ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ

КАФЕДРА ПРОГРАМНИХ ЗАСОБІВ І ТЕХНОЛОГІЙ

Пояснювальна записка

до кваліфікаційної роботи
магістра
(освітній рівень)

на тему: «Дослідження методів тестування веб-додатків для покращення їх якості»

Виконав: студент групи 6ПР1
спеціальності
121 - «Інженерія програмного забезпечення»
(шифр і назва спеціальності)

Рогатовська Анастасія Андріївна
(прізвище та ініціали)
Керівник д.т.н., проф. Жарікова М.В.
(прізвище та ініціали)
Рецензент к.т.н. доцент Козел В.М.
(прізвище та ініціали)

Хмельницький - 2025

Херсонський національний технічний університет
(повне найменування вищого навчального закладу)

Факультет, відділення Інформаційних технологій та дизайну
Кафедра Програмних засобів і технологій
Освітній рівень магістр
(шифр і назва)
Напрям підготовки ОПП - Програмне забезпечення систем
Спеціальність 121 – Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗіТ
к.т.н. доц. О.Є. Огнєва
“____” _____ 2025 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Рогатовській Анастасії Андріївні
(прізвище, ім’я, по батькові)

1. Тема роботи «Дослідження методів тестування веб-додатків для покращення їх якості»

керівник роботи д.т.н., проф. Жарікова М.В.,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджена наказом вищого навчального закладу від 15.09.2025 р. №417-С

2. Строк подання студентом роботи 15.12.2025

3. Вихідні дані до роботи аналіз літературних джерел, інструментальне моделювання, створення тестових сценаріїв, експериментальне тестування, порівняльний аналіз, мови програмування JavaScript/TypeScript, технології Jest, React Testing Library, середовище розробки JetBrains Webstorm

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): 1) аналіз та порівняння існуючих підходів до модульного, інтеграційного та E2E-тестування SPA; 2) вибір оптимальних інструментів і стратегій тестування; 3) проектування графової моделі UI-станів та переходів; 4) генерація граф-орієнтованих тестових сценаріїв; 5) реалізація та автоматизація тестів; 6) проведення експериментів із вимірюванням покриття, продуктивності й надійності; 7) аналіз і візуалізація результатів.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень) 1) діаграма варіантів; 2) діаграма класів; 3) діаграма діяльності; 4) діаграма станів; 5) діаграма компонентів; 6) побудова графової моделі; 7) аналіз

результатів дослідження

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 29.09.2025

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів виконання роботи	Термін виконання етапів роботи	Примітки
1.	Отримання завдання	29.09.2025	Виконано
2.	Підбір літератури	05.10.2025	Виконано
3.	Аналіз предметної області	19.10.2025	Виконано
4.	Розробка та обґрунтування завдання	26.10.2025	Виконано
5.	Розробка концептуальної моделі	05.11.2025	Виконано
6.	Розробка алгоритму	12.11.2025	Виконано
7.	Проектування програми	26.11.2025	Виконано
8.	Розробка інтерфейсу програми	30.11.2025	Виконано
9.	Тестування програми	05.12.2025	Виконано
10.	Оформлення пояснювальної записки	10.12.2025	Виконано
11.	Захист кваліфікаційної роботи	15.12.2025	Виконано

Студент _____ Рогатовська А.А.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Жарікова М.В.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка: 96 сторінок, ‘15 рисунків, 6 таблиць, 1 додаток, 30 джерел.

Об'єкт дослідження: процес тестування елементів інтерфейсу користувача у веб-додатках.

Предмет дослідження: використання графових моделей для побудови тестових сценаріїв та підвищення ефективності тестування.

Мета дослідження: комплексний аналіз та порівняння методів тестування елементів веб-застосувань, розроблених у технологічному стеку JavaScript (React, Redux) в поєднанні з графовими моделями для організації тестових сценаріїв, а також у демонстрації ітеративного покращення додатку на основі результатів тестування.

Новизна отриманих результатів полягає в обґрунтуванні доцільності використання графів для формалізації поведінки веб-додатків у процесі тестування, а також в інтеграції цього підходу з сучасними фреймворками автоматизації.

Практична цінність результатів роботи полягає у створенні інструменту, який може бути застосований у реальних умовах для підвищення ефективності тестування SPA-додатків, зниженні витрат на підтримку тестів і забезпечення більшої гнучкості у змінному середовищі розробки.

Перелік ключових слів: ТЕСТУВАННЯ, ВЕБ-ДОДАТОК, АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ, МЕТОДИ ТЕСТУВАННЯ, ТЕСТОВІ СЦЕНАРІЇ, REACT, REDUX, SINGLE PAGE APPLICATION.

АНОТАЦІЯ

Кваліфікаційна робота магістра складається зі вступу, чотирьох розділів, висновку, переліку використаних джерел та додатків.

Роботу присвячено аналізу ефективності графових моделей для автоматизованого тестування SPA-додатків, розроблених із використанням фреймворків React та Redux. Застосовано графовий підхід: подання логіки веб-застосунку у вигляді орієнтованого графа, де вершини відповідають інтерфейсним станам, а ребра – подіям або діям користувача. Це дозволило формалізувати модель поведінки системи, генерувати тести на основі маршрутів у графі, визначати ступінь покриття у вигляді кількості відвіданих вершин/ребер та уникати дублювання в сценаріях.

Реалізовано прототип SPA-додатку на основі React, який імітує типову логіку інтерфейсу з формами, маршрутами, станами користувача та зовнішніми запитами.

У результаті роботи було створено систему для автоматичної генерації тестів на основі графів станів та переходів, яка інтегрується з Cypress для end-to-end тестування. Проведено експериментальні дослідження та отримано результати щодо ефективності використання графового підходу у тестуванні веб-застосунків.

ABSTRACT

The master's thesis consists of an introduction, four chapters, a conclusion, a list of sources and appendices.

The thesis is devoted to the analysis of the effectiveness of graph models for automated testing of SPA applications developed using the React and Redux frameworks. A graph approach was applied: representing the logic of a web application in the form of a directed graph, where vertices correspond to interface states, and edges correspond to events or user actions. This allowed us to formalize the system behavior model, generate tests based on routes in the graph, determine the degree of coverage in the form of the number of visited vertices/edges, and avoid duplication in scenarios.

A prototype of a SPA application based on React was implemented, which simulates typical interface logic with forms, routes, user states, and external requests.

As a result of the work, a system was created for automatic test generation based on state and transition graphs, which integrates with Cypress for end-to-end testing. Experimental studies were conducted and results were obtained on the effectiveness of using the graph approach in testing web applications.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ І СКОРОЧЕНЬ

SPA – Single Page Application

DOM – Document Object Model

E2E – End-to-End (тестування від початку до кінця)

AST – Abstract Syntax Tree

JSX – JavaScript XML

DFS – Depth-First Search (пошук в глибину)

BFS – Breadth-First Search (пошук в ширину)

API – Application Programming Interface

WCAG – Web Content Accessibility Guidelines

JSON – JavaScript Object Notation

CSV – Comma-Separated Values

CI/CD – Continuous Integration / Continuous Deployment

UI – User Interface

HOC – Higher-Order Component

SSE – Server-Sent Events BDD (Behavior-Driven Development)

TCO – Total Cost of Ownership

ВСТУП

Швидкий розвиток веб-технологій і постійне збільшення складності веб-додатків, особливо односторінкових застосунків (SPA), створених із використанням фреймворків на основі JavaScript (наприклад, React) та бібліотек управління станом (Redux), висувають безпрецедентні вимоги до якості, надійності та безпеки програмного забезпечення.

Такі додатки характеризуються складною структурою, високим рівнем інтерактивності, активним використанням асинхронної логіки, що значно ускладнює процес їхнього тестування.

Традиційні методи тестування, які часто передбачають ручний аналіз функціоналу, стають дедалі менш ефективними. Вони не здатні повністю охопити багатогранну поведінку інтерфейсу, особливо в умовах швидкого циклу релізів та змінної архітектури застосунків. Як наслідок, ризики виявлення критичних помилок на більш пізніх етапах розробки або вже після впровадження залишаються високими.

Актуальність питань ефективності веб-додатків обумовлена стрімким розвитком інтернет-технологій та зростаючими вимогами користувачів до швидкодії, зручності та функціональності. Сьогодні навіть незначні затримки у відображенні контенту чи помилки в роботі веб-додатків можуть призвести до втрати користувачів та, як наслідок, зниження конкурентоспроможності компанії. Автоматизоване тестування стає критично важливим компонентом у забезпеченні якості веб-додатків, дозволяючи своєчасно виявляти дефекти, підвищувати швидкість розробки та знижувати витрати на підтримку.

Автоматизовані підходи до тестування (зокрема використання інструментів Selenium, Cypress, Postman, Jest) поступово витісняють ручні сценарії. Вони дозволяють досягти більшого охоплення, повторюваності й стабільності у виявленні помилок, а також знижують витрати на супровід. Однак, навіть у межах автоматизованого тестування зберігаються виклики,

пов’язані з підтримкою тестів при зміні DOM-структури, адаптивності UI та варіативності взаємодій користувача з інтерфейсом.

Особливої актуальності набуває проблема систематизації та оптимізації тестових сценаріїв. Для багатьох веб-додатків важливо не лише перевірити окремі компоненти, а й протестувати весь спектр можливих переходів між станами, залежно від дій користувача; через це доцільним є застосування графового підходу: подання логіки веб-застосунку у вигляді орієнтованого графа, де вершини відповідають інтерфейсним станам (наприклад, сторінкам, діалоговим вікнам, формам), а ребра – подіям або діям користувача (кліки, заповнення форм, зміна маршруту тощо).

Такий підхід дозволяє формалізувати модель поведінки системи, генерувати тести на основі маршрутів у графі, визначати ступінь покриття (у вигляді кількості відвіданих вершин/ребер) та уникати дублювання в сценаріях. Він також відкриває перспективи для часткової або повної автоматичної генерації тестових сценаріїв, що підвищує масштабованість процесу тестування.

У межах кваліфікаційної роботи буде реалізовано прототип SPA-додатку на основі React, який імітує типову логіку інтерфейсу з формами, маршрутами, станами користувача та зовнішніми запитами.

До цього застосунку буде розроблено графову модель, що дозволяє формалізувати тестові сценарії та здійснити їх запуск через інтеграцію з сучасними інструментами тестування (наприклад, Playwright або Cypress), що дозволить продемонструвати переваги використання формальних моделей для генерації, автоматизації та аналізу тестів у сучасних веб-застосунках.

Мета дослідження полягає в комплексному аналізі й порівнянні методів тестування елементів веб-застосувань, розроблених у технологічному стеку JavaScript (React, Redux) в поєднанні з графовими моделями для організації тестових сценаріїв, а також у демонстрації ітеративного покращення додатку на основі результатів тестування.

Для досягнення поставленої мети в роботі вирішуються такі завдання:

- проаналізувати існуючі методи та інструменти тестування веб-додатків (Selenium, Cypress, Postman, Jest, мануальне тестування тощо);
- формалізувати підхід до опису тест-сценаріїв за допомогою графа станів/переходів;
- розробити або адаптувати невеликий веб-застосунок на React, у якому можна буде покроково виявляти й виправляти помилки під час тестування;
- провести експеримент: порівняти швидкість, точність і покриття різних методів тестування в зазначеному застосунку; виміряти динаміку покращення коду;
- розробити практичні рекомендації щодо вибору та застосування методів тестування з урахуванням моделі, спрямовані на підвищення якості ПЗ.

Об’єктом дослідження є процес тестування елементів інтерфейсу користувача у веб-додатках.

Предметом дослідження є використання графових моделей для побудови тестових сценаріїв та підвищення ефективності тестування.

Методи дослідження: аналіз літературних джерел, інструментальне моделювання, реалізація тестових сценаріїв, експериментальне тестування, порівняльний аналіз, побудова графів, моделювання покриття, використання автоматизованих фреймворків (Jest, Cypress, Playwright).

Наукова новизна роботи полягає в обґрунтуванні доцільності використання графів для формалізації поведінки веб-додатків у процесі тестування, а також в інтеграції цього підходу з сучасними фреймворками автоматизації.

Практична значущість полягає у створенні інструменту, який може бути застосований у реальних умовах для підвищення ефективності тестування SPA-додатків, зниженні витрат на підтримку тестів і забезпечення більшої гнучкості у змінному середовищі розробки.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАДАЧІ

1.1. Аналіз предметної області дослідження

У сучасному світі, де цифрові технології проникли в усі сфери життя, веб-додатки стали інструментами доступу до інформації та послуг, а також критично важливими елементами інфраструктури в більшості галузей [1].

Веб-додатки відіграють важливу роль у повсякденному житті кожного, хто займається комунікаціями. Наприклад, веб-додатки є основою електронної комерції. Інтернет-магазини, платформи для продажу товарів і послуг та системи управління замовленнями залежать від веб-додатків.

Веб-додаток, який є одночасно зручним для користувача і функціональний, має важливе значення для успіху будь-якого підприємства, яке займається електронною комерцією, оскільки він дозволяє клієнтам здійснювати покупки, а власникам продавати товари та послуги онлайн, управляти запасами, обробляти платежі та взаємодіяти з клієнтами [2]. Прикладами є відомі всім гіганти: Amazon, eBay, Shopify та багато інших.

Фінансовий сектор також зазнав впливу діджиталізації. Сьогодні все більше послуг, пов'язаних з грошима та матеріальними активами, представлені на різних веб-сайтах, щоб полегшити клієнтський досвід. Веб-додатки використовуються для онлайн-банкінгу, інвестиційних платформ, платіжних систем та управління портфелями. Вони дозволяють клієнтам переглядати залишки на рахунках, здійснювати грошові перекази, інвестувати в цінні папери та управляти своїми фінансами онлайн. Прикладами можуть слугувати веб-сайти більшості банків, інвестиційних компаній та платіжних систем.

Крім того, не забуваймо про найважливішу сферу, яка стрімко розвивається для забезпечення кращого життя для всього населення – охорону здоров'я. Веб-додатки набувають все більшого значення в охороні здоров'я і використовуються для ведення електронних медичних записів, телемедицини, управління аптеками та аналізу медичних даних. Вони

дозволяють лікарям і пацієнтам отримувати доступ до медичної інформації онлайн, проводити дистанційні консультації та покращувати якість обслуговування. Прикладами є системи електронних медичних записів та телемедичні платформи.

Таким чином бачимо, що практично кожна галузь залежить від безперебійного функціонування веб-додатків. Дослідження McKinsey Global Institute показує, що у 2019 році Інтернет зробив 3,4% внеску у ВВП найбільших розвинених країн, що більше, ніж внесок сільського господарства або комунальних послуг [3]. Якби Інтернет розглядався як окремий сектор економіки, його внесок у світовий ВВП був би більш значним, ніж енергетика чи сільське господарство. Стрімкий розвиток веб-технологій, їх складність та функціональність значно підвищили вимоги до якості, надійності та безпеки програмного забезпечення. Саме тому вивчення ефективних методів тестування веб-додатків є особливо актуальним.

Однак стрімкий розвиток веб-технологій та постійне вдосконалення функціональності веб-додатків призвели до безпрецедентного ускладнення їх архітектури. Сучасні веб-додатки часто є складними розподіленими системами, які включають багато взаємопов'язаних компонентів, працюють з великими обсягами даних та інтегруються з різними зовнішніми сервісами. Забезпечення їх масштабованості, продуктивності та безпеки вимагає використання передових технологій і методологій розробки, а також застосування сучасних інструментів і методів тестування.

Станом на 2024 рік, JavaScript та HTML/CSS залишаються найпопулярнішими мовами програмування серед розробників усього світу. Згідно з даними Statista, понад 62% фахівців використовують JavaScript у своїй роботі, а близько 53% - HTML/CSS. До п'ятірки лідерів також увійшли Python, SQL та TypeScript, що свідчить про їх широке застосування в сучасній розробці. Клієнт взаємодіє з веб-додатком через спеціальний інтерфейс - веб-API, який дозволяє користувачам (людям або іншим програмам) отримувати доступ до функціоналу додатку через комп'ютерну

мережу. Найпоширенішим підходом для роботи з HTTP-API є REST (Representational State Transfer) - архітектурний стиль, що визначає стандартизований інтерфейс для взаємодії з ресурсами.

Враховується багато характеристик програми: простота використання, узгодженість та універсальність застосовуваних методів, вартість, дизайн, вимоги до апаратного забезпечення та операційної системи, можливість взаємодії з іншими поширеними програмами та розширення функціоналу [3].

Хоча REST-API концептуально є безстановним (stateless), на практиці вони часто взаємодіють з системами зберігання даних (базами даних, кешем), що надає їм певні характеристики стану.

Мова розмітки XML використовується для обробки та перетворення метаданих. Для зберігання метаданих застосовується база даних, яка підтримує збережені процедури та роботу з XML.

Фронтенд – це клієнтська частина вебзастосунку, з якою взаємодіє користувач. Вона відповідає за візуальне відображення інформації, збирання введених даних і надсилання запитів до сервера. Фронтенд реалізується зазвичай з використанням HTML, CSS, JavaScript або фреймворків типу React, Angular, Vue тощо. Коли користувач натискає кнопку, заповнює форму або відкриває сторінку – саме фронтенд ініціює звернення до сервера.

Бекенд – це серверна частина застосунку, яка обробляє бізнес-логіку, працює з базами даних, перевіряє права доступу та формує відповіді на запити клієнта. Бекенд реалізується мовами програмування, такими як Python (Django, Flask), Node.js, Java

(Spring), PHP, Ruby тощо. Зазвичай бекенд надає API — набір кінцевих точок (endpoint), які приймають HTTP-запити від фронтенду (наприклад, GET /products, POST /order), обробляють їх і повертають результат у вигляді JSON або HTML.

Розробка сучасних веб-додатків, особливо односторінкових додатків (SPA), що базуються на JavaScript фреймворках, таких як React та Redux, створює значні проблеми з тестуванням. SPA – це веб-додаток, який

взаємодіє з користувачем шляхом динамічного переписування поточної сторінки, а не завантаження цілих нових сторінок з сервера [4]. Такі додатки характеризуються високим рівнем складності через динамічні зміни DOM, асинхронні операції, велику кількість залежностей між компонентами та специфічну архітектуру Redux. Традиційних підходів до тестування, які часто базуються на ручному аналізі, недостатньо для ефективного виявлення помилок у складних SPA-додатках. Наприклад, є такі важкі пункти:

- тестування асинхронних операцій, які часто передбачають взаємодію з API, вимагає спеціальних підходів, таких як використання async/await, обіцянок та імітацій відповідей API;
- динамічні зміни в DOM через оновлення стану Redux ускладнюють перевірку того, що DOM знаходиться в очікуваному стані;
- велика кількість залежностей між компонентами;
- використання вищих порядків (HOC) і декораторів;
- специфіка маршрутизації в SPA-додатках значно ускладнюють тестування.

Традиційні методи тестування, засновані на ручному аналізі, все частіше стають недостатніми для забезпечення високої якості та надійності сучасних веб-додатків. Ручний аналіз займає багато часу, схильний до помилок і часто не може повністю охопити всю функціональність складної системи, особливо з огляду на високі вимоги до продуктивності та масштабованості. Крім того, він не завжди може повністю виявити приховані помилки, що може призвести до серйозних проблем після випуску програмного забезпечення.

Ефективне тестування вимагає нових підходів, які враховують ці особливості. Окрім того, автоматизоване тестування дозволяє скоротити витрати на окрему позицію тестувальника і «використовувати код для перевірки коду»[5]. Необхідно використовувати сучасні інструменти автоматизованого тестування, такі як Selenium, інструмент для автоматизації веб-браузерів, Cypress, інструмент тестування інтерфейсу нового покоління, створений для сучасного Інтернету, і Jest, чудовий фреймворк для

тестування JavaScript з акцентом на простоту, а також інструменти тестування API, такі як Postman, платформа для спільної роботи для розробки API. Ретельне планування структури тестування, використання макетів для імітації залежностей, тестування інтеграції та використання тестування знімків для перевірки стану DOM − все це необхідні заходи для забезпечення високої якості та надійності SPA-додатків. Метою даного дослідження є аналіз та порівняння ефективності різних методів тестування для виявлення найкращих підходів до забезпечення якості та надійності SPA-додатків, розроблених на React та Redux.

Тому пошук нових, більш ефективних та автоматизованих підходів до тестування, що враховують особливості сучасних веб-додатків та забезпечують глибокий аналіз їх функціональності, є не просто завданням, а нагальною місією. Це вимагає використання сучасних інструментів і методологій, зокрема автоматизованого тестування, яке дозволяє ефективно виявляти помилки на різних етапах розробки, забезпечуючи високу точність і повне покриття функціоналу.

Вивчення ефективних методів тестування є важливим напрямком розвитку програмної інженерії, оскільки спрямоване на визначення оптимальних стратегій тестування, які дозволяють ефективно виявляти помилки та забезпечувати високу якість і надійність програмного забезпечення з урахуванням особливостей різних типів веб-додатків та їх компонентів. Глибоке розуміння цих методів є ключовим для розробників, які прагнуть створювати надійні, стабільні та масштабовані веб-додатки, що відповідають високим вимогам сучасного ринку.

Ефективні методи тестування дозволяють значно скоротити час і витрати, підвищити якість тестів і забезпечити більш глибоке покриття функціональності веб-додатків, що є критично важливим для забезпечення конкурентоспроможності та успіху на ринку програмного забезпечення. Тому глибоке розуміння методів тестування та їх застосування є основою для створення якісного та надійного програмного забезпечення.

Сучасне тестування веб-додатків стикається з низкою складностей, пов'язаних із динамічним характером веб-технологій та зростаючими вимогами до якості продуктів. Однією з ключових проблем є крос-браузерна та крос-платформна сумісність. Веб-додатки повинні коректно відображатися та функціонувати в різних браузерах (Chrome, Firefox, Safari) та на різних пристроях (десктопи, смартфони, планшети), що значно ускладнює процес тестування.

Адаптивність та респонсивність інтерфейсу становить окрему проблему. Тестувальники повинні перевіряти коректність відображення контенту на екранах різних розмірів і роздільних здатностей. Це потребує використання спеціальних інструментів емуляції або тестування на реальних пристроях, що може бути ресурсомістким процесом.

Продуктивність і навантажувальні тести виявляються ще одним складним аспектом. Розрив між тестовою та продакшен-реальністю досягає катастрофічних масштабів. Тестовий платіжний шлюз фіксує час транзакції 0.5 сек, але на продакшені 98-й персентиль становить 17 секунд через AML-перевірки, валютні конвертації та міжбанківські комісії. Ця розбіжність спричиняє 40% відмов від кошика – клієнти не готові чекати. Тестування в ідеальних умовах створює хибне відчуття надійності, яке розбивається об реальність під час пікових навантажень.

Піраміда тестування (рис. 1.1) – це концепція в розробці програмного забезпечення запропонована Майком Коном у книзі "Succeeding with Agile", котра представляє різні типи автоматизованих тестів у вигляді піраміди. Її мета полягає у забезпеченні ефективного та збалансованого підходу до автоматизованого тестування, де кожен рівень тестів має свою мету, область застосування та деталізацію [6].

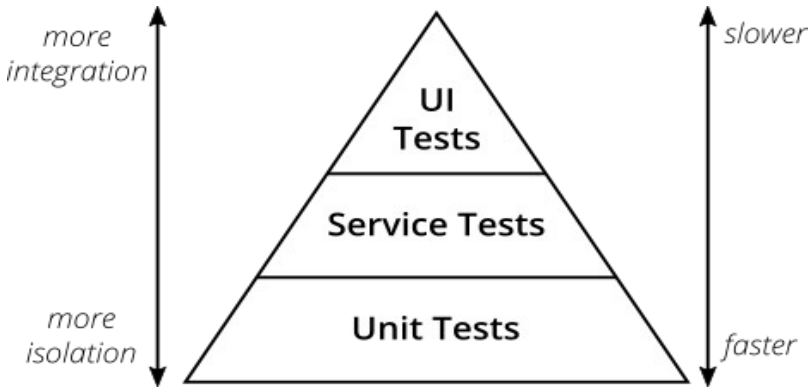

Рисунок 1.1. Піраміда тестування Майка Кона (за даними [7])

Оригінальна піраміда тестування складається з трьох основних рівнів:

– модульні тести (нижній рівень);

– сервісні тести (середній рівень);

– тести користувацького інтерфейсу (верхній рівень).

Основний принцип піраміди полягає в тому, що кількість тестів має зменшуватися при русі від нижніх рівнів до верхніх. Це означає, що найбільша кількість тестів повинна бути на рівні модульного тестування, менша кількість - на рівні сервісних тестів, і найменша - на рівні тестування користувацького інтерфейсу.

Зростаюча складність бізнес-логіки сучасних додатків вимагає від тестувальників глибокого розуміння предметної області. Це особливо актуально для фінансових систем, медичних рішень або складних корпоративних додатків.

Вирішення цих проблем вимагає комплексного підходу, що включає використання сучасних інструментів тестування, ретельне планування тестових сценаріїв та постійне вдосконалення процесів забезпечення якості. Важливим аспектом є також баланс між автоматизованим та ручним тестуванням, оскільки кожен з цих підходів має свої переваги та обмеження.

Результати цього дослідження можуть покращити процеси розробки та тестування веб-додатків, сприяти створенню більш надійних та стабільних систем, а також забезпечити розвиток індустрії програмної інженерії в

цілому. Зокрема, отримані дані можуть бути використані для вдосконалення існуючих інструментів автоматизації, оптимізації ресурсів, підвищення безпеки програмного забезпечення та скорочення часу, необхідного для впровадження нових рішень. Крім того, вони можуть стати основою для розробки нових підходів до управління проєктами та інтеграції сучасних технологій у робочі процеси.

У межах кваліфікаційної роботи планується не лише аналіз методів тестування веб-додатків, а й створення власного прикладного SPA-застосунку (на React, Redux), до якого буде побудована графова модель для формування автоматичних сценаріїв тестування. Це дозволить на практиці оцінити переваги запропонованого підходу та порівняти його ефективність із традиційними методами.

1.2. Постановка задачі

Дане дослідження спрямоване на виявлення та порівняння ефективності різних підходів до автоматизованого та ручного тестування елементів SPA-додатків, розроблених на JavaScript з використанням фреймворку React та бібліотеки Redux.

Метою є визначення оптимальних стратегій тестування, які дозволяють ефективно виявляти помилки, забезпечувати високу якість та надійність програмного забезпечення, а також враховують особливості сучасних веб-технологій.

В ході виконання роботи треба виконати наступні завдання:

а) проаналізувати існуючі методи тестування (Selenium, Cypress, TestCafe, тестування за допомогою Postman, мануальне);

б) провести порівняння цих методів в різних умовах складності роботи Single Page Application;

в) провести планування експериментального дослідження методів тестування, а саме:

1) розробка тестових сценаріїв для перевірки функціональності та якості обраних веб-додатків, використовуючи обрані методи та інструменти;
2) проведення експерименту з використанням обраних методів та інструментів;
3) визначення критеріїв оцінки: Визначте ключові критерії для оцінки ефективності різних методів тестування (швидкість, покриття, надійність, вартість);
4) провести статистичну обробку отриманих результатів та проаналізуйте їх з урахуванням обраних критеріїв;

г) виконати реалізацію програмного забезпечення на якій буде проведено експериментальне дослідження;

д) провести експериментальне дослідження методів;

е) розробити рекомендації щодо вибору та застосування методів тестування та запропонувати можливі покращення для цих методів;

ж) розробити (або використати) ітераційний підхід до тестування та покращення коду веб-застосунку, перевіривши ефективність graph-based testing у виявленні дефектів і підвищенні стабільності ПЗ;

з) спроєктувати архітектуру демо-додатку (B2C інтернет-магазину) у вигляді SPA з використанням React/Redux, що слугуватиме базовою платформою для реалізації тестових сценаріїв і аналізу ефективності методів тестування.

У результаті отримаємо рекомендацій щодо вибору та застосування методів тестування елементів SPA-додатків, розроблених на JavaScript із використанням фреймворків React і Redux.

Очікуваними результатами цього дослідження є:

- детальний порівняльний аналіз ефективності різних методів тестування, що включає кількісні показники (швидкість, точність, покриття функціональності, витрати часу тощо) та якісний аналіз переваг та недоліків кожного методу;
- наочні графіки та діаграми, що ілюструють результати тестування та полегшують порівняння різних методів;

- висновок щодо вибору оптимальних методів тестування SPA-додатків на React та Redux, з урахуванням особливостей конкретної розробки та задачі тестування;
- рекомендації щодо вибору та застосування оптимальних методів тестування SPA-додатків з врахуванням їхньої архітектури та функціональності;
- оцінка обмежень та проблем, які виникають під час тестування SPAдодатків на React та Redux (наприклад, складнощі тестування асинхронних операцій, динамічна зміна DOM, велика кількість залежностей між компонентами);
- аналіз застосовності різних методів тестування для різних типів завдань та компонентів SPA-додатку.

Дослідження буде проводитися з урахуванням певних обмежень. Час на проведення дослідження обмежений тривалістю семестру, а також доступні ресурси (комп'ютерні можливості тощо) можуть вплинути на обсяг та глибину дослідження.

Необхідні ресурси для виконання проекту:

а) апаратні ресурси:

1) потужний комп'ютер (процесор не менше 8 ядер, RAM не менше 16 ГБ) для ефективного виконання тестів та обробки великих обсягів даних;
2) надійне швидке інтернет-з'єднання для завантаження необхідних бібліотек та взаємодії з тестовими серверами;

б) програмні ресурси:

1) ліцензії на комерційні інструменти тестування;
2) Node.js та npm (або yarn) для управління залежностями проекту;
3) середовище розробки (наприклад, Visual Studio Code або WebStorm);
4) необхідні бібліотеки для розробки та тестування (React, Redux, Selenium WebDriver, Cypress, Jest, Postman API тощо);
5) програмне забезпечення для візуалізації даних (наприклад, Chart.js, Matplotlib, або інші).

Для досягнення мети дослідження необхідно розробити систему критеріїв оцінки інструментів тестування, яка враховуватиме такі аспекти як

функціональність, зручність використання, швидкість виконання тестів, підтримку різних технологій та інтеграцію з іншими інструментами розробки. Важливим елементом є також розробка методики порівняльного аналізу, яка дозволить об'єктивно оцінити кожен інструмент. Ця система враховуватиме аспекти функціональності, зручності використання, швидкості виконання тестів, підтримки різних технологій, можливостей налаштування, а також інтеграції з іншими інструментами розробки.

Необхідно проаналізувати існуючу інформацію про те, які конкретні застосунки для тестування веб додатків існують та які з них треба обрати для дослідження. Автоматизоване тестування стає ключовим інструментом забезпечення якості, дозволяючи значно прискорити випуск продуктів та підвищити їх надійність. Вибір оптимальних інструментів потребує ретельного аналізу їх характеристик та впливу на ефективність тестування.

Критерії, за якими можна оцінити технології для тестування: крос-браузерність, наявність якісної документації, можливості формування звітів, функціональні можливості інструменту, простота опанування інструментом, вартість використання, підтримувані мови для написання тестів. Цей набір критеріїв забезпечує всебічну оцінку інструментів автоматизованого тестування та допомагає обрати найефетивніше рішення для конкретних потреб проекту.

Крос-браузерне тестування є критично важливим для сучасних веб-додатків. Інструмент повинен забезпечувати:

- підтримку всіх актуальних версій популярних браузерів (Chrome, Firefox, Safari, Edge);
- можливість тестування на різних версіях браузерів;
- емуляцію різних роздільних здатностей екрану.

Якісна документація значно прискорює процес освоєння інструменту та вирішення проблем:

- повнота офіційної документації (API Reference, Tutorials, Examples);

- наявність покрокових гайдів для початківців; - приклади реальних тестових сценаріїв.

Вартість використання є дуже важливою складовою для всіх проектів. Якщо ціна зависока і проект не зможе її покрити, то ніякі технічні переваги це не перекриють.

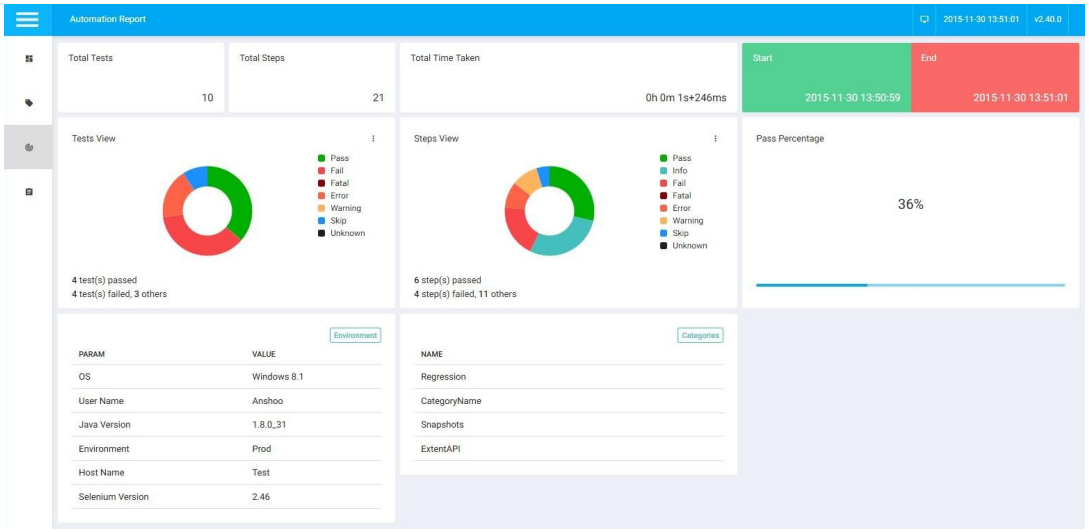

Рисунок 1.2. Приклад візуалізації результатів тестування [8]

РОЗДІЛ 2. ОГЛЯД ІСНУЮЧИХ ІНСТРУМЕНТІВ ТЕСТУВАННЯ ВЕБ ЗАСТОСУНКІВ

2.1. Загальні аспекти тестування

Ефективне тестування веб-додатків є критично важливим для забезпечення їхньої якості, надійності та безпеки. У сучасному світі, де веб-додатки відіграють ключову роль у найрізноманітніших галузях економіки, будь-який збій у їхній роботі може мати серйозні наслідки.

Зростання складності та функціональності веб-додатків, особливо single-page applications (SPA), розроблених на основі сучасних JavaScript фреймворків, таких як React та Redux, призводить до необхідності застосування різноманітних підходів та методологій тестування, що враховують особливу архітектуру та функціональність цих систем.

Традиційні підходи, засновані на ручному аналізі, часто виявляються неефективними для забезпечення високої якості сучасних веб-додатків, оскільки ручний аналіз є трудомістким, помилковим та часто не в змозі повністю охопити всю функціональність складної системи.

Типи тестування [6]:

– функціональне тестування: Перевірка коректності роботи функцій вебдодатку відповідно до заданих вимог. Включає тестування інтерфейсу користувача (UI testing), тестування API та інтеграційне тестування;
– тестування продуктивності: Оцінка продуктивності веб-додатку під різними навантаженнями. Включає тестування навантаження (load testing), тестування на витривалість (stress testing) та тестування швидкодії (performance testing);
– тестування безпеки: Виявлення уразливостей веб-додатку, що можуть бути використані зловмисниками. Включає тестування на проникнення (penetration testing), тестування на безпеку автентифікації та авторизації, а також тестування на захист від різних типів атак;

– тестування юзабіліті: Оцінка зручності та простоти користування вебдодатком. Включає спостереження за користувачами під час використання додатку та збір їхніх відгуків.

Вибір методології тестування залежить від конкретних умов та вимог до проекту. Найбільш поширеними методологіями тестування є каскадна (Waterfall), ітераційна (Agile), V-подібна та спіралеподібна. Кожна з цих методологій має свої переваги та недоліки, і вибір залежить від специфіки проекту та доступних ресурсів.

Автоматизація тестування дозволяє значно скоротити час та витрати на тестування, покращити якість та забезпечити більш глибоке покриття функціональності. Вибір інструментів автоматизації залежить від конкретних умов та вимог до проекту. Selenium, Cypress, Jest та інші інструменти забезпечують різноманітні можливості для автоматизації тестування, але їх ефективність залежить від правильного планування та використання.

Для ефективного тестування SPA-додатків, розроблених на React та Redux, важливо враховувати особливості їхньої архітектури, такі як управління станом, асинхронні операції та роботу з API. Це вимагає застосування спеціальних підходів та інструментів, які дозволяють ефективно тестувати асинхронні запити, динамічну зміну DOM та складну взаємодію компонентів.

Ефективне тестування веб-додатків є комплексним процесом, що вимагає ретельного планування, використання різних методів та інструментів, а також глибокого розуміння особливостей архітектури та функціональності системи, що тестується [7].

Правильний вибір методології та інструментів дозволяє забезпечити високу якість, надійність та безпеку веб-додатків, що є критично важливим для успіху сучасних цифрових проєктів.

2.2. Автоматизоване тестування веб-додатків

Автоматизація тестування є необхідною складовою успішної розробки сучасних веб-додатків, бо може значно підвищити якість і знизити витрати на розробку та тестування програмного забезпечення [8]. Зростання складності та функціональності цих додатків, особливо single-page applications (SPA), розроблених на основі JavaScript фреймворків, таких як React та Redux, призвело до необхідності застосування передових методів та інструментів автоматизації, які дозволяють ефективно виявляти помилки, забезпечуючи високу якість та надійність програмного забезпечення [9]. Просте використання інструментів автоматизації не гарантує успіху; необхідна чітка стратегія, ретельне планування та використання відповідних методологій.

Ефективність автоматизованого тестування безпосередньо залежить від правильного вибору інструментів. Серед найпопулярніших інструментів тестування веб-додатків:

– Selenium, відомий своїм гнучким та потужним механізмом для автоматизації взаємодії з веб-сторінками, що підтримує різні браузери та мову програмування;
– Cypress, сучасний фреймворк, орієнтований на тестування сучасних вебдодатків, відомий своїми швидкістю та простотою використання, а також інтеграцією з різними інструментами розробки;
– Playwright, ще один потужний фреймворк, що підтримує різні браузери та мову програмування, відомий своїми надійністю та підтримкою різних функцій, важливих для тестування сучасних веб-додатків;
– Puppeteer, фреймворк від Google, орієнтований на тестування вебдодатків за допомогою Node.js, що забезпечує контроль над браузером на рівні API.

Вибір конкретного інструменту залежить від конкретних умов проекту, навиків команди розробників та специфіки тестованих додатків.

Для ефективного автоматизованого тестування важливо використовувати відповідну методологію. Поширені методології, такі як тестування зверху вниз (Top-Down), тестування знизу вверх (Bottom-Up), тестування великих блоків (Big Bang) та тестування складових частин (Incremental), мають свої переваги та недоліки, і вибір залежить від складності веб-додатку та доступних ресурсів.

Ретельне планування та розробка тестових сценаріїв, що чітко визначають дії користувача, очікувані результати та критерії успішного завершення тесту, є основою ефективного автоматизованого тестування.

Для імітації залежностей та зовнішніх систем у тестуванні часто використовуються моки, що дозволяють ізолювати тестований компонент від зовнішніх залежностей, спрощуючи процес тестування та покращуючи його надійність [10].

Таким чином, автоматизація тестування є важливим етапом розробки сучасних веб-додатків, але вимагає ретельного підходу, що включає правильний вибір інструментів та методологій, ретельне планування тестування та використання відповідних технік, таких як моки. Це дозволяє забезпечити високу ефективність та точність тестування, що є критично важливим для створення якісного та надійного програмного забезпечення.

2.3. Огляд інструментів автоматизованого тестування веб-додатків

2.3.1. Selenium

Selenium є відкритим проєктом, що об'єднує набір інструментів та бібліотек для автоматизації роботи з браузерами. Він надає розробникам та тестувальникам потужні засоби для створення функціональних тестів без необхідності вивчення спеціальних мов сценаріїв [9].

Selenium WebDriver, як основний компонент фреймворка, надає API для керування браузерами, що робить його ідеальним вибором для

реалізації складних сценаріїв автоматизованого тестування. Завдяки відкритому коду та активної спільноті, Selenium продовжує залишатися одним з найпопулярніших рішень для тестування веб-додатків.

Недоліком Selenium є його схильність до нестабільності (flakiness), особливо при взаємодії з динамічними елементами DOM. Тести можуть падати через повільне завантаження сторінок, затримки в анімаціях або зміну структури HTML. Для мінімізації цих ризиків необхідно додатково реалізовувати очікування (explicit/implicit waits), перевірки на наявність елементів та обробку виключень.

Selenium є open-source проєктом, що постійно підтримується та оновлюється спільнотою. Він сумісний з більшістю інструментів безперервної інтеграції, таких як Jenkins, GitLab CI, Azure Pipelines. Це робить його універсальним вибором для тестування веб-додатків у великих та середніх командах.

2.3.2. Cypress

Cypress працює безпосередньо в середовищі браузера. Така архітектура забезпечує більш тісну інтеграцію з тестованим додатком та підвищує ефективність процесу тестування (Cypress, 2021).

Ключові переваги Cypress включають функціонал реального часу (наприклад, миттєве оновлення тестів), автоматичне очікування елементів інтерфейсу та можливість контролю мережевого трафіку. Ці особливості значно спрощують розробку тестових сценаріїв та прискорюють виконання тестів (Gordon & Baker, 2020).

Однак при роботі з Cypress варто враховувати деякі обмеження. Інструмент орієнтований переважно на JavaScript та має обмежену підтримку браузерів [10].

2.3.3. Playwright

Playwright – це сучасний фреймворк для автоматизації тестування вебдодатків, розроблений компанією Microsoft. Він дозволяє імітувати взаємодію користувача з браузером, такі як кліки, введення тексту, навігація по сторінках тощо. Playwright підтримує роботу з різними браузерами, включаючи Chromium, Firefox та WebKit, що робить його універсальним інструментом для кросс-браузерного тестування.

Playwright працює через WebSocket-з'єднання з браузером, що дозволяє керувати ним програмно. Він використовує:

– Browser Context – ізольоване середовище (подібне до інкогніто-режиму).

– Page – окрема вкладка браузера.

– Selectors – механізми пошуку елементів (CSS, XPath, text).

@playwright/test – це інструмент для запуску тестів, який надає асершени, схожі на Jest, і повністю інтегрований з Playwright.

Він оптимізований для end-to-end тестування і пропонує такі функції, як:

– підтримка різних браузерів;

– паралельне виконання тестів;

– гнучкі налаштування середовища браузера;

– можливість знімкового тестування (snapshots);

– автоматичні спроби повторного запуску.

Цей раннер розроблений командою Playwright і використовує їхній API для максимальної ефективності.

2.3.4. Katalon Studio

Katalon Studio – це інструмент для автоматизації тестування, який дозволяє створювати та запускати UI-тести без написання коду. Він працює на основі Selenium і був представлений у 2015 році.

Ця платформа підтримує автоматизацію тестування для веб-, мобільних, десктопних додатків і API, пропонуючи low-code підхід для оптимізації процесів розробки.

Серед ключових переваг Katalon – вбудовані інструменти CI/CD, сумісні з DevOps, а також Object Spy для ідентифікації елементів інтерфейсу та аналізу їх параметрів. Система інтегрується з популярними сервісами, такими як JIRA, Git і Jenkins, що робить її зручною для керування тестами та помилками.

Katalon підходить для проєктів будь-якого масштабу – від індивідуальних розробників до великих команд. Його екосистема постійно оновлюється, забезпечуючи користувачів сучасними інструментами.

Також платформа має численні вбудовані інтеграції, що дозволяє легко налаштовувати різні види тестування, зокрема API-тестування [11].

2.3.5. TestComplete

TestComplete забезпечує гнучку структуру проєктів, що дозволяє організовувати тести у вигляді ієрархічних елементів.

Користувачі можуть створювати ключові тестові сценарії, об’єднувати їх із низькорівневими скриптами або модулями, а також моделювати складні послідовності дій. Такий підхід забезпечує масштабованість і повторне використання логіки тестів, що особливо корисно при тестуванні великих корпоративних систем [12].

Вбудований планувальник виконання (Execution Plan) дозволяє формувати складні послідовності запуску тестів, призначати параметри, обробляти виняткові ситуації, а також повторно запускати тести в разі збоїв. Це особливо цінно для автоматизованого регресійного тестування, де важливо керовано перевіряти десятки сценаріїв перед кожним релізом.

Крім цього, TestComplete підтримує розподілене виконання тестів завдяки модулю Network Suite. Тестові сценарії можуть запускатися

паралельно на різних машинах в мережі, що значно скорочує загальний час перевірки великого проєкту. Такий підхід дозволяє ефективно масштабувати автоматизацію, особливо в середовищах CI/CD.

2.4. Специфіка тестування SPA-додатків на React та Redux

Тестування SPA-додатків, розроблених на JavaScript з використанням фреймворку React та бібліотеки Redux, має низку специфічних особливостей, що відрізняють його від тестування традиційних веб-додатків. Це пов’язано з унікальною архітектурою цих додатків, що базується на компонентному підході, управлінні станом за допомогою Redux store, асинхронному завантаженні даних та взаємодії з API.

Ефективне тестування таких додатків вимагає застосування спеціальних підходів та інструментів, які враховують цю специфіку.

Чиста архітектура React Redux [11] наголошує на розробці незалежних та модульних компонентів. Для тестування окремих компонентів часто використовуються unit-тести з використанням бібліотек, таких як Jest та React Testing Library. React Testing Library дозволяє тестувати компоненти через їхній інтерфейс користувача, що наближає тестування до реальних умов використання додатку і дозволяє перевірити, чи компоненти відображають дані коректно та реагують на дії користувача відповідним чином. Важливо враховувати асинхронність при тестуванні компонентів, які взаємодіють з API чи Redux store, використовуючи, наприклад, async/await, promises та моки.

Redux забезпечує централізоване управління станом додатку, що спрощує процес тестування. Ви можете тестувати reducers ізольовано, перевіряючи, чи вони коректно обробляють дії та оновлюють стан. Для цього можна використовувати Redux Mock Store або sinon. Централізоване управління станом також дозволяє легше відстежувати зміни в стані додатку під час тестування. Для цього часто використовуються unit-тести з

використанням бібліотек, таких як Redux Mock Store або sinon. Необхідно перевірити, що дії обробляються коректно та що стан додатку змінюється відповідно до очікуваного результату.

Після тестування окремих компонентів та Redux store, необхідно провести інтеграційне тестування для перевірки коректності взаємодії між різними частинами додатку. Це дозволяє виявити проблеми на ранніх етапах розробки, перш ніж вони стануть більш складними у виправленні. SPA-додатки часто взаємодіють з зовнішніми API для отримання та обробки даних. Тестування API передбачає перевірку коректності роботи API, включаючи перевірку кодів відповідей, формату даних та загальної функціональності. Для цього часто використовується Postman або інші спеціалізовані інструменти.

Ефективне тестування SPA-додатків на React та Redux вимагає комплексного підходу, що включає тестування окремих компонентів, Redux store, інтеграційне тестування та тестування API.

Використання відповідних інструментів та відпрацьованих методологій дає змогу досягти значно вищого рівня якості та надійності створюваних додатків. Ретельне планування структури тестування, яке включає чіткий розподіл на різні типи перевірок із продуманою організацією, використання моків для імітації залежностей, інтеграційне тестування та використання snapshot testing для перевірки стану DOM є необхідними заходами для забезпечення високої якості та надійності SPA-додатків.

Сучасні інструменти автоматизованого тестування, такі як Selenium, Cypress та Jest, допомагають у цьому процесі, але вимагають ретельного розуміння особливостей архітектури React та Redux.

2.5. Переваги графового підходу в контексті SPA

Графова модель поведінки користувача в інтерфейсі – це спосіб формалізованого представлення переходів між станами. Вершини графа

відображають стани інтерфейсу, а ребра – події або дії користувача. У контексті складних SPA-додатків цей підхід дозволяє структурувати поведінку користувача як послідовність переходів між екранами, компонентами або маршрутами.

Основні переваги:

- контроль повноти покриття – тестувальник отримує змогу кількісно оцінити, чи всі вершини (стани) та ребра (переходи) були охоплені тестами. Це особливо важливо для SPA, де користувацькі шляхи не завжди очевидні;
- автоматизована генерація маршрутів тестування – за допомогою обхідних алгоритмів (DFS, BFS, китайського листоноші, пошуку в глибину з обмеженням) можна згенерувати набір тестових сценаріїв, що ефективно покривають усі або обрані частини графа;
- зниження дублювання – граф дозволяє ідентифікувати надлишкові переходи та усунути дублікати сценаріїв, що економить час на супровід тестів;
- оптимізація тестування – можна виділити мінімальні множини маршрутів, які охоплюють усю функціональність, що особливо корисно в умовах обмежених ресурсів;
- рання валідація логіки додатку – графова модель може бути створена ще до початку реалізації, на стадії проєктування, дозволяючи проаналізувати послідовність дій, виявити «мертві» стани або неконсистентності;
- формалізація тестів для автоматизації – кожен маршрут у графі можна представити у вигляді автоматизованого тесту, що робить підхід надзвичайно ефективним для регресійного тестування.

У випадку SPA, де динаміка інтерфейсу часто залежить від стану Redux store, асинхронних запитів та маршрутизації, побудова графа дозволяє моделювати варіативність взаємодії та охопити навіть нетипові сценарії використання. Наприклад, умовна активація кнопок, зміна

доступності форм або динамічне завантаження підсторінок – усе це може бути представлено як гілки графа.

Окрім тестування, графова модель також може використовуватись для аналізу продуктивності (визначення найкоротших/найдовших шляхів), визначення критичних маршрутів (via weight assignment) або навіть для статичного аналізу безпеки.

Таким чином, графовий підхід дозволяє перейти від імпровізованого тестування до структурованої, системної перевірки поведінки додатку, що особливо важливо у світі складних, інтерактивних SPA-додатків.

РОЗДІЛ 3. ТЕОРЕТИЧНЕ ОБГРУНТУВАННЯ МЕТОДУ ТЕСТУВАННЯ

3.1. Вибір об’єкта тестування

Дане дослідження зосереджується на тестуванні функціональності типового інтернет-магазину Business-to-Consumer (B2C). Вибір саме B2C моделі обумовлений її поширеністю та актуальністю для дослідження базової функціональності електронної комерції.

Функціональність інтернет-магазину, що підлягає тестуванню, охоплює ключові етапи користувацького досвіду. До них належать:

- перегляд каталогу товарів із можливостями пошуку та фільтрації;
- перевірка коректності відображення детальної інформації про товари на сторінці товару (назва, опис, ціна, зображення, наявність тощо);
- додавання товарів у кошик і оновлення загальної суми замовлення;
- перевірка відображення товарів у кошику, можливості зміни кількості товарів, видалення товарів та коректного розрахунку загальної суми;
- перевірка коректності роботи форми оформлення замовлення та обробки введених даних з валідацією всіх необхідних полів.

Усі ці аспекти критично важливі для забезпечення позитивного користувацького досвіду та успішного функціонування інтернет-магазину.

Дослідження зосереджується на тестуванні переходів між сторінками інтернет-магазину. Це передбачає ретельну перевірку: коректності посилань та доступності сторінок; відображення коректних даних на кожній сторінці; та відповідності функціональності кожної сторінки заданим вимогам. Такий підхід дозволяє оцінити якість та надійність роботи інтернет-магазину в цілому, виявити потенційні проблеми та забезпечити позитивний користувацький досвід.

3.2. Аналіз та моделювання предметної області

Для представлення концептуальної моделі предметної області соціальної мережі було розроблено наступну Use Case- діаграму (див. рис. 3.1).

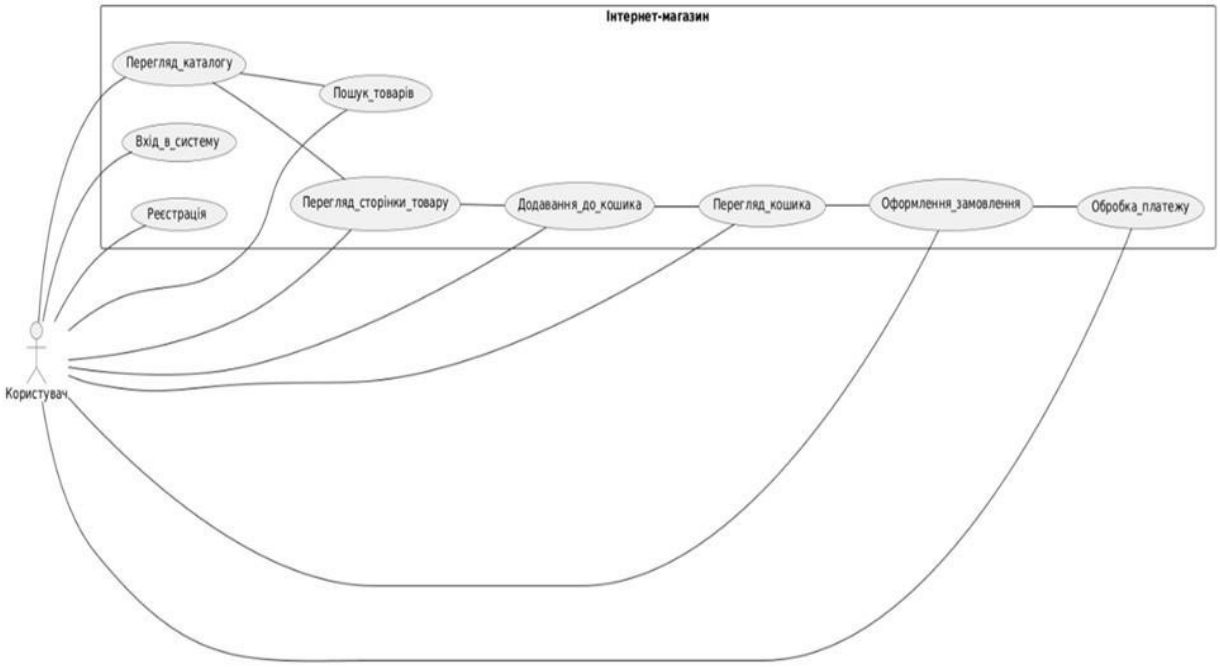

Рисунок 3.1. Діаграма варіантів

Модель інтернет-магазину базується на взаємодії між користувачами та системою, що реалізується через набір ключових функцій.

Користувач може виконувати такі дії:

- перегляд каталогу: перегляд доступних товарів із можливістю сортування та фільтрації за різними критеріями (ціна, бренд, категорія тощо). Система забезпечує швидкий та ефективний пошук товарів;
- перегляд сторінки товару: отримання детальної інформації про обраний товар (назва, опис, ціна, зображення, характеристики, відгуки тощо). Система відображає всю необхідну інформацію з високою якістю;
- додавання товару до кошика: додавання обраних товарів до кошика з можливістю зміни кількості товарів. Система оновлює загальну суму замовлення в реальному часі;

- перегляд кошика: перегляд списку товарів у кошику з можливістю видалення товарів, зміни кількості та оновлення суми. Система забезпечує зручний та інтуїтивно зрозумілий інтерфейс;
- обробка платежу: здійснення платежу за допомогою обраного способу оплати (кредитна карта, PayPal тощо). Система інтегрується з платіжними системами та забезпечує безпечну обробку платежів;
- оформлення замовлення: заповнення форми оформлення замовлення (контактні дані, адреса доставки, спосіб оплати). Система валідує введені дані та забезпечує коректну обробку замовлення.

Система інтернет-магазину забезпечує збереження та обробку всієї необхідної інформації, забезпечує безпеку та надійність роботи, а також забезпечує зручний та інтуїтивно зрозумілий інтерфейс для користувача.

На основі цієї Use Case діаграми ми можемо виокремити необхідні сторінки магазину, які ми бажаємо протестувати за допомогою наших методів та інструментів.

Основні сторінки (стани) інтернет-магазину:

- головна сторінка. Це початкова сторінка інтернет-магазину;
- каталог товарів. Ця сторінка містить повний перелік товарів. Вона забезпечує функціонал пошуку та фільтрації товарів за різними критеріями (ціна, категорія, бренд тощо);
- сторінка товару. Відображає детальну інформацію про конкретний товар: назва, опис, ціна, зображення, характеристики, відгуки користувачів;
- кошик. На цій сторінці відображається список товарів, що були додані до кошика користувачем. Дозволяє редагувати кількість товарів, видаляти товари з кошика та переглядати загальну суму замовлення;
- оформлення замовлення. Ця сторінка містить форму для оформлення замовлення;

- підтвердження замовлення. На цій сторінці підтверджується замовлення користувача. Зазвичай відображається загальна сума замовлення, контактна інформація та інші деталі;
- оплата. Сторінка, де користувач здійснює оплату замовлення за допомогою вибраного способу оплати;
- авторизація/реєстрація. Сторінки для входу в систему існуючих користувачів та реєстрації нових користувачів.

Переходи між цими станами визначаються діями користувача (клік на посилання, натискання кнопки) або системою (наприклад, успішна оплата).

Наприклад, з головної сторінки користувач може перейти до каталогу товарів, а зі сторінки товару – до кошика.

Після оформлення замовлення користувач переходить до оплати, а потім – до підтвердження замовлення.

Ці взаємодії відображено на UML діаграмі станів (див. рис. 3.2).

На основі цих даних можна перетворювати тестові сценарії в графи, які ілюструють переходи між станами.

Такі графи дозволяють визначати ключові точки перевірки, оптимізувати тестове покриття, а також збирати метрики, що забезпечують аналіз ефективності тестування та ідентифікацію критичних вузлів системи.

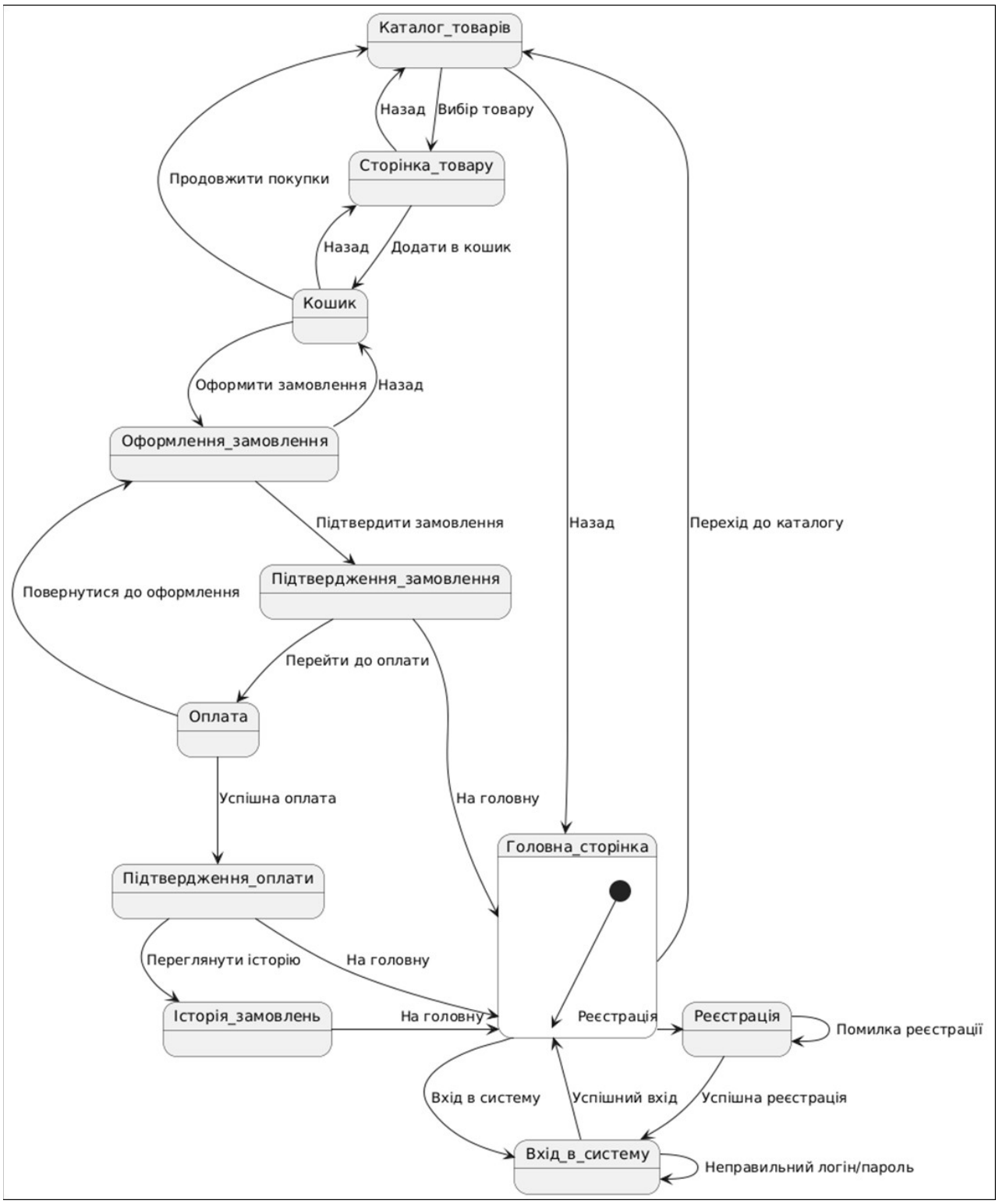

Рисунок 3.2. UML діаграма станів

3.3. Графова модель веб-додатку

Теорія графів є розділом дискретної математики, що досліджує властивості та застосування структур, які складаються з об'єктів (вершин) та зв'язків між ними (ребер).

Формально, граф визначається як упорядкована пара G = (V, E), де V – множина вершин, а E – множина ребер, які з'єднують пари вершин.

Види графів:

- орієнтований граф (оргграф) – кожне ребро має напрямок, тобто є впорядкованою парою (u, v);
- неорієнтований граф – ребра не мають напрямку;
- зважений граф – кожне ребро має числову вагу, яка може означати вартість переходу, час виконання дії тощо;
- циклічний/ациклічний граф – визначає, чи є шляхи, що повертаються до початкової вершини;
- планарний граф – такий, що його можна зобразити на площині без перетину ребер.

У контексті програмного забезпечення, графи часто використовуються для моделювання складних взаємозв’язків між об’єктами або станами. Зокрема, у тестуванні графова модель дозволяє представити логіку переходів між екранами або станами додатку у вигляді зв'язної структури, що значно спрощує аналіз покриття тестами та генерацію тест-кейсів.

Основні компоненти графа у контексті SPA-додатку:

- вершини (стани) – екрани, компоненти або внутрішні стани інтерфейсу користувача;
- ребра (переходи) – взаємодії користувача або зміни стану, що ініціюють переміщення між вершинами (наприклад, кліки, сабміти форм, навігація); – маркування ребер – можуть включати умови (наприклад, «коректно введено пароль»), тип дії (успішна/неуспішна), ролі користувачів тощо.

Граф можна представити у вигляді:

- матриці суміжності – квадратної матриці розміром n×n (де n – кількість вершин), де елемент a[i][j] вказує на наявність ребра з вершини i в вершину j;
- списку суміжності – кожна вершина має список суміжних з нею вершин;
- edge list (список ребер) – перелік усіх пар вершин, з'єднаних ребрами.

Приклад графа для моделі авторизації SPA:

- вершини: "Головна сторінка", "Форма входу", "Кабінет користувача", "Помилка входу";
- ребра: "Клік на 'Увійти'", "Успішне введення даних", "Помилкове введення";
- умови: авторизація успішна / невдала.

Завдяки графовому представленню можливо:

- описати всі можливі сценарії переходів між елементами UI;
- формально задати набір сценаріїв, що має бути протестований;
- провести аналіз покриття – чи всі вершини/ребра були охоплені тестами;
- автоматично генерувати тестові маршрути за різними стратегіями (наприклад, повне покриття станів або мінімальний шлях).

У наступних підрозділах буде розглянуто, як саме побудова такої моделі дозволяє покращити якість тестування SPA-додатку та спростити процес написання та супроводу автоматизованих тестів.

3.4. Побудова графа для тестування SPA-додатку

3.4.1. Модель станів і переходів для SPA-додатків

Модель станів і переходів є ключовою концепцією у тестуванні SPA-додатків. Вона дозволяє формалізувати логіку навігації та взаємодії користувача з інтерфейсом. На відміну від багатосторінкових сайтів, SPA-додатки не перезавантажують сторінку повністю, а динамічно змінюють DOM у межах одного HTML-документа. Це означає, що класична модель переходів між сторінками замінюється внутрішніми переходами між станами, які можуть залежати як від дій користувача, так і від внутрішніх умов додатку [12].

У SPA-додатках стан інтерфейсу не пов’язаний з повною перезагрузкою сторінки, а реалізується через внутрішню маршрутизацію, стан Redux store та асинхронну взаємодію з API. Через це класичні підходи до моделювання навігації часто не дають повного уявлення про логіку додатку. Застосування графової моделі дозволяє формалізувати всі можливі стани та переходи в системі, створити основу для автоматизованого тестування та аналізу покриття.

Стан у SPA-додатку – це визначена конфігурація інтерфейсу користувача, зокрема рендер компонентів, значення стану у Redux, активні маршрути та модальні вікна.

Перехід – це подія, яка змінює поточний стан: натискання кнопки, сабміт форми, завершення запиту до API або навігація за посиланням.

Граф станів додатку будується за такою логікою:

- вершини графа – унікальні стани UI (наприклад, "Головна", "Увійти", "Помилка", "Кабінет користувача");
- ребра графа – події або логічні дії, що спричиняють перехід ("onClick", "onSubmit", "navigate", "dispatch action").

Основні властивості такої моделі:

- орієнтованість: кожен перехід має напрям (від одного стану до іншого);
- умовність переходів: ребра можуть мати умови ("авторизація пройдена", "помилка API", "користувач – адміністратор");
- циклічність: деякі переходи повертають користувача в попередній стан ("скасувати", "повернутися").

Класифікація переходів у SPA:

- детерміновані – передбачувані переходи на основі подій;
- недетерміновані – залежать від відповіді API або асинхронних дій;
- внутрішні – всередині одного маршруту або компонента;
- зовнішні - зміна маршруту або глобального стану (наприклад, логін → редирект).

Такий підхід дає змогу чітко формалізувати логіку навігації всередині додатку, охопити всі можливі переходи між станами, включаючи менш очевидні або рідкісні сценарії, забезпечити автоматизовану генерацію тест-кейсів для повного покриття функціональності та здійснювати аналіз покриття з метою оцінки ефективності тестування.

3.4.2. Алгоритм побудови графа

Процес побудови графа для SPA-додатку починається з аналізу інтерфейсу користувача, маршрутизації, взаємодій з API та внутрішніх змін стану, які впливають на рендеринг компонентів.

Кожен унікальний візуальний або логічний стан інтерфейсу відповідає вершині графа. Кожна взаємодія користувача або автоматичний перехід – ребру між відповідними вершинами.

Основні етапи побудови графа:

а) ідентифікація станів додатку:

1) використання даних з React Router (URL-маршрути);
2) виявлення модальних вікон, спливаючих повідомлень, внутрішніх режимів (редагування, перегляд);

б) формалізація переходів:

1) опис подій, які викликають зміну стану (onClick, Redux action);
2) облік асинхронних викликів (API, useEffect, таймери);

в) позначення умов переходів: результати валідації, перевірка прав доступу, відповіді API;

г) структуризація переходів: прямі, умовні, циклічні та зворотні переходи;

д) побудова графа у вигляді структури: використання списків суміжності або edge list: збереження у форматах GraphML, JSON або DOT;

е) візуалізація: застосування draw.io, Graphviz, Mermaid.js або інших графових бібліотек.

Такий підхід дає змогу чітко формалізувати логіку навігації всередині додатку, охопити всі можливі переходи між станами, включаючи менш очевидні або рідкісні сценарії, забезпечити автоматизовану генерацію тест-кейсів для повного покриття функціональності та здійснювати аналіз покриття з метою оцінки ефективності тестування.

3.4.3. Інструменти моделювання графів

Для побудови графів тестування існує низка інструментів, які можуть бути використані як у фазі проєктування, так і безпосередньо у процесі тестування:

- Graphviz – потужний інструмент для опису графів мовою DOT, підтримує складну візуалізацію, групування та стилізацію елементів;
- Mermaid.js – дозволяє інтегрувати графи у Markdown-документацію, підтримує flowchart, state diagram, sequence diagram;
- draw.io (diagrams.net) – зручний веб-інструмент для ручного креслення графів та блок-схем;
- Cytoscape.js – JavaScript-бібліотека для інтерактивної роботи з графами в браузері;
- yEd Graph Editor / Gephi – настільні застосунки для складних графових структур і аналітики зв’язків.

Інструмент вибирається залежно від цілей – від ручної побудови для документування до автоматизованої генерації на основі структури коду.

3.4.4. Побудова графа на основі SPA-додатку

Для прикладу побудови графа тестування було взято умовний SPA-додаток, реалізований на React з використанням Redux і React Router [13 - офіційна документація React Router та Redux Toolkit].

Основні компоненти додатку включають головну сторінку, форму входу, кабінет користувача, сторінку редагування даних, вікно підтвердження та сторінку з помилкою.

Всі переходи реалізовані через маршрути та зміну стану в Redux Store.

На рисунку 3.3 представлено діаграму станів користувацької взаємодії з SPA-додатком.

Вона відображає можливі шляхи переходів між ключовими екранами та результативні варіанти взаємодії.

На основі діаграми можна побудувати орієнтований граф переходів, де:

- вершини відповідають унікальним станам (екранам);
- ребра – подіям, що запускають зміну стану;
- маркування – умовам або сценаріям переходу.

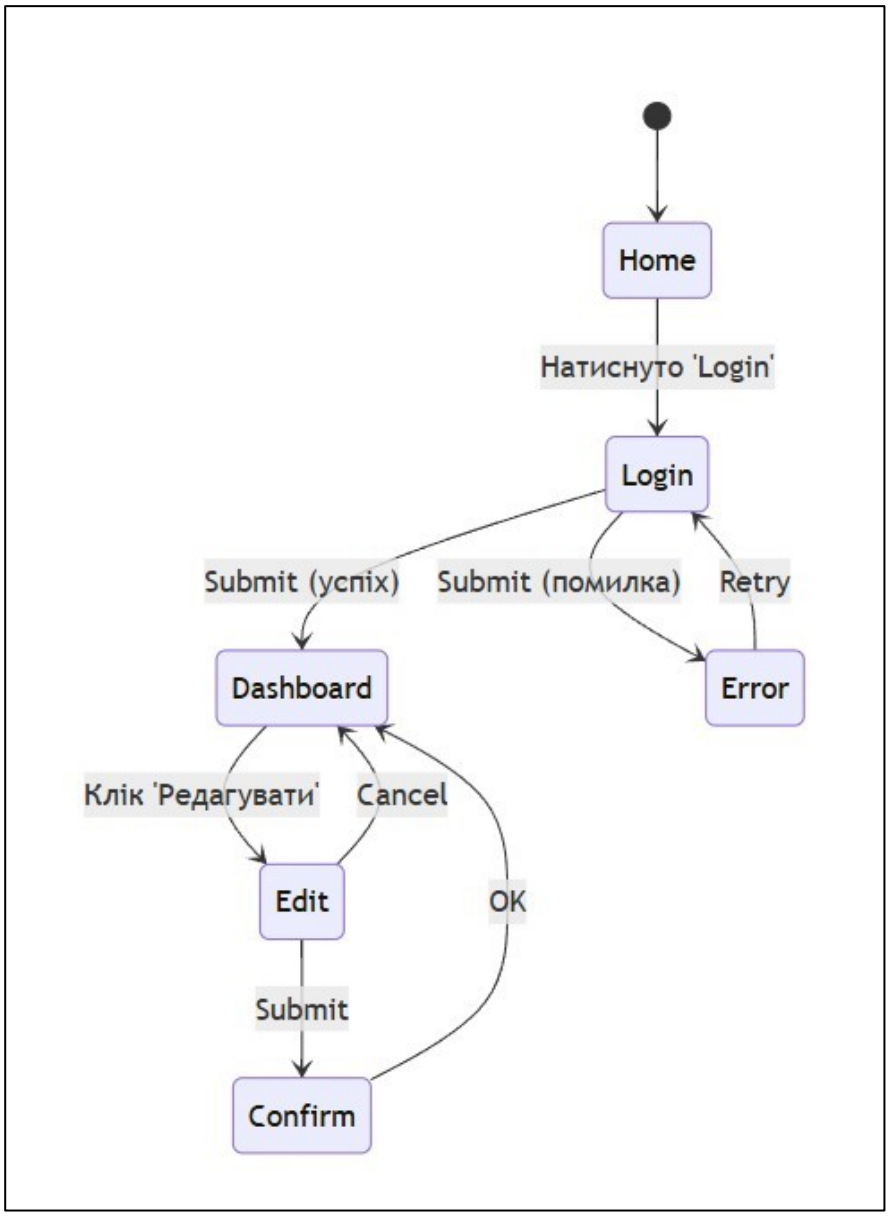

Рисунок 3.3. Діаграма станів для SPA додатку

Модель переходів формалізується як список ребер (edge list):

– (Home Page) --[Натиснуто 'Login']--> (Login Form);
– (Login Form) --[Submit: Успіх]--> (Dashboard);
– (Login Form) --[Submit: Помилка]--> (Error Page);
– (Error Page) --[Повернення]--> (Login Form);
– (Dashboard) --[Клік 'Редагувати']--> (Edit Form);
– (Edit Form) --[Submit]--> (Confirmation);
– (Confirmation) --[OK]--> (Dashboard);
– (Edit Form) --[Скасування]--> (Dashboard).

Ця модель буде мати такий вигляд у візуальному представленні (див. рис. 3.4).

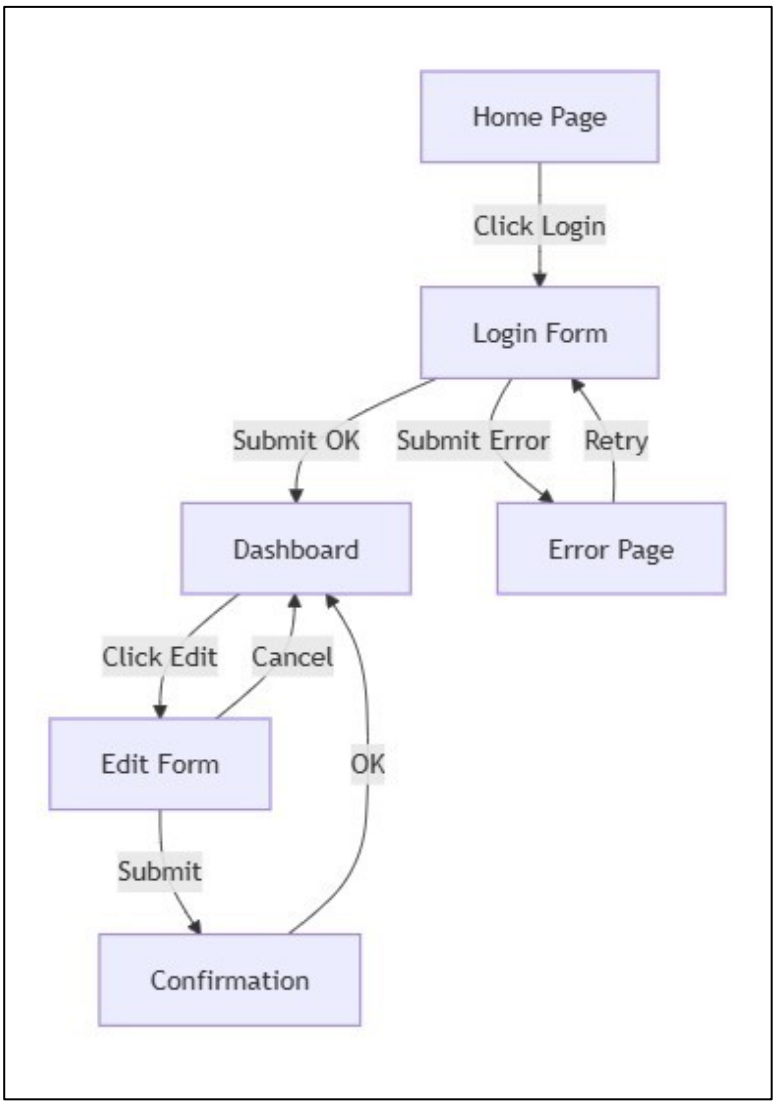

Рисунок 3.4. Графове представлення переходів станів для SPA додатку

Відповідно до рисунку можемо сформувати таблицю відповідності переходу із стану в стану за подіями (див. табл. 3.1)

Таблиця 3.1. Таблиця відносності станів та подій SPA-додатку

Стан	Подія	Перехід на стан
Головна сторінка	Натиснуто 'Login'	Форма входу
Форма входу	Submit (успіх)	Кабінет користувача
Форма входу	Submit (помилка)	Помилка авторизації
Кабінет користувача	Клік «Редагувати»	Форма редагування
Форма редагування	Submit	Підтвердження

Побудований граф дає змогу описати логіку роботи SPA-додатку у формалізованому вигляді, що дозволяє автоматично генерувати тестові сценарії, перевіряти покриття станів і переходів, а також адаптувати тести при зміні логіки додатку. Це створює основу для ефективного автоматизованого тестування з мінімальними витратами на підтримку тест-кейсів при зміні структури інтерфейсу.

3.5. Обґрунтування емпіричного методу дослідження

Теорія у поточному дослідженні представляє собою доступну інформація про технології автоматизованого тестування веб додатків: особливості, їх можлива реалізація, їх переваги та недоліки.

У поточному дослідження емпіричний метод був обраний задля порівняння різних технологій у обраному контексті.

Це метод наукового дослідження навколишньої реальності шляхом досвіду за допомогою експериментів та спостережень. Саме цей метод є найкращим серед інших через необхідність реальних вимірювань і саме у такий спосіб ми будемо мати результат, більше всього наближений до реальності.

Логічний метод пізнання було обрано головним у дослідженні. Кожна людина використовує логіку у своєму мисленні як інструмент, засіб при виконаннні різноманітних інтелектуальних дій.

Наведемо характеристики та їх значення в таблиці 3.2.

Таблиця 3.2. Опис характеристик

Характеристика	Значення
Сумісність з ОС	Підтримувані операційні системи
Характеристика	Значення
Кросбраузерність	Підтримувані інструменти браузера
Якість документації	Оцінка повноти та доступності технічної документації
Економічна ефективність	Чи є безкоштовним або ліцензованим
Популярність у спільноті	Кількість зірочок на GitHub
Мова скриптів	Мови програмування що використовуються для створення тестових скриптів
MTTR	Середній час, необхідний для відновлення роботи системи або усунення збою після його виникнення.
Chaos Experiment Success Rate	Оцінка надійності тестів при збоях

РОЗДІЛ 4. АНАЛІЗ ІНСТРУМЕНТІВ ТЕСТУВАННЯ

4.1. Вибір методів тестування

Вибір методів тестування став багатокритеріальною задачею, яку було вирішено за допомогою лінійної адитивної згортки з ваговими коефіцієнтами.

Причини вибору цього методу:

– інтуїтивна простота: Цей метод дозволяє просто підсумувати ваги й оцінки, що робить його доступним для розуміння;

– чітке представлення результатів: Лінійна модель надає просту формулу для обчислення загальної корисності;

– гнучкість: Можливість адаптувати ваги під різні проектні вимоги та специфікації;

– врахування важливості критеріїв: Загортка з ваговими коефіцієнтами є ідеальною, оскільки вона дозволяє враховувати, що не всі критерії однаковою мірою важливі для особливого випадку.

Для аналізу початково було обрано 5 варіантів тестування веб додатків (ручне, з використанням різних інструментів та різних фреймворків). Серед них:

– Cypress – сучасний E2E фреймворк, який дозволяє тестувати сучасні фронтенд-додатки (зокрема SPA). Підтримує інтерактивну перевірку DOM, легко інтегрується в CI-процеси. Має зручний GUI і підтримку роботи з моками API;
– Jest – фреймворк для unit- та інтеграційного тестування JavaScript-коду. Ідеально підходить для проєктів на React, має потужну підтримку snapshot testing, моків, таймерів, асинхронного коду;

– Playwright – інструмент для E2E тестування від Microsoft. Дозволяє тестувати декілька браузерів одночасно, має гнучкий API, підтримує візуальні тести і роботу з мобільними емуляціями;
– Selenium – класичний інструмент автоматизованого тестування браузерів. Підтримує багато мов програмування та браузерів. Вимагає значно більше налаштувань, але залишається стандартом для багатьох великих систем;
– Postman – платформа для тестування REST API. Дозволяє створювати інтеграційні тести, проводити тестування запитів, перевіряти структуру відповіді та статус-коди. Добре підходить для мікросервісної архітектури.

4.1.1. Критерії вибору інструментів

Для задачі багатокритеріального вибору були визначенні такі критерії:

– надійність (R1) – стабільність виконання тестів, здатність виявляти та фіксувати помилки без великої кількості хибнопозитивних або хибнонегативних результатів. Цей критерій вважається найважливішим, оскільки тест, який не забезпечує точності, не має практичної цінності;
– швидкодія (R2) – оцінює час виконання тестів та можливість інтеграції в CI/CD пайплайни. В умовах Agile та DevOps важливо, щоб тести не затримували розгортання продукту та швидко виявляли регресії;
– інтеграція з технологіями (R3) – легкість та простота інтеграції з React, Redux та іншими технологіями проєкту. Інструмент має добре працювати з компонентною структурою додатків, сучасними роутерами та асинхронними запитами;
– документація та підтримка (R4) – наявність актуальної документації, великої та активної спільноти, наявність прикладів, готових конфігурацій, можливість швидко знаходити відповіді на технічні питання;

– функціональне охоплення (R5) – наскільки добре інструмент підтримує різні типи тестування: юніт-тестування, інтеграційне тестування, тестування API та кінцеве тестування (E2E). Бажано, щоб один інструмент міг покрити декілька рівнів.

Ці критерії разом дають комплексну картину того, як інструмент тестування може бути інтегрований у реальні умови роботи. Вони допомагають вразити різні аспекти, які можуть бути критичними для успіху проекту. Охоплюють технічні, економічні та організаційні чинники, що дозволяє провести комплексний аналіз.

4.1.2. Застосування методу Парето для вибору інструментів

Для легшої обробки відображення показників кожної з альтернатив за нашими критеріями було заповнено векторний опис альтернатив, що дозволяє співставити кожну з альтернатив разом із оцінками за кожним критерієм (див. табл. 4.1).

З таблиці видно, що Cypress та Jest є лідерами з максимальними сумами балів – 46 та 45 відповідно. Cypress виявився найефективнішим для автоматизації інтеграційного та end-to-end тестування інтерфейсу користувача, оскільки він легко працює з реальним DOM, забезпечує стабільну симуляцію взаємодій та дозволяє спостерігати виконання тестів у реальному часі. Jest, у свою чергу, виявився оптимальним рішенням для модульного тестування, особливо в екосистемі React, завдяки вбудованій підтримці моків, snapshot-тестування, асинхронного коду та логічної перевірки стану компонентів.

Таблиця 4.1. Векторний опис альтернатив

Інструмент тестування	Критерії					Сума балів
	Надійність (R1)	Швидкодія (R2)	Інтеграція (R3)	Документація (R4)	Покриття (R5)	
Cypress	9	9	10	9	9	46
Jest	10	10	9	9	7	45
Playwright	8	9	8	8	8	41
Selenium	7	6	6	7	9	35
Postman	6	8	7	8	6	35

Отже, відповідно до принципу Парето, Cypress і Jest можна вважати найбільш обґрунтованими виборами при розгляді елементів для подальшого тестування вебдодатків. Але не будемо відмітати Playwright, який виступає майже на їхньому рівні. Тобто наша таблиця критеріїв приймає наступний вигляд (див. табл. 4.2).

Таблиця 4.2. Скорочена таблиця за методом Парето

Методи тестування	Надійність (R1)	Швидкодія (R2)	Інтеграція (R3)	Документація (R4)	Покриття (R5)
Cypress	9	9	10	9	9
Jest	10	10	9	9	7
Playwright	8	9	8	8	8

4.1.3. Нормування оцінок та введення вагових коєфіцієнтів

Нормування оцінок проводиться для забезпечення об'єктивності порівняння альтернатив. Для кожного критерію нормування здійснюється з використанням методу "за максимумом. Це дозволяє шкалувати оцінки в межах 0-1, де 1 є найкращою оцінкою за критерієм.

Для приведення значень до принципу "за максимумом" виконується наступне:

– визначити максимальні оцінки для кожного критерію;
– нормалізувати оцінки за формулою 4.1:

$$N_{ij} = \frac{X_{ij}}{X_{max,j}},$$

де N_{ij} – нормалізоване значення,

X_{ij} – початкове значення,

$X_{max,j}$ – максимальне значення для критерію j.

Нормалізація за цим методом має на меті:

– привести всі значення до шкали від 0 до 1, де 0 позначає найгірший результат (найнижче значення) для критерію, а 1 – найкращий (найвище значення);
– дозволити для подальшого порівняння й обчислення корисності альтернатив.

Отримаємо такі результати Отримаємо такі результати (див. табл. 4.3).

Ваги були визначені на основі експертної оцінки та аналізу критично важливих аспектів для проекту:

– надійність (0.30) – критичний критерій, оскільки помилкові результати тестів можуть призвести до серйозних дефектів у продакшн середовищі. Інструмент, що демонструє стабільність і повторюваність результатів, є беззаперечно цінним;

Таблиця 4.3. Нормалізовані значення за шкалами

Методи тестування	Нормалізовані значення за критеріями				
	Надійність (R1)	Швидкодія (R2)	Інтеграція (R3)	Документація (R4)	Покриття (R5)

Cypress	0,9	0,9	1,0	1,0	1,0
Jest	1,0	1,0	0,9	1,0	0,77
Playwright	0,8	0,9	0,8	0,88	0,88

– швидкодія (0.25) – дозволяє оперативно реагувати на зміни коду, зменшує час виконання перевірок. Особливо важлива для великих CIпайплайнів із великою кількістю тестів;
– інтеграція (0.20) – наявність сумісності з використовуваними у проекті технологіями (наприклад, React, Redux, GraphQL) дозволяє уникнути складних конфігурацій та затримок при запуску тестів;
– документація (0.15) – добре структурована документація скорочує час на освоєння інструменту, спрощує навчання нових членів команди та зменшує кількість помилок під час написання тестів;
– покриття (0.10) – інструмент повинен підтримувати різні типи тестів (unit, e2e, API), оскільки це дозволяє комплексно забезпечити якість продукту.

4.1.4. Розрахунок корисності альтернатив за лінійною адитивною згорткою з ваговими коефіцієнтами

Розрахунки були виконані на основі нормалізованих значень та відповідних вагових коефіцієнтів.

В результаті кожна альтернатива отримала оцінку, що відображає її загальну корисність (Z).

Загортка формулюється формулою 4.2.

$$Z = \max_{i=1,m} \sum_{j=1}^{n} \alpha_j \beta_j a_{ij}$$

де α_j – нормуючі множники,

β_j – вагові коефіцієнти, що відображають відносний внесок окремих критеріїв до загального критерію.

Вагові коефіцієнти прийнято вказувати вже нормованими величинами ($\Sigma\beta_j = 1$).

Нормуючі множники обчислюються за формулою 4.3.

$$\alpha_j = \frac{1}{\sum_{i=1}^{m} a_{ij}}$$

Підрахуємо нормуючі множники для наших альтернатив (формули 4.4 – 4.8).

$$\alpha_1 = \frac{1}{\sum_{i=1}^{5} a_{i1}} \approx 0{,}235$$

$$\alpha_2 = \frac{1}{\sum_{i=1}^{5} a_{i2}} \approx 0{,}233$$

$$\alpha_3 = \frac{1}{\sum_{i=1}^{5} a_{i3}} \approx 0{,}238$$

(4.6)

$$\alpha_4 = \frac{1}{\sum_{i=1}^{5} a_{i4}} \approx 0{,}212$$

(4.7)

$$\alpha_5 = \frac{1}{\sum_{i=1}^{5} a_{i5}} \approx 0{,}225$$

(4.8)

Тепер перейдемо до підрахунку корисності кожної з альтернатив. Так як підрахунки досить об’ємні, не будемо заносити їх до звіту, а виконаємо з застосуванням додаткових програмних продуктів.

Отримаємо результат:

Cypress: Z = 0,3235;

Jest: Z = 0,312;

Playwright: Z = 0,285;

Postman: Z = 0,265;

Selenium: Z = 0,222;

Результати адитивної згортки з урахуванням нормуючих множників і ваг показали, що Cypress є найефективнішим методом для тестування SPA-додатків. Він має найвищу швидкість, чудове покриття та інтеграцію, зберігаючи високу надійність. Jest займає друге місце та ідеально підходить для модульного тестування. Postman є найкращим для API. Selenium доцільно використовувати для широкої кросбраузерної перевірки, хоча він поступається за іншими параметрами. Playwright демонструє збалансовані результати, підходить як універсальний інструмент для проєктів середнього масштабу.

Таким чином, використання комбінації Cypress (для E2E і інтеграційного тестування) та Jest (для unit-тестування) є найбільш доцільним для сучасних SPAпроєктів.

4.2. Функціональні вимоги

Для проведення практичної частини дослідження, спрямованого на оцінку ефективності методів тестування веб-застосунків, зокрема тестування графами, було вирішено розробити програмну систему, що складається з двох основних компонентів:

– інтернет-магазин (тестовий застосунок): Простий інтернет-магазин, розроблений на React.js, який буде використовуватися як об'єкт для тестування. Він має базову функціональність, таку як перегляд товарів, додавання до кошика, оформлення замовлення;

– допоміжний інструмент для візуалізації: Застосунок, розроблений на React.js, який дозволяє візуалізувати граф станів інтернет-магазину, генерувати тестові сценарії на основі графа та аналізувати результати автоматизованих тестів.

Система дозволяє досліджувати ефективність тестування графами для виявлення дефектів та покращення якості програмного забезпечення.

Функціональні можливості інтернет-магазину:

– перегляд каталогу товарів з можливістю фільтрації та сортування;
– перегляд детальної інформації про товар;
– додавання товарів до кошика;
– оформлення замовлення;
– авторизація/реєстрація користувачів (за потреби).

Функціональні можливості інструменту для тестування графами:

– візуалізація графа станів інтернет-магазину;
– генерація тестових сценаріїв на основі графа (наприклад, на основі покриття вершин, ребер, шляхів);
– запуск автоматизованих тестів (з використанням Jest або Cypress); – відображення результатів тестування (покриття коду, виявлені помилки);
– збереження звітів у форматі .csv для подальшого аналізу.

Для забезпечення зручності проведення експериментів та достовірності отриманих даних, обидва застосунки мають простий та інтуїтивно зрозумілий інтерфейс.

4.3. Створення тестових додатків для проведення експерименту

4.3.1. Метрики для дослідження

Основною метою експерименту є оцінка ефективності тестування графами для виявлення дефектів та покращення якості програмного забезпечення вебзастосунків.

Вимірювання ефективності тестування графами є критично важливим, оскільки дозволяє оцінити, наскільки добре цей метод допомагає виявляти помилки та покращувати покриття коду.

Метою вимірювання ефективності є визначення оптимальних стратегій тестування, які забезпечують високу якість та надійність веб-застосунків.

Для оцінки ефективності тестування графами, будуть проводитися наступні вимірювання:

– кількість виявлених помилок – ця метрика показує, наскільки добре метод тестування графами допомагає виявляти дефекти у веб-застосунку. Помилки будуть класифікуватися за серйозністю (критичні, високі, середні, низькі) для більш детального аналізу;

– під час виконання тестових сценаріїв, згенерованих на основі графа станів, буде фіксуватися кожна виявлена помилка. Для автоматизованих тестів будуть використовуватися assertion-и (твердження) в Cypress, які автоматично фіксують помилки;

– покриття коду (вершин, ребер, шляхів графа) − ця метрика показує, наскільки повно тестові сценарії охоплюють різні частини вебзастосунку. Покриття буде вимірюватися на рівні графа станів (вершини, ребра, шляхи);

– для вимірювання покриття коду буде використовуватися інструменти для аналізу покриття коду. Ці інструменти дозволяють визначити, які частини коду були виконані під час виконання тестових сценаріїв. Для вимірювання покриття графа станів буде розроблено

спеціальний модуль, який відстежує, які вершини та ребра графа були відвідані під час виконання тестів;

– час виконання тестових сценаріїв – ця метрика показує, скільки часу потрібно для виконання тестових сценаріїв;

– час виконання тестових сценаріїв буде вимірюватися за допомогою функцій performance.now() в JavaScript. Для автоматизованих тестів час виконання буде фіксуватися автоматично, а для ручного тестування буде використовуватися секундомір;

– вартість розробки та підтримки тестових сценаріїв – ця метрика показує, скільки часу та ресурсів потрібно для розробки та підтримки тестових сценаріїв;

– буде вестися облік часу, витраченого на розробку та підтримку тестових сценаріїв. Також буде враховуватися складність розробки та підтримки тестових сценаріїв для різних методів тестування.

Отримані дані будуть використані для порівняння ефективності тестування графами з іншими методами тестування (наприклад, ручним тестуванням, функціональним тестуванням).

4.3.2. Структура проєкту та архітектурні особливості

Для проведення експериментальних досліджень з тестування веб-застосунків на основі графових моделей було розроблено програмну систему, що складається з двох основних компонентів: тестового інтернет-магазину та інструменту для тестування графами. Обидва компоненти реалізовані з використанням фреймворку React.js, що забезпечує модульність, гнучкість та зручність розробки користувацьких інтерфейсів.

Тестовий інтернет-магазин, розроблений для проведення експериментів з тестування, є спрощеною моделлю реального веб-

застосунку. Це дозволяє зосередитися на ключових аспектах тестування в контрольованому середовищі, мінімізуючи вплив зовнішніх факторів. Архітектура інтернет-магазину базується на наступних технологіях, які забезпечують гнучкість, масштабованість та зручність розробки:

– React.js Фреймворк для створення користувацького інтерфейсу, що забезпечує компонентний підхід та декларативний опис інтерфейсу. Це дозволяє розбивати складний інтерфейс на невеликі, незалежні компоненти, які легко тестувати та повторно використовувати;

– Redux: Бібліотека для управління станом застосунку, що дозволяє централізовано зберігати та оновлювати дані, доступні для всіх компонентів. Redux забезпечує передбачуваність та легкість відстеження змін стану, що є важливим для тестування;

– React Router: Бібліотека для реалізації навігації між сторінками вебзастосунку. React Router дозволяє створювати складні маршрутизації та забезпечує зручну навігацію для користувачів;

– Styled Components: Бібліотека для стилізації компонентів, що дозволяє писати CSS-код безпосередньо в JavaScript-файлах. Це спрощує процес стилізації та дозволяє створювати компоненти з власним стилем, що підвищує їх незалежність.

Для кращого розуміння структури та взаємодії компонентів системи, було розроблено діаграму компонентів React (див. рис. 4.1), яка візуалізує ієрархію компонентів інтернет-магазину та інструменту для тестування графами. Вона показує, як компоненти організовані в деревоподібну структуру, та які компоненти є батьківськими або дочірніми.

Як видно з цієї діаграми, основні компоненти інтернет-магазину, такі як Home, Catalog та Product, є дочірніми компонентами компонента App, що

забезпечує загальну структуру інтерфейсу. Це дозволяє легко змінювати та оновлювати окремі частини інтерфейсу без впливу на інші компоненти.

Для візуалізації структури Redux store було розроблено діаграму станів Redux (див. рис. 4.2), яка показує, які дані зберігаються в стані, та як вони змінюються в результаті дій користувача.

Діаграма станів демонструє, що стан кошика (cart) проходить через різні етапи, від початкового стану до оформлення замовлення, що дозволяє відстежувати зміни в кошику під час тестування.

Інструмент для тестування графами призначений для автоматизації процесу тестування інтернет-магазину на основі графової моделі. Архітектура інструменту базується на наступних технологіях:

– React.js: фреймворк для створення користувацького інтерфейсу;

– Vis.js (або React Flow): бібліотека для візуалізації графа станів;

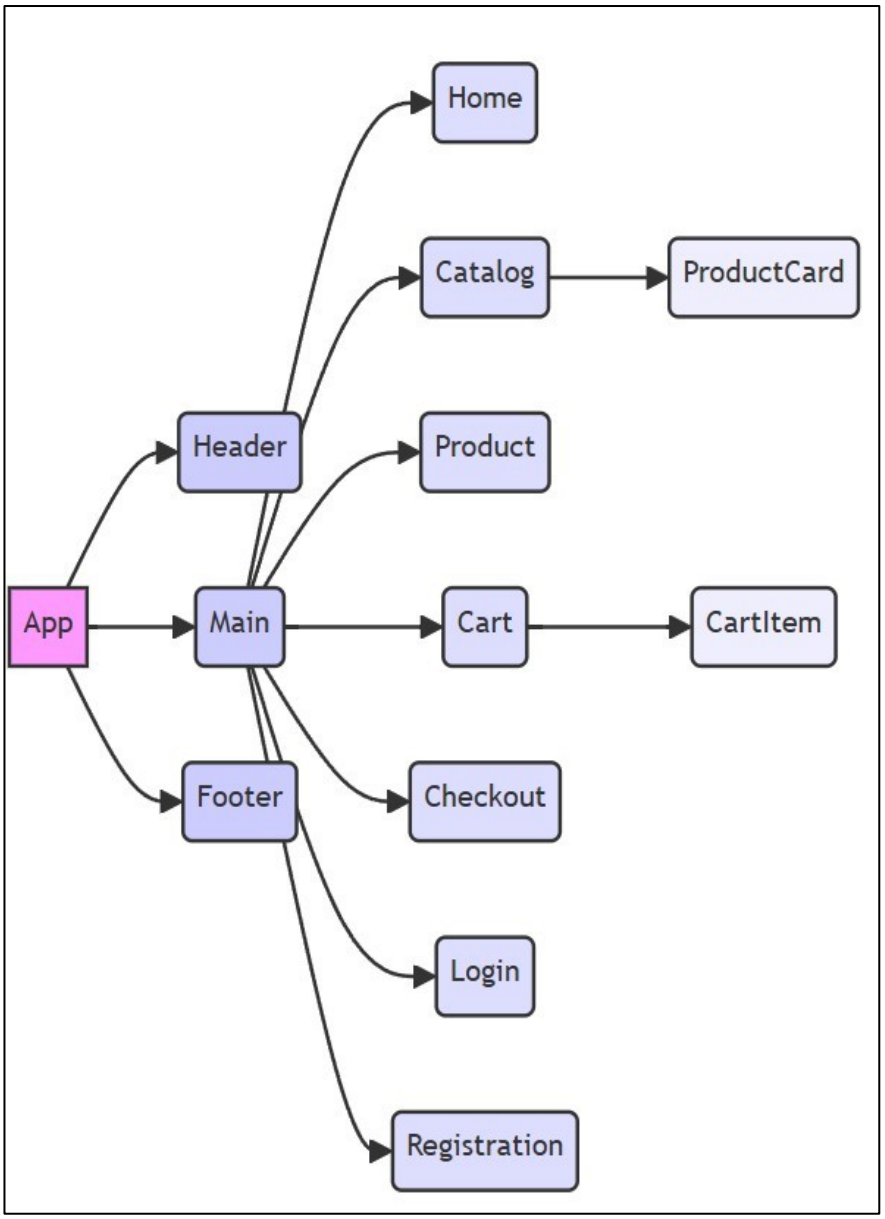

Рисунок 4.1. Діаграма компонентів інтернет-магазину

– Cypress: фреймворк для end-to-end тестування, що дозволяє перевіряти взаємодію між різними частинами системи.

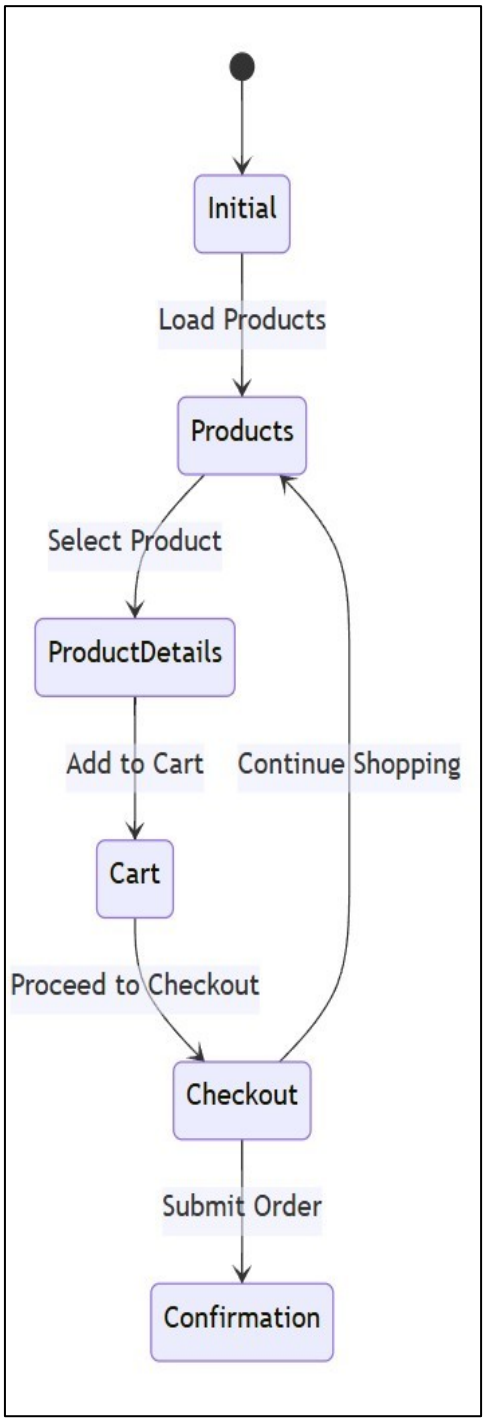

Рисунок 4.2. Діаграма станів Redux store інтернет-магазину

Для опису структури класів ScenarioGenerator, TestRunner та ReportGenerator було розроблено діаграму класів (див. рис. 4.3), яка показує їх атрибути та методи.

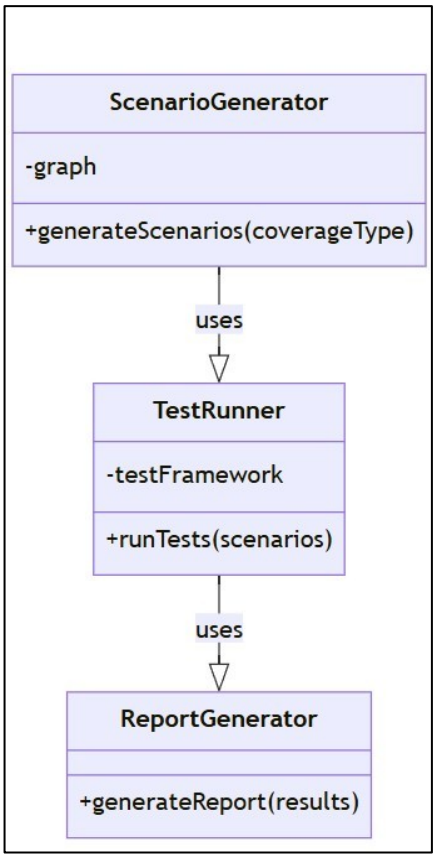

Рисунок 4.3. Діаграма класів для допоміжної програми

Діаграма показує, що клас ScenarioGenerator використовується класом TestRunner для генерації тестових сценаріїв, а клас TestRunner використовується класом ReportGenerator для генерації звітів.

Архітектурні особливості:

- модульність: кожен компонент React виконує свою окрему функцію, що полегшує розробку, тестування та підтримку коду;
- розділення відповідальності: кожен клас в каталозі src/utils/ відповідає за виконання певної задачі (генерація сценаріїв, запуск тестів, генерація звітів), що робить код більш зрозумілим та легким у використанні;
- використання бібліотек: використання популярних бібліотек, таких як Vis.js, Cypress, дозволяє спростити розробку та забезпечити високу якість коду.

Ця структура проекту та архітектурні особливості дозволяють створити ефективну та гнучку систему для проведення експериментів з тестування вебзастосунків на основі графових моделей.

4.3.3. Врахування особливостей реального об’єкта

Для забезпечення реалістичності експериментів, тестовий інтернет-магазин було розроблено з урахуванням типової поведінки користувачів та основних функціональних можливостей, притаманних сучасним веб-застосункам електронної комерції.

Інтернет-магазин реалізує наступний сценарій взаємодії з користувачем:

- перегляд каталогу товарів: користувач має можливість переглядати список доступних товарів, використовуючи різні фільтри (ціна, категорія, бренд) та сортування (за популярністю, новизною). Це дозволяє швидко знаходити потрібні товари;
- додавання товарів до кошика: користувач може додавати обрані товари до кошика, вказуючи необхідну кількість. Кошик відображає поточний перелік товарів, їх ціну та загальну суму замовлення;
- перехід на сторінку оформлення замовлення: користувач переходить на сторінку оформлення замовлення, коли готовий здійснити покупку;
- введення контактних даних та вибір способу доставки: на сторінці оформлення замовлення користувач вводить необхідні контактні дані (ім'я, прізвище, адресу доставки, номер телефону) та вибирає зручний спосіб доставки (кур'єр, пошта, самовивіз);
- підтвердження замовлення: після перевірки введених даних та вибору способу доставки, користувач підтверджує замовлення. Замовленню

присвоюється унікальний номер, а користувач отримує повідомлення про успішне оформлення.

Інструмент для тестування графами автоматизує процес тестування інтернетмагазину, використовуючи графову модель для генерації тестових сценаріїв та оцінки покриття коду. Логіка роботи інструменту має наступні етапи:

а) аналіз структури інтернет-магазину та побудова графа станів: Інструмент автоматично аналізує структуру компонентів React та маршрутизацію React Router, щоб побудувати граф станів веб-застосунку. Кожна вершина графа відповідає певному стану інтерфейсу (наприклад, "Головна сторінка", "Каталог", "Сторінка товару", "Кошик"), а ребра - переходам між цими станами (наприклад, "Натиснути кнопку 'Додати до кошика'", "Перейти на сторінку оформлення замовлення");

б) вибір типу покриття: Користувач може вибрати тип покриття, який використовуватиметься для генерації тестових сценаріїв:

1) покриття вершин (Node Coverage): тестові сценарії повинні відвідати кожну вершину графа хоча б один раз;
2) покриття ребер (Edge Coverage): тестові сценарії повинні пройти по кожному ребру графа хоча б один раз;
3) покриття шляхів (Path Coverage): тестові сценарії повинні пройти по всіх можливих шляхах графа певної довжини.

в) генерація тестових сценаріїв: на основі графа станів та вибраного типу покриття, інструмент автоматично генерує тестові сценарії. Кожен сценарій представляє собою послідовність дій користувача, які необхідно виконати для досягнення певного стану або переходу;

г) запуск автоматизованих тестів: згенеровані тестові сценарії запускаються автоматично за допомогою фреймворків Jest (для unit-тестування компонентів) та Cypress (для end-to-end тестування взаємодії між сторінками);

д) збір результатів тестування та генерація звіту: після завершення тестування, інструмент збирає результати (кількість виявлених помилок, покриття коду, час виконання) та генерує звіт у форматах .json та .csv. Звіт містить детальну інформацію про кожен тестовий сценарій, включаючи перелік виконаних дій, результат тестування та покриття коду.

Для візуалізації процесу генерації тестових сценаріїв на основі графа станів було розроблено діаграму діяльності (див. рис. 4.4), яка показує, які кроки виконуються для створення сценаріїв з різним покриттям.

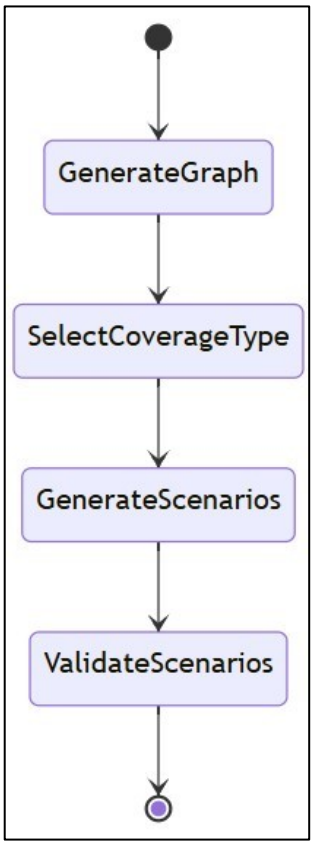

Рисунок 4.4. Діаграма діяльності

Діаграма діяльності демонструє, що процес генерації тестових сценаріїв починається з генерації графа станів, після чого вибирається тип покриття та генеруються тестові сценарії.

4.4. Генерація моделі графа станів та переходів

Графо-орієнтований підхід до E2E-тестування SPA-додатків починається з побудови формальної моделі станів і переходів користувача. У програмній системі ця модель формується автоматично, без ручного опису сценаріїв. Реалізуємо автоматичний збір моделі маршрутів та переходів SPA-додатка, використовуючи конфігурацію React Router як єдине джерело істини. Переходи між станами формуються за двома механізмами:

– маршрутизовані посилання <Link to="..."> – клієнтські посилання, що перемикають URL без повного перезавантаження Single Page Application, а лише віртуальний DOM;

– програмні навігаційні виклики navigate('/next') з React Router. Усі виявлені парні (from, to) записуються як об’єкти { from: string, to: string } у множину edges.

В результаті формується JSON-файл graph.json, що містить масив рядків nodes, у якому перераховано маршрути, таких як /, /login, /dashboard, /edit, /confirm, /error, /coverage, та масив об’єктів edges з парами {from, to}, що відображають реактивні перемикання між цими маршрутами.

Для реалізації цього механізму використовується скрипт scripts/extractgraph.cjs, який спочатку читає вміст файлу src/App.tsx за допомогою Node.js API (fs.readFileSync), а потім передає його до Babel Parser. Конфігурація парсера включає плагіни typescript і jsx, що дозволяють коректно обробити сучасний JS-код із TypeScript-типами та розмітку JSX. Після побудови AST виконується обхід дерева за допомогою @babel/traverse, який в алгоритмічному вигляді описується так:

```
parse code → AST;
for кожного вузла
AST:
  якщо це JSXOpeningElement і name === 'Route':     знайти
атрибут 'path' → nodes.add(path);   якщо name === 'Link':
```

```
знайти атрибут 'to' → edges.add({from: current, to:
path}); for кожного CallExpression:
  якщо callee.name === 'navigate':
    взяти аргумент → edges.add({from: current, to: arg});
serialize {nodes, edges} → graph.json;
```

У межах цієї процедури застосовується Set для унікалізації значень, що дозволяє уникнути дублювань маршрутів та переходів. Після обходу AST множини конвертуються в масиви через Array.from() і записуються у файл з відступом у два пробіли для зручності читання. Консольний вивід повідомляє про кількість зібраних вершин та ребер, що спрощує відлагодження.

Увесь код скрипту ви можете переглянути у додатку Ґ.

Після виконання цих операцій скрипт створює файл graph.json, який потім служить основою для генерації E2E-тестів.

Генерація тестів відбувається наступним скриптом scripts/generate-tests.cjs, який читає graph.json і будує список суміжності для обходу. У залежності від параметрів командного рядка (--coverage=node|edge|path, --strategy=dfs|bfs|random) алгоритм формує набір унікальних шляхів до заданої глибини MAX_DEPTH. Після цього кожен шлях розгортається на конкретні URL з підстановкою значень динамічних параметрів з файлу route-values.js. Для кожного тестового сценарію скрипт створює файл-спек у cypress/e2e/generated, в якому послідовно виконуються кроки:

```
cy.visitNode(route);          // запис вузла
cy.injectAndCheckA11y(route); // перевірка доступності axe-
core cy.measurePerformance(route); // збір часу завантаження
сторінки cy.url().should('include', route); // перевірка
правильності URL cy.wait(500);
cy.transitionTo(prev, route); // запис переходу
```

Під час виконання Cypress-раннера всі зібрані дані проходять через низку спеціалізованих тасків: recordNode, recordEdge, recordPerf і recordA11y. Кожен із них відповідає за окремий аспект тестового прогону: перший фіксує відвідані маршрути та UI-стани, другий – послідовність переходів між ними,

третій – ключові показники продуктивності (від часу першої байт-відповіді до завершення візуального рендерингу), а четвертий – кількість та типи порушень доступності згідно зі стандартами WCAG. Після того, як усі тести завершено, зібрані метрики автоматично зберігаються у відповідних JSON-файлах, забезпечуючи чітку структуру даних для подальшого аналізу та візуалізації:

– coverage/graph-coverage.json – переліки відвіданих вершин і ребер;

– coverage/perf-metrics.json – масиви часів завантаження по маршрутах;

– coverage/a11y-metrics.json – кількість порушень WCAG для кожного маршруту.

Отримані файли автоматично підтягуються в React-додатку CoverageDashboard.tsx, де за допомогою ReactFlow [14] формується інтерактивний граф станів із можливістю масштабування та фільтрації відвіданих шляхів, а Recharts створює діаграми – PieChart для відображення рівня покриття, BarChart і LineChart для порівняльного аналізу часу завантаження та таблицю з деталями доступності.

Користувач може в будь-який момент експортувати повні звіти у формати JSON або CSV через Blob API, який генерує файли «на льоту», а також ініціювати повторний прогін тестів без виходу з інтерфейсу, під’єднавшись до SSEендпоінта /api/run-tests. Така комплексна інтеграція гарантує повний цикл автоматизованого тестування, динамічну візуалізацію результатів та оперативний аналіз продуктивності і доступності SPA-додатка (див. рис. 4.5).

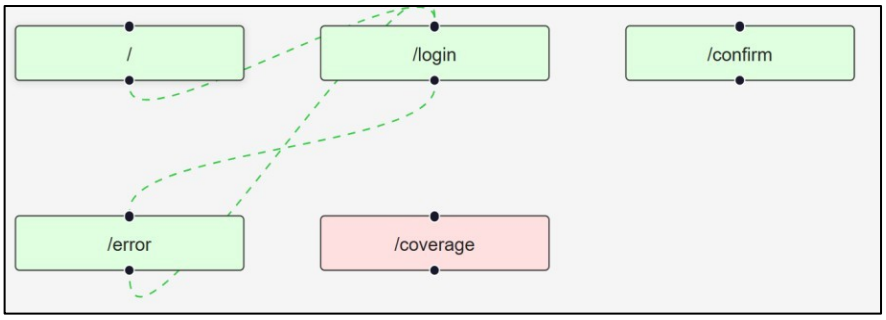

Рисунок 4.5. Візуалізація графа додатку

4.5. Автоматична генерація E2E-тестів

Програмна система забезпечує повністю автоматизований конвеєр від створення моделей маршрутів до виконання сценаріїв і відображення результатів. Основним компонентом цього процесу є скрипт scripts/generate-tests.cjs, який перетворює вміст файлу graph.json у набір Cypress-спеків, паралельно збираючи критичні метрики якості.

Перший крок полягає у зчитуванні графової моделі:

```
const graph = fs.readJSONSync('graph.json');
```

Тут graph.nodes – масив рядків із маршрутами /, /login, /dashboard тощо, а graph.edges – масив об’єктів {from, to}. Наступним кроком формується список суміжності (adjacency list), який прискорює пошук сусідів для кожної вершини:

```
const adj = {};
graph.nodes.forEach(n => adj[n] = []);
graph.edges.forEach(e => adj[e.from].push(e.to));
```

Користувачу доступні два конфігураційних параметри: максимальна довжина шляху MAX_DEPTH (за замовчуванням 5) і вибір алгоритму обходу: глибинного пошуку (DFS), пошуку в ширину (BFS) або випадкового обходу (Random Walk). Ці параметри передаються скрипту через змінні оточення або бібліотеку yargs.

Після ініціалізації початкових змінних скрипт запускає одну з функцій обходу, реалізацію яких можна подивитися у Додатку А.

У разі BFS або Random Walk метод реалізується відповідно до класичних алгоритмів: BFS проходить рівнями від кожної початкової вершини до заданої глибини, а Random Walk генерує випадкові послідовності переходів довжиною до MAX_DEPTH.

Кожна згенерована послідовність переходів (tests) перетворюється у Cypressспек, де visitNode фіксує відвідання вершини, transitionTo – ребро; injectAndCheckA11y перевіряє доступність на кожному кроці, а measurePerformance використовує window.performance.timing для вимірювання часу завантаження (див. рис. 4.6).

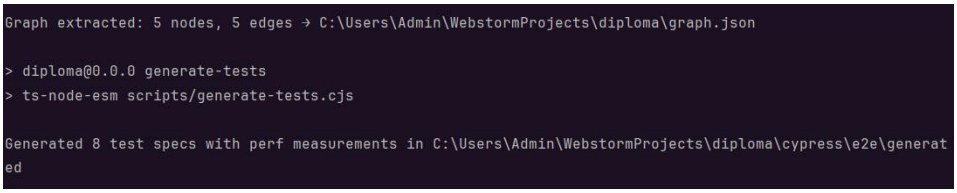

Рисунок 4.6. Консольний вивід результату виконання генерації тестів

Під час прогону тестів Cypress налаштовано на виконання таких плагін-тасків (setupNodeEvents):

- recordNode(route): запис вузлів у масив cov.visitedNodes → coverage/graphcoverage.json;
- recordEdge({from,to}): запис ребер у масив cov.visitedEdges;
- recordPerf({route,loadTime}): збереження часу завантаження кожного маршруту → coverage/perf-metrics.json;
- recordA y({ route,violationsCount}): кількість порушень доступності → coverage/a 1 1 y – metrics.json.

Після завершення всіх спеців плагіни автоматично записують JSON-файли, які є основою для візуалізації у компоненті CoverageDashboard, де відображатимуться такі метрики:

- Node Coverage: відношення унікальних visitedNodes до загальної кількості graph.nodes, відображається Pie Chart;
- Edge Coverage: відношення visitedEdges до graph.edges, також у Pie Chart;
- Performance Metrics: масиви часів завантаження, із середнім, мінімальним та максимальним значеннями, в Bar Chart та таблиці;
- Accessibility Metrics: агрегація кількості порушень WCAG, виводиться у таблиці.

Результати генерації та виконання тестів потрапляють у інтерактивний дашборд, де їх можна аналізувати, експортувати та порівнювати ефективність різних підходів до тестування SPA-додатків.

4.6. Збір та обробка метрик

Програмна система дозволяє автоматично збирати детальні метрики продуктивності та доступності під час прогону E2E-тестів, що дає змогу оцінити реальний користувацький досвід та відповідність стандартам.

Для точного вимірювання часу завантаження кожної сторінки у тестах використовується команда measurePerformance, яка звертається до об’єкта window.performance.timing. Ця команда ін'єктує код у контекст браузера та обчислює різницю між loadEventEnd та navigationStart, отримуючи час у мілісекундах. В результаті формується файл perf-metrics.json, де для кожного шляху зберігається масив значень часу. Приклад виклику:

```
Cypress.Commands.add('measurePerformance', (route)
=> {   return cy.window().then(win => {     const
t = win.performance.timing;
    const loadTime = t.loadEventEnd - t.navigationStart;
return cy.task('recordPerf', { route, loadTime });
```

```
  });
});
```

В результаті формується файл perf-metrics.json, де для кожного шляху зберігається масив значень часу, який може бути представлено наступною таблицею (див. табл. 4.4)

Таблиця 4.4. Дані про швидкість завантаження для сторінок тестового застосунку

Стан	Відвідування	Сер. швидкість	Мін. швидкість	Макс. швидкість
/	3	125	94	178
/login	7	75	61	92
/dashboard	4	64	56	74
/error	4	74	62	84
/confirm	1	94	94	94

Після збору даних в інтерфейсі дашборду ці метрики візуалізуються:

- BarChart (див. рис. 6.7) для середнього часу завантаження по маршрутах, з позначенням мінімальних і максимальних значень у таблиці;
- таблиця для a11y-metrics.json, що відображає кількість порушень доступності для кожного шляху.

Гістограма демонструє розподіл часу завантаження: довгі хвости вказують на повільні сторінки, що потребують оптимізації. Таблиця доступності дозволяє швидко виявити маршрути з високою кількістю помилок WCAG (див. рис. 4.7).

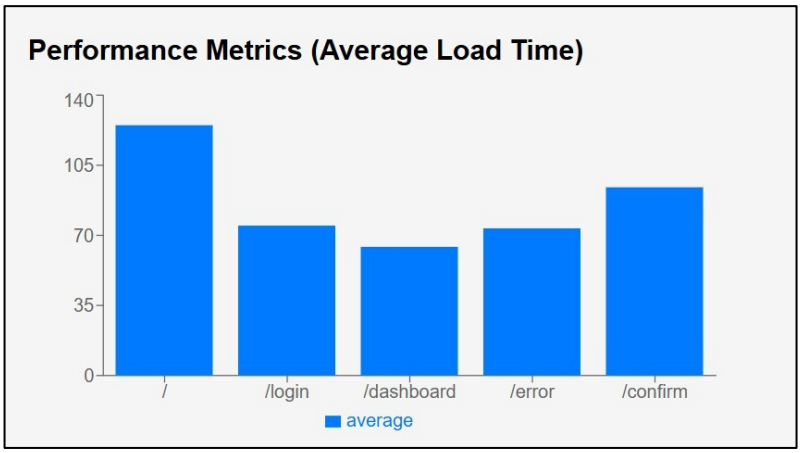

Рисунок 4.7. Діаграма продуктивності тестового застосунку

На основі цих метрик можна приймати рішення щодо оптимізації SPA:

– зменшити розмір бандла на /edit;

– додати альтернативний текст до зображень на /login;

– повторно запускати тести після впровадження виправлень і відстежувати динаміку змін у JSON-метриках.

Цей підхід забезпечує гнучкий механізм вимірювання якості веб-додатка в режимі CI/CD, дозволяючи вручну чи автоматично відстежувати ключові показники протягом усієї розробки

4.7. Інтерактивна візуалізація та дашборд

Візуалізація результатів тестування реалізована у компоненті CoverageDashboard.tsx, що поєднує ReactFlow для графа станів і Recharts для діаграм, забезпечуючи адаптивний і зручний інтерфейс.

ReactFlow використовується для відображення драґабельних вузлів і контролів масштабування (див. рис. 4.8). Вузли можна переміщувати мишею, а панель Controls дозволяє зум та панорамування.

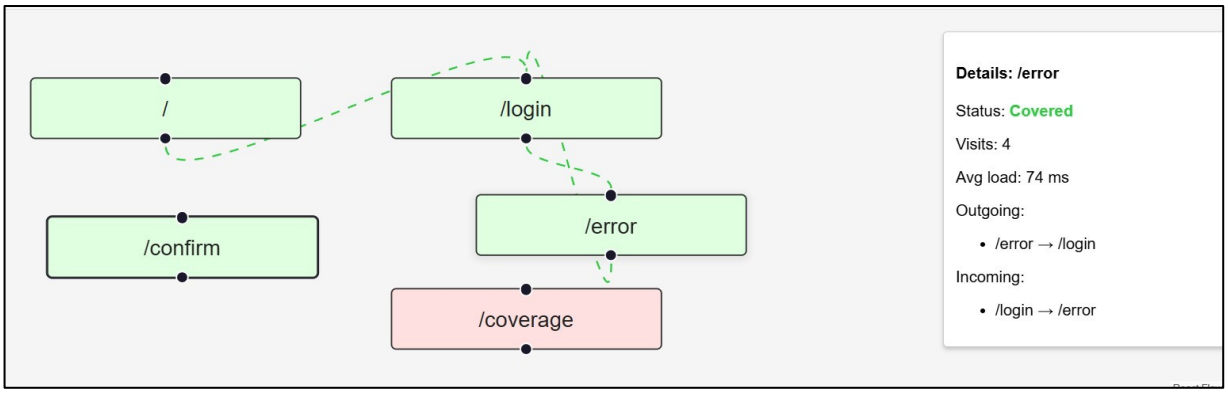

Рисунок 4.8. Візуалізація результатів тестування у вигляді графу

Додаткові елементи інтерфейсу дашборду представлені на рисунку 4.9.

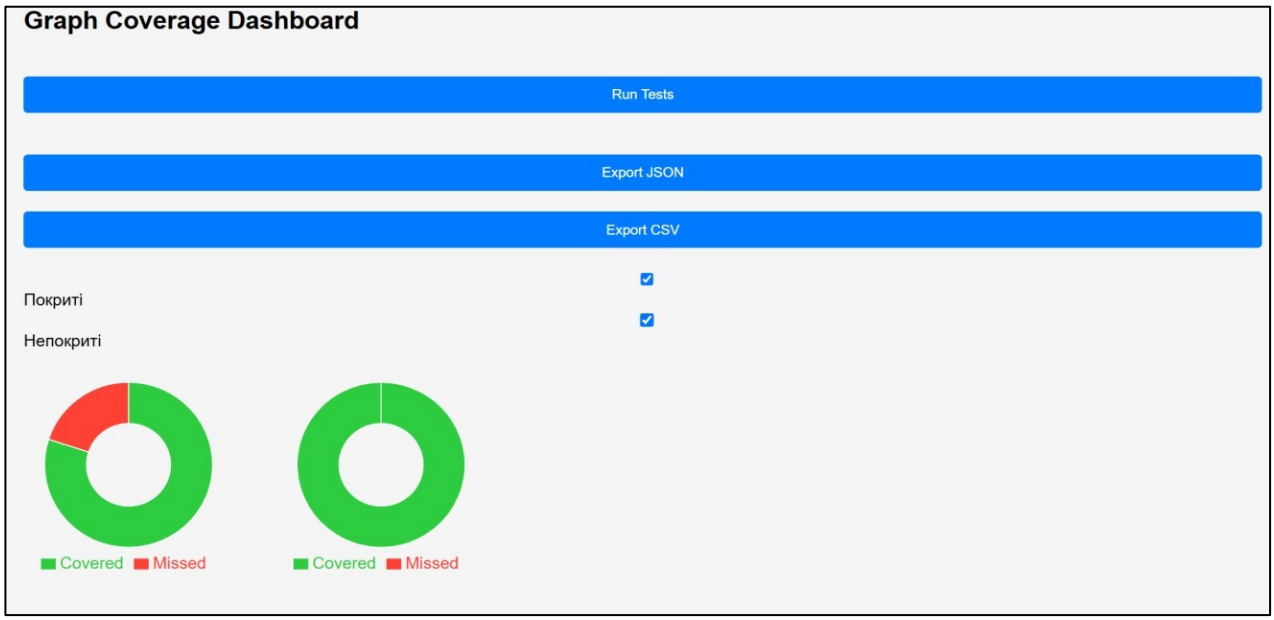

Рисунок 4.9. Додаткові елементи інтерфейсу дашборду

Інтерактивність дашборду включає:

- фільтри покриття (чекбокси): два чекбокси showCovered і showMissed керують відображенням вузлів і ребер, що дозволяє миттєво бачити лише покриті або лише непокриті елементи;
- hover-деталі: наведення на вузол викликає onNodeMouseEnter, що зберігає hoveredNode, і виводить фіксовану панель з інформацією про статус

(Covered/Missed), кількість візитів, середній час та списки вхідних/вихідних ребер. onNodeMouseLeave повертає стан у null;

- експорт звітів: кнопки Export JSON та Export CSV використовують Blob API для формування та завантаження звітів;
- Run Tests: кнопка викликає SSE-ендпоінт /api/run-tests, отримані логи виводяться в окремий тег.

4.8. Переваги, обмеження та перспективи

Розвинена автоматизація програмної системи приймає на себе всі рутинні операції: від автоматичного аналізу маршрутизації React Router до запуску Cypresstестів та збору метрик. Це дозволяє значно скоротити час ручної розробки та підтримки тестів, мінімізувати ризик людських помилок та гарантувати консистентність прогона сценаріїв при кожній збірці. Для структурованої оцінки сильних та слабких сторін, можливостей і загроз рекомендовано застосувати загальноприйняті підходи до SWOT-аналізу [15].

Сильні сторони:

- автоматизація: єдиний виклик до npm-скриптів запускає весь конвеєр, що робить тестування відтворюваним та незалежним від виконавця;
- інтеграція метрик: одночасний збір Node/Edge Coverage, Performance Metrics та Accessibility рішень забезпечує комплексний огляд якості;
- модульність: чітке розділення коду – скрипти Node.js для екстракції графа та генерації тестів, конфігурація Cypress для збору метрик, Reactкомпонент для візуалізації – полегшує розширення та підтримку.

Обмеження:

- статична маршрутизація: система концентрується на <Route>конфігурації у src/App.tsx; динамічно згенеровані чи умовні маршрути (наприклад, з параметрами, залежними від стану) можуть не

враховуватися без додаткової налаштування route-values.js;

- необхідність запущеного сервера: для виконання Cypress E2E-тестів потрібен живий dev-сервер (npm run dev), що ускладнює паралельні прогони у CI без додаткової конфігурації контейнерів або серверів.

Можливості подальшого розвитку:

- Mutation Testing: інтеграція Stryker для оцінки міцності згенерованих сценаріїв – наскільки вони виявляють штучно внесені мутації;
- історія покриття: накопичення JSON-метрик у базі даних або файловому сховищі, побудова трендів графового покриття, часу завантаження та доступності;
- API-контрактне тестування: автоматична генерація тестів для бекендендпоінтів за допомогою тих самих графових алгоритмів, перевірка схеми відповіді через cy.request та AJV.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було проведено детальний аналіз існуючих методів тестування SPA-додатків, розроблених на React і Redux.

Було визначено, що традиційні методи недостатньо ефективні для тестування динамічних і складних веб-застосунків через високі вимоги до покриття сценаріїв та інтеграції з сучасними технологіями.

Розроблено та реалізовано програмну систему, що використовує графову модель для автоматизації тестування SPA-додатків.

Основні переваги розробленого підходу:

- повна автоматизація процесу від генерації графів до виконання тестів;
- інтеграція метрик продуктивності та доступності для комплексної оцінки якості;
- гнучкість і масштабованість завдяки модульній архітектурі та чітко розмежованим обов'язкам компонентів.

За допомогою експериментального дослідження було підтверджено, що використання графових моделей значно підвищує ефективність автоматизованого тестування.

Виявлено та кількісно оцінено переваги використання Cypress та Jest порівняно з іншими інструментами (наприклад, Selenium, Postman).

Встановлено, що Cypress найкраще підходить для інтеграційного та E2E-тестування завдяки високій швидкодії та інтеграції з технологіями React і Redux, а Jest є ідеальним рішенням для модульного тестування завдяки підтримці моків та snapshot-тестування.

Під час виконання роботи визначено обмеження методу, пов’язані з необхідністю запущеного сервера та статичною маршрутизацією, і запропоновано подальші напрямки розвитку: впровадження Mutation Testing, створення історії покриття та інтеграція API-контрактного тестування.

Отримані результати підтверджують актуальність обраного методу та ефективність застосування графових моделей для тестування сучасних SPA-додатків. Це дозволяє рекомендувати розроблену систему до впровадження у процеси розробки та тестування веб-застосунків.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Rahman S. S., Dekkati S. Revolutionizing Commerce: The Dynamics and Future of E-Commerce Web Applications. *Asian Journal of Applied Science and Engineering*. 2022. Vol. 11, no. 1. P. 65–73.

URL: https://doi.org/10.18034/ajase.v11i1.58

2. Advancements in Natural Language Processing (NLP) Revolutionize Data Analysis and Decision-Making. *https://datasciconnect.com/*.

URL: https://datasciconnect.com/advancements-in-natural-language-processing-nlprevolutionize-data-analysis-and-decision-making/

3. Maniyka J., Roxburgh C. https://datasciconnect.com/advancements-innatural-language-processing-nlp-revolutionize-data-analysis-and-decisionmaking/. *https://datasciconnect.com/advancements-in-natural-language-processingnlp-revolutionize-data-analysis-and-decision-making/*. 2011. No. 2. P. 1–10.

4. Single page application. *avskedad utan varning - valutacwzv.netlify.app*.

URL: https://valutacwzv.netlify.app/80194/43694.html

5. Page A. Don't Blame Me. *The Weasel Speaks | Alan Page | Substack*. URL: https://angryweasel.substack.com/p/dont-blame-me

6. Turevska O., Shubin, I. Improving the automated testing of Web-based services by reflecting the social habits of target audiences // 2015 Information Technologies in Innovation Business Conference, ITIB 2015 - Proceedings, 2015, с. 9396, 7355062

7. Web Application Testing: Types, Process, and Best Practices. *Simform - Product Engineering Company*. URL: https://www.simform.com/blog/web-applicationtesting/

8. Verisign Domain Name Industry Brief [Електронний ресурс] - URL:

https://blog.verisign.com/domain-names/verisign-q1-2023-the-domain-name-industry-brief /

9. How To Automate Testing Web Application: Step-by-Step Instructions. QA and Software Testing Company - Luxe Quality. URL: https://luxequality.com/blog/howto-automate-testing-web-application/

10. Otálora J. A clean frontend architecture for React and Redux. Medium. URL: https://juanoa.medium.com/a-clean-frontend-architecture-for-react-and-reduxa11ee1db5560

11. Testing Single Page Applications with Cypress. Cypress.io Blog. URL: https://www.cypress.io/blog/2020/10/09/testing-spa-applications-with-cypress/

12. Потьомкін, М. М., Седляр, А. А., Дейнега, О. В., & Зварич, А. О. (2021). Комплексне використання принципу Парето та методу аналізу ієрархій для підвищення обґрунтованості результатів ранжування альтернатив. *Кібернетика та системний аналіз*.

13. React Router Redux. Офіційна документація. URL: https://github.com/reactjs/react-router-redux

14. Recharts. Official Documentation. URL: https://recharts.org/en-US/guide

15. Conducting a SWOT Analysis. MindTools. URL: https://www.mindtools.com/pages/article/newTMC_05.htm

16. Kozhevnikov, Andrii & Nataliya, Bilous. (2022). Research of methods for determining the accuracy of metrological measurements. Technology audit and production reserves. 3. 18-23. 10.15587/2706-5448.2022.259139

17. Shulika, Oleksiy & Safonov, Ivan & Ivanov, P.S. & Lysak, Volodymyr & Sukhoivanov, Igor & Lesna, Natalya. (2003). Comprehensive simulation of MQW semiconductor lasers by using laser CAD III. 80- 83. DOI: 10.1109/LFNM.2003.1246081

18. Client-Server Architecture - URL: https://darvishdarab.github.io/cs421_f20/docs/readings/client_server/

19. Dharmaadi, I & Athanasopoulos, Elias & Turkmen, Fatih. (2024). Fuzzing Frameworks for Server-side Web Applications: A Survey. 10 с.

20. Common Web Application Architectures. [Електронний ресурс] - URL: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-we b-application-architectures

21. The Practical Test Pyramid / Martin Fowler. [Електронний ресурс] - URL: https://martinfowler.com/articles/practical-test-pyramid.html

22. Graphical reporting for Selenium Webdriver and TestNG. [Електронний ресурс] - URL:

https://stackoverflow.com/questions/34607135/graphical-reporting-for-selenium-webdriver -and-testng

23. Selenium (software). [Електронний ресурс] - URL: https://en.wikipedia.org/wiki/Selenium_(software)

24. Bhimanapati, Viharika & Goel, Punit & Jain, Ujjawal. (2024). Leveraging Selenium and Cypress for Comprehensive Web Application Testing. Journal of Quantum Science and Technology. 66-79 с.

25. What is Katalon Studio? An Introduction [Електронний ресурс] - URL: https://q-pros.com/what-is-katalon-studio-an-introduction/

26. An Introduction To TestComplete And Its Features! [Електронний ресурс] - URL: https://thinkpalm.com/blogs/an-introduction-to-testcomplete-and-its-features/

27. Dharmaadi, I & Athanasopoulos, Elias & Turkmen, Fatih. (2024). Fuzzing Frameworks for Server-side Web Applications: A Survey. 22-32 с.

28. Правило максимізації корисності і крива попиту [Електронний ресурс] - URL: https://stud.com.ua/13511/ekonomika/pravilo_maksimizatsiyi_korisnosti_kriva_popitu

29. Flaky Test Management [Електронний ресурс] - URL: https://docs.datadoghq.com/tests/flaky_test_management/

30. Chaos Experiments [Електронний ресурс] - URL: https://developer.harness.io/docs/chaos-engineering/use-harness-ce/experiments/

31. Bunkley, Nick (2008). "Joseph Juran, 103, Pioneer in Quality Control, Dies". 10-12 с.

32. М.О Медиковський, О.Б. Шуневич (2011). Дослідження ефективності методів визначення вагових коефіцієнтів важливості. с. 176-177.

ДОДАТОК А. ТЕКСТ ПРОГРАМИ

Код скрипта /extract-graph.cjs

```
      const code = fs.readFileSync('src/App.tsx', 'utf-8');

      const ast = parse(code, { sourceType: 'module', plugins:
      ['typescript','jsx']
}); const nodes = new

      Set(); const

      edges = new

      Set();

      traverse(ast, {

        JSXOpeningElement({node}) {

      if (node.name.name === 'Route')

      {       node.attributes.forEach

      (attr => {

              if (attr.name.name === 'path' && attr.value.type ===
      'StringLiteral')
{           nodes.add(attr.value.va

      lue);

              }

            });     }     if

      (node.name.name === 'Link') {

      node.attributes.forEach(attr =>

      {

              if (attr.name.name === 'to' && attr.value.type ===
      'StringLiteral')
{

                edges.add(JSON.stringify({ from: '<dynamic>', to:
      attr.value.value
}));

              }
```

```
      });
    }
  },
  CallExpression({ node })
{     if (node.callee.name ===
'navigate') {       const arg =
node.arguments[0];       if
(arg.type === 'StringLiteral') {
edges.add(JSON.stringify({ from:
'<dynamic>', to: arg.value }));
      }
    }
  }
});
```

Код скрипта /generate-tests.cj

```
#!/usr/bin/env node

/**

*   scripts/generate-tests.cjs

*   CommonJS script to generate Cypress E2E test specs from
    graph.json

*   with performance measurements inserted automatically.

*   Usage:

*   npm install --save-dev fs-extra  *   node scripts/generate-

    tests.cjs

 */ const fs =

require('fs-extra');

const path =

require('path');


// Configurable max depth via environment variable or default

to 5 const MAX_DEPTH = parseInt(process.env.MAX_DEPTH, 10) ||

5;

 // Paths const projectRoot = path.resolve(__dirname,

'..'); const graphPath = path.join(projectRoot,

'graph.json'); const outDir = path.join(projectRoot,

'cypress', 'e2e', 'generated');


// Load graph if (!fs.existsSync(graphPath))

{     console.error('Error: graph.json not found. Run extract-

graph first.');     process.exit(1);

}
```

```
const graph = fs.readJSONSync(graphPath);


// Build adjacency list const adj =
{}; graph.nodes.forEach(n =>
{ adj[n] = []; });
graph.edges.forEach(e => {     if
(adj[e.from])
adj[e.from].push(e.to);
});


// Collect all simple paths up to
MAX_DEPTH via DFS const paths = [];
function dfs(current, visited, pathArr) {
if (pathArr.length > 1)
{          paths.push(pathArr.slice());
    }     if (pathArr.length >=
MAX_DEPTH) return;
(adj[current] || []).forEach(next
=> {          if (!
visited.has(next))
{              visited.add(next);
pathArr.push(next);
dfs(next, visited, pathArr);
pathArr.pop();
visited.delete(next);
        }
    });
```

```
}

graph.nodes.forEach(start => {
dfs(start, new Set([start]),
[start]);
});


// Prepare output directory
fs.ensureDirSync(outDir);


// Generate a Cypress spec for each path, including performance
measurements paths.forEach((p, idx) => {     const filename =
`test_path_${idx + 1}.cy.js`;     const lines = [
        `describe('Path ${idx + 1}: ${p.join(' -> ')}', () => {`,
        `  it('navigates sequence and measures performance', ()
=> {`
    ];


    p.forEach((route, step) => {         if (step ===
0) {             lines.push(`    cy.visitNode('$
{route}');`);             lines.push(`
cy.injectAndCheckA11y('${route}');`);
lines.push(`    cy.measurePerformance('${route}');`);
         } else {              lines.push(`    cy.transitionTo('$
{p[step  -  1]}',  '${route}');`);                    lines.push(`
cy.injectAndCheckA11y('${route}');`);               lines.push(`
cy.measurePerformance('${route}');`);
```

```
            }         lines.push(`
      cy.url().should('include', '${route}');`);
      lines.push(`    cy.wait(500);`);

          });
      lines.push('  });',
      '});');
          fs.writeFileSync(path.join(outDir, filename), lines.join('\
n'), 'utf8');
      }
      )
      ;


            console.log(`Generated ${paths.length} test specs with perf
                                                          measurements in
${outDir}`);
```