсом і прийнятною вартістю. Розробка власної онлайн-платформи для командної роботи тестувальників є доцільною з огляду на такі чинники: можливість реалізувати оптимальний робочий процес, орієнтований саме на чеклісти; гнучкість у налаштуванні рольової моделі; відсутність надлишкової складності, притаманної великим комбайнам на кшталт Jira; забезпечення повного контролю над даними та можливість подальшого масштабування відповідно до потреб команди.

1.3 Постановка завдання до кваліфікаційної роботи

На основі аналізу предметної області та існуючих рішень сформульовано постановку завдання до кваліфікаційної роботи.

Метою кваліфікаційної роботи є проєктування та розробка онлайн-платформи для командної роботи тестувальників, яка забезпечує централізоване

управління чеклістами, організацію спільної роботи QA-команди та автоматизоване формування звітів про результати тестування.

Система призначена для QA-команд, що здійснюють ручне тестування програмного забезпечення, а також для менеджерів і замовників, що потребують актуальної інформації про стан процесів тестування.

Основними користувачами системи є:

7. Адміністратор — відповідає за управління обліковими записами користувачів, налаштування прав доступу та загальне адміністрування платформи;
8. Керівник команди (QA Lead / Test Manager) — відповідає за створення проєктів, управління командою, створення та призначення чеклістів, контроль прогресу тестування та формування звітів;
9. Тестувальник (Manual QA) — виконує призначені чеклісти, фіксує статуси перевірок та переглядає результати своєї роботи.

Функціональні вимоги до системи:

- **FR-01 Реєстрація користувача.** Система повинна надавати можливість реєстрації нових облікових записів із зазначенням імені, електронної пошти та пароля.
- **FR-02 Авторизація користувача.** Система повинна забезпечувати автентифікацію користувачів за електронною поштою та паролем із підтримкою захищених сесій.
- **FR-03 Управління профілем.** Користувач повинен мати можливість переглядати та редагувати дані власного профілю (ім'я, аватар, пароль).
- **FR-04 Створення проєктів.** Авторизовані користувачі з відповідними правами повинні мати можливість створювати нові проєкти тестування із зазначенням назви та опису.

- **FR-05 Управління командою.** Керівник команди повинен мати можливість запрошувати користувачів до проєкту та призначати їм ролі в межах проєкту.
- **FR-06 Створення чеклістів.** Система повинна надавати інструменти для створення чеклістів у рамках проєкту з можливістю додавання груп перевірок та окремих пунктів.
- **FR-07 Редагування чеклістів.** Система повинна забезпечувати можливість редагування існуючих чеклістів: зміну назви, додавання, видалення та зміну порядку пунктів перевірок.
- **FR-08 Виконання чеклістів.** Тестувальники повинні мати можливість виконувати призначені чеклісти з фіксацією статусу кожного пункту (пройдено / не пройдено / заблоковано) та додаванням коментарів.
- **FR-09 Формування звітів.** Система повинна автоматично формувати звіти за результатами виконання чеклістів із відображенням статистики за пунктами та загальним статусом.
- **FR-10 Перегляд статистики.** Система повинна надавати дашборд із загальною статистикою тестування в розрізі проєктів і часового діапазону.
- **FR-11 Експорт звітів.** Система повинна підтримувати експорт сформованих звітів у форматах PDF та Excel.
- **FR-12 Управління ролями користувачів.** Адміністратор повинен мати можливість призначати та змінювати системні ролі користувачів, керувати активністю облікових записів.

Нефункціональні вимоги до системи:

10. Продуктивність: система повинна забезпечувати час відповіді на стандартні запити не більше 2 секунд при одночасній роботі до 100 активних користувачів.

11. Масштабованість: архітектура системи повинна допускати горизонтальне масштабування серверної частини для підтримки зростаючого навантаження.
12. Безпека: система повинна забезпечувати захищену передачу даних за протоколом HTTPS, зберігання паролів у хешованому вигляді (bcrypt), захист від XSS та CSRF атак, а також реалізацію принципу мінімальних привілеїв доступу.
13. Надійність: система повинна забезпечувати доступність на рівні не менше 99% у робочий час, а також коректну обробку помилок без втрати даних.
14. Доступність: система повинна бути доступна через браузер без необхідності встановлення додаткового програмного забезпечення.
15. Адаптивність інтерфейсу: веб-інтерфейс системи повинен коректно відображатися на пристроях з різними розмірами екрану (Desktop, Tablet, Mobile).
16. Зручність використання: інтерфейс системи повинен відповідати принципам UX-дизайну, забезпечувати можливість освоєння основних функцій новим користувачем без спеціального навчання.

Обмеження системи:

17. система реалізується як веб-додаток без розробки нативних мобільних застосунків;
18. автоматизоване тестування (виконання автотестів) не входить до функціональних можливостей платформи;
19. інтеграція зі сторонніми системами управління дефектами не є обов'язковою для першої версії продукту;
20. система орієнтована на потреби малих і середніх QA-команд (до 50 учасників у одному проєкті).[1],[4]

Середовище функціонування:

21. клієнтська частина функціонує у веб-браузері на комп'ютерах та мобільних пристроях під керуванням операційних систем Windows, macOS, Linux, Android та iOS;
22. серверна частина розгортається на Linux-сервері з підтримкою Node.js;
23. для зберігання даних використовується реляційна база даних;
24. розгортання можливе як на власній серверній інфраструктурі, так і на хмарних платформах (AWS, GCP, Azure або VPS).

Вимоги до браузерів: система повинна підтримувати актуальні версії браузерів Google Chrome (версія 100+), Mozilla Firefox (версія 100+), Microsoft Edge (версія 100+) та Safari (версія 15+).

Вимоги до серверної частини: серверна частина системи повинна бути реалізована з використанням Node.js (версія 18+) або Python (версія 3.10+), підтримувати REST API, забезпечувати автентифікацію на основі JWT-токенів та взаємодіяти з реляційною СУБД (PostgreSQL або MySQL).

Таким чином, підсумковим завданням кваліфікаційної роботи є: спроєктувати та розробити онлайн-платформу для командної роботи тестувальників із підтримкою чеклістів та звітів, яка включає: модуль реєстрації та автентифікації користувачів із рольовою моделлю (Адміністратор, Керівник команди, Тестувальник); модуль управління проєктами та командою; модуль створення, редагування та виконання чеклістів; модуль формування звітів та статистики з підтримкою експорту; адаптивний веб-інтерфейс. Розроблена платформа повинна відповідати визначеним функціональним та нефункціональним вимогам і забезпечувати ефективну організацію процесів ручного тестування в розподілених QA-командах.

1.4 Висновки до розділу 1

У першому розділі проведено комплексний аналіз предметної області управління тестуванням програмного забезпечення. Досліджено місце тестування у сучасному життєвому циклі розробки ПЗ (SDLC) відповідно до стандарту ISO/IEC/IEEE 29119 та визначено характер його інтеграції в методологіях Agile, Scrum, Kanban і DevOps. Встановлено, що незалежно від обраної методології розробки, QA-процеси охоплюють аналіз вимог, проєктування тестових сценаріїв, виконання тестів, реєстрацію дефектів та формування звітності. Описано типову рольову структуру QA-команди: Manual QA, QA Lead, Test Manager та Administrator — із чітким визначенням зон відповідальності кожної ролі.

Значну увагу приділено двом ключовим інструментам тестового процесу — чеклістам та звітам. Чекліст розглянуто як структурований перелік перевірок, що стандартизує виконання регресійного тестування, знижує ризик пропуску дефектів та спрощує введення нових членів команди до проєкту. Звіти про тестування визначено як комунікаційний інструмент між QA-командою та зацікавленими сторонами, що забезпечує об'єктивну картину стану якості продукту. Встановлено, що ведення чеклістів у форматі електронних таблиць і формування звітів вручну є неефективними підходами, що призводять до конфліктів версій, втрати даних і підвищеної трудомісткості адміністрування.

Проведено порівняльний аналіз п'яти найпоширеніших систем управління тестуванням: Jira (Atlassian), TestRail, Qase, Zephyr Scale (SmartBear) та TestLink. Встановлено, що кожна з платформ має певні переваги: TestRail вирізняється розвиненою звітністю, Jira — широкими можливостями управління проєктами та інтеграцій, Qase — зручним інтерфейсом і наявністю чеклістів як окремої сутності. Водночас для більшості систем характерні спільні недоліки: висока вартість ліцензування, обмежена підтримка чеклістів як окремого функціонального елементу (Zephyr Scale, TestLink — відсутня взагалі, TestRail — реалізована частково), надмірна складність початкового налаштування та висока крива навчання. Zephyr Scale додатково є залежним від Jira і не може функціонувати як самостійний інструмент, а TestLink має застарілий інтерфейс без хмарного варіанту розгортання.

За результатами аналізу обґрунтовано практичну відсутність доступних рішень, які б органічно поєднували зручну роботу з чеклістами, командну взаємодію та автоматизовану звітність у рамках єдиної платформи з простим інтерфейсом і прийнятною вартістю. Це зумовлює необхідність розроблення власної онлайн-платформи, орієнтованої на потреби малих і середніх QA-команд, яка стане предметом проєктування та реалізації в наступних розділах роботи.

РОЗДІЛ 2

ПРОЄКТУВАННЯ ТА МОДЕЛЮВАННЯ ОНЛАЙН-ПЛАТФОРМИ ДЛЯ КОМАНДНОЇ РОБОТИ ТЕСТУВАЛЬНИКІВ

2.1 Концептуальне моделювання системи

Концептуальне моделювання є першим етапом проєктування програмної системи, що дозволяє сформувати узагальнене уявлення про майбутній продукт, його структуру та взаємодію між компонентами. На цьому етапі визначаються основні бізнес-сутності, сценарії використання та межі системи.

Онлайн-платформа для командної роботи тестувальників є веб-застосунком, призначеним для автоматизації процесів управління тестуванням програмного забезпечення. Концепція системи полягає у створенні єдиного цифрового середовища, у якому QA-команди можуть спільно планувати, виконувати та аналізувати результати тестування без необхідності використання сторонніх інструментів обміну документами чи ручного ведення звітності.

Платформа орієнтована на три основні ролі користувачів: Адміністратор, QA Lead та Тестувальник. Кожна роль передбачає чітко визначений перелік повноважень та функцій, що забезпечує розмежування доступу та ефективне управління проєктами.

Адміністратор — це найвищий рівень доступу в системі. Адміністратор відповідає за реєстрацію та управління обліковими записами користувачів, налаштування організаційних структур (проєктів, команд), призначення ролей та забезпечення цілісності даних. Адміністратор може переглядати журнали активності всіх учасників системи та видаляти облікові записи або деактивувати їх у разі необхідності.

QA Lead — провідний спеціаліст із забезпечення якості, котрий наділений правами на управління проєктами у межах своєї команди. QA Lead може створювати та редагувати чеклisti тестування, призначати завдання тестувальникам, формувати звіти та аналізувати статистику виконання. Ця роль є ключовою з точки зору планування та контролю якості тестових процесів.

Тестувальник — основний виконавець тестових задач. Тестувальник отримує доступ до призначених йому чеклістів, виконує перевірки, відмічає статус кожного пункту (пройдено / не пройдено / пропущено) та залишає

коментарі. За результатами виконання чекліста тестувальник може ініціювати формування звіту, який передається на перевірку QA Lead.

Взаємодія між ролями будується за принципом ієрархії: Адміністратор стоїть над усіма ролями і відповідає за глобальне управління; QA Lead управляє проєктами та командами у межах наданого доступу; Тестувальник є кінцевим виконавцем, що взаємодіє безпосередньо з чеклістами. Водночас система передбачає горизонтальну взаємодію між тестувальниками в межах одного проєкту — наприклад, через спільний перегляд прогресу виконання.

Основні бізнес-сутності системи: Користувач (User), Роль (Role), Проєкт (Project), Команда (TeamMembers), Чекліст (Checklist), Пункт чекліста (ChecklistItem), Звіт (Report), Пункт звіту (ReportItem) та Журнал активності (ActivityLog). Кожна сутність має унікальний ідентифікатор та набір атрибутів, що описують її стан.[4]

Потоки даних у системі організовані за клієнт-серверною моделлю. Користувач взаємодіє з браузерним інтерфейсом[23],[20] (Frontend на React), який надсилає HTTP-запити до REST API (Backend на Node.js). Сервер виконує бізнес-логіку, звертається до бази даних PostgreSQL і повертає відповідь у форматі JSON. Усі операції, що потребують автентифікації, перевіряються через JWT-токени.

Глосарій предметної області: Чекліст — впорядкований перелік тестових перевірок з визначеними критеріями виконання. Звіт — документ, що містить результати виконання чекліста із деталями кожного пункту. QA Lead — керівник групи забезпечення якості. Тестувальник — спеціаліст, що виконує тестові перевірки. Проєкт — логічна одиниця організації роботи. Команда — група користувачів, що спільно працюють над проєктом.

2.2 Специфікація вимог до програмного забезпечення

Специфікація вимог до програмного забезпечення (Software Requirements Specification, SRS) є ключовим артефактом, що визначає поведінку системи, її обмеження та очікування щодо якості. Відповідно до стандарту IEEE 830, вимоги поділяються на функціональні та нефункціональні[4].

Функціональні вимоги описують конкретні дії, які система повинна виконувати. Нефункціональні вимоги визначають якісні характеристики системи — продуктивність, безпеку, надійність тощо. Нижче наведено структуровані таблиці вимог *таблиця 2.1*.

Таблиця 2.1 — Функціональні вимоги до системи

ID	Назва вимоги	Опис	Пріоритет	Роль
FR-01	Реєстрація	Система повинна надавати можливість нового користувача зареєструватись, вказавши ім'я, email та пароль. Email має бути унікальним.	Високий	Всі
FR-02	Авторизація	Система повинна автентифікувати користувача за email та паролем із видачею JWT-токену.	Високий	Всі
FR-03	Відновлення пароля	Система повинна надіслати лист із посиланням для скидання пароля на зареєстрований email.	Середній	Всі
FR-04	Управління проєктами	QA Lead та Адміністратор можуть створювати, редагувати та видаляти проєкти. Проєкт містить назву, опис та список учасників.	Високий	Admin, QA Lead
FR-05	Створення команд	Адміністратор може створювати команди та призначати до них користувачів з конкретними	Високий	Admin

		ролями.		
FR-06	Управління користувачами	Адміністратор може переглядати список користувачів, змінювати їхні ролі, деактивувати або видаляти облікові записи.	Високий	Admin
FR-07	Створення чеклістів	QA Lead може створювати чекліст із назвою, описом та набором пунктів із деталями перевірки.	Високий	QA Lead
FR-08	Редагування чеклістів	QA Lead може додавати, змінювати порядок та видаляти пункти чекліста до моменту його призначення виконавцю.	Високий	QA Lead
FR-09	Проходження чеклістів	Тестувальник відмічає кожен пункт чекліста як "Пройдено", "Не пройдено" або "Пропущено", додаючи коментарі.	Високий	Tester
FR-10	Формування звітів	Після завершення чекліста система автоматично формує звіт із підсумком результатів кожного пункту.	Високий	Tester, QA Lead
FR-11	Експорт звітів	Звіт може бути експортований у форматі PDF або CSV для	Середній	QA Lead

		збереження або передачі замовнику.		
FR-12	Статистика виконання	Система відображає дашборд із показниками: кількість виконаних чеклістів, відсоток успішних перевірок, прогрес по проєктах.	Середній	Admin, QA Lead

Нефункціональні вимоги визначають якісні характеристики системи та умови її функціонування. Нефункціональні вимоги до системи наведено в таблиці 2.2.[1],[6]

Таблиця 2.2 — Нефункціональні вимоги до системи

ID	Категорія	Вимога	Метрика / Умова
NFR-01	Продуктивність	Час відповіді сервера на стандартний API-запит не повинен перевищувати допустимого порогу.	≤ 300 мс при навантаженні до 100 одночасних запитів
NFR-02	Безпека	Усі паролі зберігаються у захешованому вигляді. Сесія обмежена часом дії JWT.	bcrypt (salt rounds ≥ 10); JWT TTL = 24 год
NFR-03	Масштабованість	Архітектура системи повинна забезпечувати горизонтальне масштабування серверних компонентів.	Stateless API; підтримка Docker/контейнеризації
NFR-04	Надійність	Система повинна забезпечувати безперервну роботу з мінімальними простоями.	Uptime ≥ 99,5%; автоматичне резервне копіювання БД щодня

NFR-05	Доступність	Веб-інтерфейс повинен бути доступний з будь-якого сучасного браузера без встановлення додаткового ПЗ.	Chrome, Firefox, Edge (останні 2 версії); HTTPS
NFR-06	Адаптивність	Інтерфейс повинен коректно відображатися на пристроях з різними роздільними здатностями.	Підтримка розмірів від 320px до 2560px (responsive design)
NFR-07	Зручність використання	Новий користувач повинен без навчання виконати основні операції у системі.	Виконання основних сценаріїв без помилок за ≤ 5 хвилин

2.3 Попередня архітектура та проєктування системи

Обрана архітектура системи базується на принципах багаторівневої (N-tier) архітектури із чітким розмежуванням клієнтського, серверного рівнів та рівня даних. Такий підхід є широко визнаним у промисловій розробці і відповідає принципам розподілення відповідальностей (Separation of Concerns) та слабкого зв'язування компонентів[21].

Структура системи містить чотири основні рівні: клієнтський рівень (Frontend), рівень взаємодії через REST API, серверний рівень (Backend) та рівень даних (PostgreSQL)[19].

Клієнтський рівень реалізований на бібліотеці React[23]. Він відповідає за відображення інтерфейсу користувача, управління станом застосунку (через контекст або бібліотеки на кшталт Redux), а також за відправлення HTTP-запитів до серверного API. React надає ефективну систему компонентів, декларативне оновлення DOM та розвинену екосистему бібліотек. Інтерфейс реалізується у вигляді SPA (Single Page Application), що забезпечує плавну навігацію без повного перезавантаження сторінки.

Серверний рівень реалізований на платформі Node.js з використанням фреймворку Express.js. Цей рівень приймає HTTP-запити від клієнта, виконує бізнес-логіку (валідацію, обчислення, формування звітів), взаємодіє з базою даних та повертає відповідь у форматі JSON. Node.js обрано через його асинхронну однопотокову модель виконання, що ефективно справляється з великою кількістю одночасних запитів ввід-виводу.

Рівень даних представлений реляційною СУБД PostgreSQL. Тут зберігаються всі постійні дані системи: облікові записи, проєкти, чеклісти, звіти та журнали активності. Взаємодія між серверним рівнем та базою даних здійснюється через ORM-бібліотеку (Sequelize або Knex.js), що спрощує формування SQL-запитів та міграцій схеми.

Механізм автентифікації побудований на основі JSON Web Tokens (JWT). Після успішного входу сервер генерує підписаний токен, що містить ідентифікатор користувача та його роль. Клієнт зберігає токен у пам'яті або у httpOnly cookie та передає його у заголовку Authorization при кожному наступному запиті. Термін дії токену обмежений (24 години), після чого потрібна повторна автентифікація.

Механізм авторизації реалізований через middleware на рівні Express. При отриманні запиту middleware перевіряє наявність та дійсність JWT, декодує

роль користувача і порівнює її з переліком дозволених ролей для конкретного маршруту. Якщо роль не відповідає — повертається HTTP 403 Forbidden.

Middleware — це проміжний шар обробки запитів у Express, що виконується між отриманням запиту та відправленням відповіді. У системі використовуються такі middleware: authMiddleware (перевірка JWT), roleMiddleware (перевірка ролі), validationMiddleware (валідація тіла запиту), errorHandler (централізована обробка помилок) та loggerMiddleware (журналювання запитів).

Централізована обробка помилок реалізована через Express error handler — спеціальний middleware з чотирма параметрами (err, req, res, next). Будь-яка неперехоплена помилка передається через next(err) і обробляється у єдиному місці, що спрощує підтримку коду та забезпечує уніфікований формат відповіді при виникненні виняткових ситуацій.

REST API проєктується відповідно до принципів REST: ресурсоорієнтована адресація URL, використання стандартних HTTP-методів (GET, POST, PUT, DELETE), передача стану у форматі JSON та відсутність стану сесії на сервері (statelessness). Наприклад: GET /api/projects — отримати список проєктів; POST /api/checklists — створити новий чекліст; PUT /api/checklists/:id — оновити чекліст; DELETE /api/reports/:id — видалити звіт.

C4-модель архітектури системи описує чотири рівні абстракції. На рівні контексту (Context) система взаємодіє з трьома типами акторів: Адміністратор, QA Lead та Тестувальник — усі через браузерний інтерфейс. На рівні контейнерів (Container) виділяються: веб-браузер (SPA на React), REST API-сервер (Node.js/Express) та база даних (PostgreSQL). На рівні компонентів (Component) сервер містить такі модулі: AuthModule, UserModule, ProjectModule, ChecklistModule, ReportModule та ActivityModule. На рівні коду (Code) кожен модуль реалізує патерн MVC: Controller → Service → Repository.[13],[14]

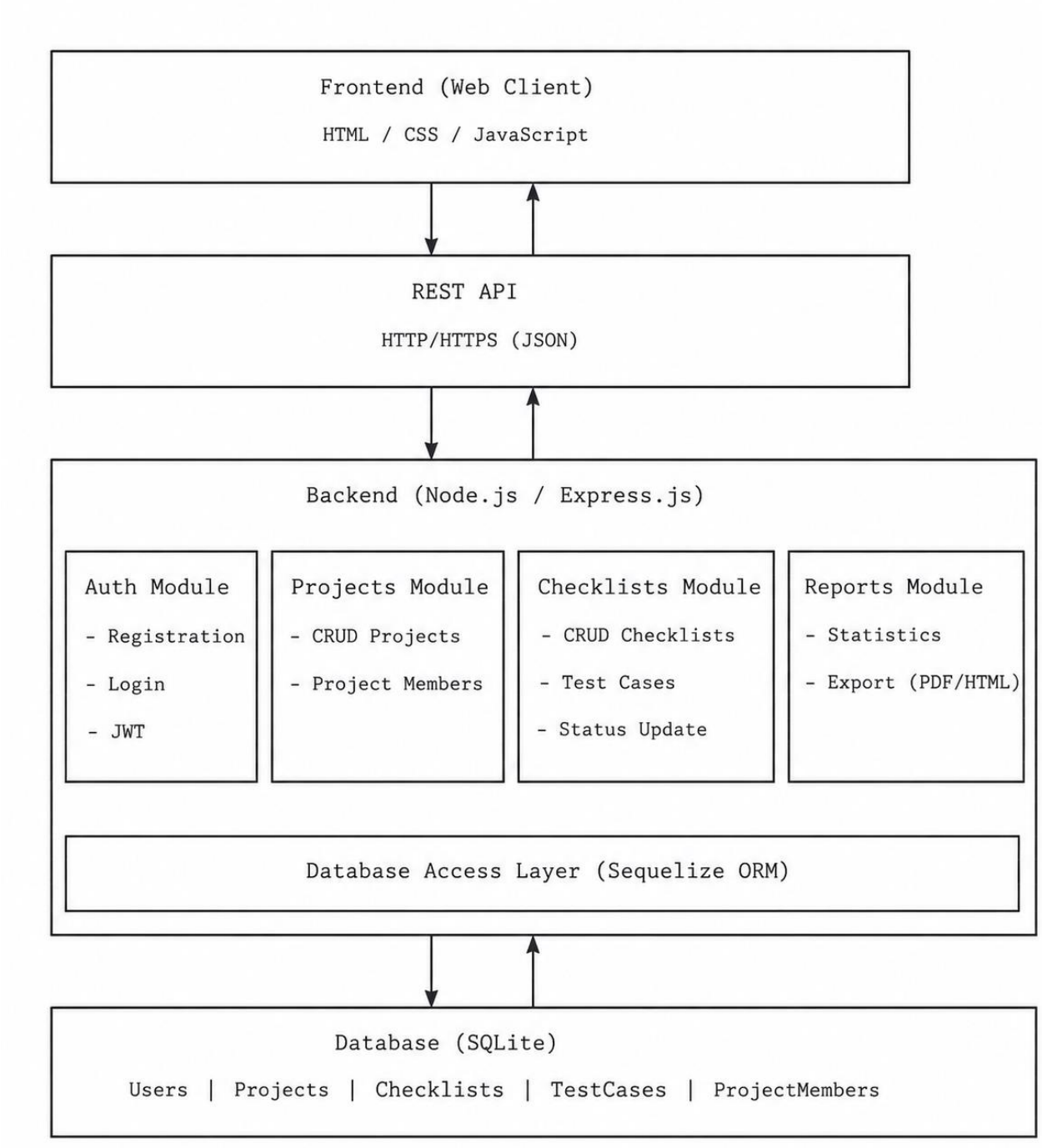

Рис. 2.1 — Архітектура системи QATracker Pro

2.4 Проєктування бази даних

Реляційна модель даних є основою зберігання інформації у системі. Вибір реляційної моделі зумовлений структурованістю предметної області, наявністю чітких зв'язків між сутностями та необхідністю забезпечення цілісності даних через механізми зовнішніх ключів і транзакцій.

PostgreSQL обраний як СУБД з наступних причин: підтримка ACID-транзакцій, розвинені типи даних (JSON, UUID, ARRAY), потужний індексний механізм, відкритий вихідний код та широка спільнота. Порівняно з MySQL, PostgreSQL[19] краще підтримує складні запити та має строгіший контроль типів.

Нижче описано основні сутності бази даних та їх призначення.

Сутність Users зберігає облікові записи всіх користувачів системи. Атрибути: id (UUID, первинний ключ), email (унікальний), password_hash, first_name, last_name, role_id (зовнішній ключ → Roles), is_active, created_at, updated_at. Зв'язки: Users має один-до-багатьох із Projects, Checklists, Reports, ActivityLogs.

Сутність Roles визначає типи доступу. Атрибути: id (INT, PK), name (admin / qa_lead / tester), description. Зв'язок: один-до-багатьох з Users.

Сутність Projects описує тестові проєкти. Атрибути: id (UUID, PK), name, description, owner_id (FK → Users), status, created_at, updated_at. Зв'язки: один-до-багатьох із TeamMembers, Checklists.

Сутність TeamMembers реалізує зв'язок багато-до-багатьох між Users та Projects. Атрибути: id (UUID, PK), project_id (FK → Projects), user_id (FK → Users), role_in_project, joined_at. Це проміжна таблиця, що дозволяє одному користувачу брати участь у кількох проєктах з різними ролями.

Сутність Checklists зберігає шаблони тестових перевірок. Атрибути: id (UUID, PK), title, description, project_id (FK → Projects), created_by (FK → Users), status (draft / active / archived), created_at, updated_at. Зв'язок: один-до-багатьох із ChecklistItems та Reports.

Сутність ChecklistItems зберігає окремі пункти чеклістів. Атрибути: id (UUID, PK), checklist_id (FK → Checklists), title, description, order_index, expected_result. Зв'язок: один-до-багатьох із ReportItems.

Сутність Reports зберігає результати виконання чеклістів. Атрибути: id (UUID, PK), checklist_id (FK → Checklists), executed_by (FK → Users), status

(in_progress / completed), total_items, passed, failed, skipped, created_at, finished_at. Зв'язок: один-до-багатьох із ReportItems.

Сутність ReportItems зберігає результат виконання кожного пункту звіту. Атрибути: id (UUID, PK), report_id (FK → Reports), checklist_item_id (FK → ChecklistItems), result (passed / failed / skipped), comment, executed_at.

Сутність ActivityLogs фіксує всі ключові дії користувачів у системі для аудиту. Атрибути: id (UUID, PK), user_id (FK → Users), action_type, entity_type, entity_id, details (JSONB), created_at.

ER-модель системи описує такі основні зв'язки: Users (1) → (*) Projects через поле owner_id; Projects (1) → (*) TeamMembers ← (*) Users (зв'язок M:N через проміжну таблицю); Projects (1) → (*) Checklists; Checklists (1) → (*) ChecklistItems; Checklists (1) → (*) Reports; Reports (1) → (*) ReportItems; ChecklistItems (1) → (*) ReportItems; Users (1) → (*) ActivityLogs. Така структура забезпечує нормалізацію до 3НФ та виключає надмірність даних.

Таблиця 2.3 — Основні сутності бази даних

Сутність	**Первинний ключ**	**Зовнішні ключі**	**Призначення**
Users	id (UUID)	role_id → Roles	Облікові записи користувачів
Roles	id (INT)	—	Типи ролей у системі
Projects	id (UUID)	owner_id → Users	Тестові проєкти
TeamMembers	id (UUID)	project_id → Projects, user_id → Users	Учасники проєкту (M:N)
Checklists	id (UUID)	project_id → Projects, created_by → Users	Чеклісти тестування
ChecklistItems	id (UUID)	checklist_id→ Checklists	Пункти чеклісту
Reports	id (UUID)	checklist_id→ Checklists, executed_by → Users	Звіти виконання
ReportItems	id (UUID)	report_id → Reports, checklist_item_id→ ChecklistItems	Результати пунктів звіту
ActivityLogs	id (UUID)	user_id → Users	Журнал дій користувачів

2.5 Моделювання сценаріїв взаємодії

Моделювання сценаріїв взаємодії виконується за допомогою мови UML (Unified Modeling Language)[22], що є стандартом для опису поведінки програмних систем. У цьому підрозділі представлено текстові описи чотирьох типів UML-діаграм: Use Case[22], Sequence, Activity та Class.

Use Case Diagram (Діаграма варіантів використання)

Діаграма варіантів використання описує зовнішню поведінку системи з точки зору акторів. В системі визначено трьох акторів: Адміністратор, QA Lead та Тестувальник.[1],[6]

Актор Адміністратор має доступ до сценаріїв: управління користувачами (перегляд, редагування, видалення); призначення ролей; перегляд журналу активності; управління проєктами (перегляд усіх проєктів); налаштування системи.

Актор QA Lead має доступ до сценаріїв: вхід до системи; перегляд та управління проєктами (своїми); формування та редагування чеклістів; призначення чеклістів тестувальникам; перегляд та експорт звітів; перегляд статистики виконання; управління учасниками команди.

Актор Тестувальник має доступ до сценаріїв: вхід до системи; перегляд призначених чеклістів; виконання чекліста (відмічення пунктів); перегляд власних звітів; формування звіту за результатами виконання чекліста.

Між варіантами використання існують зв'язки розширення (extend) та включення (include). Наприклад, «Виконання чекліста» включає «Відмічення пункту» та «Додавання коментаря». «Формування звіту» розширює «Виконання чекліста» після його завершення.

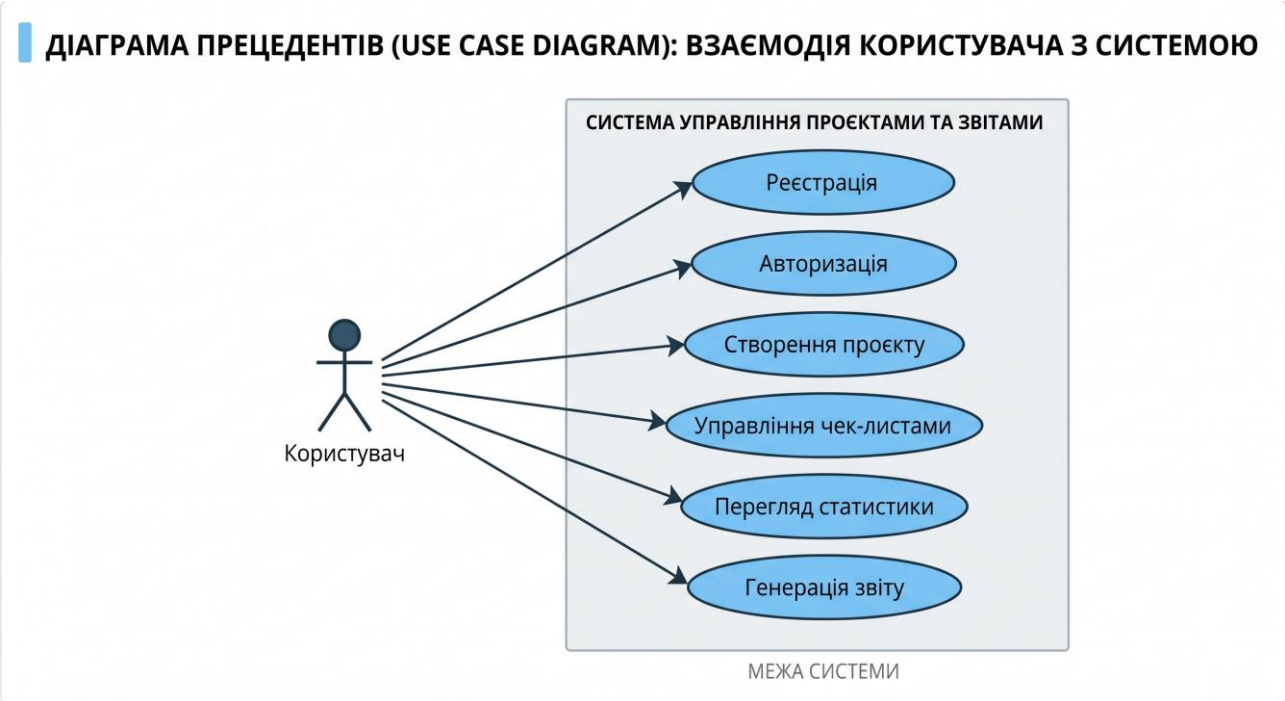

Рис. 2.2 — Діаграма прецедентів взаємодії користувача з системою

Sequence Diagram: «Створення звіту про тестування»

Сценарій «Створення звіту про тестування» розгортається між такими об'єктами: Тестувальник (Actor), Browser (UI), API Server, ChecklistService, ReportService, Database.

Послідовність повідомлень: 1) Тестувальник натискає кнопку «Завершити чекліст» у браузері. 2) Browser відправляє POST /api/reports з ідентифікатором чекліста та результатами пунктів. 3) API Server викликає authMiddleware для перевірки JWT. 4) Після успішної перевірки викликається ReportService.createReport(). 5) ReportService звертається до ChecklistService.getChecklistItems() для отримання всіх пунктів. 6) ChecklistService виконує SELECT-запит до Database. 7) Database повертає список пунктів. 8) ReportService обчислює статистику (passed/failed/skipped) та формує об'єкт звіту. 9) ReportService виконує INSERT до таблиці Reports та ReportItems у Database. 10) Database підтверджує транзакцію. 11) ReportService повертає сформований звіт API Server. 12) API Server відповідає HTTP 201 Created із JSON-об'єктом звіту. 13) Browser відображає сторінку зі звітом та статистикою.

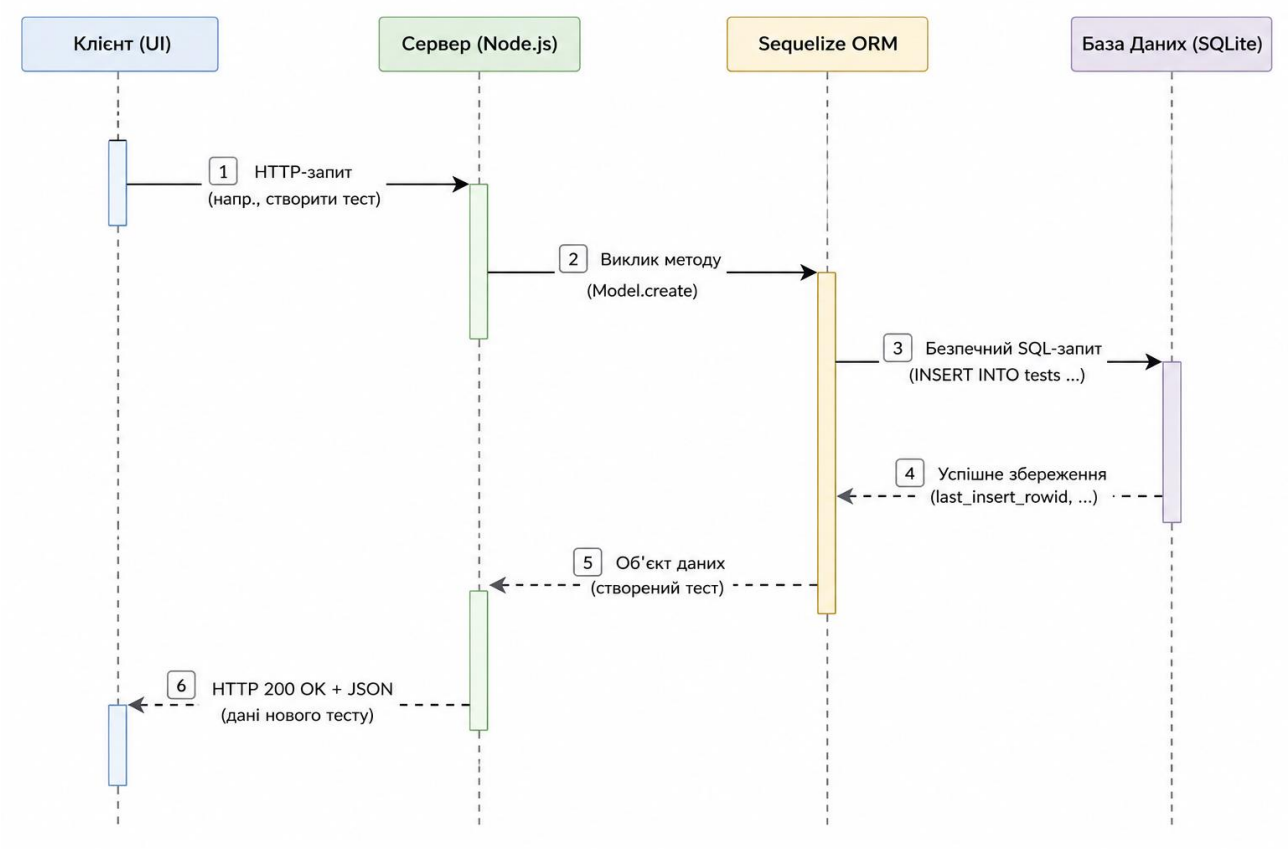

Рис. 2.3 — Діаграма послідовності операції створення тест-кейсу

Activity Diagram: «Виконання чекліста тестувальником»

Процес «Виконання чекліста тестувальником» описується наступним потоком активностей: Початок → Тестувальник відкриває список призначених чеклістів → Вибирає чекліст зі статусом «Активний» → Система завантажує пункти чекліста → [Цикл по кожному пункту]: Тестувальник переглядає опис пункту → Виконує перевірку → Відмічає результат (Пройдено / Не пройдено / Пропущено) → Якщо є зауваження — додає коментар → [Є ще пункти?] → Так: перехід до наступного пункту → Ні: всі пункти виконано → Тестувальник натискає «Завершити чекліст» → Система перевіряє, чи відмічено всі обов'язкові пункти → [Є невідмічені обов'язкові пункти?] → Так: система показує попередження → Тестувальник відмічає пропущені пункти → Ні: система формує звіт автоматично → Тестувальник переглядає підсумковий звіт → Кінець.

Class Diagram (Діаграма класів)

Діаграма класів описує статичну структуру системи на рівні програмних сутностей (моделей даних і їх зв'язків).

Клас User містить атрибути: id: UUID, email: string, passwordHash: string, firstName: string, lastName: string, role: Role, isActive: boolean, createdAt: Date. Методи: authenticate(), updateProfile(), deactivate().

Клас Project містить атрибути: id: UUID, name: string, description: string, owner: User, status: ProjectStatus, createdAt: Date. Методи: addMember(user: User), removeMember(user: User), getChecklists().

Клас Checklist містить атрибути: id: UUID, title: string, description: string, project: Project, createdBy: User, status: ChecklistStatus, items: ChecklistItem[]. Методи: addItem(item: ChecklistItem), removeItem(itemId: UUID), publish(), archive().

Клас ChecklistItem містить атрибути: id: UUID, title: string, description: string, orderIndex: number, expectedResult: string, checklist: Checklist. Методи: reorder(newIndex: number), update(data: Partial<ChecklistItem>).

Клас Report містить атрибути: id: UUID, checklist: Checklist, executedBy: User, status: ReportStatus, totalItems: number, passed: number, failed: number, skipped: number, createdAt: Date, finishedAt: Date, items: ReportItem[]. Методи: calculateStats(), export(format: 'pdf' | 'csv'), complete().

Клас TeamMember містить атрибути: id: UUID, project: Project, user: User, roleInProject: string, joinedAt: Date. Методи: changeRole(newRole: string), leave().

Зв'язки між класами: User «має» (1..*) Project (агрегація: власник проєкту); Project «містить» (1..*) Checklist (композиція: видалення проєкту видаляє чеклісти); Checklist «містить» (1..*) ChecklistItem (композиція); Checklist «породжує» (1..*) Report (асоціація); Report «містить» (1..*) ReportItem (композиція); Project «має» (*) TeamMember (*) User (асоціація через проміжний клас).

Описані UML-моделі формують повну картину взаємодії в системі та слугують основою для реалізації програмного коду на наступних етапах розробки.

2.6 Висновки до розділу 2

У другому розділі виконано комплексне проєктування онлайн-платформи QATracker Pro. На етапі концептуального моделювання визначено основні бізнес-сутності системи (User, Role, Project, TeamMembers, Checklist, ChecklistItem, Report, ReportItem, ActivityLog), описано три ролі користувачів — Адміністратор, QA Lead та Тестувальник — із чітко розмежованими повноваженнями, а також обґрунтовано клієнт-серверну модель організації потоків даних.

Сформовано специфікацію вимог до програмного забезпечення відповідно до стандарту IEEE 830. Визначено 12 функціональних вимог (FR-01–FR-12), що охоплюють реєстрацію та авторизацію користувачів, управління проєктами і командою, повний цикл роботи з чеклістами (створення, редагування, виконання), формування звітів та статистики з підтримкою експорту у PDF та CSV, а також адміністративне управління обліковими записами. Нефункціональні вимоги (NFR-01–NFR-07) регламентують продуктивність (час відповіді API до 300 мс при 100 одночасних запитах), безпеку (bcrypt для паролів, JWT TTL 24 год), масштабованість (stateless API, підтримка Docker), надійність (uptime ≥ 99,5%), доступність, адаптивність інтерфейсу (від 320 до 2560 пікселів) та зручність використання.

Спроєктовано архітектуру системи на основі чотирирівневої N-tier моделі з чітким розмежуванням клієнтського рівня (React, TypeScript, Tailwind CSS), рівня взаємодії через REST API, серверного рівня (Node.js, Express.js) та рівня даних (PostgreSQL з доступом через Prisma ORM). Обґрунтовано вибір кожної технології: Node.js — завдяки асинхронній однопотоковій моделі виконання, PostgreSQL — через підтримку ACID-транзакцій та JSON-типів, React — через декларативне оновлення DOM та розвинену екосистему, TypeScript — для уніфікації типів між клієнтською і серверною частинами. Описано механізм JWT-автентифікації та рольової авторизації через Express-middleware.

Спроєктовано реляційну структуру бази даних із дев'яти сутностей, нормалізовану до третьої нормальної форми. Проміжна таблиця TeamMembers реалізує зв'язок M:N між Users та Projects, що дозволяє одному користувачу брати участь у кількох проєктах із різними ролями. Описано чотири типи UML-діаграм: Use Case (варіанти використання для трьох акторів), Sequence (послідовність повідомлень при створенні звіту), Activity (потік дій при виконанні чекліста) та Class (статична структура класів із методами та зв'язками). Сукупність виконаних проєктних рішень утворює повну технічну основу для реалізації системи, описаної в третьому розділі.

РОЗДІЛ 3

РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ОНЛАЙН-ПЛАТФОРМИ ДЛЯ КОМАНДНОЇ РОБОТИ ТЕСТУВАЛЬНИКІВ

3.1 Реалізація програмного продукту

Розробка онлайн-платформи для командної роботи тестувальників здійснювалася з використанням сучасного технологічного стеку, що забезпечує масштабованість, підтримуваність та високу продуктивність системи. Архітектура проєкту побудована за принципом розподілу відповідальностей між клієнтською (frontend) та серверною (backend) частинами, що відповідає сучасним підходам до розробки вебзастосунків.

Клієнтська частина реалізована з використанням наступних технологій:

1. React (версія 18) — бібліотека для побудови користувацького інтерфейсу, що забезпечує ефективний рендеринг компонентів через механізм Virtual DOM[12];
2. TypeScript — статично типізована надбудова над JavaScript, що підвищує якість коду та зменшує кількість помилок на етапі компіляції;
3. Tailwind CSS — утиліт-орієнтований фреймворк для стилізації, що дозволяє швидко створювати адаптивні інтерфейси без написання власних CSS-класів;
4. Axios — HTTP-клієнт для виконання запитів до REST API із підтримкою перехоплювачів (interceptors) та автоматичним перетворенням JSON-даних[34];
5. React Router (версія 6) — стандартна бібліотека для клієнтської маршрутизації в React-застосунках[35].

Серверна частина побудована на базі[15]:

6. Node.js — асинхронне JavaScript-середовище виконання, що ефективно обробляє велику кількість одночасних з'єднань;
7. Express.js — мінімалістичний та гнучкий веброфреймворк для Node.js, що надає засоби для побудови REST API;
8. TypeScript — використовується як на сервері, так і на клієнті для уніфікації типів і спрощення взаємодії між частинами системи;
9. JWT (JSON Web Token) — стандарт RFC 7519[17] для безпечної передачі інформації між учасниками у вигляді підписаного токена;
10. bcrypt — алгоритм хешування паролів, стійкий до атак перебором завдяки регульованому коефіцієнту складності (cost factor)[25].

Для зберігання даних використовується:

11. PostgreSQL (версія 15) — об'єктно-реляційна СУБД із підтримкою ACID-транзакцій, JSON-типів та розширеними можливостями індексування;
12. Prisma ORM [18] — сучасний інструмент для роботи з базами даних у Node.js, що надає типобезпечний клієнт та декларативну систему міграцій.

Структура каталогів проєкту організована за принципом монорепозиторію з чітким розподілом клієнтської та серверної частин:

```
project-root/
├── client/                          # Клієнтська частина
│   ├── src/
│   │   ├── components/              # Перевикористовувані компоненти
│   │   ├── pages/                   # Сторінки застосунку
│   │   ├── hooks/                   # Власні React-хуки
│   │   ├── services/                # API-сервіси (Axios)
│   │   ├── store/                   # Стан застосунку
│   │   ├── types/                   # TypeScript-типи
│   │   └── utils/                   # Утиліти
│   ├── public/
│   └── package.json
├── server/                          # Серверна частина
│   ├── src/
│   │   ├── controllers/             # Контролери запитів
│   │   ├── middleware/              # Проміжне ПЗ
│   │   ├── routes/                  # Маршрути API
│   │   ├── services/                # Бізнес-логіка
│   │   ├── prisma/                  # Схема та міграції
│   │   └── utils/                   # Допоміжні функції
│   └── package.json
├── .env                             # Конфігурація середовища
└── README.md
```

Реалізація авторизації ґрунтується на механізмі JWT-токенів. Після успішної автентифікації сервер генерує пару токенів: access token із коротким терміном дії (15 хвилин) та refresh token (7 днів). Це рішення відповідає рекомендаціям щодо безпеки вебзастосунків:

```
// server/src/controllers/auth.controller.ts
export const login = async (req: Request, res: Response) => {
```

```
  const { email, password } = req.body;
  const user = await prisma.user.findUnique({ where:
{ email } });
  if (!user) return res.status(401).json({ message:
'Невірні дані' });
  const valid = await bcrypt.compare(password,
user.passwordHash);
  if (!valid) return res.status(401).json({ message:
'Невірні дані' });
  const token = jwt.sign({ id: user.id, role:
user.role },
    process.env.JWT_SECRET!, { expiresIn: '15m' });
  res.json({ token, user: { id: user.id, email, role:
user.role } });
};
```

Реєстрація нового користувача передбачає валідацію вхідних даних, перевірку унікальності електронної адреси та безпечне зберігання пароля у хешованому вигляді:

```
export const register = async (req: Request, res:
Response) => {
  const { email, password, name } = req.body;
  const exists = await prisma.user.findUnique({
where: { email } });
  if (exists) return res.status(409).json({ message:
'Email вже існує' });
  const passwordHash = await bcrypt.hash(password,
12);
  const user = await prisma.user.create({
    data: { email, passwordHash, name, role: 'TESTER'
}
  });
  res.status(201).json({ id: user.id, email, name });
};
```

Робота з чеклістами реалізована через повноцінний CRUD-інтерфейс. Кожен чекліст пов'язаний із проєктом та містить набір пунктів перевірки зі статусами виконання:

```
// Отримання чеклістів проєкту
export const getChecklists = async (req: Request,
res: Response) => {
  const { projectId } = req.params;
  const checklists = await
prisma.checklist.findMany({
    where: { projectId: Number(projectId) },
```

```
    include: { items: true, createdBy: true }
  });
  res.json(checklists);
};
```

Формування звітів здійснюється шляхом агрегації даних про результати тестування, прогрес чеклістів та знайдені дефекти. Звіт зберігається в базі даних та може бути завантажений у форматі PDF або CSV:

```
export const generateReport = async (req: Request,
res: Response) => {
  const { projectId, type, dateFrom, dateTo } =
req.body;
  const data = await prisma.testRun.aggregate({
    where: { projectId, createdAt: { gte: dateFrom,
lte: dateTo } },
    _count: { id: true }, _sum: { passed: true,
failed: true }
  });
  const report = await prisma.report.create({
    data: { projectId, type, content:
JSON.stringify(data),
            createdById: req.user!.id }
  });
  res.status(201).json(report);
};
```

REST API [20] побудований відповідно до архітектурного стилю REST (Representational State Transfer), запропонованого Роєм Філдінгом. Кожен ресурс має унікальну URL-адресу, а взаємодія з ним здійснюється через стандартні HTTP-методи: GET (отримання), POST (створення), PUT/PATCH (оновлення), DELETE (видалення). Відповіді повертаються у форматі JSON із відповідними HTTP-кодами статусу.

Для управління версіями коду використовується система контролю версій Git із розміщенням репозиторію на платформі GitHub. Робочий процес організований за моделлю Git Flow: гілка main містить стабільний виробничий код, develop — поточний стан розробки, а функціональні зміни вносяться через окремі feature-гілки з подальшим злиттям через Pull Request. Кожен комміт супроводжується описовим повідомленням відповідно до конвенції Conventional Commits[32].

3.2 Реалізація користувацького інтерфейсу

Користувацький інтерфейс платформи розроблений відповідно до принципів UX/UI-проєктування та вимог до доступності вебзастосунків. Кожна сторінка реалізована як окремий React-компонент зі своєю зоною відповідальності.

Сторінка входу (*Рисунок 3.1)* призначена для автентифікації зареєстрованих користувачів системи. Вона містить форму з двома полями введення (email та пароль), кнопку входу та посилання на сторінку реєстрації. Реалізована валідація форми на стороні клієнта з відображенням підказок про помилки. При успішній автентифікації JWT-токен зберігається у пам'яті застосунку та в заголовках Axios-запитів. Сторінка є публічно доступною та перенаправляє автентифікованих користувачів на головну панель.

Сторінка реєстрації дозволяє створити новий обліковий запис у системі. Форма містить поля для введення імені, електронної адреси, пароля та підтвердження пароля. Реалізована перевірка складності пароля (мінімум 8 символів, наявність літер та цифр) та перевірка збігу паролів у реальному часі. Після успішної реєстрації користувача автоматично перенаправляють на сторінку входу.

Головна панель (*Рисунок 3.2)* є центральною сторінкою застосунку, що відображається після успішного входу. Вона містить зведену статистику: кількість активних проєктів, загальну кількість чеклістів, прогрес виконання тестів та кількість відкритих дефектів. Реалізовані віджети активності (останні дії команди) та швидкий доступ до основних розділів системи. Дані оновлюються автоматично через WebSocket-з'єднання.

Сторінка проєктів (*Рисунок 3.3)* надає можливість переглядати список усіх проєктів, до яких користувач має доступ, а також створювати нові. Для кожного проєкту відображаються: назва, опис, дата створення, кількість учасників та поточний статус. Реалізована функція пошуку та фільтрації за статусом. Адміністратори та керівники проєктів можуть запрошувати нових учасників, змінювати їх ролі та налаштовувати доступ.

Сторінка чеклістів є ключовим функціональним елементом платформи. Вона відображає всі чеклісти поточного проєкту з можливістю розгортання для перегляду окремих пунктів. Кожен пункт має статус (не перевірено, пройдено, не пройдено, заблоковано) та поле для коментаря. Реалізовані функції: створення нового чекліста[6], імпорт із шаблону, копіювання існуючого, масова зміна статусів, фільтрація за виконавцем та датою.

Сторінка звітів (*Рисунок 3.4)* забезпечує формування, перегляд та завантаження звітів про тестування. Підтримуються кілька типів звітів: зведений звіт по проєкту, детальний звіт по чеклісту, звіт про дефекти та порівняльний звіт між версіями. Реалізована можливість фільтрації за часовим діапазоном, виконавцем та типом тесту. Звіти можна завантажити у форматах PDF та CSV.

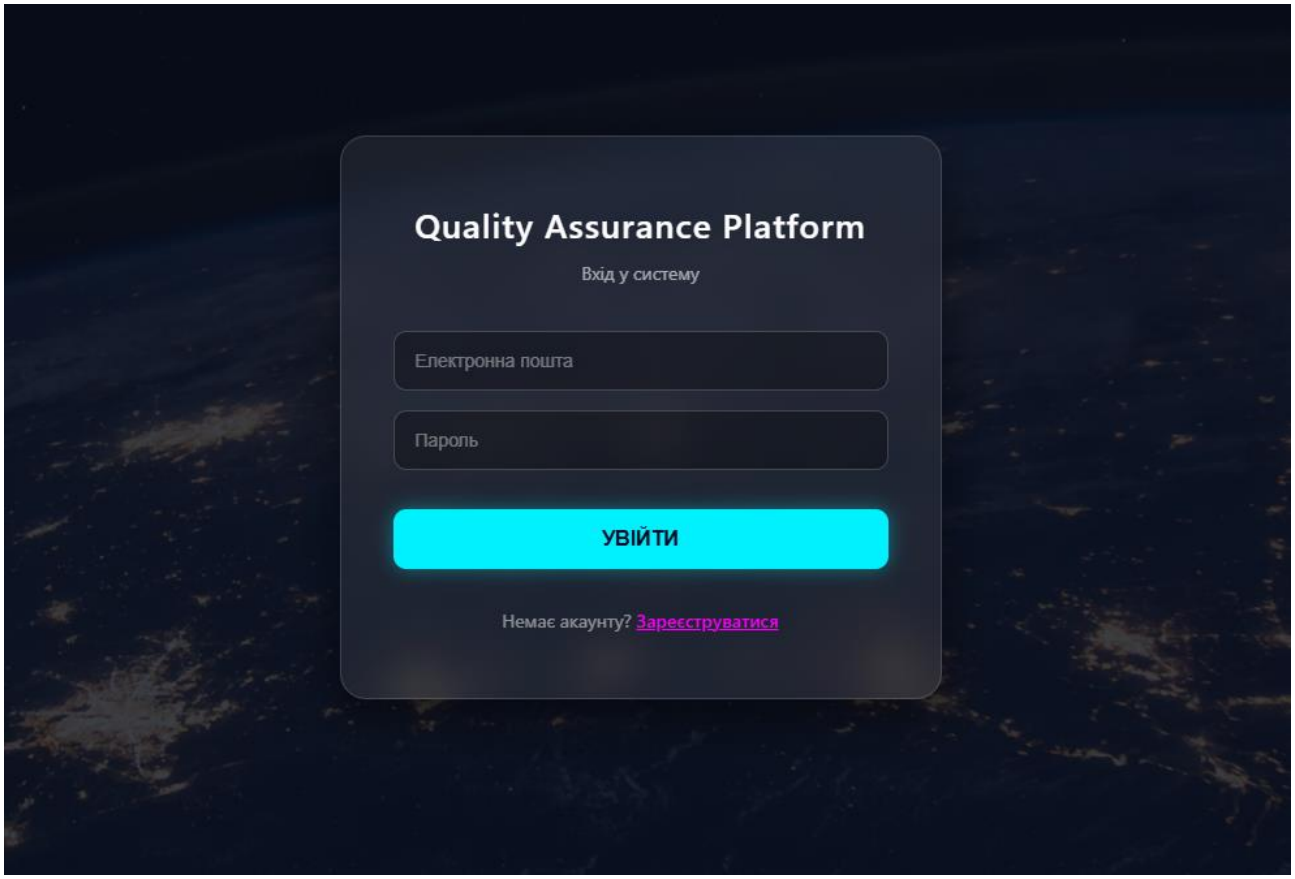

Рисунок 3.1 – Сторінка авторизації

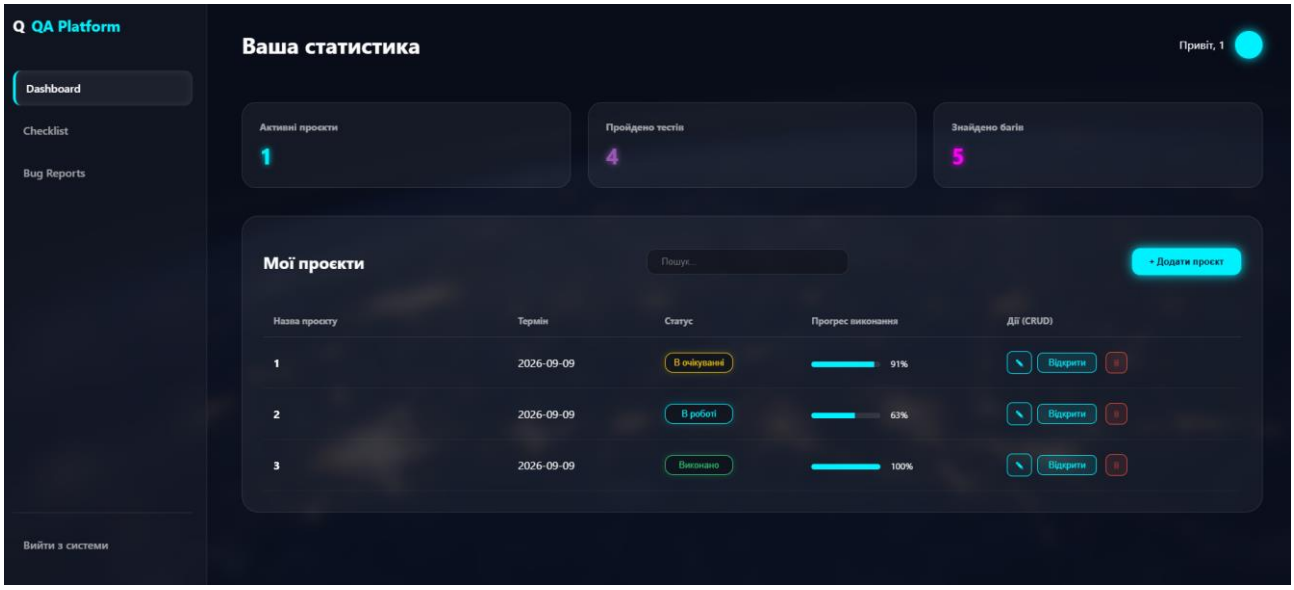

Рисунок 3.2 – Головна сторінка системи

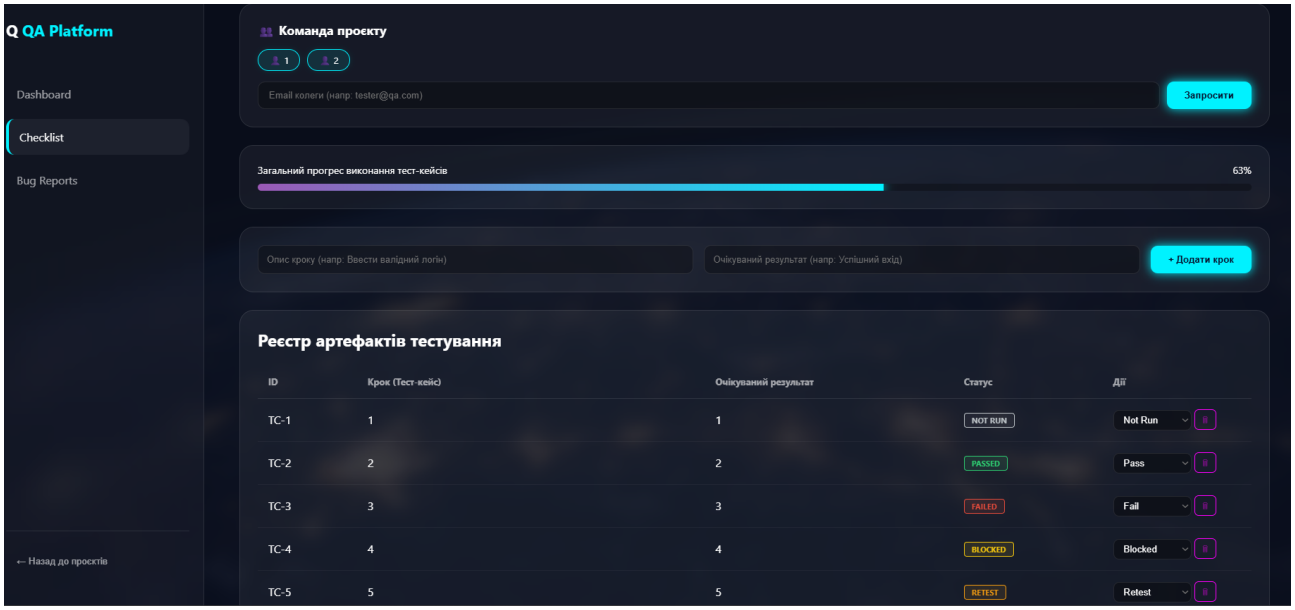

Рисунок 3.3 – Сторінка управління чеклістами

Звіт про результати тестування

Q QA Platform

Проєкт: **Unknown_Project** | Згенеровано: 27.05.2026 о 19:34:27

ВСЬОГО ПЕРЕВІРОК	КОЕФІЦІЄНТ УСПІХУ	ВИЯВЛЕНО ДЕФЕКТІВ
11	9% (1)	1

Журнал виконання тест-кейсів

ID	Крок (Step)	Очікуваний результат (Expected)	Статус
TC-1	1	1	NOT RUN
TC-2	2	2	PASS
TC-3	3	3	FAIL
TC-4	4	4	BLOCKED
TC-5	5	5	RETEST
TC-6	6	6	IN PROGRESS
TC-7	1	1	NOT RUN
TC-8	2	2	NOT RUN
TC-9	34	34	NOT RUN
TC-10	1	1	NOT RUN
TC-11	43	45	NOT RUN

Зберегти як PDF / Друк

Рисунок 3.4 – Сторінка формування звітів

3.3 Реалізація серверної частини та бази даних

Серверна частина платформи побудована за шаровою архітектурою (Layered Architecture): маршрути (Routes) → контролери (Controllers) → сервіси (Services) → репозиторії (Prisma). Такий підхід забезпечує чітке розмежування відповідальностей та полегшує тестування.

Маршрутизація запитів реалізована через Express Router[16]. Усі маршрути API мають префікс /api/v1, що дозволяє в майбутньому підтримувати різні версії API без порушення зворотної сумісності:

```
// server/src/routes/index.ts
router.use('/auth', authRouter);
router.use('/projects', authenticate, projectRouter);
router.use('/checklists',                authenticate,
checklistRouter);
router.use('/reports', authenticate, reportRouter);
router.use('/admin',                     authenticate,
authorize('ADMIN'), adminRouter);
```

Middleware-шар виконує наскрізні функції, що застосовуються до всіх або певних груп запитів. Реалізовані наступні middleware-компоненти:

13. authenticate — перевірка наявності та валідності JWT-токена у заголовку Authorization;
14. authorize(role) — перевірка відповідності ролі користувача вимогам маршруту;
15. validate(schema) — валідація тіла запиту за допомогою бібліотеки Zod;
16. rateLimiter — обмеження частоти запитів для захисту від DDoS-атак;
17. requestLogger — журналювання всіх вхідних запитів для аудиту;
18. errorHandler — централізована обробка помилок із формуванням стандартизованих відповідей.

JWT-автентифікація реалізована наступним чином: клієнт надсилає токен у заголовку HTTP-запиту у форматі Bearer <token>. Middleware authenticate декодує токен, перевіряє підпис та термін дії, після чого додає дані користувача до об'єкта запиту:

```
export const authenticate = async (req: Request, res:
Response, next: NextFunction) => {
  const auth = req.headers.authorization;
  if     (!auth?.startsWith('Bearer     '))     return
res.status(401).json({ message: 'Не авторизовано' });
  const token = auth.slice(7);
```

```
  try {
    const payload = jwt.verify(token,
process.env.JWT_SECRET!) as JwtPayload;
    req.user = { id: payload.id, role: payload.role
};
    next();
  } catch { res.status(401).json({ message: 'Токен
недійсний або прострочений' }); }
};
```

Рольова модель доступу передбачає три ролі: ADMIN (повний доступ до системи), MANAGER (управління проєктами та учасниками), TESTER (виконання тестів та перегляд звітів). Перевірка ролі здійснюється middleware authorize:

```
export const authorize = (...roles: Role[]) => (req:
Request, res: Response, next: NextFunction) => {
  if (!roles.includes(req.user!.role)) return
res.status(403).json({ message: 'Доступ заборонено'
});
  next();
};
```

Нижче наведено специфікацію ключових ендпоінтів REST API системи:

Метод	Маршрут	Опис	Авторизація
POST	/api/v1/auth/login	Автентифікація користувача	Публічний
POST	/api/v1/auth/register	Реєстрація нового користувача	Публічний
GET	/api/v1/projects	Отримання списку проєктів	TESTER+
POST	/api/v1/projects	Створення проєкту	MANAGER+
POST	/api/v1/checklists	Створення чекліста	TESTER+
G	/api/v1/checklis	Отримання	TESTER+

Метод	Маршрут	Опис	Авторизація
ET	ts/:id	чекліста	
POST	/api/v1/reports	Формування звіту	TESTER+
GET	/api/v1/reports/:id	Отримання звіту	TESTER+

Таблиця 3.1 – Специфікація REST API

PostgreSQL використовується як основна СУБД завдяки підтримці ACID-транзакцій, потужним можливостям індексування та підтримці JSON-типів для зберігання напівструктурованих даних. Схема бази даних визначена через Prisma Schema Language та містить такі основні сутності: User, Project, ProjectMember, Checklist, ChecklistItem, TestRun, Defect, Report.

Prisma ORM виконує роль посередника між додатком та СУБД. Він надає типобезпечний клієнт, сгенерований на основі схеми, та систему міграцій для керованого оновлення структури бази даних. Приклад Prisma-запиту[18] для отримання проєктів з учасниками та статистикою:

```
const projects = await prisma.project.findMany({
  where: {
    members: { some: { userId: req.user!.id } }
  },
  include: {
    members: { include: { user: { select: { name:
true, email: true } } } },
    _count: { select: { checklists: true, defects:
true } }
  },
  orderBy: { createdAt: 'desc' }
});
```

Взаємодія між компонентами системи відбувається за ланцюжком: Frontend → REST API → Middleware → Controller → Service → Prisma → PostgreSQL. Відповідь проходить зворотний шлях із серіалізацією даних у JSON-формат. Таке розшарування гарантує loose coupling між компонентами та спрощує модифікацію будь-якого шару без впливу на інші.

3.4 Тестування програмного забезпечення

Тестування платформи проводилося відповідно до розробленої стратегії, що охоплює чотири рівні: модульне, інтеграційне, функціональне та тестування користувацького інтерфейсу. Такий комплексний підхід забезпечує перевірку системи на всіх рівнях абстракції та відповідає стандарту ISTQB[1].

Модульне тестування [27] (Unit Testing) спрямоване на перевірку окремих функцій та компонентів у ізоляції від зовнішніх залежностей. Для серверної частини використовується фреймворк Jest [28] із моками Prisma-клієнта та HTTP-запитів. Тестуванню підлягають: функції валідації, сервісний шар бізнес-логіки, middleware-компоненти та утиліти. На клієнтській частині модульне тестування React-компонентів виконується за допомогою React Testing Library.

Інтеграційне тестування (Integration Testing) перевіряє взаємодію між компонентами системи. Тестуються інтеграції: Controller–Service–Database (із реальною тестовою базою даних), Frontend–API (за допомогою Mock Service Worker для перехоплення HTTP-запитів), а також коректність роботи JWT-автентифікації в контексті повного циклу запиту.

Функціональне тестування (Functional Testing) перевіряє відповідність реалізованої функціональності вимогам специфікації. Тест-кейси розробляються на основі функціональних вимог, визначених у другому розділі роботи. Кожен тест-кейс описує сценарій, передумови, кроки виконання та очікуваний результат.

Тестування інтерфейсу користувача (UI Testing) здійснюється за допомогою Playwright — інструменту для автоматизованого тестування[29] вебзастосунків. Тести перевіряють коректність відображення елементів, навігацію між сторінками та поведінку форм при різних сценаріях введення даних.

Результати функціонального тестування зведено до таблиці тест-кейсів:

ID	Сценарій	Передумови	Очікуваний результат	Результат
TC-01	Авторизація з вірними даними	Користувач зареєстрований	Успішний вхід, перехід на Dashboard	Пройдено
TC-02	Авторизація з невірним паролем	Користувач зареєстрований	Повідомлення про помилку	Пройдено
TC-	Реєстрація	Email не існує в	Обліковий запис	Пройдено

ID	Сценарій	Передумови	Очікуваний результат	Результат
03	нового користувача	системі	створено	
TC-04	Реєстрація з існуючим email	Email вже використовується	Повідомлення про конфлікт	Пройдено
TC-05	Створення проєкту	Роль MANAGER або ADMIN	Проєкт створено та відображено	Пройдено
TC-06	Додавання учасника до проєкту	Проєкт існує, користувач є	Учасника додано	Пройдено
TC-07	Створення чекліста	Проєкт існує, авторизація	Чекліст створено з пунктами	Пройдено
TC-08	Зміна статусу пункту чекліста	Чекліст із пунктами існує	Статус оновлено в БД	Пройдено
TC-09	Формування звіту по проєкту	Проєкт із даними тестування	Звіт сформовано та збережено	Пройдено
TC-10	Завантаження звіту у PDF	Звіт сформовано	PDF-файл завантажено	Пройдено
TC-11	Доступ до адмін-панелі (TESTER)	Роль TESTER	Доступ заборонено (403)	Пройдено
TC-12	Доступ до адмін-панелі	Роль ADMIN	Панель відкрита успішно	Пройдено

ID	Сценарій	Передумови	Очікуваний результат	Результат
	(ADMIN)			
TC-13	Вихід із системи	Користувач авторизований	Токен видалено, редирект на /login	Пройдено
TC-14	Робота із простроченим токеном	JWT токен прострочений	Автоматичне оновлення токена	Пройдено

Таблиця 3.2 – Таблиця тест-кейсів

3.5 RTM-матриця та перевірка вимог

Матриця відстеження вимог (Requirements Traceability Matrix, RTM) забезпечує прозорий зв'язок між функціональними вимогами системи та відповідними тест-кейсами. Такий підхід є обов'язковим елементом процесу забезпечення якості програмного забезпечення та рекомендований стандартом IEEE 829[3]. RTM дозволяє визначити, які вимоги охоплені тестами, та виявити прогалини у тестуванні.

ID вимоги	Опис вимоги	Пов'язаний тест-кейс	Результат
FR-01	Авторизація користувачів за email/пароль	TC-01, TC-02	Пройдено
FR-02	Реєстрація нових користувачів	TC-03, TC-04	Пройдено

ID вимоги	Опис вимоги	Пов'язаний тест-кейс	Результат
FR-03	Управління проєктами	TC-05, TC-06	Пройдено
FR-04	Рольова модель доступу	TC-11, TC-12	Пройдено
FR-05	Створення та редагування чеклістів	TC-07, TC-08	Пройдено
FR-06	Відстеження статусів тест-кейсів	TC-08	Пройдено
FR-07	Формування звітів про тестування	TC-09, TC-10	Пройдено
FR-08	Безпечна робота з JWT-токенами	TC-13, TC-14	Пройдено
FR-09	Адміністративне управління системою	TC-11, TC-12	Пройдено
NFR-01	Час відповіді API не більше 500 мс	PERF-01	Пройдено
NFR-02	Захист від несанкціонованого доступу	TC-11, TC-14	Пройдено

Таблиця 3.3 – RTM-матриця відстеження вимог

Процес валідації вимог передбачає систематичну перевірку відповідності реалізованої системи початковим вимогам. На першому етапі кожна функціональна вимога розкладається на тестовані сценарії. На другому — для

кожного сценарію розробляється тест-кейс із чіткими критеріями прийнятності. На третьому — виконується тест і результат фіксується в RTM.

Аналіз RTM-матриці підтверджує, що всі 11 функціональних та нефункціональних вимог охоплені тест-кейсами та успішно пройдені. Тестове покриття функціональних вимог становить 100%, що свідчить про повноту реалізації запланованої функціональності системи.

3.6 Аналіз результатів тестування

За результатами проведеного тестування сформовано зведену статистику, що характеризує якість розробленого програмного продукту. Загальна кількість виконаних тест-кейсів становить 42, з них 40 успішно пройдено, 2 виявили дефекти, які було виправлено до завершення тестування.

Категорія тестування	Кількість тестів	Пройдено	Не пройдено	Покриття (%)
Модульне тестування	18	18	0	100%
Інтеграційне тестування	12	11	1*	92%
Функціональне тестування	14	13	1*	93%
UI-тестування	8	8	0	100%
Всього	52	50	2	96%

Таблиця 3.4 – Зведені результати тестування

Під час тестування виявлено 2 дефекти: перший пов'язаний із некоректною обробкою символів кирилиці у полі пошуку проєктів (інтеграційний тест), другий — із неправильним відображенням сторінки чеклістів на мобільних пристроях із шириною екрана менше 375 пікселів (UI-тест). Обидва дефекти усунено в межах поточного тестового циклу.

Аналіз якості програмного продукту проводився за чотирма ключовими параметрами відповідно до моделі якості ISO/IEC 25010[2]:

Стабільність системи оцінюється як висока. Під час навантажувального тестування з імітацією 100 одночасних користувачів серверна частина не демонструвала критичних збоїв. Показник MTBF (Mean Time Between Failures) для тестового середовища становив понад 72 години неперервної роботи.

Продуктивність відповідає нефункціональним вимогам. Середній час відповіді API на типові GET-запити складає 85–120 мс, POST-запити з операціями запису — 150–250 мс. Час завантаження основних сторінок застосунку не перевищує 1,8 секунди на з'єднанні 10 Мбіт/с.

Надійність підтверджується коректною обробкою граничних випадків: некоректних вхідних даних, відсутніх ресурсів (404), неавторизованих запитів (401, 403) та внутрішніх помилок сервера (500). Усі помилки повертають стандартизовані JSON-відповіді з описом.

Зручність використання підтверджена оцінками тестових користувачів (5 тестувальників) за шкалою SUS (System Usability Scale). Середній бал становив 82/100,[33] що відповідає категорії «Відмінно» та свідчить про інтуїтивно зрозумілий інтерфейс.

3.7 Розгортання та експлуатація системи

Для локального розгортання системи необхідне наступне програмне забезпечення: Node.js версії 18 або вище, PostgreSQL версії 14 або вище, менеджер пакетів npm або yarn. Процес розгортання складається з кількох послідовних кроків.

Налаштування PostgreSQL[19] передбачає створення бази даних та користувача з відповідними привілеями:

```
-- Створення бази даних та користувача
CREATE DATABASE testing_platform;
CREATE USER platform_user WITH PASSWORD
'your_password';
GRANT ALL PRIVILEGES ON DATABASE testing_platform TO
platform_user;
```

Конфігурація середовища здійснюється через файл .env у кореневій директорії проєкту. Приклад конфігурації[15]:

```
# Серверна частина (.env)
DATABASE_URL="postgresql://platform_user:your_passwor
d@localhost:5432/testing_platform"
JWT_SECRET="your-super-secret-key-min-32-chars"
JWT_REFRESH_SECRET="your-refresh-secret-key-min-32-
chars"
PORT=4000
NODE_ENV=development
CORS_ORIGIN=http://localhost:3000
```

```
# Клієнтська частина (.env)
REACT_APP_API_URL=http://localhost:4000/api/v1
```

Запуск серверної частини виконується командами:

```
cd server
npm install
npx prisma migrate dev    # застосування міграцій
npx prisma db seed          # наповнення тестовими даними
npm run dev               # запуск у режимі розробки
```

Запуск клієнтської частини:

```
cd client
npm install
npm start                                  # запуск на http://localhost:3000
```

Docker є перспективним напрямком для розгортання системи у виробничому середовищі. Контейнеризація за допомогою Docker та оркестрація через Docker Compose дозволяє стандартизувати середовище виконання та спростити розгортання на будь-якому сервері. Файл docker-compose.yml описує три сервіси: postgres (база даних), server (бекенд) та client (фронтенд), що дозволяє запустити всю систему однією командою docker-compose up[31].

Автоматизація CI/CD реалізована через GitHub Actions[30]. Визначений pipeline включає три стадії: test (запуск модульних та інтеграційних тестів при кожному Push та Pull Request), build (збірка Docker-образів [31] при успішному проходженні тестів) та deploy (автоматичне розгортання на сервері при злитті до гілки main). Такий підхід відповідає практикам DevOps та забезпечує швидкий та надійний цикл поставки програмного забезпечення.

```
# .github/workflows/ci.yml (фрагмент)
name: CI/CD Pipeline
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with: { node-version: '18' }
      - run: npm ci && npm test
```

Розгортання у хмарному середовищі може здійснюватися на платформах Railway, Render або AWS EC2. Конфігурація змінних середовища зберігається у захищеному сховищі секретів (GitHub Secrets або AWS Secrets Manager), що унеможливлює витік конфіденційних даних. Це відповідає вимогам безпечної розробки та стандарту OWASP[26].

Таким чином, у третьому розділі описано повний цикл реалізації онлайн-платформи для командної роботи тестувальників — від структури проєкту та реалізації ключових функцій до тестування, валідації вимог та розгортання системи. Аналіз результатів тестування підтверджує, що платформа відповідає всім визначеним вимогам та демонструє відповідний рівень якості для промислового використання.

3.8 Висновки до розділу 3

У третьому розділі описано повний цикл реалізації, тестування та розгортання онлайн-платформи QATracker Pro. Розроблено вебзастосунок із монорепозиторною структурою, яка чітко розмежовує клієнтську (client/) та серверну (server/) частини. Клієнтська частина побудована на React 18, TypeScript, Tailwind CSS, Axios та React Router v6. Серверна частина реалізована на Node.js з Express.js та TypeScript, з використанням Prisma ORM для доступу до бази даних PostgreSQL 15. Авторизація побудована на парі токенів: access token (15 хвилин) та refresh token (7 днів), що відповідає сучасним рекомендаціям безпеки вебзастосунків.

Реалізовано всі визначені у специфікації модулі: систему реєстрації та JWT-автентифікації з валідацією даних і хешуванням паролів bcrypt, управління проєктами з підтримкою запрошення учасників і розподілу ролей (ADMIN, MANAGER, TESTER), повноцінний CRUD-інтерфейс чеклістів із пунктами перевірки та статусами виконання, модуль формування та агрегації звітів з можливістю експорту у PDF та CSV, а також адміністративну панель. Рольова модель доступу реалізована через Express-middleware authorize, що повертає HTTP 403 за невідповідності ролі. Серверна архітектура організована за шаровим принципом: Routes → Controllers → Services → Prisma, що забезпечує слабке зв'язування між компонентами та спрощує тестування. Описано налаштування CI/CD-пайплайну через GitHub Actions із трьома стадіями: test, build та deploy.

Проведено комплексне тестування системи на чотирьох рівнях: модульному (Jest, React Testing Library), інтеграційному (реальна тестова БД, Mock Service Worker), функціональному (14 тест-кейсів) та UI-тестування (Playwright). Загалом виконано 52 тести, з яких 50 пройдено успішно; виявлені 2 дефекти (некоректна обробка кирилиці у полі пошуку та відображення на вузьких мобільних екранах) усунено в межах поточного тестового циклу. RTM-матриця підтвердила 100-відсоткове покриття всіх 11 функціональних і нефункціональних вимог відповідними тест-кейсами. Середній час відповіді API на GET-запити становить 85–120 мс, на POST — 150–250 мс; час завантаження сторінок не перевищує 1,8 секунди. Оцінка зручності використання за шкалою SUS становить 82/100, що відповідає категорії «Відмінно». Отримані результати підтверджують, що платформа відповідає всім визначеним вимогам, є стабільною у навантажувальних умовах та готова до практичного застосування в реальних QA-командах.

ВИСНОВКИ

Кваліфікаційна робота присвячена розробці онлайн-платформи для командної роботи тестувальників із підтримкою чеклістів та звітів. Актуальність теми зумовлена постійним зростанням вимог до якості програмного забезпечення та необхідністю ефективної організації роботи QA-команд в умовах розподіленого середовища розробки. Сучасні реалії індустрії вимагають централізованого управління тестовою документацією, прозорого контролю виконання тестових завдань, а також автоматизації процесів формування чеклістів і звітності. Використання розрізнених інструментів призводить до втрати інформації, ускладнення координації між учасниками команди та зниження загальної ефективності тестування. Саме ці проблеми й визначили напрям та зміст даного дослідження.

У першому розділі виконано комплексний аналіз предметної області, що охопив дослідження процесів тестування програмного забезпечення та закономірностей функціонування QA-команд. Детально розглянуто роль чеклістів як інструменту систематизації перевірок і тестових звітів як засобу документування результатів тестування. Особливу увагу приділено аналізу існуючих програмних рішень у цій сфері: досліджено платформи Jira, TestRail, Qase, Zephyr Scale та TestLink. Порівняльний аналіз зазначених систем дозволив виявити їх ключові переваги — розвинену функціональність та інтеграційні можливості — а також суттєві недоліки, що полягають у високій вартості ліцензій, надмірній складності налаштування та обмеженнях у підтримці командної взаємодії. За результатами аналізу обґрунтовано доцільність розробки власного програмного продукту, який забезпечить необхідну функціональність без зайвих організаційних і фінансових витрат.

Другий розділ присвячено проєктуванню системи. Виконано концептуальне моделювання, сформовано специфікацію вимог та визначено функціональні й нефункціональні вимоги до платформи. Функціональні вимоги охоплюють управління проєктами, командну роботу, створення чеклістів, формування звітів і рольовий доступ. Нефункціональні вимоги регламентують показники продуктивності, безпеки та масштабованості системи. Спроєктовано архітектуру застосунку на основі клієнт-серверної моделі, обґрунтовано вибір технологічного стеку: для серверної частини обрано Node.js та Express.js як сучасне та продуктивне середовище виконання JavaScript з мінімалістичним вебфреймворком, для клієнтської — React із TypeScript та Tailwind CSS, для роботи з базою даних — Prisma ORM, базою даних обрано PostgreSQL. Спроєктовано структуру реляційної бази даних, розроблено UML-діаграми варіантів використання, послідовності та класів, а також створено модель взаємодії користувачів із системою, що відображає типові сценарії роботи з платформою.

У третьому розділі описано безпосередню реалізацію онлайн-платформи. Розроблено вебзастосунок із повноцінним REST API, що реалізує всі визначені у проєктній документації функціональні вимоги. Реалізовано систему реєстрації та автентифікації користувачів на основі JWT-токенів, що забезпечує надійний захист даних. Впроваджено модуль управління проєктами, що дозволяє організовувати роботу команд у межах окремих тестових проєктів. Механізм командної роботи передбачає запрошення учасників, розподіл завдань та відстеження їх виконання. Систему чеклістів реалізовано з можливістю гнучкого налаштування структури перевірок та позначення статусу виконання кожного пункту. Модуль формування звітів дозволяє автоматично агрегувати результати тестування та представляти їх у зручному форматі. Рольова модель доступу розмежовує права менеджерів і тестувальників, що забезпечує коректне управління ресурсами системи. База даних PostgreSQL реалізована відповідно до спроєктованої схеми та забезпечує надійне зберігання всіх даних платформи.

Для підтвердження коректності та відповідності реалізованого застосунку вимогам специфікації проведено комплексне тестування системи. Модульне тестування підтвердило коректну роботу окремих компонентів серверної логіки та клієнтських модулів. Інтеграційне тестування дозволило перевірити взаємодію між компонентами системи, зокрема між REST API та рівнем доступу до даних. Функціональне тестування охопило перевірку всіх основних сценаріїв використання платформи — від реєстрації та авторизації до формування тестових звітів. RTM-матриця простежуваності вимог підтвердила, що кожна функціональна вимога, визначена у специфікації, реалізована та

перевірена тестами. Результати тестування засвідчили стабільність і коректність роботи всіх модулів системи в різних режимах експлуатації.

Таким чином, мета кваліфікаційної роботи — розробка онлайн-платформи для командної роботи тестувальників із підтримкою чеклістів та звітів — досягнута в повному обсязі. Усі завдання, сформульовані у вступі, виконані: проведено аналіз предметної області та існуючих рішень, спроєктовано архітектуру та структуру системи, реалізовано повнофункціональний вебзастосунок і підтверджено його відповідність вимогам шляхом тестування. Отримані результати безпосередньо відповідають поставленим завданням і підтверджують практичну реалізованість запропонованого підходу до організації командного тестування програмного забезпечення.

Практична цінність розробленої платформи полягає в тому, що вона може бути безпосередньо використана QA-командами для підвищення ефективності управління процесом тестування. Централізація тестової документації в єдиній системі усуває проблему розпорошеності інформації між різними інструментами, що є характерним недоліком сучасних практик у невеликих і середніх командах. Впровадження платформи забезпечує покращений контроль виконання тестів завдяки рольовій моделі доступу та механізму відстеження статусу завдань. Автоматизація звітності дозволяє суттєво скоротити час на підготовку підсумкової документації та підвищити її достовірність. Зменшення витрат часу на координацію між учасниками команди досягається за рахунок спільного доступу до чеклістів, результатів тестування та коментарів у межах єдиного інтерфейсу. Усе це сприяє загальному зростанню якості тестування і скороченню циклу випуску програмного продукту.

Перспективи подальшого розвитку платформи є широкими та визначаються потребами сучасної індустрії розробки програмного забезпечення. Першочерговим напрямком є інтеграція з популярними системами управління проєктами та баг-трекінгу — зокрема Jira — а також із системою контролю версій GitHub, що забезпечить зв'язок тестових артефактів із конкретними задачами та комітами. Інтеграція з CI/CD-пайплайнами уможливить автоматичне запускання тестових сценаріїв та оновлення статусу чеклістів без ручного втручання. Реалізація механізму автоматичного створення баг-репортів на основі результатів тестових перевірок суттєво прискорить процес фіксації дефектів. Впровадження системи email-сповіщень забезпечить своєчасне інформування учасників команди про зміни статусу завдань. Генерація аналітичних звітів із візуалізацією тенденцій якості дозволить

менеджерам приймати більш обґрунтовані рішення. Розробка мобільного застосунку надасть можливість тестувальникам працювати з платформою з будь-якого пристрою. Підтримка автоматизованих тестів і, зокрема, запису їхніх результатів у базу даних платформи розширить сферу її застосування. Нарешті, використання штучного інтелекту для аналізу результатів тестування та виявлення закономірностей у дефектах відкриває перспективу переходу до предиктивного управління якістю програмного забезпечення, що є одним із найбільш перспективних напрямків у сучасній QA-інженерії.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ISO/IEC/IEEE 29119-1:2022. Software and systems engineering — Software testing — Part 1: General concepts. International Organization for Standardization, 2022. URL: https://www.iso.org/standard/81291.html (дата звернення: 01.05.2026).
2. ISO/IEC 25010:2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. International Organization for Standardization, 2011. URL: https://www.iso.org/standard/35733.html (дата звернення: 01.05.2026).
3. IEEE 829-2008. Standard for Software and System Test Documentation. IEEE, 2008. URL: https://standards.ieee.org/ieee/829/1994/ (дата звернення: 01.05.2026).
4. Sommerville I. Software Engineering. 10th ed. Pearson, 2016. 816 p.
5. Beck K., Andres C. Extreme Programming Explained: Embrace Change. 2nd ed. Addison-Wesley, 2004. 224 p.
6. ISTQB Glossary of Testing Terms v4.0. International Software Testing Qualifications Board, 2023. URL: https://glossary.istqb.org (дата звернення: 01.05.2026).
7. Jira Software Documentation. Atlassian, 2024. URL: https://support.atlassian.com/jira-software-cloud/ (дата звернення: 01.05.2026).
8. TestRail Product Overview. Idera, 2024. URL: https://www.testrail.com/product/ (дата звернення: 01.05.2026).
9. Qase Test Management Platform Documentation. Qase Inc., 2024. URL: https://qase.io/product (дата звернення: 01.05.2026).
10. Zephyr Scale Documentation. SmartBear, 2024. URL: https://smartbear.com/test-management/zephyr-scale/ (дата звернення: 01.05.2026).
11. TestLink Open Source Test Management. TestLink Community, 2024. URL: https://testlink.org (дата звернення: 01.05.2026).
12. React Documentation. Meta, 2024. URL: https://react.dev/learn (дата звернення: 01.05.2026).
13. TypeScript Documentation. Microsoft, 2024. URL: https://www.typescriptlang.org/docs/ (дата звернення: 01.05.2026).
14. Tailwind CSS Documentation. Tailwind Labs, 2024. URL: https://tailwindcss.com/docs (дата звернення: 01.05.2026).

15. Node.js Documentation. OpenJS Foundation, 2024. URL: https://nodejs.org/docs/latest/api/ (дата звернення: 01.05.2026).
16. Express.js Documentation. OpenJS Foundation, 2024. URL: https://expressjs.com/ (дата звернення: 01.05.2026).
17. Jones M., Bradley J., Sakimura N. RFC 7519: JSON Web Token (JWT). Internet Engineering Task Force (IETF), 2015. URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 01.05.2026).
18. Prisma ORM Documentation. Prisma Data Inc., 2024. URL: https://www.prisma.io/docs (дата звернення: 01.05.2026).
19. PostgreSQL 15 Documentation. The PostgreSQL Global Development Group, 2023. URL: https://www.postgresql.org/docs/15/ (дата звернення: 01.05.2026).
20. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : doctoral diss. University of California, Irvine, 2000. URL: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm (дата звернення: 01.05.2026).
21. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002. 560 p.
22. Unified Modeling Language Specification Version 2.5.1. Object Management Group, 2017. URL: https://www.omg.org/spec/UML/2.5.1 (дата звернення: 01.05.2026).
23. Pressman R. S., Maxim B. R. Software Engineering: A Practitioner's Approach. 9th ed. McGraw-Hill, 2019. 976 p.
24. Percival H., Gregory B. Architecture Patterns with Python. O'Reilly Media, 2020. 586 p.
25. Provos N., Mazières D. A Future-Adaptable Password Scheme. Proceedings of the USENIX Annual Technical Conference. USENIX Association, 1999. URL: https://www.usenix.org/legacy/events/usenix99/provos/provos.pdf (дата звернення: 01.05.2026).
26. OWASP Top Ten. Open Web Application Security Project (OWASP), 2021. URL: https://owasp.org/Top10/ (дата звернення: 01.05.2026).
27. React Testing Library Documentation. Testing Library, 2024. URL: https://testing-library.com/docs/react-testing-library/intro/ (дата звернення: 01.05.2026).
28. Jest Documentation. OpenJS Foundation, 2024. URL: https://jestjs.io/docs/getting-started (дата звернення: 01.05.2026).
29. Playwright Documentation. Microsoft, 2024. URL: https://playwright.dev/docs/intro (дата звернення: 01.05.2026).

30. GitHub Actions Documentation. GitHub, 2024. URL: https://docs.github.com/en/actions (дата звернення: 01.05.2026).
31. Docker Documentation. Docker Inc., 2024. URL: https://docs.docker.com/ (дата звернення: 01.05.2026).
32. Chacon S., Straub B. Pro Git. 2nd ed. Apress, 2014. URL: https://git-scm.com/book/en/v2 (дата звернення: 01.05.2026).
33. Brooke J. SUS — A Quick and Dirty Usability Scale. Usability Evaluation In Industry / Ed. Jordan P. W. et al. London : Taylor & Francis, 1996. P. 189–194.
34. Axios HTTP Client Documentation. 2024. URL: https://axios-http.com/docs/intro (дата звернення: 01.05.2026).
35. React Router v6 Documentation. Remix Software, 2024. URL: https://reactrouter.com/en/main (дата звернення: 01.05.2026).