Міністерство освіти і науки України

Харківський національний університет імені В.Н. Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту

Рекомендовано до захисту:
протокол засідання кафедри
№ _____ від _____._____.2025 р.
Завідувач кафедри ММтАД
_____ Струков В. М.
(підпис) (прізвище та ініціали)

Пояснювальна записка

до кваліфікаційної роботи бакалавра

на тему «Розробка комбінованого засобу автоматизованого тестування для веб-застосунків»

Захищено на засіданні ЕК № _____
протокол № _____ від _____._____.2025 р.
Оцінка _____ / _____
Голова ЕК
_____ Фесенко Г.В.
(підпис) (прізвище та ініціали)

Виконала:
студентка 4 курсу, групи КС-42
Спеціальності:
122 Комп'ютерні науки
(шифр і назва спеціальності підготовки)
Василенко Софія Романівна
(прізвище, ім'я та по батькові, підпис)
Керівник ст. викладач Мішин О.В.
(ступень, звання, прізвище та ініціали, підпис)
Рецензент _____
(ступень, звання, прізвище та ініціали, підпис)

Харків – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В. Н. Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Кафедра математичного моделювання та аналізу даних
Рівень вищої освіти (освітньо-кваліфікаційний рівень) бакалавр
Галузь знань: 12 Інформаційні технології
Спеціальність: 122 Комп'ютерні науки
Освітня програма: Комп'ютерні науки

ЗАТВЕРДЖУЮ
В.О. завідувача кафедри ММАД
Володимир СТРУКОВ
«____» _____2025 року

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ (ПРОЕКТ)

Василенко Софії Романівни
(прізвище, ім'я, по батькові студента)

1. Тема роботи Розробка комбінованого засобу автоматизованого тестування для веб-застосунків

керівник роботи старший викладач Мішин Олександр Вікторович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від "16" квітня 2025 року № 4101-5/962

2. Строк подання студентом роботи 05 червня 2025 року

3. Перелік питань, які потрібно розробити
1) Аналіз сучасних інструментів і методів автоматизованого тестування веб-додатків.
2) Проектування архітектури універсального засобу тестування, що охоплює модульні, інтеграційні, API та UI-тести
3) Реалізація прототипу засобу тестування мовою програмування Python.
4) Інтеграція засобу з системами управління тестами.
5) Валідація засобу на проєкті з оцінкою його ефективності за відповідними метриками.

4. План роботи

№ з/п	Назви етапів роботи
1	Формулювання мети та повного переліку завдань для виконання кваліфікаційної роботи (КР).
2	Пошук та аналіз теоретичної літератури та сучасних підходів за темою КР.
3	Концептуальне моделювання архітектури і висування вимог до програмного засобу.
4	Програмна імплементація прототипу програмного засобу.
5	Інтеграція та валідація засобу.
6	Підсумок отриманих результатів та оформлення пакету документів для процедури захисту КР в екзаменаційній комісії ННІ КН та ШІ.

5. Дата видачі завдання 21 січня 2025 року

Студент С. Р. Василенко _____
ініціали, прізвище підпис

Керівник роботи О. В. Мішин _____
ініціали, прізвище підпис

АННОТАЦІЯ

Дипломна робота обсягом 76 сторінок пояснювальної записки, що складається з 4 розділів, містить 21 рисунок, 6 таблиць та 25 джерел згідно з переліком літератури. Сучасні методи автоматизованого тестування веб-додатків характеризуються фрагментованістю та вузькою спеціалізацією, що ускладнює їх інтеграцію та використання в проектах з обмеженими ресурсами. Метою роботи є створення, реалізація та впровадження прототипу комбінованого засобу тестування веб-додатків, який реалізує повне покриття піраміди тестування та сприяє ефективності інтеграційних процесів розробки програмного забезпечення.

Для реалізації використано інструменти та фреймворки: Pytest для модульного тестування, HTTPX для тестування API, Testcontainers для інтеграційного тестування, Playwright для UI-тестування, а також Docker Compose для розгортання тестової інфраструктури. Код написано мовою Python. Запропонований засіб дозволяє виконувати модульні, інтеграційні, API та UI-тести в єдиній системі, забезпечує генерацію тестових даних та інтеграцію з системами управління тестами, такими як TestRail і ReportPortal. Він демонструє ефективність у забезпеченні якості веб-додатків і може бути адаптований до різних проектів.

Запропонований інструмент можна застосовувати для автоматизованого тестування веб-додатків у стартапах і невеликих командах, інтегрувати в процеси безперервної інтеграції та доставки, а також розширювати для підтримки нових типів тестів і платформ.

Ключові слова: АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ, ПІРАМІДА ТЕСТУВАННЯ, МОДУЛЬНЕ ТЕСТУВАННЯ, ІНТЕГРАЦІЙНЕ ТЕСТУВАННЯ, API-ТЕСТУВАННЯ, UI-ТЕСТУВАННЯ, СИСТЕМИ УПРАВЛІННЯ ТЕСТАМИ, БЕЗПЕРЕРВНА ІНТЕГРАЦІЯ, БЕЗПЕРЕРВНА ДОСТАВКА.

ABSTRACT

The diploma thesis, consisting of an explanatory note of 76 pages, includes 4 chapters, 21 figures, 6 tables, and 25 sources according to the list of references. Modern methods of automated testing for web applications are characterized by fragmentation and narrow specialization, which complicates their integration and use in projects with limited resources. The aim of the work is the creation, implementation, and deployment of a prototype of a combined testing tool for web applications, which provides full coverage of the testing pyramid and contributes to the efficiency of integration processes in software development.

For implementation, the following tools and frameworks were used: Pytest for unit testing, HTTPX for API testing, Testcontainers for integration testing, Playwright for UI testing, and Docker Compose for deploying the testing infrastructure. The code is written in Python. The proposed tool allows performing unit, integration, API, and UI tests in a single system, provides test data generation, and integration with test management systems such as TestRail and ReportPortal. It demonstrates effectiveness in ensuring the quality of web applications and can be adapted to various projects.

The proposed tool can be applied for automated testing of web applications in startups and small teams, integrated into continuous integration and delivery processes, and also extended to support new types of tests and platforms.

Keywords: AUTOMATED TESTING, TESTING PYRAMID, UNIT TESTING, INTEGRATION TESTING, API TESTING, UI TESTING, TEST MANAGEMENT SYSTEMS, CONTINUOUS INTEGRATION, CONTINUOUS DELIVERY.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

АТ	–	Автоматизоване Тестування
ПЗ	–	Програмне Забезпечення
AI	–	Artificial Intelligence
API	–	Application Programming Interface
CI/CD	–	Continuous Integration / Continuous Delivery
CLI	–	Command-Line Interface
E2E	–	End-to-End
REST	–	Representational State Transfer
TDD	–	Test-Driven Development
TMS	–	Test Management System
UI	–	User Interface
VS Code	–	Visual Studio Code

ВСТУП

Сучасна розробка програмного забезпечення (ПЗ), зокрема веб-додатків, характеризується швидкими темпами розвитку та ускладненням систем. Оперативний випуск стабільних версій є ключовою вимогою для забезпечення конкурентоспроможності, що робить автоматизоване тестування (АТ) невід'ємною частиною процесу розроблення.

Веб-додатки потребують комплексного тестування на всіх рівнях тестової піраміди[1][2]. Проте стартапи та невеликі команди часто мають обмежені ресурси, і внаслідок, стикаються з проблемами: доступні інструменти, для прикладу, такі як Selenium чи Pytest, є складними в інтеграції, вузько спеціалізованими, та поодинці не забезпечують повноцінного покриття всіх ступенів піраміди. Це створює потребу в комбінованому засобі тестування, який би об'єднував усі щаблі тестування, легко вбудовувався в сучасні процеси розроблення ПЗ та був економічно вигідним для впровадження в проєктах подібного масштабу.

Інструменти, наприклад, такі як Selenium, широко застосовуються для UI-тестування, але мають обмеження, пов'язані зі стабільністю та продуктивністю. Pytest є де-факто стандартом для Unit-тестів у Python[3] і, хоча він надає міцну основу для API та Integration-тестів, останні часто потребують зовнішніх бібліотек або специфічних конфігурацій для взаємодії із зовнішніми системами. Playwright пропонує сучасний підхід до кросбраузерного тестування, але його інтеграція з іншими рівнями тестування залишається викликом, хоча він і надає можливість написання інтеграційних або тестів Application Programming Interface-ів (API). Системи управління тестами (TMS), і Continuous Integration/Continuous Deployment платформи (CI/CD), активно використовуються для автоматизації робочих процесів, але брак універсальних рішень ускладнює їхнє швидке розгортання в проектах. Проблема полягає у відсутності комбінованого інструменту, який би

забезпечував комплексне покриття тестами, легку адаптацію до різних проєктів і ефективну інтеграцію з TMS та CI/CD.

Об’єктом дослідження є автоматизоване тестування як засіб забезпечення якості веб-додатків у сучасних процесах розробки програмного забезпечення.

Предметом дослідження є універсальний засіб автоматизованого тестування веб-додатків, що забезпечує виконання модульних, сервісних і візуальних тестів із можливістю інтеграції в системи керування тестуванням і процеси безперервної інтеграції та постачання.

Метою роботи є створення, реалізація та впровадження прототипу комбінованого засобу тестування веб-додатків, який реалізує повне покриття піраміди тестування та сприяє ефективності інтеграційних процесів розробки програмного забезпечення.

Основні завдання включають:

- аналіз сучасних інструментів і методів тестування;
- проектування архітектури універсального засобу;
- реалізація прототипу мовою програмування Python;
- інтеграція з TMS, платформами аналізу і візуалізації результатів;
- валідація на проєкті із відкритим вихідним кодом.

Актуальність теми пов’язана із постійним зростанням потреби IT-компаній, зокрема стартапів, у швидкому випуску якісних релізів. За результатами досліджень, AT може скоротити час на тестування до 40% і зменшити кількість помилок у продакшені до 10%[3].

Новизна полягає у описі створення засобу, що об’єднує всі етапи тестування в єдину систему, інтегрується із TMS, забезпечує спеціалізовані звіти тестування й має можливість масштабування. Пропонований засіб орієнтований на команди з обмеженими ресурсами, надаючи адаптивність і економічність.

Ключова ідея роботи полягає в синтезі теоретичного аналізу та практичної реалізації універсального засобу тестування. Вона включає аналіз сучасних підходів, проєктування, створення прототипу, його інтеграцію та валідацію на реальному проєкті. Результати роботи підтверджують ефективність інструменту та його придатність до впровадження в реальні проєкти.

1 АНАЛІЗ СУЧАСНИХ ПІДХОДІВ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ

1.1 Методологія та піраміда тестування

1.1.1 Принципи та особливості автоматизованого тестування

Автоматизоване тестування стало невід'ємною частиною сучасної розробки програмного забезпечення, використовуючи спеціалізовані інструменти та скрипти для виконання тестових сценаріїв. Цей підхід значно зменшує ручну працю, прискорює перевірку якості та підвищує точність виявлення дефектів, забезпечуючи стабільні й повторювані результати. Автоматизація рутинних перевірок, наприклад, таких як регресійні тести, дозволяє командам заощаджувати час і зосереджуватися на складніших аспектах, наприклад, тестування нових функцій чи аналіз поведінки системи в нестандартних умовах.

Повторюваність і консистентність є ключовими перевагами АТ. Тести щоразу виконуються за однакових умов, усуваючи вплив людських факторів, таких як втома чи неуважність. Це спрощує відстеження змін у якості програмного забезпечення та дозволяє швидко виявляти нові дефекти після оновлень коду. Така стабільність дає змогу командам порівнювати результати різних запусків тестів і точно визначати проблемні ділянки.

АТ ідеально інтегрується з сучасними методологіями розробки, такими як безперервна інтеграція та доставка (CI/CD). Тести можуть запускатися автоматично після кожного оновлення коду (комміту), надаючи розробникам миттєвий зворотний зв'язок. Це сприяє оперативному виправленню помилок, що підвищує загальну якість продукту та прискорює процеси розробки, особливо в умовах швидких ітерацій.

Одним із важливих принципів є раннє виявлення дефектів, відоме як підхід "shift-left". Тестування на початкових етапах розробки дозволяє знаходити проблеми до їх усугублення ситуації. Це значно знижує витрати на виправлення, адже дефекти, виявлені на пізніх етапах, зазвичай значно

дорожчі в усуненні. Такий підхід особливо цінний у гнучких методологіях, як Agile та DevOps, де швидкість і якість є пріоритетами.

АТ також підтримує масштабування. Воно дозволяє ефективно перевіряти великі кодові бази та складні архітектури, що критично для сучасних проєктів із великою кількістю компонентів. Завдяки цьому команди можуть охоплювати тисячі тестових сценаріїв без значного збільшення ресурсів, забезпечуючи всебічне тестування системи.

Серед переваг АТ – значна економія часу. У швидких циклах випуску, де ручне тестування може займати години чи дні, автоматизація скорочує цей час до хвилин, прискорюючи вихід продукту на ринок. Зменшення людського втручання мінімізує помилки, викликані неуважністю, а детальне логування результатів забезпечує точні дані для аналізу збоїв і прийняття рішень.

Проте АТ має й недоліки. Початкові витрати на створення скриптів, налаштування інструментів та інтеграцію в робочі процеси можуть бути суттєвими, що створює труднощі для невеликих команд чи проєктів із обмеженим бюджетом. Зміни в програмному забезпеченні вимагають регулярного оновлення тестів, що може бути трудомістким. Без належного супроводу тести стають неактуальними, що призводить до хибних результатів або пропусків дефектів.

Крім того, автоматизоване тестування менш ефективне для перевірки аспектів, які потребують людської оцінки, наприклад, зручності інтерфейсу чи візуальної привабливості. У таких випадках ручне тестування залишається незамінним, що змушує команди комбінувати обидва підходи для досягнення найкращих результатів.

Роль АТ в сучасній розробці важко переоцінити. Воно прискорює випуск продуктів завдяки швидкому й надійному тестуванню, забезпечує підтримку складних систем за допомогою масштабованості та підвищує конкурентоспроможність компаній, дозволяючи швидко адаптуватися до ринкових змін. Зменшення ручної праці та раннє виявлення дефектів знижують витрати, даючи змогу спрямовувати ресурси на вдосконалення

продукту. Автоматизоване тестування – це не просто інструмент, а стратегічний підхід, який підвищує продуктивність команд, скорочує час виходу на ринок і гарантує високу якість ПЗ, що відповідає сучасним стандартам.

1.1.2 Піраміда тестування: структура та рівні

Одним із ключових підходів до організації АТ є піраміда тестування – концепція, формалізована Майком Коном у 2009 році в книзі Succeeding with Agile: Software Development Using Scrum[5]. Ця модель була розроблена для управління тестуванням в Agile-середовищах, де ітеративна розробка та часті релізи вимагають швидкого зворотного зв’язку та ефективного використання ресурсів. Піраміда тестування структурує тести за рівнями складності та обсягу, пропонуючи оптимальний баланс між швидкістю виконання, покриттям коду та витратами на тестування. Модель візуально представлена на рисунку 1[6].

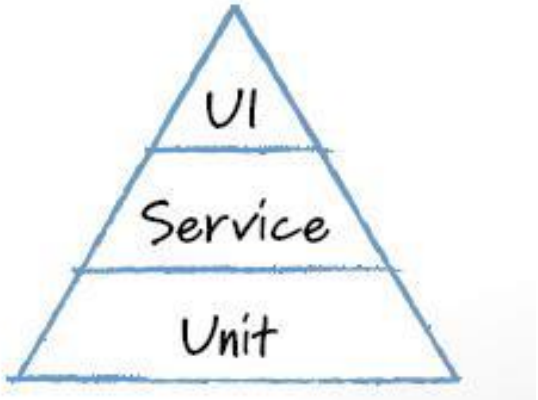

Рисунок 1.1 – Піраміда АТ

Піраміда тестування складається з трьох основних рівнів: юніт-тестів (Unit), сервісні (Service) тестів та User Interface (UI) тестів, із розподілом[7], де юніт-тестів 70-80%, сервісних – 15-20%, а UI-тестів – 5-10%, проте це не є чітким правилом і залежить від контексту. Зважене використання розподілу

забезпечує швидкість, ефективність та надійність тестового процесу, що є ключовим у розробленні ПЗ.

Юніт-тести, розташовані в основі піраміди, перевіряють окремі компоненти коду, такі як функції чи методи, в ізольованому середовищі. Вони є найчисленнішими, найшвидшими у виконанні та найдешевшими, що дозволяє запускати їх при кожному коміті коду[8]. Юніт-тести допомагають виявляти помилки на ранніх етапах розробки, коли їх виправлення є найменш витратним. Наприклад, інструменти, такі як JUnit для Java або Pytest для Python, широко використовуються для створення юніт-тестів, забезпечуючи швидке тестування окремих модулів. Дослідження показують, що юніт-тести можуть покривати до 70-80% коду, значно знижуючи ризик дефектів у подальших етапах [7].

Вище в піраміді розташовані сервісні (Service) тести, або іноді їх ще називають інтеграційними (Intregration). Вони відіграють ключову роль у перевірці коректної взаємодії між компонентами системи, такими як модулі, бази даних, API чи зовнішні сервіси. Їхня основна мета – виявити дефекти, які неможливо ідентифікувати на рівні юніт-тестів, наприклад, некоректну передачу даних між модулями чи збої в обробці запитів через затримки в мережі. На практиці це може бути тестування в Python, де, наприклад, за допомогою бібліотеки requests перевіряється, чи система правильно відправляє запит до API, отримує відповідь із кодом 200 і розбирає JSON із потрібними полями, такими як ідентифікатор чи статус. Для зручності й точності часто застосовують інструменти, як Wiremock для створення моків або Pact для верифікації контрактів у мікросервісах, що дозволяє уникнути залежності від нестабільних зовнішніх систем. В класичній піраміді, кількість інтеграційних тестів менша за юніт-тести, вони також є складнішими в налаштуванні, проте їхня цінність полягає в забезпеченні впевненості, що окремі частини програми працюють як єдиний механізм[8]. Вони стають незамінними в складних архітектурах, де навіть незначна неузгодженість між компонентами може призвести до критичних збоїв, гарантуючи стабільність і

надійність продукту в умовах, максимально наближених до реального використання[9].

На вершині піраміди тестування розташовані UI (User Interface) або E2E (End-to-End) тести, які відтворюють реальні сценарії взаємодії користувачів із ПЗ, перевіряючи функціональність усієї системи від початку до кінця. Такі тести є найскладнішими та найповільнішими через їхню здатність охоплювати весь технологічний стек – від графічного інтерфейсу до серверної логіки та баз даних. Наприклад, E2E-тест може симулювати процес входу в систему, додавання товару до кошика та завершення покупки, перевіряючи коректність роботи всіх інтегрованих компонентів. Їхня висока ресурсомісткість обумовлена необхідністю взаємодії з реальними або емульованими браузерами, а також залежністю від зовнішніх факторів, таких як стабільність мережі чи доступність API. Через це UI-тести зазвичай запускаються рідше[8], переважно перед релізами, щоб переконатися, що система функціонує як єдине ціле. Для автоматизації таких тестів використовують інструменти, як-от Selenium або Playwright, однак їхнє виконання може тривати від кількох хвилин до годин, що обмежує їхнє застосування в швидких CI/CD-пайплайнах. Також, вони мають крихкість і схильність до помилкових спрацьовувань, спричинену змінами в інтерфейсі чи затримками в мережі, що ускладнює підтримку тестів[2]. Нестабільність можна частково зменшити за допомогою ретельно спланованої архітектури тестів, наприклад, із використанням патерна Page Object Model, , хоча це не є вичерпним вирішенням проблеми.

1.1.3 Значення піраміди в Agile/DevOps та альтернативні моделі

Піраміда тестування тісно пов’язана з сучасним циклом розробки ПЗ, особливо в контексті Agile-методологій, таких як Scrum або Kanban. Agile-розробка передбачає ітеративний підхід, де кожна ітерація (спринт) має доставляти робоче ПЗ[5]. Піраміда тестування підтримує цей процес, забезпечуючи швидкий зворотний зв’язок на різних етапах розробки. Юніт-тести дозволяють розробникам перевіряти код одразу після його написання,

що відповідає практиці тест-направленої розробки (TDD), коли тести створюються до написання коду. Інтеграційні тести забезпечують коректну взаємодію компонентів під час інтеграції нових функцій у спринтах, а E2E-тести підтверджують, що система відповідає вимогам користувачів перед релізом. Такий підхід дозволяє командам швидко виявляти та виправляти помилки, що є ключовим для Agile-принципів швидкості та адаптивності.

У сучасних CI/CD-пайплайнах піраміда тестування відіграє центральну роль, автоматизуючи тестування на різних етапах розробки та забезпечуючи високу якість ПЗ у відповідності до принципів DevOps. DevOps акцентує увагу на співпраці між командами розробки та експлуатації, сприяючи автоматизації процесів, безперервній інтеграції, доставці та моніторингу[10]. Піраміда тестування ідеально вписується в цю парадигму, дозволяючи командам спільно працювати над створенням надійного продукту. Наприклад, у CI/CD-пайплайні, побудованому на платформі, такій як GitHub Actions, юніт-тести запускаються автоматично після кожного коміту, щоб переконатися, що нові зміни не порушують існуючу функціональність – це відображає DevOps-підхід до безперервної інтеграції. Інтеграційні тести виконуються після успішної збірки, перевіряючи коректну взаємодію компонентів системи, а E2E-тести проводяться перед розгортанням у стейджинг або продакшн, гарантуючи готовність продукту до релізу. Автоматизація на всіх рівнях піраміди тестування усуває вузькі місця в процесі розробки, прискорюючи зворотний зв’язок і підвищуючи ефективність. Така інтеграція забезпечує безперервне тестування, що є ключовим для принципів безперервної доставки, де ПЗ має бути готовим до розгортання в будь-який момент. Таким чином, піраміда тестування не лише підтримує DevOps-цілі, а й сприяє швидкій та надійній доставці високоякісного продукту, зміцнюючи зв’язок між розробкою та експлуатацією.

Із розвитком технологій та зміною архітектур ПЗ, з’явилися нові підходи до адаптації піраміди тестування. Так, мікросервісні архітектури підняли значущість API-тестів, які перевіряють взаємодію між незалежними

сервісами. Контрактне тестування[2], реалізоване через інструменти на зразок Pact, дозволяє верифікувати відповідність API їхнім специфікаціям, зменшуючи залежність від зовнішніх систем. Наприклад, Pact створює контракти між споживачем і постачальником API, що дає змогу тестувати сумісність без повного розгортання системи. Віртуалізація сервісів за допомогою WireMock також полегшує тестування, дозволяючи симулювати поведінку API в ізольованому середовищі. Це забезпечує швидкий зворотний зв’язок і відповідає принципам піраміди щодо раннього виявлення помилок.

Сучасні фронтенд-фреймворки, такі як React і Angular, розмивають межу між юніт- і UI-тестами. Компоненти інтерфейсу можна тестувати як ізольовані одиниці за допомогою юніт-тестів, що підвищує їхню швидкість і зменшує ресурсомісткість. Наприклад, Jest із функцією снепшот-тестування дозволяє порівнювати рендеринг компонентів із попередніми версіями, виявляючи неочікувані зміни. Такий підхід робить тестування інтерфейсу ефективнішим.

Альтернативою піраміді є модель тестового трофею[11], яка акцентує на інтеграційних тестах. Запропонована Кентом К. Доддсом, ця модель включає статичний аналіз, обмежену кількість юніт-тестів, основний фокус на інтеграційних тестах і мінімальну кількість E2E-тестів. Інтеграційні тести вважаються більш цінними, оскільки вони охоплюють ширший спектр поведінки системи, уникаючи крихкості юніт-тестів, прив’язаних до деталей реалізації. Тестовий трофей зображений на рисунку 1.2[11], де видно зміщення акценту на інтеграційні тести.

Рисунок 1.2 – тестовий трофей

Отже, як підсумок, піраміда тестування залишається фундаментальною моделлю для структурування автоматизованих тестів, адаптуючись до сучасних парадигм розробки ПЗ. У контексті розподілених архітектур, таких як мікросервіси, зростає значення тестів, що перевіряють взаємодію компонентів, забезпечуючи узгодженість системи. Сучасні підходи до розробки інтерфейсів користувача сприяють інтеграції швидких тестів компонентів, що поєднують характеристики юніт- і UI-тестів, оптимізуючи баланс між швидкістю та покриттям. Альтернативні моделі, які акцентують на інтеграційних тестах, пропонують переосмислення пріоритетів для ширшої верифікації поведінки системи. У рамках безперервної інтеграції та доставки піраміда підтримує автоматизацію тестування, сприяючи швидкій і надійній розробці. Водночас складність управління даними та нестабільність комплексних тестів потребують стратегічного підходу до планування, щоб забезпечити ефективність і якість тестового процесу.

1.2 Огляд сучасних інструментів АТ

1.2.1 Інструменти автоматизованого тестування

Автоматизоване тестування вимагає різноманітних інструментів для покриття всіх аспектів розробки ПЗ. Інструменти поділяються на кілька категорій залежно від рівня тестування, який вони підтримують, або функцій, які вони виконують. Кожна категорія відповідає певному аспекту піраміди тестування, описаної в розділі 1.1, або специфічним потребам процесу тестування. Ці категорії включають інструменти для модульного тестування, тестування API, інтеграційного тестування, UI-тестування, генерації тестових даних, інтеграції з CI/CD, управління тестами (TMS) і звітності.

1.2.1.1 Інструменти для Unit-тестування

Unit-тестування є основою забезпечення якості, оскільки воно перевіряє окремі компоненти коду, такі як функції чи методи класів, ізольовано від зовнішніх залежностей. Популярними інструментами для цього рівня є Pytest і Unittest для Python[3], JUnit для Java та NUnit для .NET. Pytest вирізняється своєю гнучкістю, підтримкою плагінів і простотою синтаксису, що дозволяє створювати параметризовані тести та використовувати фікстури для підготовки тестового середовища. Unittest, хоча й менш гнучкий, пропонує стандартний підхід, який добре підходить для простих проєктів. JUnit і NUnit є аналогами для інших мов програмування, забезпечуючи подібні функції, але адаптовані до екосистем Java і .NET відповідно. Розширене порівняння подане у таблиці 1.1.

Таблиця 1.1 – Порівняння інструментів для Unit-тестування

Критерій	Pytest	Unittest	JUnit
Мова програмування	Python	Python	Java
Гнучкість плагінів	Висока	Низька	Середня
Простота налаштування	Складна	Проста	Середня

Закінчення таблиці 1.1

Підтримка асинхронності	Так	Ні	Ні
Масштабованість	Висока	Середня	Висока
Документація і спільнота	Висока	Низька	Висока

1.2.1.2 Інструменти для API-тестування

Тестування API є критично важливим для перевірки взаємодії між компонентами системи, особливо в мікросервісних архітектурах. Інструменти, такі як HTTPX, Requests, Postman і SoapUI, дозволяють створювати, виконувати та аналізувати тести для REST, GraphQL і SOAP API. HTTPX, сучасна бібліотека для Python, підтримує асинхронні запити, що робить її ідеальною для тестування високопродуктивних систем[12]. Requests, хоча й не асинхронна, залишається популярною завдяки своїй простоті та широкій підтримці. Postman пропонує графічний інтерфейс, який спрощує створення тестів і автоматизацію запитів[13], тоді як SoapUI спеціалізується на тестуванні SOAP API, якщо існує така необхідність. Ці інструменти забезпечують гнучкість і підтримують інтеграцію з CI/CD, що дозволяє автоматизувати перевірку API в процесі розробки. Наочне порівняння продемонстроване в таблиці 1.2.

Таблиця 1.2 – Порівняння інструментів для API-тестування

Функціональність	HTTPX	Requests	Postman
Підтримка асинхронних запитів	Так	Ні	Так
Легкість використання	Складна	Проста	Проста
Інтеграція з CI/CD	Так	Так	Так
Підтримка форматів API	REST, GraphQL	REST, GraphQL	REST, SOAP, GraphQL

Закінчення таблиці 1.2

Генерація звітів	Ні	Ні	Так

1.2.1.3 Інструменти для Integration-тестування

Інтеграційні тести перевіряють взаємодію між модулями, включаючи сервіси, бази даних тощо. Часто для них потрібні зовнішні сервіси – бази даних, брокери повідомлень, кеші. Інструмент Testcontainers (доступний для Python, Java та .NET) дозволяє автоматично запускати ці залежності у Docker-контейнерах під час тестів[14]. Наприклад, із Testcontainers можна підняти контейнер PostgreSQL чи Redis, виконати інтеграційні тести і потім знищити середовище. Це забезпечує високу ізоляцію та повторюваність тестів. До переваг належить можливість використовувати реальні сервіси без ручного налаштування; недолік – потреба в інсталяції Docker і додатковий оверхед на запуск контейнера. Testcontainers надає програмний API, який легко викликати з тестів (CLI не потрібен), а результати тестів інтегруються з CI.

1.2.1.4 Інструменти для UI-тестування

Інструменти UI-тестування (End-to-End) автоматизують взаємодію з інтерфейсом програми (найчастіше – вебінтерфейсом). Найвідоміший з них:

– Selenium WebDriver – бібліотека для управління браузерами (Chrome,

Firefox, Edge тощо) з кодом на різних мовах. Selenium WebDriver «є багатомовною бібліотекою автоматизації браузерів» (підтримує офіційно Java, JavaScript, Python, .Net, Ruby)[15]. Він забезпечує крос-браузерне тестування за відкритим стандартом W3C WebDriver. Серед недоліків Selenium – необхідність явного керування очікуваннями та доволі громіздкий API;

– Playwright від Microsoft підтримує мультипроцесний запускач із

нативною підтримкою паралельних тестів і автоматичними очікуваннями (доки елементи не завантажаться). Playwright також мультиплатформовий (Chrome, Firefox, WebKit) та мультиплатформно-клієнтний (JavaScript, Python, Java, .NET)[15];

– Cypress (JavaScript/TypeScript) – фреймворк для E2E-тестів, який працює безпосередньо в браузері і «надає автоматичне очікування та вбудовані високорівневі фічі». Однак Cypress обмежений тим, що може виконуватись лише в JavaScript-середовищі і обмежено підтримує кросбраузерність (переважно Chromium).

Підсумовуючи, Selenium є стандартом для UI-тестування завдяки широкій підтримці браузерів, але його продуктивність і стабільність можуть бути нижчими через складність налаштування. Playwright пропонує високу швидкість і стабільність, підтримуючи сучасні браузери. Cypress вирізняється простотою використання і автоматичним очікуванням, що робить його ідеальним для JavaScript-проєктів, але обмежена підтримка браузерів може бути недоліком для кросбраузерного тестування. Згідно з TestGuild, Playwright набирає популярності завдяки продуктивності, але Cypress залишається вибором для фронтенд-команд. Деталізоване порівняння інструментів наведено у таблиці 1.3.

Таблиця 1.3 – Порівняння інструментів для UI-тестування

Характеристика	Selenium WebDriver	Playwright	Cypress
Підтримка браузерів	Chrome, Firefox, Safari, Edge, IE	Chromium, Firefox, WebKit	Chrome, Firefox, Edge
Мови програмування	Java, C#, Python, Ruby, JavaScript, Kotlin	JavaScript/TypeScript, Python, Java, C#	JavaScript/TypeScript

Закінчення таблиці 1.3

Архітектура	WebDriver protocol	Пряме управління браузером через WebSocket, мульти-таб	Виконується всередині браузера, один таб за замовчуванням
Швидкість виконання	Середня	Висока	Висока
Автоматичні очікування	Немає	Вбудовані	Вбудовані
Паралелізація	Немає	Вбудована	Платна функція
Звітність та дебаг	Немає	Трейсинг, відео, скріншоти	Time Travel, відео, скріншоти

1.2.1.5 Інструменти для генерації тестових даних

Генерація тестових даних є важливою для створення реалістичних сценаріїв тестування. Інструменти, такі як Mimesis [16] і Faker [17] для Python, дозволяють генерувати випадкові дані, такі як імена, адреси чи номери, які відповідають певним критеріям. Mimesis особливо корисний для локалізації даних, наприклад, для створення даних українською мовою, що відповідає вимогам до реалістичності, але може мати обмеження у підтримці специфічних форматів. Faker пропонує подібні функції, але з ширшою підтримкою різних мов і форматів, хоча іноді потребує додаткових налаштувань для кастомізації. Ці інструменти спрощують підготовку тестів, зменшуючи час на створення даних вручну, але їх ефективність залежить від точності вимог до даних.

1.2.1.6 Системи CI/CD

Інтеграція з CI/CD є ключовою для автоматизації тестування в процесі розробки. Наприклад, GitHub Actions – це CI/CD-платформа, інтегрована з GitHub, яка дозволяє автоматизувати збірку, тестування та розгортання проєкту. У GitHub Actions робота описується у YAML-файлах; вона може запускатись по подіям (push, PR) і підтримує виконання у віртуальних машинах або Docker-контейнерах[17]. Інші популярні рішення: Jenkins (open-source рішення, сервер CI з великою кількістю плагінів та підтримкою будь-якого скрипту чи контейнера), GitLab CI/CD (вбудована в GitLab YAML-конфігурація з власними ранерами), Travis CI, CircleCI та інші[18]. Усі ці сервіси надають CLI або API для взаємодії. Переваги – повна автоматизація тестів при кожному коміті, можливість паралельного виконання (матриці запуску), інтеграція з Docker та зовнішніми сервісами. Недоліки – налаштування інфраструктури (особливо у Jenkins/GitLab на власних серверах) та лімітовані безкоштовні плани. Наприклад, GitHub Actions безкоштовно дає певний ліміт хвилин запуску, але пропонує Docker-образи та середовища для Linux, Windows і Mac. Багато тестових інструментів мають готові плагіни: наприклад, для Selenium, Pytest, Playwright існують офіційні модулі у маркетплейсі GitHub.

1.2.1.6 Системи управління тестуванням

Системи управління тестами (TMS) використовуються для організації тест-кейсів, планування та відстеження виконання тестів. Одним із провідних інструментів є TestRail, комерційний продукт від компанії Idera, який представляє собою веб-платформу для управління тестовими артефактами. Ця система широко застосовується в індустрії завдяки своїм розширеним можливостям, які включають інтеграцію з баг-трекерами, такими як Jira, що забезпечує безперервний обмін інформацією між командами розробки та тестування. TestRail також оснащений REST API, який дозволяє автоматизувати робочі процеси та інтегрувати платформу з системами CI/CD,

що є важливим для сучасних Agile-команд. Окрім цього, інструмент пропонує детальну звітність, яка допомагає аналізувати результати тестування, відстежувати метрики покриття та виявляти проблемні ділянки в програмному продукті. Завдяки своїй здатності підвищувати прозорість процесу тестування, TestRail дозволяє командам оперативно реагувати на дефекти, проте висока вартість ліцензії може бути суттєвим недоліком для невеликих організацій.

Ще одним ключовим інструментом, який часто використовується разом із TestRail, є Jira, розроблена компанією Atlassian, що є однією з найпопулярніших систем управління проектами та відстеження помилок. Jira забезпечує команди можливістю планувати спринти, відстежувати завдання та дефекти, а також налаштовувати робочі процеси відповідно до потреб проекту. Інтеграція з TestRail дозволяє пов’язувати тестові кейси з вимогами та помилками, що забезпечує повну простежуваність у процесі розробки та тестування[19]. Цей інструмент підтримує різні методології, такі як Agile та Scrum, що робить його універсальним рішенням для команд із різними підходами до управління. Гнучкість Jira, можливість кастомізації та велика кількість доступних плагінів роблять її незамінною для розробників і тестувальників, які прагнуть адаптувати систему під свої унікальні вимоги. Популярність Jira також підтримується активною спільнотою користувачів, яка постійно вдосконалює її функціональність через розширення та інтеграції.

Також існують системи управління тестами, такі як Qase, TestLink та Zephyr, але мають меншу популярність порівняно з TestRail та Jira. Qase є хмарною SaaS-платформою з безкоштовним базовим тарифом і можливістю імпорту даних із TestRail, що робить її привабливою для команд із обмеженим бюджетом. TestLink, у свою чергу, є open-source рішенням, яке, попри свою безкоштовність, поступається комерційним аналогам за функціональністю. Zephyr, інтегрований із Jira, пропонує зручне управління тестами в рамках цього інструменту, що може бути перевагою для команд, які вже використовують Jira. Усі ці системи мають свої особливості, але TestRail і Jira залишаються лідерами завдяки своїм широким можливостям і гнучкості.

1.2.1.6 Інструменти звітності

Звітність є важливим аспектом аналізу результатів тестування. Інструменти, такі як ReportPortal і Allure надають інтерактивні звіти, які візуалізують результати, аналізують тренди і моніторять метрики, такі як покриття коду і стабільність тестів[20]. ReportPortal особливо корисний для великих команд, оскільки пропонує розширені функції моніторингу і аналізу, включаючи кастомні метрики, такі як Test Stability Index, але може бути складним у налаштуванні. Allure є популярним завдяки своїй простоті і підтримці різних фреймворків, але іноді має обмеження у візуалізації складних даних. Ці інструменти допомагають командам швидко оцінювати якість продукту і приймати обґрунтовані рішення, але їх ефективність залежить від рівня інтеграції з іншими системами.

1.2.2 Огляд сучасних тенденцій в автоматизованого тестування

Інтеграція штучного інтелекту (AI) є однією з найвизначніших тенденцій. AI-агенти, відомі як Agentic AI, здатні автономно керувати процесами тестування, включаючи пріоритизацію тестів, виконання їх у різних середовищах і аналіз результатів. Наприклад, такі агенти можуть оптимізувати регресійне тестування для платформ електронної комерції, пропонуючи виправлення дефектів. Згідно з даними TestGuild[21], 72.3% команд досліджують або впроваджують AI-driven тестування, а 38% вважають AI вирішенням проблеми нестачі кваліфікованих тестувальників. Інструменти, такі як ChatGPT, Claude і Gemini, використовуються для генерації тест-кейсів, прогнозування дефектів і аналізу логів, що значно підвищує ефективність.

1.3 Обмеження інтеграції та універсальності тестових систем

Попри широкий вибір інструментів для АТ веб-застосунків, їхня інтеграція в єдину цілісну систему залишається значною проблемою. Різні фреймворки орієнтовані на окремі рівні тестування і працюють незалежно

один від одного, не забезпечуючи узгодженості у форматах, конфігураціях і способах виконання. У типовому випадку кожен тип тестів реалізується за допомогою окремого інструмента, що має власні правила налаштування та запуску. Це породжує фрагментацію тестової інфраструктури, коли кожен набір тестів існує ізольовано, без можливості централізованого керування або наскрізного виконання.

Відсутність єдиної модульної архітектури унеможливлює логічне об'єднання різнорівневих тестів у рамках спільного пайплайну. Тестові сценарії різних рівнів зберігаються у різних структурах, використовують різні конфігураційні формати та не мають спільного середовища виконання. Через це виникає необхідність створення складних сценаріїв оркестрації – зокрема, скриптів запуску, конфігурацій CI/CD або умовного керування залежностями – що ускладнює автоматизацію та збільшує поріг входу.

Крім того, відсутність уніфікованих підходів призводить до дублювання коду. Типові приклади – це повторне описування логіки автентифікації, ініціалізації тестових даних або обробки сесій у кожному типі тестів. Оскільки інструменти працюють незалежно, спільні частини функціоналу доводиться реалізовувати окремо, що збільшує обсяг підтримки та підвищує ризик неузгодженостей. Навіть базові параметри середовища, як-от адреса тестового сервера або токен авторизації, часто прописуються окремо для кожного фреймворку, що ускладнює масштабування і веде до конфігураційних помилок.

Ще одним викликом є необхідність застосування додаткових адаптерів і плагінів для інтеграції фреймворків між собою або з іншими системами. Наприклад, щоб запускати UI-тести у рамках середовища, яке використовується для API-тестів, потрібні сторонні бібліотеки або власноруч написаний код зв'язування. Таке рішення є технічно складним і потребує постійного оновлення, оскільки залежить від зовнішніх змін у фреймворках. Крім того, різні формати звітності, які генерують інструменти, не сумісні між собою: один фреймворк створює XML-звіт, інший – HTML, ще один – JSON

або власний консольний вивід. Це ускладнює агрегацію результатів, особливо якщо потрібно інтегрувати звіти з системами управління тестуванням, управління проєктами або CI/CD.

Усе це призводить до того, що тестування як процес втрачає цілісність. Відсутність наскрізної логіки унеможливлює побудову єдиного ланцюга перевірок, у якому різні рівні тестування логічно поєднані. У великих командах така роз’єднаність призводить до втрати узгодженості між підрозділами, а в малих командах або стартапах – до перевитрати ресурсів і зниження ефективності. Саме тому проблема фрагментації та несумісності інструментів є ключовою при створенні сучасної автоматизованої системи тестування, яка має бути не лише ефективною окремо на кожному рівні, а й узгодженою у цілому.

2 ПРОЄКТУВАННЯ ЗАСОБУ ТЕСТУВАННЯ

2.1 Постановка задачі та формування вимог до засобу

2.1.1 Задача засобу автоматизованого тестування

Аналіз, проведений у розділі 1.3, виявив низку проблем, пов’язаних із сучасними засобами АТ. Зокрема, фрагментація інструментів призводить до необхідності використання різних систем для різних рівнів тестування, що ускладнює процеси управління та підвищує ймовірність помилок. Складність інтеграції із TMS, створює додаткові перешкоди для команд, які прагнуть автоматизувати свої робочі процеси. Крім того, дублювання коду в тестових наборах знижує ефективність розробки та підтримки, особливо в умовах обмежених ресурсів.

З огляду на це, сформульовано завдання – створення універсального засобу АТ веб-додатків з урахуванням сучасних вимог CI/CD. Технічно це означає розробку системи, яка дозволяє автоматизовано виконувати та управляти Unit-, Integration-, API- та UI-тестами, можливістю інтегруватися з конвеєром безперервної інтеграції та доставки, забезпечувати гнучке генерування тестових даних і звітування про результати. Інструмент повинен коректно взаємодіяти з існуючими платформами (репозиторіями, сервісами звітності, моніторингу тощо) та бути масштабованим.

Згідно з класифікацією вимог у розробці ПЗ, усі вимоги поділяються на функціональні (що система має робити) та нефункціональні (характеристики якості системи)[22]. Функціональні вимоги описують конкретний функціонал засобу, а нефункціональні – його поведінку та обмеження (продуктивність, надійність, безпека тощо).

2.1.2 Функціональні вимоги

Засіб АТ, що проєктується повинен відповідати таким функціональним вимогам:

1) підтримка створення та виконання автоматизованих тестів для веб

додатків:

- підтримка UI-тестів (наприклад, Playwright);
- підтримка API-тестів;
- підтримка інтеграційних тестів;
- підтримка модульних тестів (наприклад, pytest);
- критерії приймання: тести виконуються через єдиний інтерфейс; пріоритет: високий;

2) забезпечення виконання тестів у єдиній схемі в різних середовищах:
   - локальне середовище;
   - можливість налаштування запуску через CI/CD сервер (наприклад, GitHub Actions);
   - контейнер Docker;
   - критерії приймання: тести виконуються без модифікації коду, налаштування через конфігураційні файли (YAML); пріоритет: високий;
3) надання механізмів параметризації та генерації даних:
   - інтеграція з бібліотеками (наприклад, Faker);
   - генерація динамічних тестових даних;
   - критерії приймання: підтримка щонайменше 5 типів даних (імена, email, номери телефонів, дати, числа); пріоритет: середній;
4) забезпечення збирання, зберігання та аналізу результатів тестування:
   - централізоване сховище результатів;
   - інтеграція з аналітичними сервісами (наприклад, ReportPortal);
   - інтеграція з TMS (наприклад, TestRail);
   - критерії приймання: звіти у форматі JSON/XML, статистика (пройдені/провалені тести, час виконання); пріоритет: високий.

2.1.3 Нефункціональні вимоги

Засіб АТ повинен відповідати таким нефункціональним вимогам:

1) забезпечення надійності шляхом коректної обробки помилок із виведенням інформації про причини збоїв;
2) забезпечення розширюваності шляхом додавання нових функцій або інтеграцій із мінімальними змінами;
3) забезпечення зручності розробки шляхом використання стандартних інструментів Pytest і Playwright;
4) забезпечення безпеки шляхом захисту даних під час доступу до тестових середовищ;
5) забезпечення кросплатформності шляхом підтримки Linux, Windows і контейнерів Docker.

2.2 Архітектура інструменту автоматизованого тестування

Засіб АТ буде побудований на основі модульної N-tier архітектури з чітким розділенням відповідальностей та застосуванням принципів Clean Architecture. Система спроектована як розширюваний Python-фреймворк, що підтримує різні типи тестування та забезпечує легку інтеграцію з зовнішніми сервісами.

Архітектура базується на принципах Domain-Driven Design (DDD) та Dependency Injection, що забезпечує слабку зв'язаність компонентів та високу тестованість системи. Основна ідея полягає в тому, щоб ізолювати бізнес-логіку тестування від конкретних реалізацій фреймворків та зовнішніх залежностей через систему адаптерів та інтерфейсів.

Архітектура складається з чотирьох основних рівнів:

– рівень представлення – забезпечує інтерфейси для взаємодії з користувачем;
– рівень застосунку – містить бізнес-логіку координації та оркестрації тестів;

– доменний рівень – визначає основні абстракції та бізнес-правила предметної області;

– інфраструктурний рівень – реалізує конкретні технічні рішення та зовнішні інтеграції;

Описана архітектура забезпечує технічну гнучкість через чіткий розподіл відповідальностей між рівнями, дозволяє незалежно змінювати технологічний стек інфраструктурного рівня без значного впливу на доменну логіку.

2.3 Вибір і обґрунтування інструментів

Python обрано як мову програмування через зрілу екосистему інструментів для АТ, інфраструктурної інтеграції та розробки веб-сервісів. Завдяки підтримці об’єктно-орієнтованого підходу, можливості структуровано організовувати код і активному розвитку open-source бібліотек (наприклад, pytest, HTTPX, Playwright, docker-py, тощо), Python дозволяє реалізувати всі рівні тестування в єдиному стеку. Його широка інтегрованість з інструментами DevOps робить автоматизацію не лише технічно повною, а й операційно ефективною. Динамічна типізація та гнучкість синтаксису спрощують початкову розробку та адаптацію сценаріїв під змінні вимоги, хоча й потребують дисципліни при масштабуванні проєкту. Завдяки широкому застосуванню мови у сферах від web до data science, засіб, побудований на Python, матиме високу сумісність і потенціал повторного використання в проєктах різного профілю.

Pytest обрано як базовий фреймворк через модульну архітектуру, зручний синтаксис і підтримку масштабованих фікстур. Він автоматично виявляє тести, формує детальні звіти про збої та легко розширюється плагінами (наприклад, pytest-asyncio для асинхронних сценаріїв).

Integration-рівень тестування реалізовано через сценарії, що імітують міжкомпонентну взаємодію: наприклад, запити до API, що обробляють дані з бази, або обробка повідомлень у чергах. Для цього застосовується docker-

compose середовище, де одночасно розгортаються всі залежні компоненти. В тестах використовується HTTPX, Postgres test-база та можливо – mock-сервіси для імітації зовнішніх API. Такий підхід забезпечує верифікацію контрактів між сервісами до виходу в staging/production.

Для API-тестування використовується HTTPX – асинхронний HTTP-клієнт із підтримкою HTTP/2, таймаутів, пулу з'єднань і гнучкої конфігурації. На відміну від requests, він дозволяє виконувати запити у неблокуючому режимі, що важливо для навантажувального або паралельного тестування. HTTPX сумісний із синхронним і асинхронним кодом, що спрощує інтеграцію в проєкти зі змішаною архітектурою. Недоліком є потреба в знаннях асинхронності, але це виправдано гнучкістю й продуктивністю.

Для UI-тестування обрано Playwright завдяки його гнучкості та можливості забезпечувати стабільне кросбраузерне тестування з мінімальними накладними витратами. Playwright дозволяє легко масштабувати тестову автоматизацію до складних, динамічних веб-застосунків, завдяки підтримці трьох основних рушіїв (Chromium, Firefox, WebKit) та універсальному API. Це забезпечує високий рівень повторного використання тестів і зменшує витрати на підтримку при розширенні функціоналу. Технологія WebSocket-з'єднання, яка використовується Playwright, скорочує час виконання тестів порівняно з традиційними HTTP-запитами, підвищуючи ефективність CI/CD-процесів. Автоматичне очікування готовності елементів знижує ймовірність нестабільності тестів, що особливо важливо для інтерфейсів із різною швидкістю завантаження сторінок чи асинхронною взаємодією.

Вибір Playwright мотивований також його активним розвитком і розширенням функціоналу, що гарантує актуальність інструмента та доступність нових можливостей для автоматизації UI-тестування різного рівня складності. Це робить Playwright оптимальним рішенням для нашого засобу як на поточному етапі, так і при подальшому масштабуванні.

Для генерації тестових даних була обрана бібліотека Faker, яка забезпечує швидке створення різноманітних, реалістичних значень (імена, адреси, дати, електронні адреси тощо). Це дозволяє автоматизувати параметризацію тестів без необхідності ручного введення або розробки власних генераторів, що підвищує підтримуваність і якість тестових сценаріїв. Завдяки широкому використанню та відкритому ліцензуванню, Faker є стабільним і надійним інструментом, що суттєво спрощує процес підготовки даних для АТ

Для системи звітності обрано ReportPortal – хмарний TestOps-сервіс для централізованого збору, аналізу та візуалізації результатів тестування[23]. ReportPortal забезпечує розгорнуті дашборди, пошук у логах, історію запусків і статистику дефектів, що спрощує моніторинг якості. Він має нативну інтеграцію з CI/CD (Jenkins, GitHub Actions) і підтримує тестовий фреймворк pytest через розширення pytest-reportportal, що дозволяє автоматизувати обробку результатів і покращити командну співпрацю, зокрема зв’язування тестів з дефектами. На відміну від локальних рішень (наприклад, Allure), ReportPortal є незалежним від CI/CD і надає розширені функції аналізу, включно з AI-тріажем та Quality Gates. Основна складність – налаштування і підтримка інфраструктури (база даних, сервер), що компенсується значно глибшим аналітичним потенціалом.

Для управління тестовими кейсами обрано TestRail – хмарний сервіс, який забезпечує централізоване планування, виконання та моніторинг тестів. Інструмент підтримує інтеграцію з CI/CD і тестовими фреймворками через API, що робить його гнучким і сумісним із різними стеком технологій. TestRail виділяється зрозумілим інтерфейсом та можливістю масштабування під потреби команд будь-якого розміру. На відміну від рішень, орієнтованих виключно на автоматизацію звітності, він фокусується на організації тестових процесів і контролі якості, що робить його універсальним для ручного і автоматизованого тестування. Хоча сервіс платний, він пропонує пробний період і різні тарифи, що дозволяє адаптуватися до бюджету проєкту.

Підсумовуючи вибір інструментів: комбінація для Python – Pytest, HTTPX, Playwright, Faker, ReportPortal, та TestRail дозволяє реалізувати усі функціональні вимоги (див. 2.1.2) – від автоматизації UI/API до збору звітів і керування тест-кейсами – і при цьому забезпечує сучасний, гнучкий стек. Альтернативи цих рішень розглянуті і виправданий їх вибір. Таке поєднання інструментів закриває сильні та слабкі сторони одне одного і відповідає вимогам проєкту.

2.4 Перспективи розширення засобу

Серед перспективних розширень на майбутнє можна виокремити такі:

- інтеграція з CI/CD пайплайнами дозволить забезпечити безперервне виконання тестів на кожному етапі життєвого циклу розробки. Завдяки підключенню автоматизованого тестування до CI/CD, усі зміни коду можуть супроводжуватись автоматичним запуском тестів, формуванням звітів та фіксацією результатів у системах типу ReportPortal або TestRail. Це сприятиме ранньому виявленню помилок, підвищенню прозорості процесу розробки та дозволить оперативно зворотно зв’язувати тестування з командами розробки. Крім того, автоматизоване розгортання середовищ і запуск тестів у контейнерах або віртуальних машинах дозволяє масштабувати тестування та зменшити час зворотного циклу.
- генерація тест-кейсів за допомогою LLM (Large Language Model) дозволить автоматично створювати первинні тест-кейси з описів вимог, документації або прикладів коду. LLM можуть аналізувати текст специфікацій або веб-інтерфейси й формувати сценарії тестування, які слугуватимуть основою для pytest. це значно прискорить покриття тестами та підвищить продуктивність команд, особливо при частих змінах вимог;
- автоматизація обробки flaky-тестів через модуль «self-healing» дасть змогу відстежувати нестабільні тести і намагатись їх стабілізувати,

коригуючи таймінги, параметри середовища або застосовуючи машинне навчання для класифікації причин нестабільності. цей сервіс міг би аналізувати результати в Report Portal, визначати патерни та автоматично створювати завдання в системі управління тестами або повторно запускати тести з адаптованими налаштуваннями. Такий модуль у рамках архітектури використовує існуючі компоненти (ReportPortal, TestRail, логіку тест-раннера) та додає «поверховий» сервіс для обробки помилок. Він підвищує стабільність CI/CD-потоку, знижуючи помилкові тривоги та економлячи ресурси на невдалі запуски;

– тестування продуктивності шляхом додавання модуля для автоматичного навантажувального тестування API та інших критичних компонентів системи допоможе контролювати швидкодію і стабільність під різними сценаріями навантаження.

3 РЕАЛІЗАЦІЯ ЗАСОБУ ТЕСТУВАННЯ

3.1 Налаштування середовища розробки та інфраструктури

Для розробки використано Visual Studio Code (VS Code) через його гнучкість та підтримку Python. У VS Code було встановлено розширення, такі як Python, Pylance (для автодоповнення та статичного аналізу коду), GitLens (для зручної роботи з Git) і Docker (для управління контейнерами). Для забезпечення якості коду налаштовано лінтери, зокрема flake8 для перевірки стилю коду і pylint для виявлення потенційних помилок. Налаштування VS Code включало конфігурацію інтерпретатора Python 3.10 для забезпечення сумісності з усіма бібліотеками, використаними в проєкті.

Для ізоляції залежностей проєкту створено віртуальне середовище за допомогою відповідного інструменту venv. Усі необхідні бібліотеки, такі як pytest, httpx, playwright, faker і pytest-reportportal, було встановлено через файл requirements.txt. Вміст файлу представлено на рисунку 3.1.

requirements.txt

```
# Тестування
pytest>=7.0.0
pytest-asyncio>=0.21.0

# HTTP клієнт для API тестів
httpx>=0.24.0

# UI тестування
playwright>=1.40.0
pytest-playwright>=0.4.3

# TestRail інтеграція
pytest-testrail>=2.9.0
trcli

# Контейнери для integration тестів
testcontainers>=3.7.0
docker>=6.0.0

# Генерація даних
faker>=18.0.0

# Звітність
reportportal-client>=5.2.0

# Моніторинг ресурсів для метрик
psutil>=5.9.0

# Інструменти якості коду
black==24.3.0
flake8>=7.0.0
mypy>=1.9.0

# Додаткові пакети
coverage>=7.5.0

# Stripe тестування (для e-commerce платежів)
stripe>=7.0.0
```

Рисунок 3.1 – залежності проєкту

Віртуальне середовище активується перед кожною сесією розробки за допомогою команди source venv/bin/activate (для Linux/Mac) або venv\Scripts\activate (для Windows). Таким чином, ми можемо уникнути конфліктів між версіями бібліотек.

Для управління кодом, будемо використовувати Git із репозиторієм на GitHub. Налаштування Git включало ініціалізацію репозиторію в директорії проєкту, створення .gitignore для виключення файлів віртуального середовища та тимчасових файлів.

Також, інструмент забезпечує можливість запуску в Docker-контейнері, що гарантує відтворюваність і стабільність процесу тестування. Завдяки цьому усувається необхідність встановлення Python, залежностей та конфігурування середовища на кожному робочому місці – достатньо лише мати встановлений Docker і запустити контейнер. Такий підхід є особливо ефективним для командної роботи: кожен розробник може проводити тестування в уніфікованому середовищі, що виключає проблеми, пов’язані з несумісністю версій бібліотек, конфігурацій операційних систем або локальних налаштувань, забезпечуючи ідентичні умови виконання тестів.

Dockerfile

```dockerfile
FROM python:3.10-slim

WORKDIR /app

ARG PLAYWRIGHT_BROWSERS=chromium,firefox
ARG DEFAULT_TEST_TYPE=ui

# Встановлюємо системні залежності
RUN apt-get update && apt-get install -y \
    wget \
    gnupg \
    unzip \
    curl \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*

# Копіюємо файли залежностей
COPY requirements.txt .
COPY pyproject.toml .
COPY setup.cfg .

# Встановлюємо Python залежності
RUN pip install --no-cache-dir -r requirements.txt

RUN python3 -m playwright install

# Копіюємо весь проект
COPY . .

# Створюємо необхідні директорії
RUN mkdir -p test_reports test_results htmlcov rp_data opensearch_data

# Встановлюємо права доступу
RUN chmod -R 755 /app

# Налаштовуємо Python path
ENV PYTHONPATH=/app
ENV PYTHONUNBUFFERED=1

# Встановлюємо змінні середовища за замовчуванням
ENV TEST_TYPE={DEFAULT_TEST_TYPE}
ENV EXTRA_ARGS=""
ENV ENVIRONMENT=test

# Команда за замовчуванням
CMD ["python", "-m", "src.presentation.cli", "--type", ${TEST_TYPE}, ${EXTRA_ARGS}]
```

Рисунок 3.2 – Docker-файл для запуску засобу

Цей Dockerfile створює зручне середовище для тестування на основі легкої версії Python 3.10. Усе розгортається в папці /app, куди копіюються файли проєкту та залежності. Спочатку встановлюються потрібні системні інструменти, як-от для роботи з мережею чи архівами, а потім – бібліотеки Python із файлу requirements.txt, включно з Playwright для перевірки інтерфейсів у браузерах. Для коректної роботи додаються файли конфігурації проєкту. Створюються папки для звітів і результатів тестів, налаштовані

правда досутпу, щоб усе працювало без збоїв. Змінні середовища дозволяють легко перемикати тип тестів (наприклад, інтерфейсні) і задавати додаткові параметри.

Для запуску потрібно спочатку збудувати image, за допомогою команди “docker build -t testing-system:latest .” . Є декілька варіантів запуску для зручної організації процесу розробки. Наприклад для запуску ui тестів, потрібно просто при запуску встановити відповідне значення у змінну TEST_TYPE через флаг -e, що встановює змінні оточення для контейнера: “docker run --rm -e TEST_TYPE=ui testing-system”. Щоб встановити додаткові флаги команди, наприклад надсилання результатів у ReportPortal, потрібно виконати “docker run –rm -e TEST_TYPE=api -e EXTRA_ARGS="—reportportal"”.

3.2 Розробка основних компонентів інструменту

Як уже було описано в 2.2, архітектура розробленого засобу АТ базується на принципах Clean Architecture та Domain-Driven Design, оскільки цей підхід забезпечує слабку зв’язаність компонентів і можливість розширення системи.

У цьому розділі детально описано створення ключових модулів інструменту, зосереджуючись на технічних аспектах реалізації, інтеграції та вирішенні практичних викликів. Кожен компонент розроблено для покриття певного рівня тестової піраміди, інтеграції з системами управління тестами (TMS) і CI/CD, а також забезпечення додаткової функціональності.

3.2.1 Реалізація базової інфраструктури тестування

У цьому підрозділі описано архітектуру модуля, технічні аспекти реалізації, конфігурацію, інтеграцію з іншими компонентами системи, а також проаналізовано переваги й недоліки обраних рішень.

До юніт-тестів зазвичай висуваються такі вимоги:

- перевірка коректності реалізації окремих функцій і класів цільового додатку;

- забезпечення швидкого зворотного зв’язку розробникам під час CI/CD;
- досягнення високого покриття коду (цільовий показник – 70-80%), що є стандартом для проєктів середньої складності.

В нашому засобі, для цільового додатку, ці тести будуть зберігатися у директорії tests/unit, яка структурована за модульним принципом: кожна директорія тестів відповідає певному модулю цільового додатку (наприклад, business_logic), тобто розподіл тестів “по функціям”.

Юніт-тестування інтегровано в загальну архітектуру інструменту через адаптер PytestAdapter, який реалізує інтерфейс TestAdapter.

На рівні домену визначено абстрактний інтерфейс TestAdapter, який забезпечує контракт для всіх типів тестових адаптерів. Цей інтерфейс включає обов'язкові методи для запуску тестів, валідації конфігурації та отримання метаінформації про фреймворк. Клас PytestAdapter на рівні інфраструктури реалізує інтерфейс і забезпечує інтеграцію з фреймворком pytest. Адаптер підтримує паттерн dependency injection для сервісу помилок, що дозволяє централізовано обробляти помилки та створювати стандартизовані результати тестування. Це забезпечує консистентність обробки помилок у всій системі.

На рівні додатка TestOrchestrator координує виконання тестів, використовуючи фабрику адаптерів IAdapterFactory для створення відповідних адаптерів. Це дозволяє системі бути розширюваною – додавання нових типів тестів не вимагає змін в оркестраторі, а лише реєстрації нового адаптера в фабриці.

TestRunner діє як фасад для всієї системи тестування, приймаючи конфігурацію та делегуючи виконання до оркестратора. Він також відповідає за підготовку середовища виконання, включаючи встановлення змінних оточення та налаштування логування.

На рисунку 3.1 представлено візуалізацію цієї архітектури, а саме, діаграму послідовності виконання тестів.

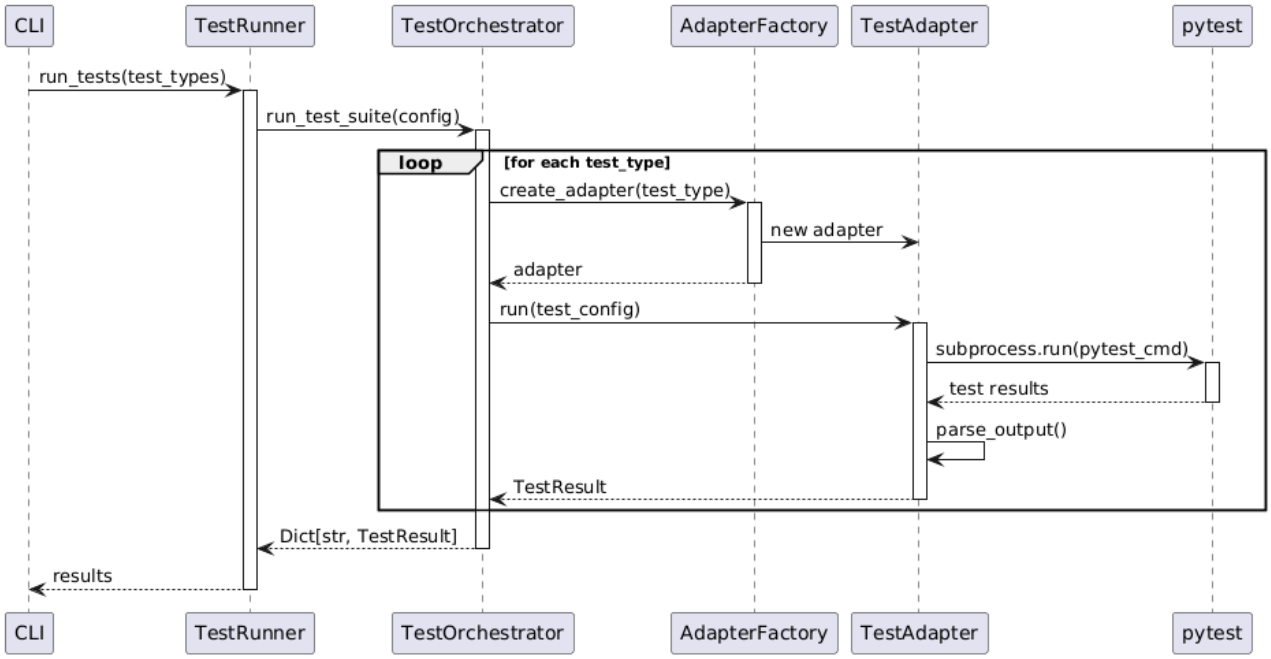

Рисунок 3.1 – діаграму послідовності виконання тестів

Тепер, проаналізуємо переваги і недоліки такого підходу.

Переваги:

- високий рівень абстракції дозволяє легко замінювати або додавати нові фреймворки тестування без змін в коді додатка;
- патерн Adapter забезпечує уніфікований інтерфейс для різних фреймворків тестування;
- Dependency Injection покращує тестованість коду та дозволяє гнучке налаштування залежностей;
- автоматична інтеграція з системами звітування (TestRail, ReportPortal) зменшує ручну роботу команди QA;
- модульна структура тестів забезпечує логічне групування та спрощує навігацію в кодовій базі;
- підтримка паралельного виконання тестів через asyncio дозволяє значно скоротити час виконання.

Недоліки:

- сильна залежність від pytest;
- додаткова складність архітектури може бути надмірною для простих проєктів;
- залежність від зовнішніх утиліт (trcli) може створювати проблеми при їх недоступності;
- необхідність підтримки множинних конфігураційних файлів ускладнює налаштування системи;
- асинхронна природа системи може ускладнювати налагодження при виникненні помилок.

3.2.2 Реалізація інфраструктури для API-тестування

API-тестування реалізоване через HttpxAdapter, який використовує HTTPX бібліотеку для асинхронного виконання HTTP запитів. Адаптер підтримує повний життєвий цикл HTTP клієнта включаючи ініціалізацію, виконання запитів та коректне закриття з'єднань.

Система включає розширені можливості конфігурації включаючи підтримку різних типів автентифікації (Basic, Bearer), налаштування SSL верифікації та керування таймаутами. Адаптер реалізує механізм health check для перевірки доступності API перед виконанням тестів.

HttpxAdapter підготовляє конфігурацію pytest з специфічними маркерами для API тестів та автоматично налаштовує змінні середовища.

3.2.3 Реалізація інфраструктури для Integration-тестування

Інтеграційне тестування реалізоване через IntegrationAdapter, який забезпечує тестування взаємодії між компонентами системи. Адаптер підтримує використання контейнерів Docker для ізоляції тестового середовища та створення відтворюваних умов тестування.

IntegrationAdapter використовує комбінацію різних технологій включаючи pytest для організації тестів, Docker для контейнеризації залежностей та HTTPX для тестування API взаємодії між компонентами. Система автоматично налаштовує тестове середовище з попередньо заповненими даними.

Адаптер реалізує складну логіку управління життєвим циклом тестового середовища включаючи запуск контейнерів, ініціалізацію бази даних, створення тестових користувачів та очищення ресурсів після завершення тестів. Це забезпечує повну ізоляцію тестів та їх незалежність від зовнішніх факторів.

3.2.4 Реалізація інфраструктури для UI-тестування

UI-тестування реалізоване через PlaywrightAdapter, який використовує Playwright фреймворк для автоматизації браузерів. Adapter підтримує Chromium, Firefox та WebKit браузери з можливістю налаштування headless режиму для CI/CD пайплайнів.

PlaywrightAdapter забезпечує розширені можливості для UI тестування включаючи автоматичне створення скріншотів при падінні тестів, запис відео для складних сценаріїв та автоматичне очікування елементів інтерфейсу.

Адаптер інтегрується з паттерном Page Object Model та підтримує генерацію тестових даних через DataGenerator для створення унікальних користувачів та сценаріїв тестування.

3.3 Інтеграція з TMS та сервісом аналізу результатів тестування

Система звітності реалізована через модульну архітектуру з окремими клієнтами для різних платформ управління тестами. Інтеграція сприяє дотриманню пункту 4 функціональних вимог до засобу що розробляється (див. 2.1.2), про централізований збір результатів, аналіз трендів та командну роботу над тестуванням.

Інтеграція ReportPortal реалізована через ReportPortalClient, який забезпечує повний життєвий цикл взаємодії з платформою за допомогою плагіну pytest-reportportal. Клієнт підтримує автоматичну конфігурацію через змінні середовища та централізовану обробку credentials. ReportPortalClient інкапсулює специфічну логіку створення запуску, відправки результатів тестів та прикріплення артефактів: Система автоматично створює launch з метаданими про тип тестів, браузер (для UI тестів) та середовище виконання. ReportPortal клієнт підтримує прикріплення скріншотів, логів та інших артефактів тестування.

TestRail інтеграція реалізована за допомогою TestRail CLI для автоматичного створення тест-кейсів. Схематичне зображення цього процесу зображено на рисунку 3. [24].

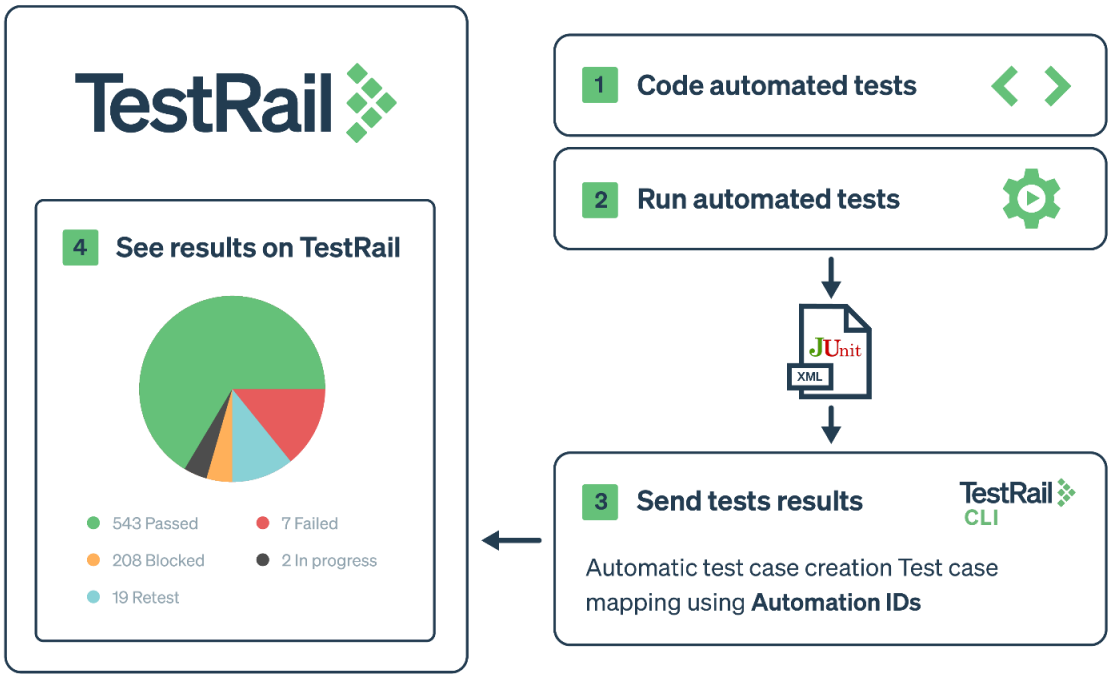

Рисунок 3.2 – Процес автоматизації тестування та відображення результатів у системі TestRail

Як видно зі схеми, система використовує JUnit XML як проміжний формат для передачі результатів. Архітектура TestRail інтеграції включає автоматичне створення тест-кейсів на основі pytest результатів, створення test runs з результатами виконання. Система підтримує Automation ID маппінг для зв'язування фактичних тестів в коді з тест-кейсами в TestRail.

3.4 Реалізація допоміжних модулів

3.4.1 Генерація тестових даних з використанням Faker

DataGenerator реалізує комплексну систему генерації тестових даних на базі Faker бібліотеки з додатковими можливостями для створення доменно-специфічних об'єктів. Генератор підтримує шаблони (DataTemplate) для стандартизації процесу створення тестових даних.

Система включає розширену типізацію даних через DataType enum та FieldConfig для детального налаштування генерації. Відповідний фрагмент коду представлено на рисунку 3.3

```python
class DataType(Enum):
    """Типи даних для генерації."""
    STRING = "string"
    INTEGER = "integer"
    FLOAT = "float"
    BOOLEAN = "boolean"
    EMAIL = "email"
    URL = "url"
    PHONE = "phone"
    DATE = "date"
    DATETIME = "datetime"
    UUID = "uuid"
    TEXT = "text"
    NAME = "name"
    ADDRESS = "address"
    COMPANY = "company"
    CURRENCY = "currency"
    IP_ADDRESS = "ip"
    MAC_ADDRESS = "mac"
    CREDIT_CARD = "credit_card"
    PASSWORD = "password"
    USERNAME = "username"


You, 4 days ago | 1 author (You)
@dataclass
class FieldConfig:
    """Конфігурація поля для генерації."""
    data_type: DataType
    required: bool = True
    min_value: Optional[Union[int, float]] = None
    max_value: Optional[Union[int, float]] = None
    max_length: Optional[int] = None
    choices: Optional[List[Any]] = None
    format_pattern: Optional[str] = None
    custom_generator: Optional[Callable] = None
    dependencies: List[str] = field(default_factory=list)
```

Рисунок 3.3 – можливість детального налаштування генерації даних

DataGenerator підтримує генерацію великих обсягів даних даних з гарантією унікальності значень, локалізацію (українська локаль за замовчуванням) та детермінований характер генерації через seed значення. Система включає готові шаблони для типових веб-додатків: користувачі, продукти, замовлення, статті.

4 ВПРОВАДЖЕННЯ ТА РЕЗУЛЬТАТИ ЗАСТОСУВАННЯ ЗАСОБУ

4.1 Огляд додатку для тестування

4.1.1 Загальна характеристика і джерело проєкту

Об'єктом тестування виступає веб-додаток «Grocery Store Site» – багатокористувацький інтернет-магазин для управління продуктовою крамницею. Додаток розроблений з використанням фреймворку Flask та бази даних SQLite, що дозволяє користувачам здійснювати пошук та замовлення товарів, а також надає функціональність управління запасами та категоріями товарів.

Система спроектована як повноцінна e-commerce платформа з підтримкою різних ролей користувачів та інтеграцією із зовнішніми сервісами. Додаток забезпечує можливість пошуку та запиту товарів покупцями, управління інвентарем та створення, редагування й видалення різних категорій менеджерами магазину після затвердження адміністратором, а також управління доставками виконавцями доставки. Розробники можуть використовувати CRUD API для товарів та категорій, використовуючи свої приватні API ключі.

Оригінальний код проекту доступний у репозиторії GitHub за адресою https://github.com/Schefflera-Arboricola/Grocery-Store-Site[25].

4.1.2 Технологічний стек та архітектура

Серверна частина додатку побудована на фреймворку Flask, який є одним з найпопулярніших мікрофреймворків для Python. Для роботи з базою даних використовується SQLite у поєднанні з SQLAlchemy ORM, що забезпечує ефективне управління даними та міграціями.

Система автентифікації та авторизації реалізована через Flask-Login та Flask-Security, що надає надійний механізм управління сесіями користувачів та контролю доступу. API функціональність забезпечується через Flask-RESTful та Flask-RESTX, що дозволяє створювати документовані RESTful

веб-сервіси з автоматичною генерацією документації. Для обробки асинхронних задач та фонових процесів використовується Celery у поєднанні з Redis. Це дозволяє ефективно обробляти такі операції як відправка електронних листів, генерація звітів та інші ресурсомісткі завдання без блокування основного потоку виконання. Кешування реалізовано через Flask-Caching з використанням Redis як бекенду, що значно покращує продуктивність додатку.

Додаток інтегрований з кількома зовнішніми сервісами для забезпечення повноцінної e-commerce функціональності. Платіжна система реалізована через Stripe API, що забезпечує безпечну обробку кредитних карток та інших платіжних методів. Для SMS та голосових повідомлень використовується Twilio API, що дозволяє відправляти OTP коди та інші важливі сповіщення. Email сервіс реалізований через Flask-Mail з підтримкою SMTP, що забезпечує автоматичну відправку нагадувань, підтверджень замовлень та періодичних звітів.

Додаток також включає модулі машинного навчання на базі Scikit-learn та NumPy для реалізації рекомендаційних систем та аналізу даних. Архітектура додатку побудована за багаторівневим принципом з чітким розділенням відповідальності.

Основна логіка додатку розміщена в директорії application, яка містить контролери для різних ролей користувачів, моделі бази даних, RESTful API ендпоінти, конфігурацію додатку, Celery задачі та налаштування бази даних. Шаблони HTML розміщені в директорії templates, статичні ресурси в директорії static, а моделі машинного навчання винесені в окрему директорію ML_models.

4.1.3 Функціональні можливості та система ролей

Додаток реалізує Role-Based Access Control (RBAC) з п'ятьма типами користувачів, кожен з яких має специфічний набір функцій та привілеїв.

Покупці мають доступ до найширшого спектру функціональності, спрямованої на забезпечення зручного процесу покупок.

Система пошуку дозволяє знаходити товари за категоріями, рейтингами та різноманітними характеристиками продукції. Покупці можуть додавати товари до віртуального кошика, оформлювати замовлення з подальшим відстеженням статусу, залишати оцінки та відгуки про товари.

Персоналізація досвіду забезпечується через систему рекомендацій на основі машинного навчання, яка аналізує попередні покупки та пропонує релевантні товари (див. рисунок 4.1).

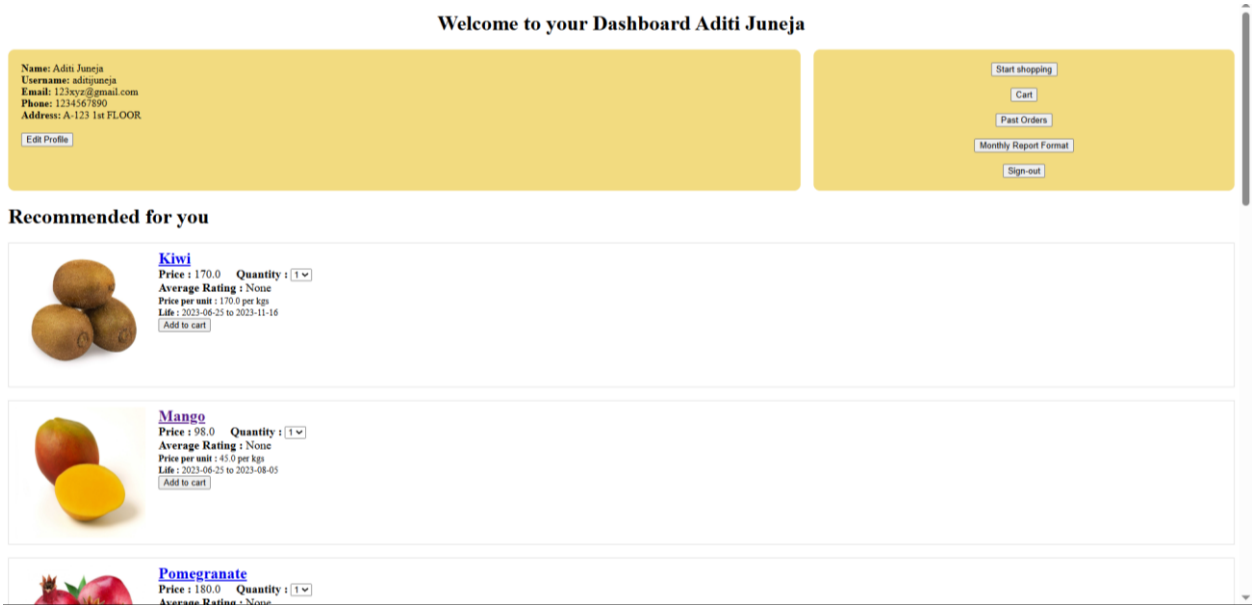

Рисунок 4.1 – система рекомендацій тестуємого додатку

Адміністратор системи має найвищі привілеї та відповідає за загальне управління платформою. До його можливостей входить перегляд та затвердження заявок від менеджерів магазину на реєстрацію в системі, затвердження запитів на створення нових категорій товарів, а також повний доступ до управління товарами та категоріями.

Менеджери магазину мають обмежені, але релевантні для роботи привілеї, а саме, вони можуть самостійно управляти товарами, проте для

управління категоріями повинні створювати запити на затвердження адміністратором. Менеджери також мають доступ до аналітичних функцій, включаючи експорт товарів у CSV формат.

Кур'єри або виконавці доставки мають спеціалізований інтерфейс для управління замовленнями та оновлення їх статусів через систему OTP верифікації.

Розробники представляють особливу категорію користувачів з програмним доступом до системи через RESTful API з використанням приватних API ключів, що дозволяє виконувати CRUD операції без веб-інтерфейсу.

4.1.4 Ключові особливості та деталі

Пошукова система додатку побудована на основі багатокритеріального аналізу з підтримкою пошуку за категоріями, рейтингами, ціновими діапазонами та текстовими характеристиками товарів. Рекомендаційна система базується на алгоритмах машинного навчання та аналізує поведінку користувачів для надання персоналізованих пропозицій, враховуючи історію покупок, переглянуті товари та інші метрики взаємодії. Додатково реалізований алгоритм аналізу схожості товарів на основі їх описів з використанням технологій обробки природної мови.

Повний цикл електронної комерції реалізований від моменту вибору товару до його доставки покупцю. Система кошика забезпечує персистентність даних між сесіями, інтеграція зі Stripe забезпечує безпечну обробку платежів, а система відстеження замовлень працює в реальному часі. Комунікаційна система забезпечує багатоканальне інформування користувачів через email та SMS, включаючи автоматичні щоденні нагадування, місячні звіти та OTP верифікацію через Twilio.

Архітектура додатку спроектована з урахуванням потреб масштабування та забезпечення високої продуктивності. Redis кешування використовується на кількох рівнях для зменшення навантаження на базу

даних, а асинхронна обробка через Celery дозволяє виносити ресурсомісткі операції за межі основного циклу обробки HTTP запитів. Багаторівнева система безпеки включає Flask-Security, rate limiting через Flask-Limiter, CSRF захист, валідацію вхідних даних та логування всіх критичних операцій.

4.1.5 Розгортання та тестові можливості

Для успішного функціонування додатку необхідне середовище Python версії 3.9 або новішої з встановленими залежностями, Redis сервер для роботи Celery черг та системи кешування, а також стабільне інтернет з'єднання для роботи зовнішніх API сервісів.

Додаток підтримує два основні варіанти розгортання: локальне середовище через скрипти local_setup.sh та local_run.sh, що є рекомендованим для розробки та тестування, та контейнеризацію через Docker з включеними Dockerfile та docker-compose.yml файлами для більш стандартизованого середовища.

Система конфігурації побудована на принципі розділення налаштувань для різних середовищ з використанням .env файлів для зберігання змінних середовища.

API додатку спроектований відповідно до принципів REST архітектури з автоматичною генерацією документації через Swagger UI. Автентифікація API здійснюється через Bearer token систему з унікальними ключами для кожного розробника. Для тестування додаток надає готові облікові записи з логіном "aditijuneja" та паролем "123456789".

Тестування платіжної функціональності можливе через тестовий режим Stripe з використанням спеціальних номерів карток, наприклад "4242 4242 4242 4242" з будь-якою майбутньою датою закінчення терміну дії та трицифровим CVC кодом.

Цей багатофункціональний додаток представляє собою сучасну e-commerce платформу з інноваційними функціями машинного навчання, надійною архітектурою та комплексним підходом до безпеки, що робить його

ідеальним об'єктом для всебічного тестування різних аспектів сучасних веб-додатків.

4.2 Планування і документація тестування

Основною метою тестування, в контексті даної роботи, є демонстрація ефективності розробленої системи АТ на реальному веб-додатку, а не досягнення значного покриття функціональності. Отже, метою тестування не є максимальне покриття коду, а демонстрація можливостей засобу комбінувати різні інструменти в одній системі для досягнення максимальної ефективності тестування в контексті процесів розробки ПЗ.

Для демонстрації написання тестової документації перед тестуванням, створимо чек-ліст для тестування модуля системи рекомендацій і декілька тест кейсів перевірки авторизації і навігації на рівні інтерфейсу користувача.

Чек-ліст – це структурований перелік елементів, умов або функцій, які необхідно перевірити під час тестування, щоб переконатися, що система відповідає вимогам. Використовується для організації тестування, але не деталізує кроки виконання.

Тест-кейс – це набір умов, кроків і очікуваних результатів, розроблений для перевірки конкретної функціональності або поведінки системи. Включає детальний опис дій тестувальника та критерії оцінки результатів.

Чекліст у таблиці 4.1 покриває аспекти тестування рекомендаційної системи від базової функціональності до граничних випадків та обробки помилок, забезпечуючи надійність та якість компонента машинного навчання. Перевіряються базові рекомендації за категоріями (перевірка структури, виключення замовлених товарів, обмеження кількості N=1,3,5,10), пошук схожих товарів (категоризація, виключення цільового товару, сортування за релевантністю), розрахунок частоти замовлень і ранжування, обробку некоректних даних (None, порожні списки, неправильні типи), продуктивність (1000+ товарів, 500+ замовлень), консистентність результатів та використання DataGenerator для складних сценаріїв.

Таблиця 4.1 – Чек-лист тестування модуля рекомендаційної системи.

№	Пункт
1	Перевірити повернення списку рекомендацій для валідних вхідних даних
2	Перевірити структуру повернутих даних (список словників)
3	Перевірити що рекомендації містять товари з тієї ж категорії
4	Перевірити виключення вже замовлених товарів з рекомендацій
5	Перевірити рекомендації при одному товарі замовленому багато разів
6	Перевірити рекомендації при різних товарах замовлених по одному разу
7	Перевірити рекомендації при змішаному патерні замовлень
8	Перевірити обмеження кількості результатів параметром N=1
9	Перевірити обмеження кількості результатів параметром N=3
10	Перевірити обмеження кількості результатів параметром N=5
11	Перевірити обмеження кількості результатів параметром N=10
12	Перевірити пошук схожих товарів за категорією
13	Перевірити виключення цільового товару з результатів пошуку
14	Перевірити правильність категоризації схожих товарів
15	Перевірити сортування рекомендацій за частотою замовлень
16	Перевірити підрахунок частоти при множинних замовленнях одного товару
17	Перевірити пріоритезацію товарів з вищою частотою замовлень
18	Перевірити отримання товарів за списком ідентифікаторів
19	Перевірити порядок товарів у результатах пошуку за ID
20	Перевірити повний процес рекомендаційного процесу
21	Перевірити обробку None замість списку замовлень
22	Перевірити обробку порожнього списку замовлень
23	Перевірити обробку некоректної структури замовлень

Продовження таблиці 4.1

24	Перевірити обробку замовлень з невірними ключами
25	Перевірити обробку порожніх словників у замовленнях
26	Перевірити обробку товару без category_id
27	Перевірити обробку негативного значення параметру N
28	Перевірити обробку нульового значення параметру N
29	Перевірити обробку N більшого за кількість доступних товарів
30	Перевірити обробку порожнього списку товарів
31	Перевірити обробку порожнього списку категорій
32	Перевірити обробку неправильних типів даних замість списків
33	Перевірити обробку строки замість списку
34	Перевірити обробку числа замість словника
35	Перевірити відсутність падіння системи при KeyError
36	Перевірити відсутність падіння системи при IndexError
37	Перевірити відсутність падіння системи при TypeError
38	Перевірити відсутність падіння системи при AttributeError
39	Перевірити повернення порожнього списку при відсутності рекомендацій
40	Перевірити граничний випадок з одним товаром у системі
41	Перевірити роботу з великою кількістю товарів (1000+)
42	Перевірити роботу з великою історією замовлень (500+)
43	Перевірити час виконання алгоритму (< 1 секунди)
44	Перевірити використання пам'яті при великих даних
45	Перевірити детермінованість результатів при повторних запусках
46	Перевірити консистентність сортування результатів
47	Перевірити використання DataGenerator для генерації тестових даних
48	Перевірити правильність роботи з згенерованими даними

Закінчення таблиці 4.1

49	Перевірити підрахунок частоти при дублікатах в одному замовленні
50	Перевірити масштабованість алгоритму при зростанні даних

Тест кейси покриття функціоналу автентифікації та навігації представлені у таблиці 4.2.

Таблиця 4.2 – Тест кейси перевірок автентифікації та навігації

№	Назва тест-кейса	Передумови	Кроки виконання	Очікуваний результат
1	Вхід адміністратора з валідними даними	1. Веб-додаток доступний 2. Користувач admin існує в системі 3. Пароль адміністратора відомий	1. Відкрити сторінку логіну 2. Ввести логін адміністратора 3. Ввести пароль адміністратора 4. Натиснути кнопку "Увійти" 5. Дочекатися переходу на дашборд	1. Система перенаправляє на дашборд 2. Відображається контент для адміністратора 3. Відсутні системні помилки 4. В інтерфейсі видно роль "admin"
2	Вхід клієнта з валідними даними	1. Веб-додаток доступний 2. Користувач customer існує в системі 3. Пароль клієнта відомий	1. Відкрити сторінку логіну 2. Ввести логін клієнта 3. Ввести пароль клієнта 4. Натиснути кнопку "Увійти" 5. Дочекатися переходу на дашборд	1. Система перенаправляє на дашборд 2. Відображається контент для клієнта 3. Відсутні системні помилки 4. В інтерфейсі видно роль "customer"

Продовження таблиці 4.2

3	Вхід з неіснуючими обліковими даними	1. Веб-додаток доступний 2. Підготовлені неіснуючі логін і пароль	1. Відкрити сторінку логіну 2. Ввести неіснуючий логін 3. Ввести неіснуючий пароль 4. Натиснути кнопку "Увійти" 5. Дочекатися відповіді системи	1. Логін не виконується 2. Користувач залишається на сторінці логіну 3. Відображається повідомлення про помилку 4. Поля логіну очищаються або підсвічуються
4	Вхід з порожніми полями	1. Веб-додаток доступний 2. Сторінка логіну відкрита	1. Відкрити сторінку логіну 2. Залишити поле логіну порожнім 3. Залишити поле пароля порожнім 4. Натиснути кнопку "Увійти" 5. Перевірити поведінку системи	1. Логін не виконується 2. Відображається валідаційне повідомлення 3. Кнопка "Увійти" неактивна або показує помилку 4. Користувач залишається на сторінці логіну
5	Вхід з правильним логіном та неправильним паролем	1. Веб-додаток доступний 2. Існуючий логін користувача відомий 3. Підготовлений неправильний пароль	1. Відкрити сторінку логіну 2. Ввести існуючий логін 3. Ввести неправильний пароль 4. Натиснути кнопку "Увійти" 5. Дочекатися відповіді системи	1. Логін не виконується 2. Відображається повідомлення "Неправильний логін або пароль" 3. Користувач залишається на сторінці логіну 4. Поле пароля очищається

Закінчення таблиці 4.2

6	Перевірка навігації до сторінки логіну	1. Веб-додаток доступний 2. Базова URL відома	1. Відкрити базову URL додатку 2. Знайти посилання/кнопку для входу 3. Клікнути на посилання входу 4. Дочекатися завантаження сторінки	1. Сторінка логіну відкривається 2. Відображаються поля логіну та пароля 3. Присутня кнопка "Увійти" 4. URL містить шлях до логіну (/login)
7	Перевірка відображення дашборду після успішного входу	1. Користувач успішно увійшов в систему 2. Перехід на дашборд виконано	1. Переконатися що вхід виконано 2. Перевірити URL дашборд 3. Перевірити наявність контенту дашборду 4. Перевірити відсутність помилок	1. URL містить шлях до дашборду 2. Контент відповідає ролі користувача 3. Відсутні JavaScript помилки 4. Сторінка повністю завантажена
8	Перевірка повідомлення про помилку при невдалому вході	1. Підготовлені неправильні дані для входу 2. Сторінка логіну відкрита	1. Ввести неправильні дані 2. Спробувати увійти 3. Дочекатися відповіді 4. Перевірити повідомлення про помилку	1. Повідомлення про помилку відображається 2. Текст помилки зрозумілий користувачу 3. Повідомлення зникає після виправлення даних 4. Стиль повідомлення відповідає дизайну

Основна увага приділяється ключовим аспектам, які впливають на стабільність інтерфейсу та відповідність системної поведінки очікуваним сценаріям для різних категорій користувачів.

Тести охоплюють сценарії успішного входу для основних ролей – адміністратора та клієнта. Це дозволяє впевнитися, що система коректно обробляє автентичні облікові дані та надає доступ до відповідного

функціоналу залежно від ролі. Такий підхід є критичним для гарантування ролевої ізоляції і правильного функціонування механізмів контролю доступу.

Проводиться валідація негативних сценаріїв, зокрема: введення неіснуючих облікових даних, залишення полів порожніми та використання неправильного пароля. Ці перевірки є необхідними для забезпечення надійності та безпеки системи, адже дозволяють своєчасно виявити можливі вразливості у верифікації даних та відображенні повідомлень про помилки.

Окремо перевіряється доступність та функціональність сторінки входу, що є критичною точкою взаємодії користувача із системою. Крім того, тести включають перевірку відображення дашборду після успішної автентифікації. Це дозволяє гарантувати, що система коректно виконує перенаправлення та завантажує інформацію згідно з контекстом користувача.

Значна увага приділяється перевірці змістовності й однозначності повідомлень про помилки. Їх коректність є важливою не лише з точки зору зручності для користувача, а й у контексті інформаційної безпеки – повідомлення мають надавати достатньо інформації для розуміння помилки, але не розкривати внутрішніх деталей реалізації або структури системи.

4.3 Розгортання проєкту і підготовка тестового середовища

Спочатку, потрібно склонувати репозиторій із Github в директорію із розробленим засобом за допомогою команди git clone https://github.com/Schefflera-Arboricola/Grocery-Store-Site, або якщо використовуються ssh – git clone git@github.com:Schefflera-Arboricola/Grocery-Store-Site.git.

Оскільки планується запуск додатку через комплексний Docker Compose файл, згідно наданої документації в репозиторії, у файлі application/config.py, класі LocalDevelopmentConfig, необхідно розкоментувати ініціалізацію змінної SQLITE_DB_DIR для використання з Docker та закоментувати її для віртуального середовища.

Для налаштування інтеграції сторонніх сервісів (Twilio, Stripe, SMTP) необхідно отримати відповідні конфігураційні параметри та зберегти їх у файлі .env. Нижче описано порядок дій для отримання кожної змінної:

Для інтеграції з сервісом Twilio виконайте такі кроки:

- перейти на офіційний сайт https://www.twilio.com/en-us, увійти до облікового запису або зареєструватися;
- на головній сторінці Dashboard скопіювати значення: TWILIO_ACCOUNT_SID – ідентифікатор облікового запису (Account SID), TWILIO_AUTH_TOKEN – токен авторизації (Auth Token);
- у розділі Phone Numbers обрати або зареєструвати номер, з якого будуть надсилатися повідомлення:
  TWILIO_SENDER_PHONE – номер телефону від Twilio.

Щоб отримати ключі для інтеграції з платіжною системою Stripe:

- Перейти до Stripe Dashboard (https://dashboard.stripe.com/login) та виконати вхід;
- Відкрити розділ Developers API keys і скопіювати STRIPE_PUBLIC_KEY (відкритий ключ) і STRIPE_SECRET_KEY (секретний ключ).

Рекомендується використовувати тестові ключі для середовища розробки.

Щоб налаштувати надсилання електронної пошти через SMTP-сервер Gmail:

- увімкнути двоетапну автентифікацію в обліковому записі Google;
- перейти на сторінку Google App Passwords;
- створити новий App Password для застосунку типу "Mail".
    - MAIL_USERNAME – адреса електронної пошти, з якої буде здійснюватися надсилання;

- MAIL_PASSWORD – згенерований App Password;
- MAIL_DEFAULT_SENDER – адреса відправника (зазвичай збігається з MAIL_USERNAME).

Щоб інтегрувати результати запусків АТ з TestRails, потрібно виконати такі дії:

– зареєструвати новий обліковий запис у системі або використати вже наявний;
– отримати хост-посилання на пробний період (або скористатися наявним), після чого здійснити автентифікацію в системі;
– створити новий проєкт із відповідними налаштуваннями (див. рисунок 4.2);
– згенерувати API-ключ у налаштуваннях облікового запису та зберегти його;
– у налаштуваннях сайту активувати підтримку API;
– в панелі адміністратора проєкта, створити нову змінну проєкту automation_id (див. рисунок 4.3[24]);
– додати у файл .env змінні з відповідними значеннями, а саме: TESTRAIL_URL, TESTRAIL_USERNAME, TESTRAIL_API_KEY, TESTRAIL_PROJECT_ID, TESTRAIL_SUITE_ID, TESTRAIL_MILESTONE_ID.

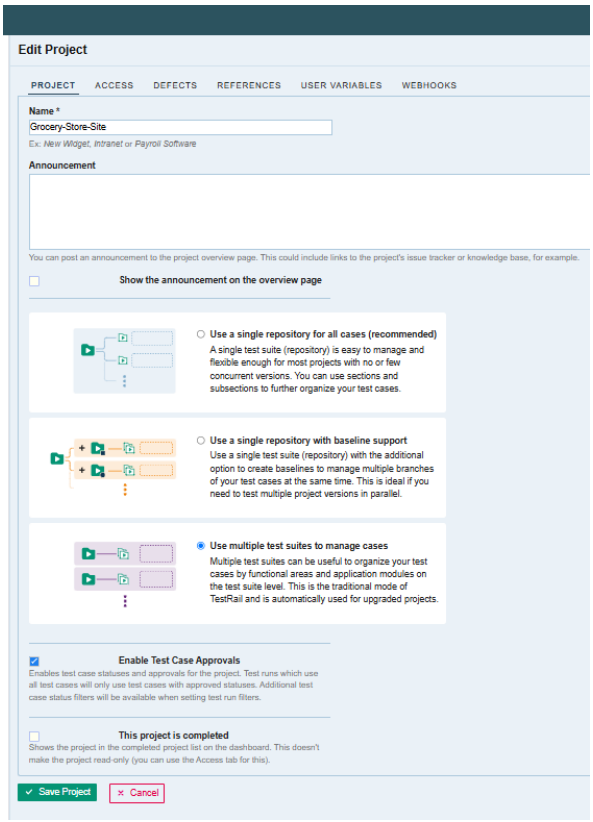

Рисунок 4.2 – налаштування нового проєкту TestRails

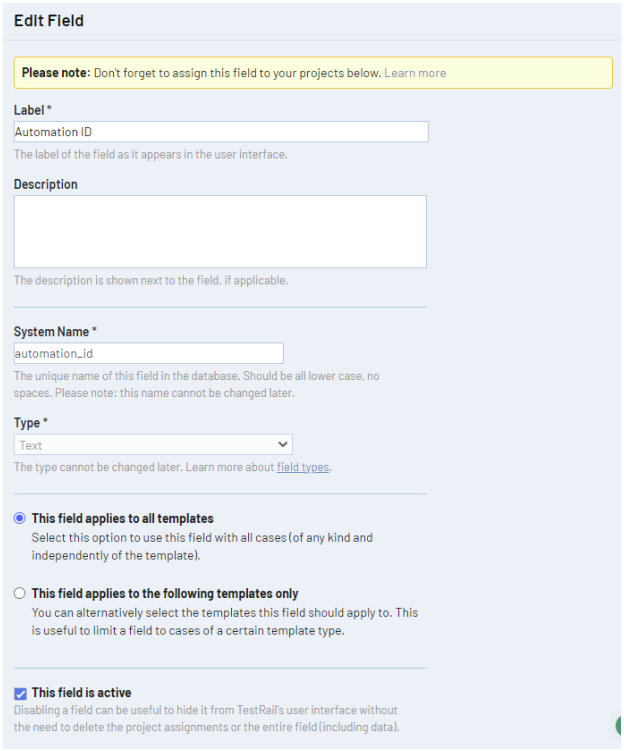

Рисунок 4.3 – створення змінної automation_id

Проєкт у TestRail підготовлено для створення тест-кейсів, запуску тестів та збереження пов’язаної з ними інформації.

Наступним етапом є налаштування інтеграції з ReportPortal. Спочатку необхідно переконатися, Docker встановлено і запущено. За допомогою bash-команди curl -LO https://raw.githubusercontent.com/reportportal/reportportal/master/docker-compose.yml або вручну з офіційного репозиторію https://github.com/reportportal/reportportal/blob/master/docker-compose.yml, потрібно завантажитти Docker Compose файл для розгортання додатку. Наступним кроком є безпосередньо запуск контейнерів за допомогою команди docker-compose -p reportportal up -d --force-recreate.

Необхідно перевірити що всі контейнери, потрібні для коректного розгортання ReportPortal, перебувають у стані виконання. Після цього можна перейти до сторінки автентифікації, яка за стандартних параметрів доступна за адресою: http://localhost:8080/ui/#login. Для доступу до системи використовується обліковий запис адміністратора зі стандартними обліковими даними: логін – superadmin, пароль – erebus[23].

Після успішного входу в систему, необхідно згенерувати API-токен, перейшовши на вкладку профіля, потім потрібно натиснути секцію API-ключі і створити новий ключ відповідною кнопкою. Після надання ключеві імені, потрібно зберегти токен локально, також додати в .env файл змінні REPORTPORTAL_ENDPOINT, REPORTPORTAL_PROJECT (за замовуванням superadmin_personal), REPORTPORTAL_TOKEN, генерацію якого було описано вище. Після цього, система буде готова для інтеграції.

Таким чином, .env файл має виглядати таким чином (див. рисунок 4.4):

env-example

```
# ═══ TestRail Configuration ═══
TESTRAIL_URL=
TESTRAIL_USERNAME=
TESTRAIL_API_KEY=
TESTRAIL_PROJECT_ID=
TESTRAIL_SUITE_ID=
TESTRAIL_MILESTONE_ID=

# ═══ ReportPortal Configuration ═══
REPORTPORTAL_ENDPOINT=
REPORTPORTAL_PROJECT=
REPORTPORTAL_TOKEN=

# ═══ Target Application ═══
TARGET_APP_URL=
TARGET_APP_JWT_SECRET=

# ═══ Application Logging ═══
DEBUG=false
LOG_LEVEL=INFO

# ═══ Twilio Integration ═══
TWILIO_ACCOUNT_SID=
TWILIO_AUTH_TOKEN=
TWILIO_SENDER_PHONE=

# ═══ Stripe Integration ═══
STRIPE_PUBLIC_KEY=
STRIPE_SECRET_KEY=

# ═══ Email (SMTP) Settings ═══
MAIL_USERNAME=
MAIL_PASSWORD=
MAIL_DEFAULT_SENDER=
```

Рисунок 4.4 – шаблон .env-файлу для налаштування проекту

Після завершення всіх налаштувань конфігураційних параметрів, можна переходити до запуску основного додатку через Docker Compose, який розгортає комплексну екосистему тестування: цільовий веб-додаток з Redis базою даних, повноцінну інфраструктуру ReportPortal (PostgreSQL, RabbitMQ, OpenSearch, Traefik Gateway), систему автоматизованого тестування(див. рисунок 4.5).

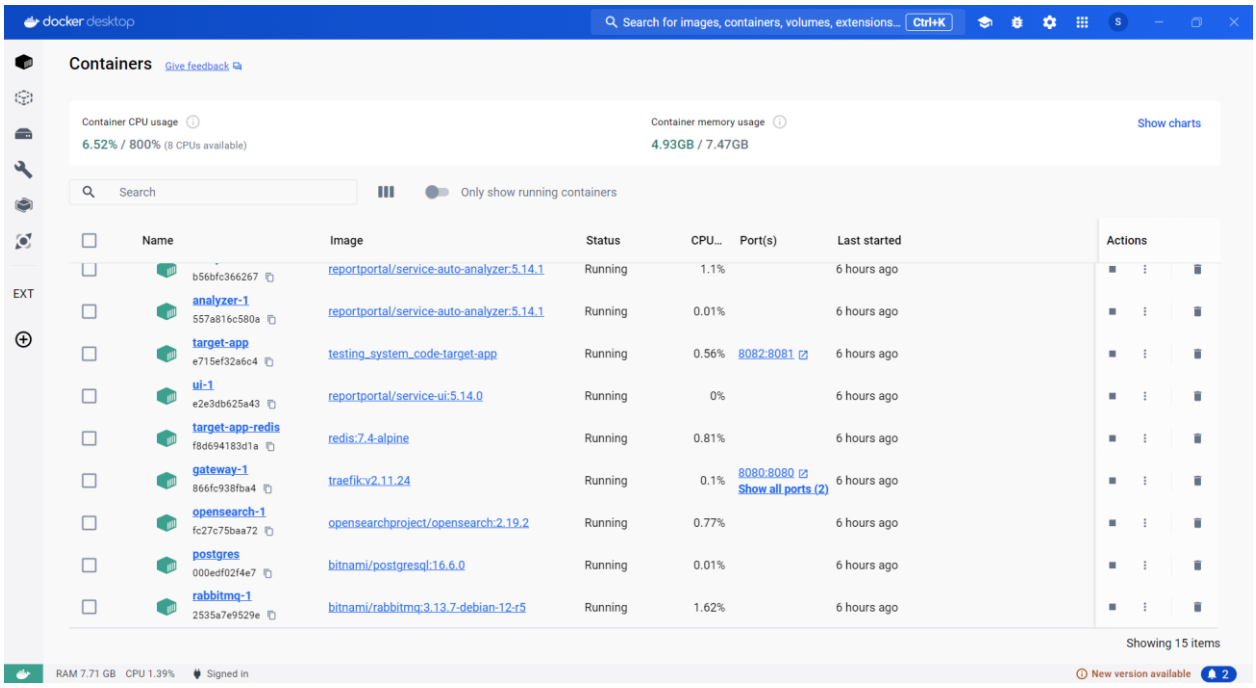

Рисунок 4.5 – запущені Dokcer-контейнери

Після успішного запуску користувачі з’явився доступ до цільового додатку за адресою http://localhost:8082, ReportPortal UI за адресою http://localhost:8080/ui. Це продемонстровано на рисунках 4.6 і 4.7.

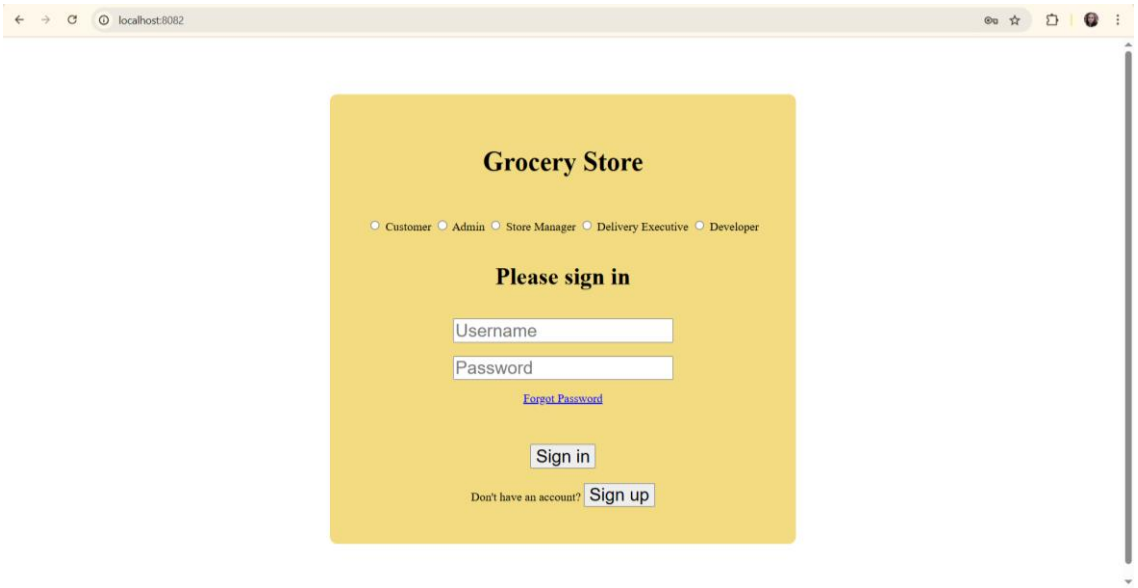

Рисунок 4.6 – розгорнутий цільовий веб-додаток

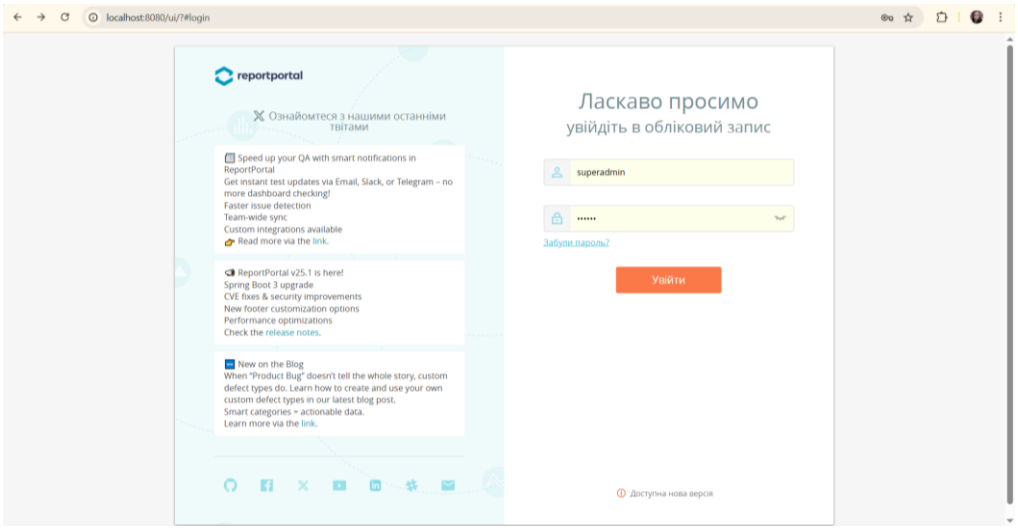

Рисунок 4.7 – розгорнутий ReportPortal

4.4 Розроблення і запуск тестів

Розглянемо практичну реалізацію тесту для рекомендаційної системи, що відповідають пункту 3 з таблиці 4.1. Реалізація даного тесту представлена на рисунку 4.8.

```python
@framework_test_case(priority='medium')
def test_similar_products_by_category(
    self, data_generator, ml_recommendation_service
):
    """
    Тест пошуку схожих товарів за категорією

    Перевіряє основну логіку схожості
    """
    # Генеруємо товари з різними категоріями
    products = data_generator.generate_bulk_by_template(
        "target_app_product", count=10, unique_fields=["product_id"]
    )
    categories = data_generator.generate_bulk_by_template(
        "target_app_category", count=3, unique_fields=["category_id"]
    )

    # Призначаємо категорії товарам
    for i, product in enumerate(products):
        product["category_id"] = (i % 3) + 1  # Категорії 1, 2, 3

    # Вибираємо цільовий товар
    target_product = products[0]  # Категорія 1

    # Знаходимо схожі товари
    similar = ml_recommendation_service.find_similar_products(
        target_product, products, categories, N=5
    )

    # Перевіряємо результат
    assert isinstance(similar, list)
    assert len(similar) ≤ 5  # Максимум 5 товарів

    # Всі схожі товари повинні мати ту ж категорію
    for similar_product in similar:
        assert similar_product["category_id"] == target_product["category_id"]
        assert similar_product["product_id"] ≠ target_product["product_id"]
```

Рисунок 4.8 – програмна реалізація тесту логіки схожості товарів

Тест реалізує перевірку логіки пошуку схожих товарів за категоріями у рекомендаційній системі. Підготовка тестових даних здійснюється через data_generator, який створює 10 унікальних товарів та 3 категорії. Кожному товару автоматично призначається категорія за формулою (i % 3) + 1, що забезпечує рівномірний розподіл товарів між категоріями 1, 2 та 3.

Основна логіка тесту полягає у виборі цільового товару з першої категорії та пошуку для нього схожих товарів через метод find_similar_products(). Параметр N=5 обмежує кількість рекомендацій максимум п'ятьма товарами, що відповідає реальним вимогам користувацького інтерфейсу.

Система перевірок включає валідацію типу результату (список), контроль кількості рекомендацій (не більше 5), та ключову бізнес-логіку: всі рекомендовані товари повинні належати до тієї ж категорії, що й цільовий товар, але мати відмінні ідентифікатори. Це забезпечує коректність алгоритму групування за схожістю.

За допомогою команди, виконаної із кореневої директорії проєкту, python -m src.presentation.cli –type unit –reportportal –testrail, ми виконуємо запуск усіх модульних тестів, реалізованих в директорії tests/unit, одразу синхронізуємо результат із налаштованими в розділі 4.3 проєктами в системах тестування. Рисунок 4.9, показують успішне виконання усіх unit тестів проєкту.

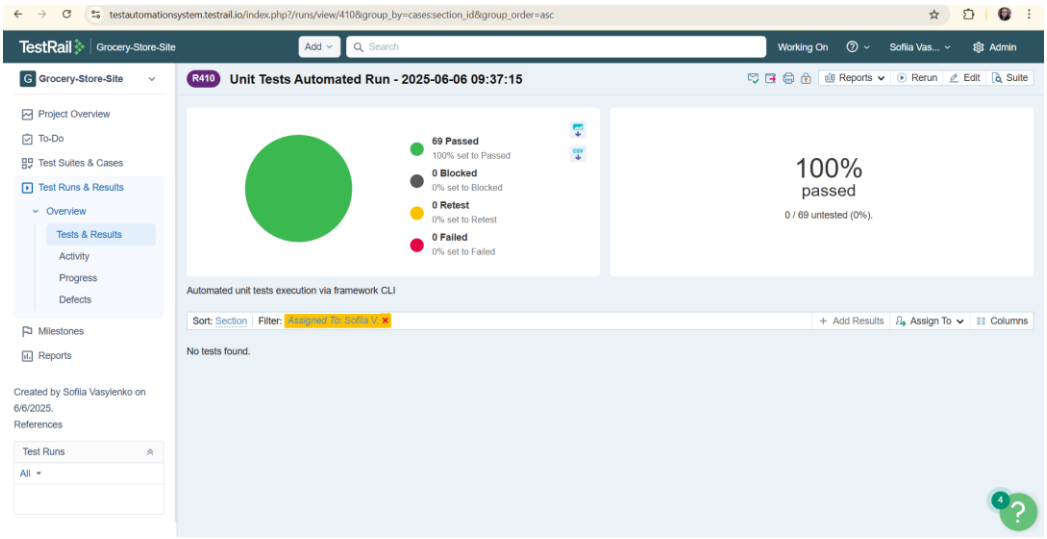

Рисунок 4.9 – успішний запуск набору unit тестів в TestRail

Автоматично створюються відповідні тест кейси(див. рисунок 4.10), для відстеження статистик успішного і провального виконання, поділені на категорії (Unit, Api, Ui, Integration), відстеження логів, для ui тестування синхронізуються прикріплені файли (наприклад, скріншот моменту на якому впав тест).

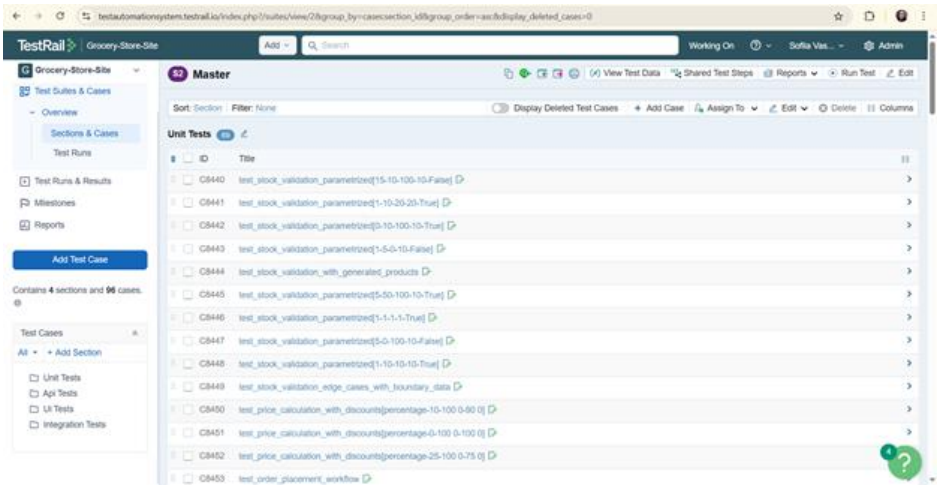

Рисунок 4.10 – згенеровані тест кейси

Статистика і інформація про реалізований раніше тест продемонстрована на рисунку 4.11.

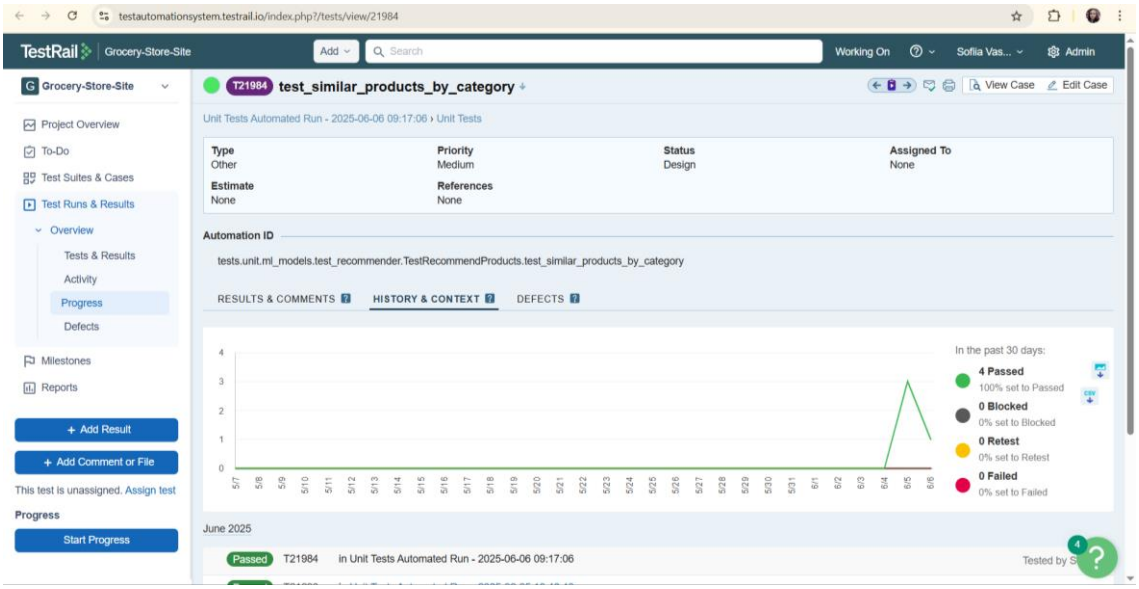

Рисунок 4.11 – інформація і статистика описаного тест кейсу в TestRail

4.5 Аналіз результатів тестування

Можливості систем TestRail та ReportPortal надають інструменти для зручного аналізу та візуалізації результатів тестування. Вони дозволяють відстежувати статус проходження тест-кейсів, виявляти нестабільні тести, аналізувати причини збоїв і оцінювати покриття вимог. Інтегровані звіти та графіки допомагають швидко отримати уявлення про загальний стан якості продукту. У рамках демонстрації наведено приклади запусків тестів і звітів – у реальних умовах, при значно більшій кількості тестів, аналітична цінність таких звітів зростає ще більше.

На рисунку 4.12 представлено візуалізацію запусків тестувань у TestRail. У ній показано окремі запуски, назву запуску (в якій зазначено рівень тестування), автора запуску, а також загальну успішність виконання у відсотках. Такий огляд дає змогу швидко оцінити стабільність тестової сесії та виявити потенційні проблеми.

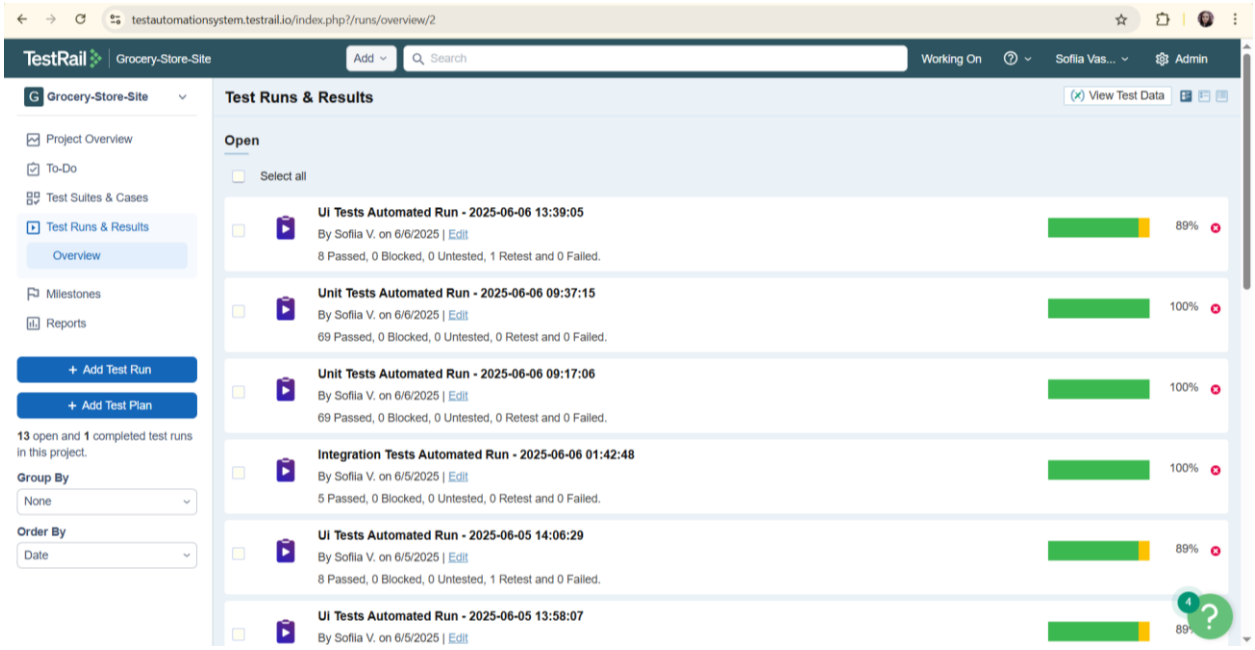

Рисунок 4.12 – візуалізація запусків тестувань у TestRail

Рисунок 4.13 демонструє згенерований звіт у TestRail із підсумковою статистикою успішності тестів за день. Діаграма дає змогу швидко оцінити співвідношення пройдених, повторно виконаних та провалених тестів. Це спрощує аналіз загального стану релізу або конкретного тестового циклу.

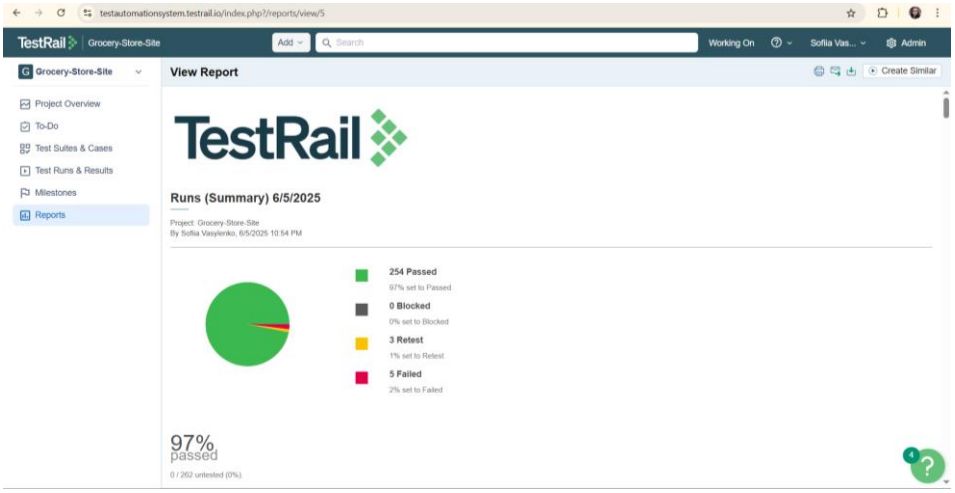

Рисунок 4.13 –згенерований у TestRail звіт про статистику успішності тестів

На рисунку 4.14 зображено список запусків у ReportPortal. Система дозволяє деталізовано переглядати інформацію про кількість тестів, їх результати, а також можливі причини збоїв – наприклад, системні помилки, дефекти продукту або автотестів. Зручна таблиця забезпечує швидку навігацію та виявлення пріоритетних проблем для розслідування.

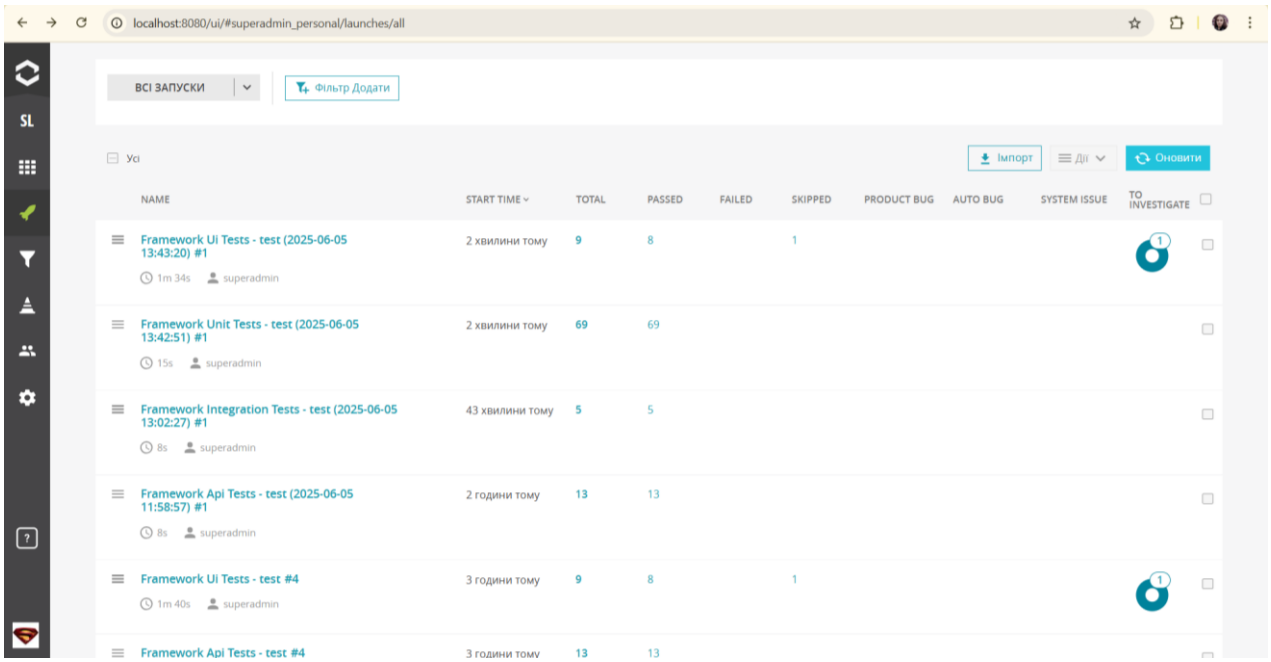

Рисунок 4.14 – запуски тестувань у ReportPortal

Рисунок 4.15 демонструє приклад налаштованого дашборду в ReportPortal. Він дозволяє візуалізувати ключові метрики: стабільність тестів, частоту збоїв, історію запусків тощо. Такий дашборд значно спрощує моніторинг якості в динаміці та підходить як для QA-команд, так і для менеджменту.

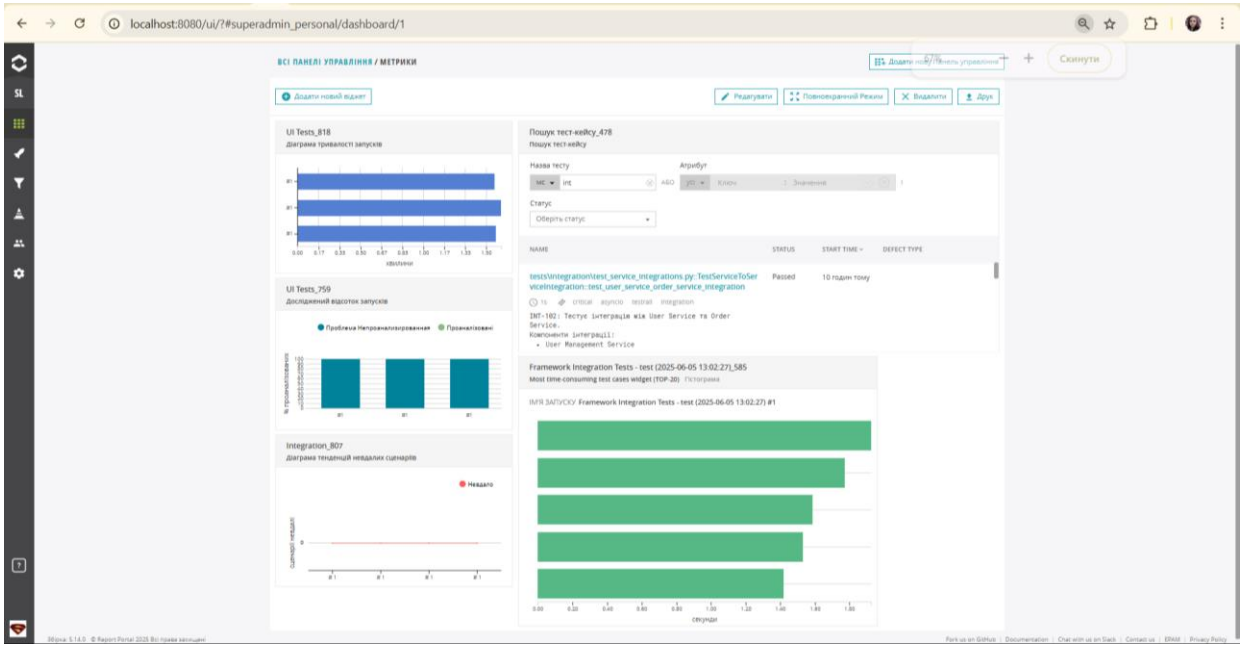

Рисунок 4.15 – приклад створеного інтерактивного дашборду ReportPortal

ВИСНОВКИ

У процесі виконання дипломної роботи було проведено комплексне дослідження, спрямоване на розробку комбінованого засобу автоматизованого тестування для веб-додатків. Цей засіб забезпечує повне покриття піраміди тестування, включаючи модульні, інтеграційні, API та UI-тести, а також інтеграцію з системами TMS і процесами безперервної інтеграції та доставки (CI/CD). Робота охоплювала аналіз сучасних підходів, проєктування архітектури інструменту, його реалізацію мовою програмування Python, інтеграцію з TMS і платформами аналізу результатів, а також валідацію на реальному проєкті з відкритим вихідним кодом.

На першому етапі було проведено аналіз сучасних інструментів і методологій автоматизованого тестування, таких як Selenium, Pytest, Playwright, а також підходів, що застосовуються в Agile та DevOps. Виявлено, що більшість наявних інструментів є вузькоспеціалізованими, що ускладнює їх інтеграцію для повноцінного покриття всіх рівнів тестування. Це створює проблему для стартапів і невеликих команд, які мають обмежені ресурси, але потребують швидкого випуску якісних релізів. Аналіз підтвердив актуальність створення універсального засобу, який би об’єднував усі етапи тестування в єдину систему, був економічно вигідним і легко адаптувався до різних проєктів.

На основі отриманих даних було спроєктовано архітектуру інструменту, яка враховує функціональні та нефункціональні вимоги. Функціональні вимоги включали підтримку модульних, API, інтеграційних і UI-тестів, а також інтеграцію з TMS, наприклад ReportPortal і TestRail. Нефункціональні вимоги стосувалися масштабованості, продуктивності та легкості налаштування. Архітектура була розроблена модульною, що забезпечує гнучкість і можливість розширення інструменту в майбутньому.

Реалізація прототипу здійснювалася мовою Python із використанням бібліотек Pytest для модульних і API-тестів, Playwright для UI-тестів і Faker

для генерації тестових даних. Інтеграція з ReportPortal дозволила організувати централізоване управління тестами та аналіз результатів, а використання Docker Compose забезпечило легке розгортання тестової інфраструктури. Для валідації інструменту було використано проєкт із відкритим вихідним кодом Grocery-Store-Site, що дозволило оцінити ефективність інструменту.

Інтеграція з TMS і системою аналізу і візуалізації ReportPoral спростила процеси моніторингу та аналізу, а модульна структура інструменту дозволила легко адаптувати його до потреб конкретного проєкту.

Таким чином, розроблений засіб є ефективним рішенням для команд із обмеженими ресурсами, що прагнуть забезпечити високу якість веб-додатків за умов швидких циклів розробки. Робота підтвердила гіпотезу про можливість створення універсального інструменту, який об’єднує всі рівні тестування та інтегрується з сучасними процесами розробки.

Для подальшого вдосконалення комбінованого засобу автоматизованого тестування веб-додатків, пропонується зосередитися на таких напрямах: розширення підтримки нових типів тестів, зокрема тестування безпеки та продуктивності, для забезпечення ширшого покриття потреб проєктів; інтеграція з додатковими CI/CD-платформами, такими як Jenkins або GitLab CI, для підвищення гнучкості розгортання; вдосконалення генерації тестових даних шляхом використання бібліотек, подібних до Mimesis, для створення реалістичніших сценаріїв; оптимізація продуктивності інструменту через паралельне виконання тестів і покращення алгоритмів обробки результатів; а також розробка плагінів для інтеграції з новими TMS, що забезпечить легшу адаптацію до різних проєктних екосистем.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Vocke H. The Practical Test Pyramid. *Martinfowler.com*. *articles*. 26.02.2018. URL: https://martinfowler.com/articles/practical-test-pyramid.html (дата звернення: 29.04.2025).
2. Jetbrains. Python Developers Survey 2023 Results. *Lp.jetbrains.com*. URL: https://lp.jetbrains.com/python-developers-survey-2023/ (дата звернення: 30.04.2025).
3. Shftrs. The Hidden Benefits of Test Automation: Uncovering Cost Savings and Boosting ROI. *shftrs.com*. *articles*. 10.04.2023. URL: https://shftrs.com/articles/the-hidden-benefits-of-test-automation-uncovering-cost-savings-and-boosting-roi (дата звернення: 29.04.2025).
4. Cohn M. Succeeding with Agile: Software Development Using Scrum. Boston : Addison-Wesley, 2009. 512 с. ISBN 978-0321579362.
5. Cohn M. The Forgotten Layer of the Test Automation Pyramid. *www.mountaingoatsoftware.com*. *blog*. URL: https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid (дата звернення: 30.04.2025).
6. Patel P. Testing Pyramid: A Successful Approach to End to End testing.. *Alphabin.co*. *blog*. 16.08.2024. URL: https://www.alphabin.co/blog/testing-pyramid (дата звернення: 30.04.2025).
7. Sambamurthy M. Test Automation Engineering Handbook: Learn and implement techniques for building robust test automation frameworks. Packt Publishing Ltd, 2023. 276 с. ISBN 9781804615492.
8. Jain J. Learn API Testing: Norms, Practices, and Guidelines for Building Effective Test Automation. Apress, 2022. 248 с. ISBN 978-1484281413.
9. Homes B. Fundamentals of Software Testing 2nd Edition, Revised and Updated. Wiley-ISTE, 2024. 400 с. ISBN 978-1786309822.

10. Dodds K. C. The Testing Trophy and Testing Classifications. *Kentcdodds.com*. *blog*. 03.06.2021. URL: https://kentcdodds.com/blog/the-testing-trophy-and-testing-classifications (дата звернення: 30.04.2025).
11. HTTPX. *Python-httpx.org*. URL: https://www.python-httpx.org/ (дата звернення: 03.05.2025).
12. Postman, Inc. Postman: Getting Started. *Learning.postman.com*. *docs*. URL: https://learning.postman.com/docs/introduction/overview/ (дата звернення: 03.05.2025).
13. Testcontainers. Getting Started. *Testcontainers.com*. URL: https://testcontainers.com/getting-started/ (дата звернення: 03.05.2025).
14. García B. Exploring Browser Automation: A Comparative Study of Selenium, Cypress, Puppeteer, and Playwright. *Quality of Information and Communications Technology*, Pisa, Italy, 12 верес. 2024. Quality of Information and Communications Technology: 17th International Conference on the Quality of Information and Communications Technology : Springer, 2024. С. 142–149.
15. Uchakaev I. Mimesis 18.0.0. *Pypi.org*. *project*. URL: https://pypi.org/project/mimesis/ (дата звернення: 06.05.2025).
16. @joke2k. Faker 37.3.0. *pypi.org*. *project*. URL: https://pypi.org/project/Faker/. (дата звернення: 06.05.2025).
17. GitHub, Inc. GitHub Actions documentation. *Docs.github.com*. URL: https://docs.github.com/en/actions (дата звернення: 07.05.2025).
18. Red Hat, Inc. What is CI/CD? *redhat.com*. *topics/devops*. 12.12.2023. URL: https://www.redhat.com/en/topics/devops/what-is-ci-cd#ci/cd-vs-devops (дата звернення: 07.05.2025).
19. Kumar A. Top 5 JIRA integrations for software testing.. *disbug.io*. *blog*. 10.07.2024. URL: https://disbug.io/en/blog/top-5-jira-integrations-for-software-testing (дата звернення: 03.05.2025).

20. Thomas S. Report Portal vs Allure Report: A Comprehensive Comparison.. *Medium.com*. 08.07.2024. URL: https://medium.com/@sarah.thoma.456/reportportal-vs-allure-report-a-comprehensive-comparison-5eb0d153ce73 (дата звернення: 08.05.2025).
21. Test Guild LLC. Top 8 Automation Testing Trends Shaping 2025. *testguild.com*. 11.01.2025. URL: https://testguild.com/automation-testing-trends/ (дата звернення: 09.05.2025).
22. Moore C. Functional vs Non Functional Requirements. *Guru99.com*. 26.09.2024. URL: https://www.guru99.com/functional-vs-non-functional-requirements.html. (дата звернення: 10.05.2025).
23. ReportPortal. What is ReportPortal?. *Reportportal.io*. *docs*. URL: https://reportportal.io/docs/ (дата звернення: 22.05.2025).
24. Testrail. Code-first workflow. *Testrail.com*. *Support*. URL: https://support.testrail.com/hc/en-us/articles/12609674354068-Code-first-workflow (дата звернення: 23.05.2025).
25. Juneja A. Schefflera-Arboricola / Grocery-Store-Site. *Github.com*. */Schefflera-Arboricola/Grocery-Store-Site*. 13.05.2024. URL: https://github.com/Schefflera-Arboricola/Grocery-Store-Site (дата звернення: 26.05.2025).