Національний університет «Полтавська політехніка імені Юрія Кондратюка»
(повне найменування закладу вищої освіти)

Навчально-науковий інститут інформаційних технологій і робототехніки
(повне найменування інституту, назва факультету (відділення))

Кафедра автоматики, електроніки та телекомунікацій
(повна назва кафедри (предметної, циклової комісії))

Пояснювальна записка

до кваліфікаційної роботи

магістр
(рівень вищої освіти)

на тему **Моделювання та впровадження фреймворку автоматизованного тестування веб-застосунку, як інструменту підвищення якості інфокомунікаційної системи**

Виконав: студент 2 курсу, групи 601-ТТ
спеціальності 172 «Електронні комунікації та радіотехніка»
(шифр і назва напряму підготовки, спеціальності)

Куденко О.О.
(прізвище та ініціали)

Керівник Косенко В.В.
(прізвище та ініціали)

Рецензент Губанов В.В.
(прізвище та ініціали)

Полтава - 2026 рік

Національний університет «Полтавська політехніка імені Юрія Кондратюка»
Інститут Навчально-науковий інститут інформаційних технологій та робототехніки
Кафедра Автоматики, електроніки та телекомунікацій
Рівень вищої освіти Магістр
Спеціальність 172 «Електронні комунікації та радіотехніка»

ЗАТВЕРДЖУЮ

Завідувач кафедри автоматики, електроніки та телекомунікацій
_____ О.В. Шефер
“ 15 ” 09 2025 р.

З А В Д А Н Н Я

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Куденку Олексію Олександровичу

1. Тема проекту (роботи) **«Моделювання та впровадження фреймворку автоматизованного тестування веб-застосунку, як інструменту підвищення якості інфокомунікаційної системи»**
керівник проекту (роботи) Косенко В.В. д.т.н., професор
затверджена наказом вищого навчального закладу від 03.09.2025 року № 1025 - ф.а.
2. Строк подання студентом проекту (роботи) 22.12.2025 р.
3. Вихідні дані до проекту (роботи) Вихідними даними є: технічна документація на обраний для тестування веб-застосунок (відкритий проєкт у вигляді панелі адміністратора); вимоги до функціоналу, що підлягає автоматизації (End-to-End, UI/API); сучасні специфікації та стандарти автоматизованого тестування (наприклад REST API специфікації); документація інструментального стеку: Playwright, TypeScript (Node.js), системи звітності та системи безперервної інтеграції (Gitgub Actions).
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Вступ з обґрунтуванням актуальності теми. Аналіз фундаментальних засад тестування ПЗ та доцільності його автоматизації. Огляд та порівняння сучасних інструментів автоматизації. Моделювання архітектури фреймворку (Page Object Model, використання фікстур та API-хелперів). Розробка та реалізація тестового фреймворку на базі Playwright/TypeScript. Впровадження механізмів конфігурації (наприклад, playwright.config.ts) та тегування тестів за середовищами. Проведення

експериментального дослідження ефективності впровадженого фреймво
(покриття, швидкість, звітність). Висновки по роботі.

5. Дата видачі завдання 01.10.2025 р.

КАЛЕНДАРНИЙ ПЛАН

Пор. №	Назва етапів магістерської роботи	Термін виконання етапів роботи			Примітка (плакати)
1.	Обгрунтування актуальності, аналіз ІКС, класифікація тестування (Manual vs. Automated).	06.10.2025	I	15%	Пл. 2 - 5
2.	Огляд фреймворків (Playwright), вибір стеку (Playwright/TS), критерії автоматизації.	31.10.2025	I	30%	Пл. 6
3.	Розробка моделі ФАТ, визначення структури проєкту (POM), архітектура UI+API інтеграції.	07.11.2025	II	45%	Пл. 7 - 8
4.	Розробка структури конфігу, моделювання механізму фікстур (Fixtures) та тегування тестових наборів.	14.11.2025	II	60%	Пл. 9 - 10
5.	Розробка та автоматизація ключових End-to-End (UI) та API сценаріїв, інтеграція тегів середовищ.	03.12.2025	III	85%	Пл. 12 - 18
6.	Налаштування CI/CD (Git), впровадження системи звітності, збір метрик для експерименту.	10.12.2025	III	95%	Пл. 19 - 21
7.	Висновки. Проведення порівняльного аналізу, оцінка ефективності, фінальне оформлення роботи та додатків.	22.12.2025	III	100%	Пл. 22

Магістрант _____ Куденко О.О.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Косенко В.В
(підпис) (прізвище та ініціали

РЕФЕРАТ

кваліфікаційної роботи магістра
“Моделювання та впровадження фреймворку автоматизованого тестування веб-застосунку як інструменту підвищення якості інфокомунікаційної системи”

Робота містить 116 с., 52 рисунка, 4 додатки, 20 джерел.

Ключові слова: автоматизоване тестування, Playwright, TypeScript, QA, UI-тестування, API-тестування, End-to-End тести, Page Object Model, Fixtures, CI/CD, GitHub Actions, тестова архітектура, інфокомунікаційні системи, веб-застосунок.

Предметом дослідження кваліфікаційної роботи є кваліфікаційної роботи є процес забезпечення якості веб-застосунків у складі інфокомунікаційних систем із використанням автоматизованого тестування.

Метою роботи є моделювання та впровадження фреймворку автоматизованого тестування на основі бібліотеки Playwright і мови TypeScript, який забезпечує інтегрований підхід до UI та API тестування, підвищує швидкість виконання тестових сценаріїв, стабільність регресійного тестування та якість контролю програмного забезпечення.

У процесі виконання роботи проведено аналіз фундаментальних засад тестування програмного забезпечення, сучасних підходів до автоматизації тестування та існуючих інструментів. Обґрунтовано доцільність вибору технологічного стеку Playwright + TypeScript. Спроєктовано архітектуру фреймворку автоматизованого тестування з використанням патерну Page Object Model та механізму Playwright Fixtures, що забезпечує гібридний підхід до тестування UI та API.

Реалізовано ключові End-to-End та API тестові сценарії, налаштовано глобальну конфігурацію фреймворку, механізми тегування тестів і підтримку різних тестових середовищ. Виконано інтеграцію фреймворку з системою безперервної інтеграції CI/CD на базі GitHub Actions та впроваджено систему звітності для збору й аналізу метрик тестування.

Практичне значення отриманих результатів полягає у створенні масштабованого та універсального фреймворку автоматизованого тестування, який може бути використаний для підвищення ефективності процесів контролю якості веб-застосунків у реальних проєктах.

ABSTRACT

of the Master's Qualification Thesis
"Modeling and Implementation of an Automated Testing Framework for a Web Application as a Tool for Improving the Quality of an Infocommunication System"

Current work contains 116 pages, 52 images, 12 tables, and 20 references.

Keywords: automated testing, Playwright, TypeScript, quality assurance, UI testing, API testing, End-to-End tests, Page Object Model, Fixtures, CI/CD, GitHub Actions, test architecture, infocommunication systems, web application.

The subject of the research is the process of quality assurance of web applications within infocommunication systems using automated testing approaches.

The aim of the thesis is to model and implement an automated testing framework based on the Playwright library and the TypeScript programming language, which provides an integrated approach to UI and API testing, increases test execution speed, improves regression testing stability, and enhances the overall quality control of software products.

During the research, a comprehensive analysis of fundamental principles of software testing, modern approaches to test automation, and existing testing tools was conducted. The feasibility of selecting the Playwright + TypeScript technology stack was substantiated. The architecture of an automated testing framework was designed using the Page Object Model pattern and the Playwright Fixtures mechanism, enabling a hybrid UI and API testing approach.

Key End-to-End and API test scenarios were implemented, along with global framework configuration, test tagging mechanisms, and support for multiple testing environments. The framework was integrated into a Continuous Integration and Continuous Delivery (CI/CD) pipeline based on GitHub Actions, and a reporting system was implemented to collect and analyze testing metrics.

The practical significance of the obtained results lies in the development of a scalable and reusable automated testing framework that can be applied to improve the efficiency and reliability of quality assurance processes for web applications in real-world projects.

ЗМІСТ

ВСТУП **8**

РОЗДІЛ 1. ФУНДАМЕНТАЛЬНІ ЗАСАДИ ТЕСТУВАННЯ ТА АНАЛІЗ ІНСТРУМЕНТАРІЮ **12**

1.1 Веб-застосунки (ВЗ) та інфокомунікаційні системи (ІКС): визначення, архітектура та їхня роль у сучасному бізнесі 12

1.2 Тестування ПЗ як інструмент підвищення якості: визначення якості та проблеми, які вирішує тестування 14

1.3 Класифікація та порівняння видів тестування: ручне (Manual) vs. автоматизоване (Automated) тестування. 19

1.4 Обґрунтування доцільності застосування автоматизованого тестування: критерії вибору та сценарії, коли автоматизація є недоречною. 23

1.5 Інструменти автоматизації тестування, історичний огляд та порівняння сучасних ФАТ. 26

1.6 Обґрунтування вибору Playwright та TypeScript/Node.js як ключових технологій 38

Висновки до розділу І 46

РОЗДІЛ 2. МОДЕЛЮВАННЯ АРХІТЕКТУРИ ТА СТРУКТУРИ ФРЕЙМВОРКУ **48**

2.1 Модель фреймворку автоматизованого тестування (ФАТ): функціональні вимоги та архітектурні принципи 48

2.2 Проєктування структури ФАТ: застосування патерну Page Object Model (POM) та організація файлів проєкту 55

2.3 Керування конфігураціями середовищ: детальний опис структури файлу конфігу та налаштування проєктів 59

2.4 Механізми підготовки тестового середовища: фікстури (Fixtures) та їхня роль у налаштуванні стану тестів 63

2.5 Моделювання інтегрованого підходу UI+API: архітектура API-хелперів для ефективної підготовки тестових даних та авторизації 66

2.6 Аналіз цільового веб-застосунку: ключовий функціонал, що підлягає автоматизації, та визначення тестового покриття 70

Висновки до розділу ІІ 73

РОЗДІЛ 3. ВПРОВАДЖЕННЯ, АВТОМАТИЗАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ **75**

3.1 Реалізація базової функціональності ФАТ з використанням Playwright та TypeScript (конфігурація, керування браузерами) 75

3.2 Реалізація модулів API-тестування: перевірка коректності обробки даних та бізнес-логіки 89

3.3 Впровадження механізму тегування тестів для фільтрації за середовищами 98

3.4 Інтеграція ФАТ із системою контролю версій (Git) та конвеєром безперервної інтеграції (CI/CD) 102

3.5 Налаштування та застосування системи звітності: візуалізація результатів, збір метрик та аналіз логів 105

ВИСНОВКИ **112**

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ **116**

ДОДАТКИ **118**

Додаток А 118

Додаток Б 129

ПЕРЕЛІК ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

АПІ (API) - Інтерфейс програмування застосунків (Application Programming Interface)

ФАТ - Фреймворк автоматизованого тестування

ІКС - Інфокомунікаційна система

ПЗ (SW) - Програмне забезпечення (Software)

ПМ (PM) - Проєктний менеджмент (Project Management)

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

QA- Забезпечення якості (Quality Assurance)

UI (ЮАЙ) - Користувацький інтерфейс (User Interface)

E2E - Наскрізне тестування (End-to-End)

POM - Модель об'єктів сторінки (Page Object Model)

CI - Безперервна інтеграція (Continuous Integration)

CD - Безперервна доставка/розгортання (Continuous Delivery/Deployment)

Git - Система контролю версій

REST - Передача репрезентативного стану (Representational State Transfer)

JSON - Нотація об'єктів JavaScript (JavaScript Object Notation)

TLS - Безпека транспортного рівня (Transport Layer Security)

HTTP - Протокол передачі гіпертексту (HyperText Transfer Protocol)

URI - Уніфікований ідентифікатор ресурсу (Uniform Resource Identifier)

TS - Мова програмування TypeScript

JS - Мова програмування JavaScript

Node.js - Середовище виконання JavaScript

HTML - Мова гіпертекстової розмітки (HyperText Markup Language)

CSS - Каскадні таблиці стилів (Cascading Style Sheets)

URL - адресаУніфікований локатор ресурсу

ВСТУП

Епоха глобальної діджиталізації та розбудови інфокомунікаційних систем (ІКС) характеризується експоненційним зростанням складності та взаємозалежності програмних рішень. Сучасні веб-застосунки (ВА) є ключовими вузлами ІКС, забезпечуючи критично важливі бізнес-процеси: від електронної комерції та фінансових транзакцій до управління життєво важливими інфраструктурами. У таких умовах, питання якості, надійності та стійкості програмного забезпечення набуває не просто бажаного, а критично необхідного статусу. Дефекти, що потрапили у виробниче середовище (production), призводять до прямих фінансових збитків, підриву довіри користувачів та, у випадку ІКС, можуть мати серйозні системні наслідки.

Тестування програмного забезпечення (ПЗ) є основним інструментом у системі управління якістю (Quality Management System, QMS), завданням якого є виявлення дефектів та мінімізація ризиків, пов'язаних з експлуатацією ПЗ. Історично тестування розпочиналося як ручний (мануальний) процес, який ґрунтувався на експертизі та досвіді тестувальника. Проте, прискорення циклів розробки в рамках методологій Agile та DevOps, а також необхідність багаторазових, швидких перевірок існуючого функціоналу (регресійне тестування) зробили цей підхід неефективним. Ручне регресійне тестування є надто трудомістким, повільним та схильним до людських помилок.

Відповіддю на ці виклики стало автоматизоване тестування (АТ). Створення надійної, масштабованої бази автотестів дозволяє перевіряти величезні обсяги функціоналу за хвилини, забезпечуючи швидкий зворотний зв'язок для команди розробки і підтримуючи принципи безперервної інтеграції та доставки (CI/CD)[16].

Особлива складність сучасних ВА полягає у їхній архітектурі, що базується на клієнт-серверній взаємодії через API. Це вимагає комплексного підходу до АТ, який не обмежується лише інтерфейсом користувача (UI), а включає пряму взаємодію з API (тестування бізнес-логіки та даних). Хоча API-тестування є швидшим та стабільнішим за UI-тестування, воно часто

недостатнє для верифікації кінцевого досвіду користувача. Таким чином, найбільш ефективним є гібридний (інтегрований) підхід, де для прискореної підготовки тестового середовища (авторизація, створення об'єктів) використовуються швидкі API-запити, а кінцева перевірка результату відбувається через UI[19].

Кваліфікаційна робота присвячена моделюванню та практичному впровадженню такого інтегрованого ФАТ із використанням сучасного та високопродуктивного інструментального стеку — Playwright та TypeScript/Node.js. Це дозволить створити інструмент, здатний кардинально підвищити швидкість та якість контролю ПЗ, що є критично важливим для забезпечення високої якості інфокомунікаційної системи.

Об'єкт досліджень: Процес забезпечення якості (QA) веб-застосунку, що є частиною інфокомунікаційної системи.

Предмет досліджень: Методи, моделі та інструментальні засоби побудови фреймворку автоматизованого тестування (ФАТ) на базі Playwright для підвищення ефективності контролю якості ПЗ.

Метою кваліфікаційної роботи є моделювання, проєктування та практичне впровадження високоефективного фреймворку автоматизованого тестування (ФАТ) на базі Playwright та TypeScript, а також експериментальне дослідження його впливу на підвищення якості та ефективності процесу тестування веб-застосунку в контексті інфокомунікаційної системи.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

1. Провести аналіз фундаментальних засад тестування, сучасних підходів (ручне/автоматизоване тестування, UI/API), інструментів (Playwright, Cypress, Selenium) та обґрунтувати доцільність вибору інструментального стеку.
2. Розробити архітектурну модель фреймворку автоматизованого тестування (ФАТ), включно з вибором патернів проєктування (POM),

моделюванням механізмів підготовки тестового середовища (Fixtures, API-хелпери) та конфігурації (playwright.config.ts).

3. Реалізувати розроблений ФАТ, впровадивши ключові End-to-End (UI) та API-сценарії для цільового веб-застосунку та забезпечивши механізми інтеграції (CI/CD), тегування та звітності.
4. Провести експериментальне дослідження ефективності впровадження ФАТ та оцінити його вплив на якість тестування системи за показниками покриття, швидкості виконання та кількості виявлених дефектів.

В процесі виконання роботи були використані наступні методи:

1. Системний аналіз: для огляду сучасного стану проблеми, порівняння існуючих інструментів (Playwright, Cypress, Selenium) та обґрунтування вибору технологій.
2. Теоретичне моделювання: для проєктування архітектури ФАТ, вибору патернів проєктування (POM) та розробки інтегрованого підходу UI+API.
3. Емпіричний метод (експеримент): для практичної реалізації ФАТ, написання тестових сценаріїв та збору кількісних даних (часу виконання тестів, покриття).
4. Порівняльний аналіз: для оцінки ефективності розробленого ФАТ шляхом зіставлення отриманих метрик (швидкість, стабільність) із показниками ручного тестування.
5. Методи об'єктно-орієнтованого програмування (ООП): для створення гнучкої та розширюваної кодової бази фреймворку на базі TypeScript[6].

Наукова новизна полягає у наступному:

1. Удосконалено модель фреймворку автоматизованого тестування (ФАТ), що забезпечує інтегрований підхід тестування, де високоефективні API-хелпери використовуються для мінімізації часу на підготовку

тестового середовища (авторизація, створення даних) для End-to-End (UI) тестів, що підвищує загальну швидкість регресійних перевірок.

2. Набула подальшого розвитку методика конфігурування та керування виконанням автоматизованих тестових наборів шляхом впровадження механізмів Playwright Fixtures та гнучкого тегування (@dev, @stage, @prod), що забезпечує адаптивність ФАТ до різних середовищ розгортання в умовах CI/CD.

Практичне значення роботи полягає у наступному:

1. Розроблено та впроваджено готовий до використання ФАТ на базі Playwright/TypeScript, який може бути безпосередньо використаний для автоматизації тестування цільового веб-застосунку та будь-якої іншої ІКС із сучасною архітектурою.
2. Продемонстровано практичне скорочення часу виконання тестових циклів завдяки гібридному підходу UI+API, що дозволяє раніше виявляти дефекти та знижувати операційні витрати, пов'язані з якістю ПЗ.
3. Отримані результати можуть бути використані іншими розробниками та QA-фахівцями для міграції застарілих тестових рішень на сучасні платформи

РОЗДІЛ 1. ФУНДАМЕНТАЛЬНІ ЗАСАДИ ТЕСТУВАННЯ ТА АНАЛІЗ ІНСТРУМЕНТАРІЮ

1.1 Веб-застосунки (ВЗ) та інфокомунікаційні системи (ІКС): визначення, архітектура та їхня роль у сучасному бізнесі

Інфокомунікаційна система (ІКС) є архітектурно складним комплексом апаратного та програмного забезпечення, телекомунікаційних мереж та організаційних структур, призначеним для збору, обробки, зберігання та обміну інформацією. ІКС є ключовою основою для будь-якого сучасного підприємства, оскільки забезпечує інтеграцію інформаційних потоків та автоматизацію критично важливих бізнес-процесів.

У цьому контексті, Веб-застосунок (ВЗ) набуває статусу головного засобу взаємодії та реалізації функціоналу ІКС. ВЗ - це програмне забезпечення, що функціонує на віддаленому сервері та доступне користувачеві через стандартний веб-браузер та протокол HTTP. Це визначає ВЗ як єдиний, універсальний та кросплатформний інтерфейс для доступу до функцій усієї ІКС[17].

Внаслідок своєї доступності та стандартизації, веб-інтерфейси стали де-факто стандартом доступу та управління для систем будь-якої складності, замінивши переважну більшість спеціалізованих клієнтських програм. Ця універсальність охоплює практично кожен критично важливий бізнес-домен:

1. Фінансовий та Страховий Домени: Управління клієнтськими портфелями, оформлення полісів, моніторинг інвестиційних операцій, інтернет-банкінг - все це реалізується через складні, багатофункціональні веб-портали, де помилка може коштувати мільйони.
2. Телекомунікації та ІКС-інфраструктура: Керування мережевим обладнанням, моніторинг трафіку, білінгові системи, налаштування телекомунікаційних хабів та агрегаторів - часто здійснюється через

адміністративні веб-панелі. Надійність цих інтерфейсів є прямою передумовою безперервності надання послуг.

3. Промисловий та Енергетичний Сектор: Системи моніторингу та диспетчеризації (SCADA, HMI), які використовуються для контролю нафтогазових станцій, електростанцій або виробничих ліній, інтегруються у веб-середовище. Оператор взаємодіє зі складними даними та елементами керування через браузер, що вимагає абсолютної точності відображення та функціоналу.
4. Електронна комерція (E-commerce): Інтернет-магазини та маркетплейси є класичним прикладом ВЗ, де критично важлива швидкість, зручність та бездоганна робота наскрізних процесів (від реєстрації до оплати).

Більшість сучасних ВЗ побудовані за триланковою (3-tier) архітектурою, яка, в умовах мікросервісів, стає ще більш розподіленою[19[:

1. Ланка Презентації (Presentation Tier): Це інтерфейс користувача (UI) в браузері.
2. Ланка Логіки (Application Tier): Тут знаходиться основна бізнес-логіка, яка оперує даними та обробляє запити. Взаємодія з клієнтом відбувається через API (Application Programming Interface).
3. Ланка Даних (Data Tier): Сховище інформації (бази даних).

Така архітектура визначає необхідність комплексного підходу до забезпечення якості. Якість ВЗ - це не лише естетична привабливість інтерфейсу, а й коректна робота бізнес-логіки на сервері, що перевіряється через API.

Таким чином, роль ВЗ у сучасній ІКС є центральною. Будь-яка нестабільність, дефект або вразливість, що виникає в будь-якій із ланок цього застосунку, безпосередньо призводить до порушення функціонування всієї

інфокомунікаційної системи. Це підкреслює стратегічну важливість розробки високоякісних інструментів для автоматизованого тестування.

1.2 Тестування ПЗ як інструмент підвищення якості: визначення якості та проблеми, які вирішує тестування

У контексті інфокомунікаційних систем (ІКС) та веб-застосунків (ВЗ), якість програмного забезпечення (ПЗ) — це багатоаспектна категорія, яка відображає ступінь відповідності програмного продукту встановленим вимогам, потребам користувачів та галузевим стандартам. Якість не є бінарним параметром ("є" або "немає"), а являє собою сукупність вимірюваних характеристик.

Ключові атрибути якості ПЗ, згідно з міжнародними стандартами (наприклад, ISO/IEC 25010), включають:

1. Функціональна придатність (Functionality): Здатність ПЗ надавати функції, які відповідають заявленим та неявним потребам. Це включає коректність, доцільність, точність та функціональну повноту.
2. Надійність (Reliability): Здатність системи безвідмовно працювати в заданих умовах протягом визначеного часу. Цей атрибут особливо критичний для ІКС, що працюють у режимі 24/7 (телекомунікаційні хаби, моніторингові системи).
3. Ефективність (Performance Efficiency): Швидкість реагування, час обробки та використання ресурсів (пам'ять, процесор) під час виконання певних функцій. Низька ефективність прямо знижує якість обслуговування користувачів.
4. Працездатність (Usability): Легкість, з якою користувачі можуть взаємодіяти з ВЗ. Включає зрозумілість, інтуїтивність інтерфейсу та зручність навчання.
5. Безпека (Security): Здатність системи захищати дані та функціональні можливості від несанкціонованого доступу або використання.

Враховуючи роботу ІКС із конфіденційною інформацією, це є критично важливим.

6. Супроводжуваність (Maintainability): Легкість модифікації, тестування, аналізу та виправлення помилок у коді. Впливає на довгострокову вартість володіння системою.
7. Переносимість (Portability): Здатність ПЗ ефективно функціонувати в різних середовищах (різні браузери, операційні системи).

Якість ПЗ визначається як відповідність вимогам і придатність до використання. Тестування є проактивним та реактивним процесом, що дозволяє виміряти та підвищити ці характеристики.

Тестування програмного забезпечення - це процес виконання програми з метою виявлення відхилень від очікуваної поведінки, що називаються дефектами, помилками або багами[20].

Тестування виконує дві ключові функції, які гарантують якість:

1. Верифікація (Verification): Перевірка відповідності продукту технічним вимогам та стандартам (Чи правильно ми створюємо продукт?). Це часто відбувається на ранніх етапах розробки (наприклад, рев'ю коду, тестування архітектури).
2. Валідація (Validation): Перевірка відповідності продукту потребам кінцевого користувача (Чи створюємо ми правильний продукт?). Це типово для тестування функціональності та призначеного для користувача інтерфейсу.

Впровадження тестування у життєвий цикл розробки (SDLC), особливо в парадигмах Agile та DevOps, перетворює його на безперервний процес. Це забезпечує:

- Раннє виявлення дефектів (Shift-Left Testing): Чим раніше виявлено дефект, тим менша вартість його виправлення. Тестування, інтегроване

в CI/CD-конвеєр, дозволяє знаходити проблеми вже на етапі злиття коду (merge request).

- Управління ризиками: Тестування оцінює ймовірність відмов системи та допомагає керівництву приймати обґрунтовані рішення щодо випуску продукту (Go/No-Go Decision).
- Зворотний зв'язок (Feedback Loop): Надає розробникам та бізнесу швидкий та об'єктивний зворотний зв'язок про стан продукту, дозволяючи оперативно коригувати напрямок розробки.

Проблеми, що виникають при розробці складних ІКС, є багатогранними. Тестування виступає як критичний механізм їх вирішення:

Проблеми, пов'язані з коректністю та надійністю функціоналу

1. Помилки в бізнес-логіці: Найдорожчі та найкритичніші дефекти. Тестування API (яке є предметом даної роботи) зосереджено саме на виявленні неточностей, неправильних розрахунків або збоїв у послідовності виконання бізнес-правил.
2. Дефекти інтеграції: В сучасній мікросервісній архітектурі, коли компоненти ІКС (наприклад, сервіс авторизації, сервіс даних) взаємодіють через API, тестування вирішує проблему забезпечення коректності цього обміну даними.
3. Регресійні помилки: Це поява нових дефектів у раніше працюючому функціоналі після внесення змін (виправлення багів або додавання нових функцій). Автоматизоване регресійне тестування є єдиним ефективним способом боротьби з цією проблемою.

Проблеми, пов'язані з продуктивністю та масштабованістю

1. Проблеми з продуктивністю (Performance Issues): ВА, що працюють в ІКС, повинні витримувати пікові навантаження. Тестування допомагає виявити "вузькі місця" (bottlenecks) у коді, базах даних або мережевій

взаємодії, що призводять до повільного часу відгуку, особливо під навантаженням.

2. Неефективне використання ресурсів: Дефекти, що призводять до витоків пам'яті (memory leaks) або надмірного навантаження на процесор. Тестування допомагає оптимізувати використання ресурсів, знижуючи експлуатаційні витрати[19].

Проблеми, пов'язані з безпекою та вразливістю

1. Загрози безпеці (Security Vulnerabilities): Тестування безпеки (включаючи спеціалізовані методи, такі як пентести) виявляє вразливості, як-от SQL-ін'єкції, міжсайтовий скриптинг (XSS), або неправильна обробка автентифікації/авторизації. В ІКС, що працюють із конфіденційними даними (фінанси, телекомунікації), це є життєво необхідним.

Проблеми, пов'язані з експлуатацією та підтримкою

1. Низька супроводжуваність (Low Maintainability): Тестування спонукає розробників писати чистіший, модульніший код, що полегшує його подальшу підтримку та розширення.
2. Недокументована поведінка: Тести (особливо автоматизовані, написані за методологією BDD) часто слугують як "жива" документація системи, описуючи очікувану поведінку функціоналу.

Таблиця 1.1 - Порівняльна характеристика ручного та автоматизованного тестування

Характеристика	Ручне тестування	Автоматизоване тестування
Швидкість виконання	Повільна, лімітована швидкістю людини.	Висока, дозволяє запускати тисячі тестів за хвилини.
Ефективність регресії	Низька, повторне виконання — монотонне та дороге.	Висока, ідеальне для частих і повторюваних перевірок.
Об'єктивність	Суб'єктивна, залежить від уваги та настрою тестувальника.	Об'єктивна, ґрунтується на заздалегідь визначених критеріях.
Інтеграція в CI/CD	Неможлива.	Повна, є невід'ємною частиною конвеєра.
Вартість (у довгостроковій перспективі)	Висока, зростає пропорційно кількості функціоналу.	Низька, початкові інвестиції швидко окупаються.

В умовах CI/CD, кожне злиття коду повинно бути негайно перевірено, щоб уникнути блокування розробки. Єдиний спосіб забезпечити якість в цьому циклі — це впровадження ФАТ. Саме цей процес, з акцентом на сучасний стек (Playwright, TypeScript) та гібридний підхід UI+API, є головним предметом дослідження цієї роботи, оскільки він дозволяє вирішити найгостріші проблеми якості у складних ІКС.

Таким чином, тестування ПЗ еволюціонувало від простого пошуку помилок до стратегічного інструменту підвищення якості, інтегрованого у всі

етапи розробки, що є критично важливим для забезпечення конкурентоспроможності та надійності ІКС у сучасному бізнесі.

1.3 Класифікація та порівняння видів тестування: ручне (Manual) vs. автоматизоване (Automated) тестування.

Тестування програмного забезпечення класифікується за різними ознаками, але фундаментальний поділ існує між методами виконання: ручним та автоматизованим.

Ручне тестування - це процес, де тестувальник виконує тестові сценарії без використання спеціалізованих інструментів автоматизації, імітуючи дії кінцевого користувача. Цей підхід незамінний там, де потрібна людська оцінка, інтуїція та креативність.

Основні види ручного тестування:

1. Функціональне тестування (Functional Testing):
    - Smoke Testing (Димове тестування): Швидка перевірка основних, критично важливих функцій системи для підтвердження, що застосунок придатний для подальшого тестування.
    - Sanity Testing (Тестування осудності): Невелика, швидка перевірка після незначних змін або виправлення дефектів, щоб переконатися, що основний функціонал не постраждав.
    - Exploratory Testing (Дослідницьке тестування): Одночасне навчання, проєктування та виконання тестів. Тестувальник імпровізує та використовує свою інтуїцію для пошуку непередбачених дефектів[1].
    - Usability Testing (Тестування працездатності): Оцінка, наскільки зручний та інтуїтивно зрозумілий інтерфейс для кінцевого користувача.

Для забезпечення максимальної ефективності та тестового покриття ручне тестування використовує техніки тест-дизайну. Вони допомагають систематизувати тестові випадки та уникнути дублювання[20]:

- Еквівалентний Поділ (Equivalence Partitioning): Розділяє вхідні дані на класи (діапазони) еквівалентності. Тестувальник обирає лише одне значення з кожного класу, оскільки очікується, що всі значення в класі будуть оброблені системою однаково (наприклад, одне правильне та одне неправильне значення всередині/поза діапазоном)[2].
- Аналіз Граничних Значень (Boundary Value Analysis): Зосереджується на перевірці поведінки системи на межах діапазонів (наприклад, 1, 10, 0, 11 для діапазону від 1 до 10), оскільки більшість дефектів виникає саме тут.
- Причина/Наслідок (Cause/Effect): Допомагає визначити мінімальний набір вхідних комбінацій (причин), які призводять до очікуваного результату (наслідку), що підвищує ефективність перевірки складних бізнес-правил[3].
- Таблиця рішень (Decision Table Testing): Використовується для тестування функцій, чия поведінка залежить від комбінації різних умов (наприклад, знижки, що залежать від статусу клієнта та суми замовлення).

Автоматизоване тестування (АТ) - це процес використання програмних інструментів для виконання тестових сценаріїв, порівняння фактичних результатів з очікуваними та генерації звітів. АТ є ключовим для забезпечення швидкості та надійності регресійних перевірок.

Автоматизації підлягають майже всі рівні тестування, але найчастіше виділяють наступні види:

1. Модульне тестування (Unit Testing): Перевірка найменших ізольованих одиниць коду (функцій, методів, класів). Найшвидший та найдешевший вид АТ.
2. Інтеграційне тестування (Integration Testing): Перевірка коректності взаємодії між двома або більше модулями (наприклад, взаємодія між клієнтом і базою даних або між двома мікросервісами).
3. Тестування API (Service Layer Testing): Автоматизована перевірка програмних інтерфейсів (REST, SOAP). Воно зосереджується на бізнес-логіці та потоці даних. Це швидше і стабільніше, ніж UI-тестування[19].
4. Тестування інтерфейсу користувача (UI Testing / End-to-End): Автоматизоване імітування дій користувача у веб-браузері. Забезпечує максимальну впевненість у тому, що система працює коректно для кінцевого споживача, але є найповільнішим і найбільш "крихким" (flaky).
5. Тестування продуктивності (Performance Testing): Автоматичне створення навантаження на систему для оцінки її стабільності, часу відгуку та використання ресурсів.

Вибір між ручним та автоматизованим тестуванням залежить від цілей, етапу розробки та частоти виконання. Оптимальний підхід полягає у їхньому розумному поєднанні.

Таблиця 1.2 - Порівняльна характеристика ручного та автоматизованого тестування.

Критерій	Ручне тестування (Manual)	Автоматизоване тестування (Automated)
Цілі	Юзабіліті, естетика, зручність, неформальні перевірки, дослідництво.	Регресія, функціональна коректність, продуктивність, API-логіка.
Початкові інвестиції	Низькі.	Високі (потрібна розробка фреймворку та написання коду).
Швидкість виконання	Повільна.	Висока (забезпечує швидкий зворотний зв'язок).
Регресійне тестування	Повільне, дороге, схильне до помилок.	Швидке, надійне, економічно вигідне у довгостроковій перспективі.
Інтеграція в CI/CD	Неможлива.	Повна, є основою конвеєра DevOps.
Крихкість (Fragility)	Відсутня.	Висока у UI-тестів (зміна елемента UI ламає тест).
Швидкість виправлення помилки	Низька (потрібно повторювати багато кроків).	Висока (зміна локатора/API-поля вимагає мало рядків коду).

Висновок: Ручне тестування необхідне для перевірки користувацького досвіду та дослідницького пошуку дефектів. Автоматизоване тестування (особливо на рівні API) є ключовим для швидких, повторюваних та великомасштабних регресійних перевірок, що є критичним для якості функціонування складних ІКС в умовах безперервної інтеграції.

1.4 Обґрунтування доцільності застосування автоматизованого тестування: критерії вибору та сценарії, коли автоматизація є недоречною.

Автоматизоване тестування (АТ) є не просто технічним інструментом, а стратегічною необхідністю для сучасних інфокомунікаційних систем (ІКС) та веб-застосунків (ВЗ), які розробляються за методологіями Agile та DevOps. Доцільність АТ обґрунтовується економічними, якісними та операційними перевагами, які ручне тестування (РТ) не може забезпечити в умовах швидкої розробки.

Основні аргументи на користь АТ:

1. Скорочення циклу регресійного тестування: АТ дозволяє запускати тисячі тестів за хвилини, тоді як ручний регресійний прогін може займати дні. Це критично важливо для підтримки безперервної інтеграції (CI), оскільки команда отримує швидкий зворотний зв'язок про внесені зміни[3].
2. Підвищення надійності та точності: Автоматизовані тести виконуються ідентично при кожному запуску, усуваючи людський фактор (втому, неуважність), що підвищує об'єктивність та надійність перевірок[1].
3. Економічна ефективність (ROI): Хоча початкові інвестиції в розробку фреймворку є значними, у довгостроковій перспективі АТ значно знижує загальну вартість контролю якості, оскільки вартість повторного виконання тестів є майже нульовою[2]. Чим частіше функція тестується (наприклад, критична функція аутентифікації), тим вищою є економічна вигода від автоматизації.
4. Розширення можливостей тестування: АТ може виконувати ті перевірки, які фізично неможливі або непрактичні для людини, наприклад, симуляція тисяч одночасних користувачів (тестування продуктивності) або багаторазове повторення складних операцій.

5. Підтримка DevOps та CI/CD: ФАТ є ключовим елементом автоматизованого конвеєра, забезпечуючи, що тільки якісний код потрапляє до наступних середовищ (staging, production).

1.4.2 Критерії вибору для автоматизації (Що потрібно автоматизувати?)

Для максимізації віддачі від інвестицій (ROI) не слід автоматизувати все підряд. Рішення про автоматизацію конкретного тестового випадку чи функціоналу має ґрунтуватися на чітких критеріях:

Таблиця 1.3 - Таблиця критеріїв доцільності впровадження та використання підходу автоматизованого тестування.

Критерій	Опис та доцільність АТ
Частота виконання	Висока. Сценарії, що виконуються регулярно (наприклад, при кожній збірці або щоденно). Регресійні тести є головним кандидатом на автоматизацію.
Критичність функціоналу	Висока. Сценарії, які перевіряють основну бізнес-логіку або критичні потоки (наприклад, вхід у систему, оформлення замовлення). Несправність цього функціоналу призведе до зупинки бізнесу.
Стабільність функціоналу	Висока. Функціонал, який вже усталений і не підлягатиме частим змінам у дизайні або логіці. Тести для нестабільного функціоналу будуть вимагати постійної підтримки, що знижує ROI.
Складність даних	Висока. Тести, які вимагають введення великого обсягу даних або складних комбінацій (наприклад, перевірка різних податкових ставок або прав доступу). АТ легко справляється з керуванням тестовими даними.
Час виконання РТ	Довгий. Сценарії, які займають багато часу при ручному виконанні. Автоматизація дозволить перерозподілити час тестувальників на дослідницьке тестування.
Використання на різних рівнях	Перевага надається API-тестам перед UI-тестами, оскільки вони швидші, стабільніші і забезпечують краще покриття бізнес-логіки.

Ідеальний кандидат для автоматизації – це тест, який часто виконується, є критично важливим, вимагає багато даних та має стабільну логіку.

Існують ситуації, коли інвестиції часу та ресурсів в автоматизацію не окупаються або вона просто не може замінити людську оцінку. У таких випадках доцільніше залишити тестування ручним.

1. Функціонал, який швидко змінюється

- Проблематика: Якщо функція або інтерфейс знаходяться на етапі активної розробки, і їхній дизайн (локатори, поля, порядок кроків) змінюється щодня, написання автотесту буде неефективним. Тест вимагатиме постійного оновлення (висока ремонтопридатність), що збільшує накладні витрати до рівня, коли дешевше виконати тест вручну.
- Рішення: Застосовувати ручне або дослідницьке тестування, поки функціонал не стабілізується.

2. Дослідницьке та Ad-hoc тестування

- Проблематика: Дослідницьке тестування (Exploratory Testing) ґрунтується на інтуїції, знаннях та креативності тестувальника. Його мета – виявити непередбачену поведінку та знайти дефекти, які не вказані у вимогах. Цей процес є унікальним при кожному запуску, що унеможливлює його кодування.
- Рішення: Ці види тестування залишаються виключно в компетенції людини.

3. Тестування юзабіліті та візуальної привабливості

- Проблематика: Жоден автоматизований тест не може оцінити естетичну привабливість, зручність навігації, правильність сприйняття кольорової гами або загальне задоволення користувача від роботи з ВЗ. Це є завданням тестування працездатності (Usability Testing).
- Рішення: Використання фокус-груп та ручних перевірок UX/UI.

4. Одноразові або рідкісні тести

- Проблематика: Тестові сценарії, які виконуються лише один раз, не окуплять часу, витраченого на написання, налагодження та інтеграцію

їхнього коду. Час, необхідний для автоматизації (наприклад, 4 години), значно перевищить час ручного виконання (наприклад, 20 хвилин).

- Рішення: Виконання вручну.

Оптимальна стратегія тестування ілюструється Пірамідою тестування, яка відображає ієрархію та співвідношення між різними рівнями тестування:

1. Широка основа: Модульне тестування (Unit Testing) — найбільш часте, швидке і дешеве (90% тестів мають бути тут).
2. Середній рівень: Інтеграційне та API-тестування — перевірка бізнес-логіки без завантаження UI (критично для швидкості регресії).
3. Верхівка: UI (End-to-End) тестування — найменша кількість тестів, оскільки вони найповільніші та найбільш "крихкі" (flaky).

Таким чином, доцільність застосування АТ у розробці ІКС не зводиться до заміни людини машиною. Вона полягає у стратегічному перерозподілі зусиль: автоматизувати повторювані, критичні та швидкі тести (Unit, API) для CI/CD, залишаючи ручному тестуванню простір для досліджень, UX та роботи з нестабільним функціоналом. Саме ця філософія лягає в основу розробки даного ФАТ на базі Playwright та TypeScript[16].

1.5 Інструменти автоматизації тестування, історичний огляд та порівняння сучасних ФАТ.

Ефективність фреймворку автоматизованого тестування (ФАТ) значною мірою залежить від обраної моделі виконання коду, архітектури інструментів тестування та способів взаємодії з браузером або системою. У сучасних ФАТ важливу роль відіграють:

- модель виконання (синхронна чи асинхронна),
- підхід до типізації (динамічна чи статична),
- механізм управління потоками,

- взаємодія тестів із зовнішніми API, UI та інфраструктурою.

У цьому контексті ключовими є три підсистеми: модель виконання JavaScript/TypeScript, асинхронна взаємодія браузера, та архітектура популярних ФАТ (Playwright, Cypress, Robot Framework).

1. Моделі виконання: Синхронний та Асинхронний код.

Розуміння моделей виконання є ключовим, оскільки взаємодія ФАТ з браузером або API є операцією введення/виведення (I/O), яка вимагає очікування.

Таблиця 1.4 - Порівняльна характеристика синхронного та асинхронного коду у мовах програмування.

Характеристика	Синхронний код (Java/Selenium)	Асинхронний код (TypeScript/Node.js)
Виконання	Послідовне та Блокуюче. Наступна операція чекає завершення попередньої. Основний потік заблоковано.	Неблокуюче та Паралельне. Операція запускається, але потік продовжує виконання інших завдань, не очікуючи результату I/O.
I/O-Операції	Якщо потік ініціює запит до браузера, він блокується і простоює до отримання відповіді. Компенсується використанням багатьох потоків.	Не блокується. Відповідь обробляється пізніше через механізм Event Loop (цикл подій). Це набагато ефективніше для веб-завдань.
Вплив на тести	Неефективне для автоматизації, оскільки очікування I/O-операцій призводить до простою ресурсів.	Висока продуктивність та ефективне використання ресурсів, що ідеально підходить для I/O-інтенсивних завдань.

У синхронному виконанні код обробляє інструкції послідовно. Якщо відбувається I/O-операція (запит до браузера чи API), потік блокується до

отримання результату. У Java це характерно для Selenium, де кожна взаємодія з вебелементом блокує потік тесту[20].

Приклад (Java) та його синхронність: Потік блокується, вимагаючи жорсткої структури Класу та методу main.

```java
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class SyncUITest {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        try {
            // Відкриваємо сторінку
            driver.get("https://example.com/login");

            // Синхронне очікування елемента
            WebElement email = driver.findElement(By.id("email"));
            email.sendKeys("test@example.com");

            WebElement password = driver.findElement(By.id("password"));
            password.sendKeys("123456");

            WebElement loginButton = driver.findElement(By.id("login"));
            loginButton.click();

            // Перевірка, що після логіну з'являється профіль
            WebElement profile = driver.findElement(By.id("profile"));
            if (profile.isDisplayed()) {
                System.out.println("Login test passed");
            } else {
                System.out.println("Login test failed");
            }
        } finally {
            driver.quit();
        }
    }
}
```

Рис. 1 Приклад синхронного коду на мові програмування Java.

Характеристика:

- Потік блокується на кожному кроці, поки Selenium виконує запит до браузера.
- Передбачуваність виконання, але низька ефективність при великій кількості тестів або I/O-операцій.

- Для підвищення продуктивності потрібне використання багатопоточності.

Приклад (TypeScript) та його асинхронність: У Node.js (TypeScript) використовується однопотоковий механізм Event Loop. Коли асинхронна I/O-операція (наприклад, запит до браузера) ініціюється, вона вивантажується в систему, а Event Loop продовжує обробляти інші завдання в основному потоці. Коли I/O-операція завершується, її результат ставиться в чергу для подальшої обробки. Це дозволяє Playwright виконувати багато тестів швидко, не витрачаючи ресурси на очікування.

```
import { test, expect } from '@playwright/test';

test('user can log in', async ({ page }) => {
  await page.goto('/login');
  await page.fill('#email', 'test@example.com');
  await page.fill('#password', '123456');

  // Очікування реальної відповіді API
  await page.waitForResponse(resp =>
    resp.url().includes('/auth') && resp.status() === 200
  );

  await expect(page.locator('#profile')).toBeVisible();
});
```

Рис.2 Приклад асинхронного коду на простому прикладі.

Характеристика:

- Event Loop дозволяє запускати інші задачі під час очікування I/O.
- Ефективність і масштабованість - ключові для великого ФАТ.
- Зменшує "flaky tests", які часто виникають у синхронних тестах через таймаути.

2. Статична та динамічна типізація в тестуванні.

   Спосіб, яким вихідний код перетворюється на машинний, має прямий зв'язок із системою типізації та швидкістю виконання.

Таблиця 1.5 - Приклад моделей обробки коду

Модель обробки	Принцип роботи	Приклади	Зв'язок із типізацією
Компілятор	Перетворює весь код у машинний/байт-код до виконання. Процес одноразовий.	Java, C++, C#	Тісно пов'язаний зі статичною типізацією, оскільки компілятор вимагає перевірки типів на етапі збірки.
Інтерпретатор	Виконує код рядок за рядком під час запуску програми.	JavaScript, Python	Часто пов'язаний із динамічною типізацією, де помилки типів виявляються лише під час виконання.
Гібрид (Transpiler)	Попередньо перетворює код (наприклад, TypeScript) в іншу форму (JavaScript) для подальшого виконання інтерпретатором.	TypeScript	Дозволяє застосовувати статичну типізацію в інтерпретованому середовищі.

Динамічна типізація (JavaScript / Python):

- Змінні можуть змінювати тип під час виконання.
- Гнучкість у роботі з API-відповідями або браузером.
- Недоліки: помилки типу об'єктів (наприклад, спроба викликати метод, якого не існує) видно лише під час виконання тесту, що знижує надійність.

```
test('API returns valid user', async () => {
  const user = await api.get('/user/1');
  expect(user.age > 0).toBe(true); // помилка проявиться тільки під час запуску
});
```

Рис.3 Приклад роботи динамічної типізації у тесті на JavaScript.

Статична типізація (TypeScript / Java / C# / C++)

- Типи перевіряються до виконання тестів.
- Зменшує кількість помилок у великому ФАТ.
- Компілятор гарантує, що клас має ті властивості, які ви намагаєтеся використати.
- Як результат це зменшує кількість помилок у великому ФАТ, забезпечує високу супроводжуваність коду.

```
interface User {
  id: number;
  name: string;
  email: string;
}

test('validate user response type', async () => {
  const user: User = await api.get('/user/1');
  expect(user.email).toContain('@'); // компілятор гарантує правильний тип
});
```

Рис.4 Приклад роботи статичної типізації у TypeScript.

Огляд інструментів:

1. **Selenium** (розробка розпочата ~ 2004 рік) є історичним фундаментом автоматизації та індустріальним стандартом W3C WebDriver.

W3C WebDriver - це офіційний стандарт, затверджений World Wide Web Consortium (W3C), який визначає протокол взаємодії тестових фреймворків із

браузерами. Його мета - уніфікувати спосіб автоматизації браузерів і забезпечити кросбраузерну сумісність[18].

Основні принципи:

a) Out-of-Process архітектура
   - Тестовий код виконується у процесі тестового фреймворку (наприклад, Selenium на Java або Python).
   - Браузер запускається окремо, а тест взаємодіє з ним через HTTP-запити.
   - Така архітектура дозволяє ізолювати браузер від тестового коду, але додає накладні затрати на комунікацію.

b) Уніфіковане API для різних браузерів
   - WebDriver підтримує Chrome, Firefox, Edge, Safari та інші.
   - Для кожного браузера існує драйвер, який реалізує WebDriver API:
     - chromedriver для Chrome,
     - geckodriver для Firefox,
     - msedgedriver для Edge,
     - safaridrivere для Safari,

c) Синхронний підхід до команд
   - Тест надсилає команду на драйвер → драйвер виконує дію в браузері → тест отримує результат.
   - Кожна взаємодія блокує тест до завершення команди, що характерно для синхронних тестів Selenium.

Типова взаємодія через W3C WebDriver

- Тест викликає команду: findElement(By.id("login"))
- Selenium надсилає HTTP-запит на chromedriver

- chromedriver керує Chrome через нативні інтерфейси (Chrome DevTools Protocol або UI Automation API).
- Браузер виконує команду → драйвер повертає результат тесту

Переваги WebDriver:

- Стандартизований, підтримується всіма основними браузерами.
- Забезпечує кросбраузерну сумісність тестів.
- Гнучкий у виборі мови програмування (Java, Python, C#, JavaScript).

Недоліки:

- Кожна команда синхронна → можливі затримки при виконанні I/O
- Out-of-Process → додаткові накладні витрати на комунікацію з браузером.
- Складніше інтегрувати асинхронні операції без спеціальних обгорток (наприклад, WebdriverIO або Selenide).

2. **Selenide** (2010 рік) - це Java-фреймворк для UI-автоматизації, який є обгорткою над Selenium і спрощує написання тестів. Набув активної популярності через вбудовані автоматичні очікування та зручний API для роботи з фреймворком.

Таблиця 1.6 - Переваги та недоліки фреймворку Selenide

Переваги	Недоліки
Автоматичні "розумні" очікування.	Успадковує архітектурні обмеження Selenium
Значно вища стабільність, ніж у "чистого" Selenium.	Обмежений лише Java-стеком.
Fluent Interface (чистий синтаксис)	Повільніший за Playwright (використовує W3C)
Мінімум бойлерплейту (немає ручного WebDriverWait)	Не має нативного вбудованого API-тестування
Автоматичне керування драйверами	Не підтримує новітні протоколи.
Легка інтеграція у будь-який існуючий Java-проєкт	Складніше інтегрувати з іншими мовами.
Надійність, властива строгій типізації Java	Великий розмір залежностей.
Автоматичні скріншоти при падінні.	Може мати конфлікти з іншими Selenium-бібліотеками.
Легкий старт.	Складнощі при роботі з тіньовим DOM.

```
import static com.codeborne.selenide.Selenide.*;
import org.openqa.selenium.Keys;

public class SelenideLoginTest {
    public static void main(String[] args) {
        open("https://example.com/login");
        $("[name='email']").setValue("test@example.com");
        $("[name='password']").setValue("123456");
        $("[id='login']").pressEnter();
    }
}
```

Рис. 5 Приклад тесту з використанням фреймворку Selenide.

У цьому прикладі Selenide автоматично чекає, поки елементи будуть готові до взаємодії, без явного WebDriverWait.

3. Cypress (2017 рік) - сучасний фреймворк для автоматизації UI-тестів, що працює безпосередньо всередині браузера.

In-Process архітектура:

- Cypress виконує тестовий код всередині браузера, в тому ж процесі, що й веб-додаток.
- Це дозволяє напряму взаємодіяти з DOM, мережевими запитами та JavaScript без зовнішніх драйверів.

Таблиця 1.7 - Переваги та недоліки використання фреймворку Cypress.

Переваги	Недоліки
Висока швидкість (через In-Process архітектуру).	Обмежена кросбраузерність (без WebKit)
Автоматичні очікування.	Не підтримує нативно роботу з кількома вкладками/вікнами
Унікальне налагодження (Time-Travel Debugging).	Обмеження при роботі з зовнішніми URL.
Нативний доступ до мережі для імітації (Stubs)	Не підтримує багатопоточність (виконання є однопотоковим)
Простий, командний синтаксис.	Підходить переважно для веб UI
Швидкий час налаштування.	Складніше тестувати завантаження файлів
Активна спільнота Node.js.	Обмежені можливості для Unit-тестування поза UI
Автоматичні скріншоти та відеозапис	Складніше керувати асинхронними операціями поза його командною чергою
Автоматичне керування драйверами	Обмеження при роботі з iFrame елементами

```
describe('Login Test', () => {
  it('should log in successfully', () => {
    cy.visit('/login');
    cy.get('#email').type('test@example.com');
    cy.get('#password').type('123456');
    cy.get('#login').click();
    cy.get('#profile').should('be.visible');
  });
});
```

Рис.6 Приклад вигляду готового тесту з використанням Cypress.

4. **Robot Framework** (2005 рік) - це фреймворк для автоматизації тестування високого рівня, який підтримує keyword-driven підхід та інтегрується з багатьма інструментами (Selenium, Appium, API).

Keyword-driven підхід:

- Тести складаються з ключових слів (keywords), що описують дії.
- Ключові слова можуть бути готовими (наприклад, Click Element) або створюватися користувачем.

Синхронна взаємодія:

- Robot Framework зазвичай працює синхронно: команда виконується повністю, поки тест чекає результат.
- Асинхронні дії підтримуються через бібліотеки (наприклад, використання Playwright або asyncio у Python).

Таблиця 1.8 - Переваги та недоліки фреймворк Robot Framework

Переваги	Недоліки
Низький поріг входження (ідеально для тестувальників без програмування)	Складнощі при створенні складних/кастомних логік (обмежена гнучкість)
Висока читабельність (сценарії схожі на природну мову)	Низька швидкість виконання порівняно з кодовими фреймворками.
Можливість розширення за допомогою Python.	Вимагає додаткових бібліотек (наприклад, SeleniumLibrary) для UI.
Легке створення звітів.	Обмеження щодо використання ООП
Висока гнучкість у застосуванні (UI, API, Desktop).	Складніше налагодження
Широкий набір вбудованих бібліотек.	Великий розмір файлів з тестами.
Можливість створення власних ключових слів.	Залежність від Python
Підходить для enterprise QA та не-код команд.	Низька ефективність для Unit-тестування.
Добре документований.	Складно інтегрувати з сучасними інструментами, що не базуються на Python
Активна спільнота Python.	Вимагає дисципліни у створенні унікальних ключових слів.

```
*** Settings ***
Library    SeleniumLibrary

*** Test Cases ***
Login Test
    Open Browser    https://example.com/login    chrome
    Input Text      id=email                     test@example.com
    Input Text      id=password                  123456
    Click Button    id=login
    Page Should Contain Element    id=profile
    Close Browser
```

Рис.7 Приклад використання Robot Framework.

Аналіз основних ФАТ показує, що кожен інструмент має свої сильні та слабкі сторони, а вибір залежить від конкретних вимог проекту.

1.6 Обґрунтування вибору Playwright та TypeScript/Node.js як ключових технологій

У процесі розробки Фреймворку Автоматизованого Тестування (ФАТ) критично важливо обрати технології, які забезпечать стабільність, масштабованість, швидкість розробки, а також гнучкість інтеграції з сучасними веб-додатками. На сьогодні екосистема автоматизації тестування налічує десятки інструментів, проте не всі вони здатні забезпечити стабільну роботу з динамічними SPA-додатками, сучасними JS-фреймворками (React, Vue, Angular), мікросервісною архітектурою, Shadow DOM, WebSockets та ін.

Playwright - це сучасний фреймворк для end-to-end тестування вебзастосунків, розроблений компанією Microsoft у 2020 році. Інструмент створювався на основі досвіду розробки Puppeteer, тому Playwright можна вважати «другим поколінням» браузерної автоматизації: швидшим, надійнішим та архітектурно сенсовнішим[9].

Playwright забезпечує:

- автоматичні очікування (auto-waiting),
- повну підтримку Chromium, Firefox, WebKit,
- роботу з версіями браузерів без WebDriver-протоколу,
- ізольовані браузерні контексти,
- паралельне виконання тестів,
- API для роботи з мережею, файлами, cookies, localStorage,
- нативну інтеграцію з TypeScript,
- стабільні локатори (locator engine),
- підтримку UI Mode (графічне виконання тестів),
- власні інструменти налагодження.

Ці характеристики суттєво вплинули на вибір саме Playwright як основи для ФАТ.

Playwright працює за моделлю **Driver → Browser → Page**.

Його архітектура не використовує JSON Wire Protocol, на відміну від Selenium/WebDriver.

Таблиця 1.9 - Коротка порівняльна характеристика між інструментами

WebDriver (Selenium)	Playwright
Взаємодіє через мережевий протокол	Використовує прямий канал зв’язку з браузером
Кожна дія створює HTTP-запит	Мінімальна затримка між командами
Затримка між діями	Повний контроль над браузерним процесом
Фокус на сумісності, а не на швидкості	Синхронізація вбудована на рівні драйвера
Вразливий до flaky-тестів	Всі очікування автоматизовані

Playwright автоматично очікує:

- видимість,
- стабільність DOM,
- readiness state,
- attach до DOM,
- відсутність обробки подій,
- завершення анімацій.

Приклад методу кліку на кнопку з текстом Login у Playwright:
await page.getByRole('button', { name: 'Login' }).click();

У Selenium довелося б явно очікувати елемент:
WebDriverWait wait = new WebDriverWait(driver, 10);
wait.until(ExpectedConditions.elementToBeClickable(locator)).click();

Playwright дозволяє створювати десятки незалежних «мікробраузерів» в одному процесі:

```
const context = await browser.newContext();
const page = await context.newPage();
```

На практиці це дає так переваги:

- тести не впливають один на одного,
- можна логінитись одночасно за різних користувачів,
- контексти працюють як вкладки реального браузера.

Playwright має власний локаторний рушій що набагто спрощує можливість використання стабільних локаторів:

```
page.getByRole('button', { name: 'Submit' })
page.getByTestId('login-field')
page.getByText('Dashboard')
```

Playwright пропонує повноцінний доступ до мережі за рахунов вбудованих можливостей які можна одразу використовувати за замовчуванням:

```
await page.route('**/api/user', route => {
  route.fulfill({ json: { name: 'Test' } });
});
```

Це дає такі можливості:

- перехоплювати запити,
- емулювати відповіді (зробити готову відбовідь від сервера замість очікування реального),
- аналізувати трафік

Також одна з ключових переваг це використання fixtures у фреймворку.

У Playwright фікстури - це спеціальні об’єкти або функції, які створюють і налаштовують середовище для тестів[4].

Вони:

- створюють дані,
- відкривають сторінки,
- роблять логін,
- налаштовують API-клієнт,
- дозволяють перевикористовувати логіку між тестами.

```
import { test } from './fixtures';

test('user can open dashboard', async ({ loggedPage }) => {
  await loggedPage.goto('/dashboard');
  await loggedPage.waitForSelector('text=Welcome');
});
```

Рис.8 Приклад використання фікстури у тесті

Як видно на рис.8, наш тест одразу починається зі сторінки Dashboard, а логіка логіну винесена у фікстуру loggedPage окремо, що дає значну перевану у чистоті тесту для якого не потрібно писати багато підготовчих кроків до самого тесту.

Playwright - єдиний фреймворк, який може тестувати WebKit на Windows/Linux.Це критично для iOS Safari сумісності.

Playwright має вбудований графічний середовище виконання тестів(рис.8):

- таймлайн,
- автопауза,
- скріншоти,
- відео,
- трасування дій.

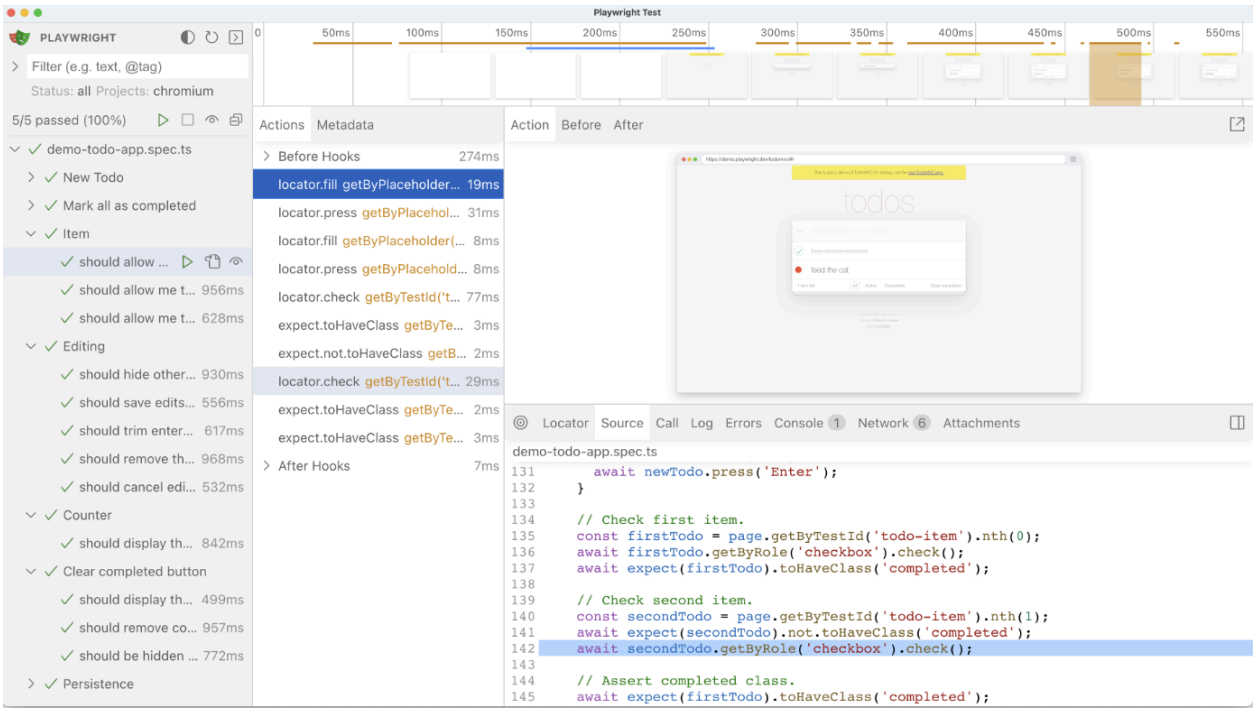


Рис.9 Приклад використання UI моду для дебагу теста

Playwright пропонує використання різних мов програмування: JavaScript, TypeScript, Python, Java, C#. Проте є одна особливість - всі нові апдейти та покращення в першу чергу випискають для Node.js а самеTypeScript і саме тому ця мова пропонується і використовуєтьсяу більшості проєктів на Playwright[8].

Сильні сторони TypeScript:

- статична типізація → менше помилок,
- строго типізовані Page Object-и,
- типізовані фікстури,
- автодоповнення у VS Code,
- контроль структури тестового фреймворку,
- нативна підтримка у Playwright API.

А Node.js рушій забезпечує:

- неблокуючу асинхронність,
- високу швидкість виконання,
- модульність,
- багатий набір бібліотек,
- нативну сумісність із TypeScript.

Також один з ключевих факторів у виборі ФАТ є грунтовна зацікавленість у розробці та підтримці з боку розробників цього програмного забезпечення. Обидва інструменти - Playwright та TypeScript були розроблені компанією Microsoft[4].

TypeScript створений Microsoft у 2012 році.
Playwright створений Microsoft у 2020 році.

Обидва інструменти:

- розвиваються під одну технічну філософію,
- мають єдиний підхід до API,
- працюють у Node.js рушії,
- оптимізовані одне під одного,
- використовують VS Code (редактор коду) як основне середовище.

Playwright написаний з урахуванням того, що код буде працювати в TypeScript. Всі методи мають:

- строгі типи,
- описані інтерфейси,
- документацію прямо у типах.

Це робить Playwright + TypeScript «ідеальною парою».

```
test('login flow', async ({ page }) => {
  await page.goto('/login');
  await page.getByTestId('email-input').fill('user@test.com');
  await page.getByTestId('password-input').fill('123456');
  await page.get le('button', { name: 'Login' }).click();
  await expect(page.getByText('Dashboard')).toBeVisible();
});
```

Рис.10 Приклад асинхронного тесту на Playwright та TypeScript для тестування корректного входу у систему.

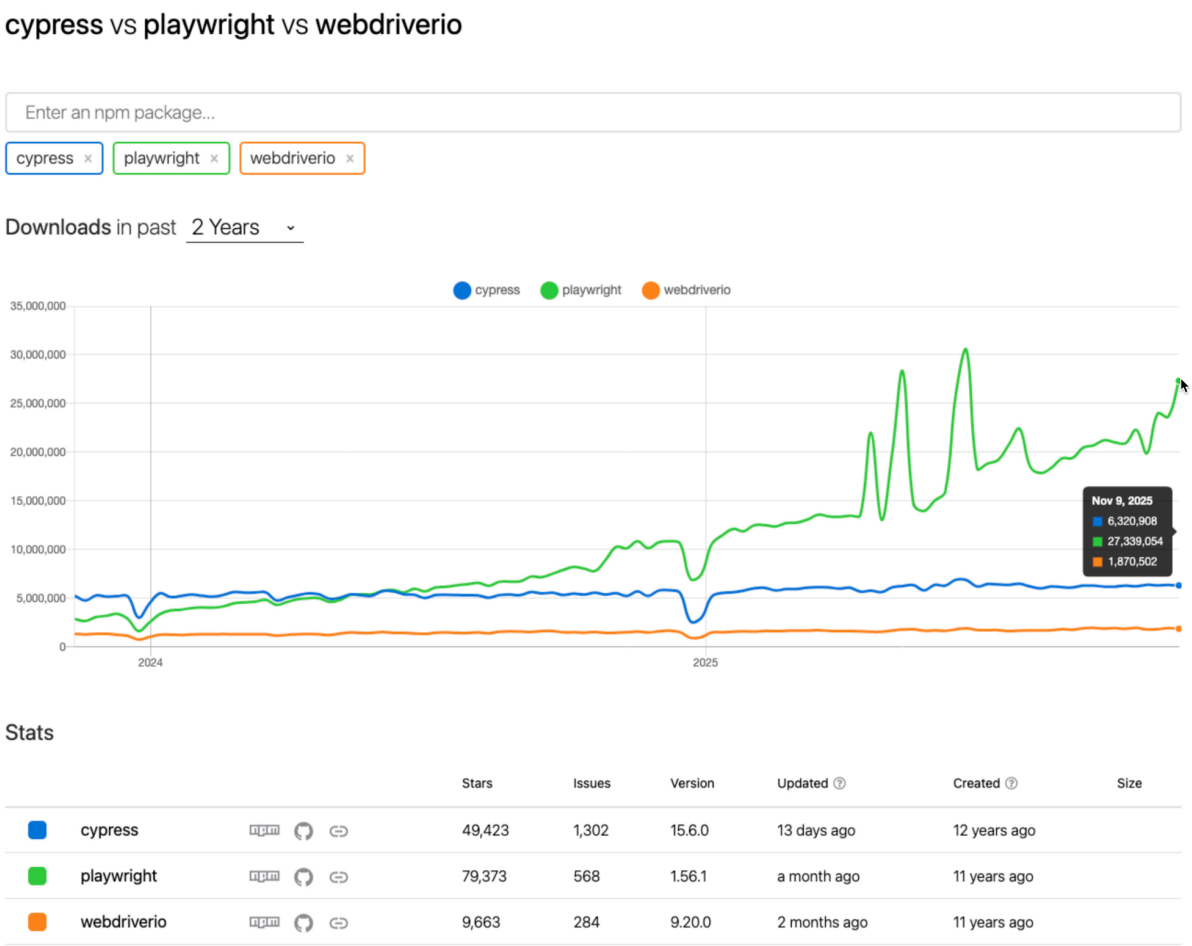


Рис.11 Інфографіка скачувань інструменту за два роки за допомогою ресурсу https://npmtrends.com/

Як видно на рис.11 популярність Playwright кожним роком росла зі значущим відривом від своїх сусідів по фреймворку з використанням JavaScript/Typescript.

Тому, вибір Playwright та TypeScript/Node.js як основних технологій для розробки ФАТ був обумовлений факторами:

- висока швидкодія завдяки прямому протоколу взаємодії з браузером,
- стабільність тестів завдяки auto-waiting,
- нативна підтримка TypeScript,
- підтримка Microsoft обох технологій,
- глибока екосистемна сумісність,
- наявність фікстур,
- підтримка WebKit, Firefox, Chromium,
- комплексність API,
- зручність побудови власного фреймворку.

Таким чином, Playwright + TypeScript є сучасним, ефективним та стратегічно виправданим рішенням для побудови фреймворку автоматизованого тестування[7].

Висновки до розділу І

У першому розділі було проведено комплексний теоретичний аналіз веб-застосунків, інфокомунікаційних систем та сучасних підходів до забезпечення якості програмних продуктів. Розглянуто фундаментальні поняття та класифікації, які формують основу для подальшого опрацювання технологій автоматизованого тестування та вибору інструментів для реалізації тестового середовища.

Показано, що веб-застосунки та інфокомунікаційні системи відіграють ключову роль у роботі сучасних підприємств, забезпечуючи автоматизацію бізнес-процесів, масштабованість, високу доступність і інтеграцію з численними зовнішніми сервісами. Їхня архітектура еволюціонувала від монолітних до мікросервісних систем, а це значно підвищило складність тестування, актуалізувавши потребу в автоматизації.

Було обґрунтовано, що якість програмного забезпечення є багатовимірною характеристикою, що включає надійність, продуктивність, безпеку, зручність використання та підтримуваність. Тестування ПЗ розглянуто як центральний інструмент забезпечення якості, який дозволяє виявляти дефекти, знижувати ризики та запобігати критичним помилкам під час експлуатації системи.

Проведено порівняння ручного та автоматизованого тестування, в ході якого визначено їх переваги, недоліки, сфери застосування та особливості планування тестових стратегій. Показано, що автоматизація є доцільною за умови достатньої стабільності продукту, повторюваності тестів та наявності технічних ресурсів, але водночас наведено приклади ситуацій, у яких ручне тестування залишається незамінним.

Історичний огляд інструментів автоматизації тестування та аналіз сучасних фреймворків продемонстрували значний прогрес галузі — від класичного Selenium WebDriver, який заклав основу браузерної автоматизації, до сучасних інструментів нового покоління (Playwright, Cypress, Selenide, Robot Framework), що забезпечують вищу стабільність, паралельність,

кросбраузерність та інтеграцію з CI/CD-платформами. Порівняльний аналіз виявив тенденцію до переходу індустрії від громіздких, складних інструментів до більш гнучких, швидких і розробник-орієнтованих фреймворків.

У підрозділі 1.6 обґрунтовано вибір Playwright та TypeScript/Node.js як ключових технологій для реалізації автоматизованого тестування у рамках цієї роботи. Зокрема продемонстровано, що Playwright, створений компанією Microsoft, забезпечує високу швидкість тестів, нативну підтримку кількох браузерів, автоматичне очікування елементів, вбудований API-клієнт, механізм фікстур та зручні інструменти налагодження. Поєднання з TypeScript — також розробкою Microsoft — створює додаткові переваги: типізацію, кращу підтримку IDE, зменшення кількості помилок часу виконання та високу масштабованість тестового проєкту.

Таким чином, у розділі 1 сформовано цілісну теоретичну основу, яка дозволяє аргументовано перейти до практичної частини роботи — розробки архітектури автоматизованих тестів, створення тестового середовища, впровадження тест-сценаріїв і виконання експериментальної перевірки якості веб-застосунку за допомогою Playwright та TypeScript/Node.js.

РОЗДІЛ 2. МОДЕЛЮВАННЯ АРХІТЕКТУРИ ТА СТРУКТУРИ ФРЕЙМВОРКУ

2.1 Модель фреймворку автоматизованого тестування (ФАТ): функціональні вимоги та архітектурні принципи

Фреймворк автоматизованого тестування - це структурований набір компонентів, правил та інструментів, призначених для:

- організації виконання тестів;
- налаштування тестового середовища;
- керування залежностями;
- взаємодії з тестованою системою (UI, API, БД, мережевими протоколами);
- формування репортів;
- забезпечення розширюваності та підтримуваності проекту.

У більш широкому сенсі модель ФАТ визначає:

1. Логічну структуру фреймворку (модулі, шари, взаємодія між ними).
2. Процеси, які фреймворк автоматизує (підготовка середовища, запуск тестів, логування, аналіз результатів).
3. Архітектурні підходи, що забезпечують масштабованість.
4. Інтеграції з інструментами DevOps та сторонніми сервісами.
5. Правила для розробників та тестувальників, які працюють з кодом.

Таким чином, модель ФАТ є фундаментом, що визначає поведінку системи і слугує основою для її реалізації.

Функціональні вимоги - це ключові можливості, які фреймворк повинен забезпечувати. Нижче виділено вимоги, сформовані на основі аналізу сучасних ФАТ та потреб підприємств, що працюють з веб-системами[20]:

1. Підтримка браузерної автоматизації

Фреймворк має забезпечувати:

- взаємодію з UI у браузерах Chromium, Firefox, WebKit;
- управління контекстами та вкладками;
- виконання дій (клік, введення тексту, прокрутка, інтерцепція запитів);
- визначення стану елементів DOM.

2. Автоматичне очікування стабільного стану сторінки

У випадку Playwright фреймворк повинен:

- виконувати дії лише після того, як елемент стане доступним;
- уникати потреби у ручних timeouts;
- гарантувати детермінованість тестів.

3. Підтримка API-тестування

Фреймворк має забезпечувати:

- надсилання HTTP-запитів;
- перевірку статус-кодів та структур відповідей;
- роботу з авторизацією та токенами;
- логування API-трафіку.

4. Механізм фікстур (fixtures)

Обов’язковими можливостями є:

- створення передумов для тестів;
- ізоляція тестових контекстів;
- управління життєвими циклами ресурсів (браузер, сторінка, тестові дані).

5. Структуроване логування у якому фреймворк повинен:

- збирати логи дій;
- фіксувати помилки та винятки;

- зберігати трасування (trace) для дебагу.

6. Формування репортів

Обов’язкові формати:

- HTML-звіти з деталізацією кроків;
- відеозаписи виконання;
- скриншоти при падінні;

7. Підтримка паралельного виконання

Фреймворк має:

- запускати тести у кількох потоках;
- керувати чергами та ресурсами;

дозволяти distributed execution (наприклад, у CI/CD).

8. Масштабованість

Проект повинен:

- підтримувати розширення тестового набору;
- забезпечувати незалежність модулів;
- містити чітку структуру директорій;
- легко адаптуватися під нові модулі системи.

9. Інтеграція з CI/CD

Необхідно забезпечити:

- запуск через GitHub Actions / GitLab CI / Jenkins;
- збір і збереження репортів;
- за потреби можливість роботи у docker-контейнерах.

10. Робота з тестовими даними

Фреймворк повинен включати:

- генерацію тестових даних;

- управління JSON / YAML конфігами;
- створення окремих data-layer або фабрик.

Нефункціональні вимоги

1. Продуктивність

- швидкість виконання тестів;
- мінімальна кількість блокуючих операцій;
- оптимізація очікувань та пошуку елементів.

2. Надійність

- відсутність flaky tests;
- відтворюваність результатів;
- контроль стабільності мережевих викликів.

3. Масштабованість

- можливість додавання нових модулів без переписування ядра;
- підтримка структурованої архітектури.

4. Портативність

- робота під Windows, Linux, macOS;
- кросбраузерність (Chrome, WebKit, Firefox).

5. Зручність розробки

- зрозумілі інтерфейси;
- читабельний код;
- мінімальна крива навчання, щоб будь-який член команди тестування міг писатичи чи виправляти код.

Архітектурні принципи моделювання ФАТ

1. Модульність

Кожен компонент відповідає за власну функцію:

- базовий клас від якого будуть наслідуватись всі наступні сторінки;
- page objects (описує сторінку з її компонентами у вигляді локаторів та методів роботи з ними.
- утиліти (різні додаткові утиліти, наприклад по роботі з емейлами і тд);
- фікстури (в яких є заготовка та очищення тестових данних);

2. Ізоляція

Тести не повинні впливати один на одного:

- у Playwright - окремий browser context на тест;
- окремі фікстури для даних[4].

3. Повторне використання

Загальні дії винесено у:

- Page Object Model (POM);
- API-сервіси;
- хелпери та утиліти.

4. Конфігуровність

Фреймворк має підтримувати:

- змінні середовища (.env);
- перемикання оточень (dev/stage/prod);
- кастомні параметри запуску.

5. Прозорість та відтворюваність

Кожен тест:

- детермінований (чітко визначений та знає що перевіряє);
- містить логування;

- може бути відтворений локально та у CI.

6. Мінімум залежностей

Архітектура має уникати зайвої складності:

- без зайвих бібліотек;
- без надмірних абстракцій;
- з фокусом на стабільність.

7. Підтримка асинхронної моделі

Оскільки система базується на Node.js:

- усі операції виконуються у non-blocking mode;
- використовується async/await;
- фреймворк працює швидше за традиційні WebDriver-рішення.

Узагальнена модель ФАТ

1. Application Layer (може бути API або UI системи)

Тестований продукт - веб-застосунок, API, мікросервіси.

2. Interaction Layer (Playwright API)

Включає:

- Browser / BrowserContext;
- Page;
- Network Interception;
- APIRequestContext.

3. Test Logic Layer:

- Test Scenarios;
- Test Suites;
- Fixtures;
- Hooks.

4. Abstraction Layer (POM / Services):

- Page Object Models;
- API Services;
- Data Factories.

5. Utilities & Helpers:

- логування;
- генератори даних;
- утиліти (дата, фейкові дані, числа).

6. Configuration Layer:

- Playwright config;
- environment variables;
- secrets management.

7. Reporting & Debug Layer:

- HTML Reports;
- Screenshots;
- Videos;
- Trace Viewer.

8. CI/CD Integration Layer:

- запуск тестів у pipeline;
- збереження артефактів;
- паралельне виконання.

Таким чином, модель ФАТ слугує системним описом структури, поведінки та взаємодії компонентів тестового фреймворку. На її основі в наступних підрозділах буде побудовано архітектуру, структуру директорій, шаблони тестів, фікстури, утиліти та взаємодію з CI/CD-середовищем.

2.2 Проєктування структури ФАТ: застосування патерну Page Object Model (POM) та організація файлів проєкту

Упродовж розвитку програмної інженерії розробники стикалися з типовими архітектурними проблемами: як зменшити дублювання коду, як підвищити читабельність та розширюваність системи, як спростити супровід великих проєктів. Щоб вирішити ці проблеми, були сформульовані архітектурні патерни (design patterns) - універсальні рішення для типових задач проєктування[18].

У контексті автоматизованого тестування патерни виконують такі ключові функції:

- Стандартизують структуру проєкту, роблячи код прогнозованим та зрозумілим.
- Зменшують дублювання, оскільки повторювані елементи винесені у спільні модулі.
- Підвищують стійкість тестів, особливо UI-тестів, які схильні до ламкості.
- Спрощують масштабування: при додаванні нових тестів або функціональності архітектура дозволяє це робити без глобальних змін.
- Розділяють відповідальності, що робить тести більш чистими, а логіку централізованою.

Найважливішим і найпоширенішим патерном у веб-автоматизації є Page Object Model (POM).

Page Object Model (POM) - це архітектурний патерн, який передбачає винесення всієї логіки роботи з елементами сторінки у спеціальні класи (об’єкти сторінок). Замість того, щоб у кожному тесті описувати селектори та дії з ними, все це зберігається в одному місці у відповідному Page Object.

Таблиця 2.1 - Переваги паттерну Page Object

Переваги	Пояснення
Зменшення дублювання коду	Один і той самий елемент не описується у десятках тестів.
Висока читабельність	Тести виглядають як бізнес-сценарії, а не хаотичні строки повторюваного коду.
Централізація логіки	У разі зміни UI потрібно оновити лише відповідний Page Object.
Зниження ламкості	Тести не падають через дрібні зміни в DOM, якщо правильно використовувати локатори та методи.
Покращення підтримки	Новий QA може швидко зрозуміти структуру проєкту.

Розглянемо приклад тестування логін сторінки без використання пейдж обжект патерна (рис.12) та з ним (рис.13).

```
import { test, expect } from '@playwright/test';

test('User can log in (без POM)', async ({ page }) => {
  await page.goto('https://example.com/login');

  await page.fill('input[name="email"]', 'user@example.com');
  await page.fill('input[name="password"]', 'Password123!');
  await page.click('button[type="submit"]');

  await expect(page.locator('h1')).toHaveText('Dashboard');
});
```

Рис.12 Позитивний тест логін сторінки без Page Object.

Проблеми цього підходу

- Селектори дублюються у кожному тесті
- Будь-яка зміна UI потребує оновлення десятків тестів
- Тест містить одночасно бізнес-логіку і технічну реалізацію
- Неможливо масштабувати

А тепер розглянемо використання Page Object.

```
import { Page } from '@playwright/test';

export class LoginPage {
  constructor(private page: Page) {}

  emailField = this.page.locator('input[name="email"]');
  passwordField = this.page.locator('input[name="password"]');
  submitButton = this.page.locator('button[type="submit"]');

  async open() {
    await this.page.goto('https://example.com/login');
  }

  async login(email: string, password: string) {
    await this.emailField.fill(email);
    await this.passwordField.fill(password);
    await this.submitButton.click();
  }
}
```

Рис.13 Приклад описаного класу для сторінки Login Page.

Тепер у тесті ми можемо переглянути наскільки змінилось відображення структури. Тест містить зрозумілі верхньорівневі кроки, де тепер чітко встановлена межа між тестовим рівнев та рівнем опису сторінок[12].

```
import { test, expect } from '@playwright/test';
import { LoginPage } from '../pages/login.page';

test('User can log in (з POM)', async ({ page }) => {
  const loginPage = new LoginPage(page);

  await loginPage.open();
  await loginPage.login('user@example.com', 'Password123!');

  await expect(page.locator('h1')).toHaveText('Dashboard');
});
```

Рис.14 Відображення патерну Page Object в тесті.

Як результат, маємо такі висновки:

- Відсутність дублювання - локатор описаний один раз і у разі його зміни потрібно замінити лише в одному місці (у класі сторінки), а не оновлювати кожен тест мануально.
- Тест читається як бізнес-сценарій - open() → login() → expect().
- Масштабованість - додавання нових сторінок чи тестів не ламає структуру.

Патерн Page Object Model забезпечує структурованість, чисту архітектуру, розширюваність і стабільність фреймворку автоматизованого тестування. У поєднанні з Playwright та TypeScript він створює сучасний, надійний та підтримуваний підхід до автоматизації UI, який відповідає інженерним стандартам на рівні промислових систем.

2.3 Керування конфігураціями середовищ: детальний опис структури файлу конфігу та налаштування проєктів

У більшості інфокомунікаційних систем (ІКС) та вебзастосунків (ВЗ) середовища розгортання (Production, Staging, Development) мають суттєві відмінності: різні URL, різні API-ключі, відмінні параметри оточення та механізми автентифікації[18]. Тому сучасний ФАТ повинен мати можливість:

- Централізовано керувати середовищами;
- Швидко перемикатися між ними;
- Забезпечувати повторюваність результатів тестування;
- Дотримуватися принципів безпеки (не хардкодити таємні значення у коді).

У Playwright ключовим елементом, що забезпечує цю керованість, є файл playwright.config.ts. Правильне проєктування конфігураційної системи значно спрощує створення масштабованого та стійкого фреймворку.

Playwright дозволяє організувати конфігурації у вигляді єдиного файлу, який за допомогою логіки Environment Variables (змінних середовища) та механізму projects адаптується до будь-якого середовища та типу тестування.

Наступний приклад демонструє структуру конфігу, який використовується для даної роботи, з інтегрованою логікою завантаження змінних середовища (.env.dev, .env.stage):

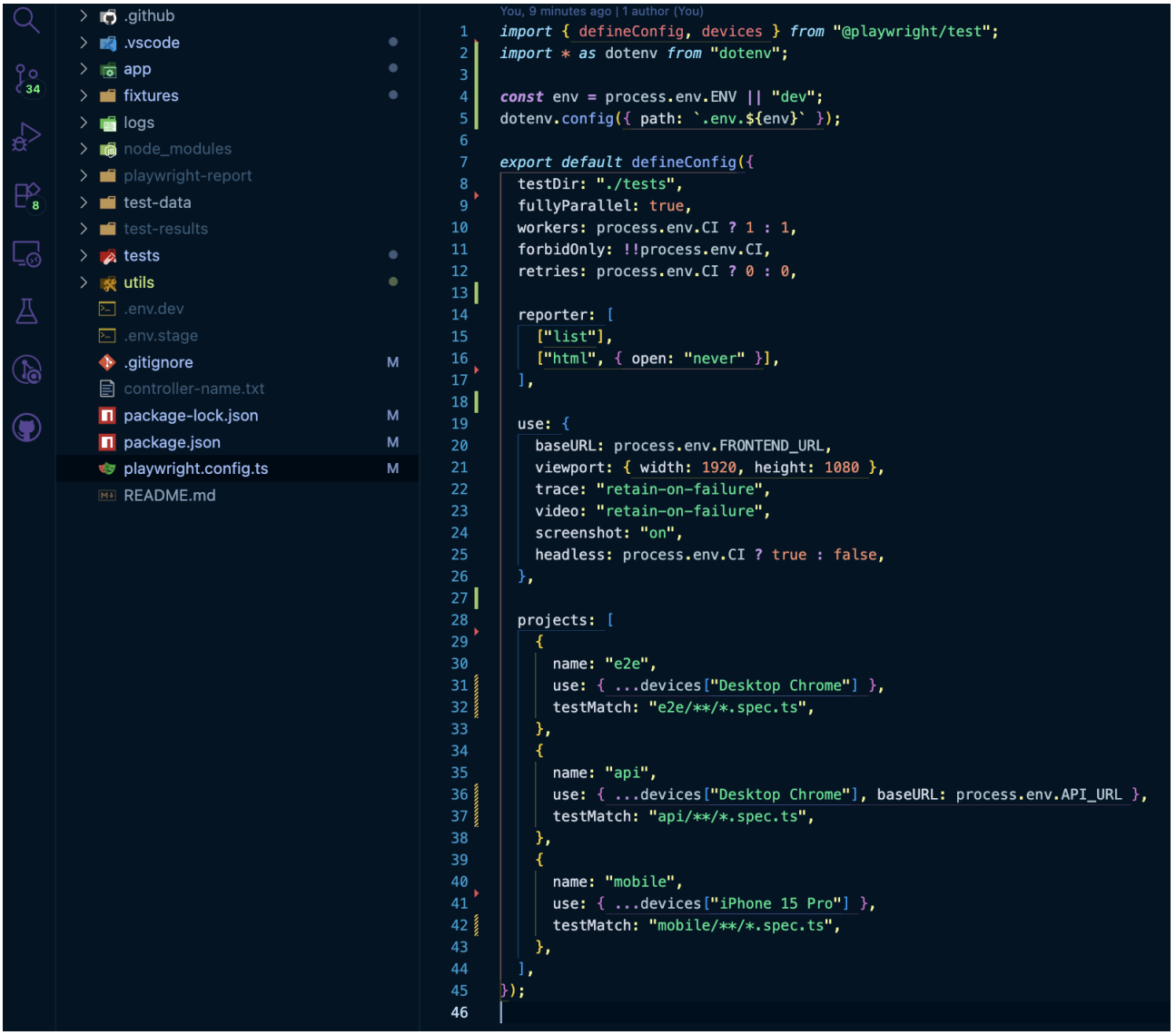


Рис.15 Структура конфігураційного файлу

Розберемо детальніше структуру конфігураційного файлу:

1. Динамічне завантаження .env файлів:

   Визначаємо, який env-файл використовувати dev чи stage:

   const env = process.env.ENV || "dev"; dotenv.config({ path: `.env.${env}` });

2. Дозволяє паралельне виконання тестів (потік не блокується):

   fullyParallel: true

3. Логіка для CI - якщо запущено в CI, зменшуємо workers та відключаємо retries:

   workers: process.env.CI ? 1 : 4,

forbidOnly: !!process.env.CI,

retries: process.env.CI ? 0 : 2,

4. Конфігурація репортів, у даному випадку вбудований HTML-репортер:
   reporter: [["list"], ["html", { open: "never" }]
5. Базовий URL для UI-тестів з .env файлу:
   baseURL: process.env.FRONTEND_URL,
6. Налаштування розмірів вікна браузера під час запуску тестів:
   viewport: { width: 1920, height: 1080 }
7. Налаштування трасування та зберігання відео зі скріншотами для певної умови, в нашому випадку записи потрібні лише для тестів які не пройшли перевірку, що економить ресурси:
   trace: "retain-on-failure",
   video: "retain-on-failure",
   screenshot: "only-on-failure",
8. Запуск браузера у хеадлес режимі на CI задля економії ресурсів:
   headless: process.env.CI ? true : false,
9. Механізм запуску різних сценраціїв тестів (десктоп, апі, мобайл):
   projects: [{ name: "e2e",
   use: { ...devices["Desktop Chrome"] },
   testMatch: "e2e/**/*.spec.ts",
   },

Механізм projects дозволяє розділити логіку запуску для різних сценаріїв. Наприклад e2e сценарії будуть запускатись лише для Chrome браузера для UI-тестів, тоді як для API тестів ми додаємо окремий базовий урл, а для мобайл тестів вказуємо також перелік девайсів.

Найкраща практика передбачає зберігання змінних конфігурації у файлах .env та їхнє завантаження під час запуску. Playwright чудово інтегрується з бібліотекою dotenv[4].

Файли .env: Створюються окремо для кожного середовища (.env.dev, .env.stage, .env.prod). Вони містять ключі, URL та інші данні які будуть повторюватись у кожному оточенні але для кожного будуть різні.

```
# Приклад .env.dev
FRONTEND_URL=https://dev.apitable.com
API_URL=https://dev.apitable.com/api/v1
API_KEY=dev-123-api-key
```

Рис.16 Приклад файлу .env для оточення розробки (dev)

Логіка завантаження: Як видно у коді вище, Playwright динамічно завантажує потрібний файл на основі системної змінної process.env.ENV (наприклад, npx playwright test --project=e2e --env=stage).

Керування конфігураціями середовищ є критичним компонентом архітектури ФАТ. Правильно організована система конфігурацій Playwright:

1. Забезпечує масштабованість тестового середовища.
2. Дозволяє швидко перемикатися між Dev/Stage/Prod.
3. Знижує ризики, пов’язані з безпекою.
4. Гарантує повторюваність і стабільність результатів тестів.

Завдяки гнучкому та модульному механізму playwright.config.ts, ФАТ на базі Playwright забезпечує високий рівень керованості, що дозволяє адаптувати фреймворк під складні корпоративні процеси.

2.4 Механізми підготовки тестового середовища: фікстури (Fixtures) та їхня роль у налаштуванні стану тестів

У контексті автоматизованого тестування, Фікстури (Fixtures) - це механізм, що використовується для встановлення та очищення початкового стану (state) тестового середовища перед виконанням тесту або набору тестів. Вони забезпечують, що кожен тест завжди виконується в ізольованому, чистому та передбачуваному середовищі, незалежно від результатів попередніх тестів.

Основне призначення фікстур у Фреймворку автоматизованого тестування (ФАТ):

1. Ізоляція: Гарантують, що тестовий випадок не має небажаних залежностей від інших тестів (наприклад, залишкових даних або стану сесії).
2. Повторюваність (Repeatability): Забезпечують, що тест дасть однаковий результат при багаторазовому запуску.
3. Ефективність: Дозволяють використовувати вже існуючі, дорогі в ініціалізації об'єкти (наприклад, об'єкт page браузера або авторизований API-клієнт) без необхідності створювати їх у кожному тесті.
4. Управління ресурсами: Фікстури відповідають за Setup (створення) та Teardown (очищення/знищення) ресурсів.

Playwright Test Runner має потужний і гнучкий механізм фікстур, який значно відрізняється від традиційних підходів (наприклад, @BeforeAll/@AfterEach у JUnit/TestNG). Фікстури Playwright реалізовані як функції, які динамічно вводяться в тестовий випадок (Dependency Injection)[5].

Типи фікстур у Playwright:

1. Built-in Fixtures (Вбудовані): Ресурси, надані Playwright "з коробки" (наприклад, page, browser, context, request, baseURL). Вони автоматично створюються та знищуються.
2. Custom Fixtures (Користувацькі): Створюються розробником ФАТ для налаштування специфічного середовища. Вони є основним інструментом для підготовки тестових даних та авторизації.

Архітектура створення користувацьких фікстур

Користувацька фікстура створюється за допомогою функції test.extend(), де визначається логіка її створення (Setup) та очищення (Teardown).

Ключові властивості:

- Асинхронність: Фікстури є асинхронними, що дозволяє їм виконувати I/O-операції, наприклад, запити до API або бази даних.
- Генератори (Generators): Фікстури використовують функцію-генератор, яка розділена ключовим словом yield (або викликом use()):
    - Код до yield (або use()): Виконується як Setup (створення ресурсу).
    - Код після yield (або use()): Виконується як Teardown (очищення/знищення ресурсу).

Роль Фікстур у налаштуванні стану тестів (UI та API)

Фікстури Playwright використовуються для реалізації гібридного підходу UI+API, який є центральною ідеєю даної роботи:

1. Підготовка Тестових Даних через API (Headless Setup)

Найбільш ефективний спосіб використання фікстур - це створення об'єктів (користувачів, проєктів, записів) через швидкі API-запити замість повільного проходження цих кроків через UI.

- Сценарій: Створення фікстури authenticatedUser, яка використовує вбудований об'єкт request для:

1. Надсилання запиту до /api/v1/auth/register для створення нового тестового користувача. (Setup)
2. Виконання тесту.
3. Надсилання запиту до /api/v1/users/delete для видалення користувача. (Teardown)

Фікстура авторизації для UI-тестування

Ця фікстура є похідною від стандартної фікстури page і використовується для автоматичного входу в систему.

Механізм:

1. Setup: Створюється користувач (через іншу API-фікстуру), отримується його токен.
2. Токен зберігається у сесії браузера (localStorage або cookie) за допомогою page.addInitScript() або context.storageState().
3. Тест виконується, починаючи вже з авторизованого стану.

Результат: Це значно скорочує час виконання тестів, оскільки усуває необхідність щоразу вводити логін і пароль через UI.

Playwright дозволяє керувати життєвим циклом фікстур (коли вони створюються та знищуються) за допомогою параметра scope:

- test (за замовчуванням): Створюється для кожного тесту та знищується після нього (найбільш ізольований).
- worker: Створюється один раз для кожного паралельного процесу та знищується після завершення всіх тестів цього процесу. Ідеально для створення об'єктів, що використовуються багатьма тестами (наприклад, чистий порожній проєкт).
- session: Створюється один раз для всіх тестів у всьому запуску. Використовується рідко, зазвичай для глобальної ініціалізації.

Отже, механізм Фікстур у Playwright є потужним інструментом для реалізації принципів якісного ФАТ. Вони забезпечують:

1. Автоматичний Teardown (очищення ресурсів).
2. Ефективну підготовку стану (через API-хелпери).
3. Високу ізоляцію та повторюваність тестів, що є критично важливим для надійності регресійних прогонів у CI/CD.

Саме завдяки фікстурам Playwright дозволяє створити ФАТ із чистою архітектурою, де логіка підготовки даних відділена від самого тестового сценарію[7].

2.5 Моделювання інтегрованого підходу UI+API: архітектура API-хелперів для ефективної підготовки тестових даних та авторизації

Моделювання інтегрованого підходу UI+API є центральним архітектурним рішенням у розробці сучасного Фреймворку автоматизованого тестування (ФАТ). Цей підхід забезпечує усунення головного недоліку традиційних End-to-End (E2E) UI-тестів: низька швидкість та нестабільність, спричинена тривалим підготовчим етапом.

Сутність інтегрованого (гібридного) підходу:

Традиційний E2E-тест виконує всі кроки, включно з підготовкою даних (наприклад, реєстрація, створення проєкту), через повільний інтерфейс користувача (UI). Інтегрований підхід передбачає стратегічне використання API-інтерфейсу для виконання початкових, невізуальних, але критично важливих завдань.

Інтегрований підхід - це (Швидкі API-хелпери для Setup) + (Надійне UI-тестування для перевірки користувацького досвіду).

Архітектура API-хелперів - відділення логіки підготовки від самого тесту.

Для реалізації цього підходу, логіка API-взаємодії виноситься в окремий шар — API-хелпери (або API-сервіси). Це забезпечує чисте відділення відповідальності (Separation of Concerns).

Принцип роботи API-хелперів:

API-хелпер - це клас або набір функцій (наприклад, у TypeScript), які використовують вбудовану фікстуру request Playwright для надсилання HTTP-запитів (GET, POST, DELETE тощо) до серверного API застосунку.

- Призначення: Вони не перевіряють функціонал API (це робиться в окремих API-тестах), а маніпулюють станом системи для потреб UI-тестів.
- Швидкість: API-запити виконуються без завантаження браузера, DOM та CSS, що робить їх виконання у десятки разів швидшим за UI-операції.

Приклад структури:

```
export class AuthController extends RequestHolder {
  @step("Login via API")
  async login(data: {
    email: string;
    password: string;
  }): Promise<LoginResponse> {      You, 9 months ago • added current progress …
    const loginResponse = await this.request.post(`${env.API_URL}/auth/login`, {
      data,
    });

    return loginResponse.json() as Promise<LoginResponse>;
  }

  @step("Sign up via API")
  async signUp(data: {
    name: string;
    email: string;
    password: string;
  }): Promise<void> {
    const res = await this.request.post(`${env.API_URL}/auth/sign-up`, {
      data,
    });

    if (!res.ok()) {
      const errorText = await res.text();
      throw new Error(`Sign-up failed: ${res.status()} ${errorText}`);
    }
  }
```

Рис. 17 Приклад структури API хелпера

Роль API-хелперів у підготовці тестових даних - це ефективна підготовка даних які є найдорожчим етапом тестування. API-хелпери вирішують цю проблему наступним чином:

1. Створення Об'єктів: Створення складних об'єктів (наприклад, 100 записів, проєкт із 5 таблицями) відбувається одним API-запитом, а не 100 кліками миші.
2. Забезпечення Чистоти: У фікстурах (Fixtures, див. Розділ 2.4) API-хелпери використовуються для очищення (Teardown): після завершення тесту вони надсилають DELETE-запит для видалення створеного об'єкта, гарантуючи чистоту наступного тестового запуску.

3. Фікстури-Фабрики: API-хелпери часто інтегруються з бібліотеками-фабриками (наприклад, Faker.js) для генерації випадкових, але валідних тестових даних (email, ім'я, адреса), які одразу передаються через API.

Найбільш критичний сценарій використання API-хелперів — це автоматична авторизація, яка забезпечує, що UI-тест завжди починається з валідного стану входу в систему.

1. Setup (API-хелпер): Тест використовує API-хелпер для:
    - Реєстрації нового користувача (POST /register).
    - Входу в систему (POST /login) та отримання авторизаційного токена (JWT/Session Cookie).
2. Збереження стану (Playwright): Отриманий токен або сесія записується у файл стану браузера за допомогою механізму storageState Playwright.
3. Використання у тесті: Фікстура authenticatedPage автоматично завантажує збережений стан перед запуском кожного тесту, ініціюючи вже авторизовану сторінку.

Це дозволяє уникнути повторного виконання UI-кроків входу в систему (введення логіна/пароля) у 95% тестових сценаріїв, що кардинально скорочує час виконання регресійного набору[4].

Отже, моделювання інтегрованого підходу UI+API на базі архітектури API-хелперів та механізму Playwright Fixtures є обов'язковою умовою для створення ефективного ФАТ:

1. Максимізація швидкості: Перенесення I/O-інтенсивних операцій (Setup/Teardown) з повільного UI на швидкий API.
2. Підвищення стабільності: Зменшення залежності тестів від візуальних елементів (локаторів) на етапі підготовки, що усуває крихкість.
3. Гнучкість: Можливість тестувати функціонал, доступний лише авторизованим користувачам, без необхідності щоразу проходити форму входу.

2.6 Аналіз цільового веб-застосунку: ключовий функціонал, що підлягає автоматизації, та визначення тестового покриття

Цільовим веб-застосунком для моделювання та впровадження ФАТ обрано **Strapi** (з відкритим вихідним кодом), сучасну Headless CMS, що базується на Node.js. Цей застосунок є репрезентативним прикладом складної інфокомунікаційної системи (ІКС), оскільки він поєднує адміністративний UI (ВЗ) та багатий на функціонал API для управління контентом.

Strapi має класичну двокомпонентну архітектуру, яка ідеально підходить для демонстрації гібридного підходу UI+API:

1. Адміністративний UI (Frontend): Надає інтерфейс для управління контентом (створення колекцій, редагування записів, керування користувачами та ролями). Це об'єкт End-to-End (E2E) UI-тестування (рис 18 та 19).
2. API (Backend): Основна логіка, побудована на Node.js. Надає RESTful та GraphQL ендпоінти для програмного доступу до контенту. Це об'єкт API-тестування та використання як API-хелпера.

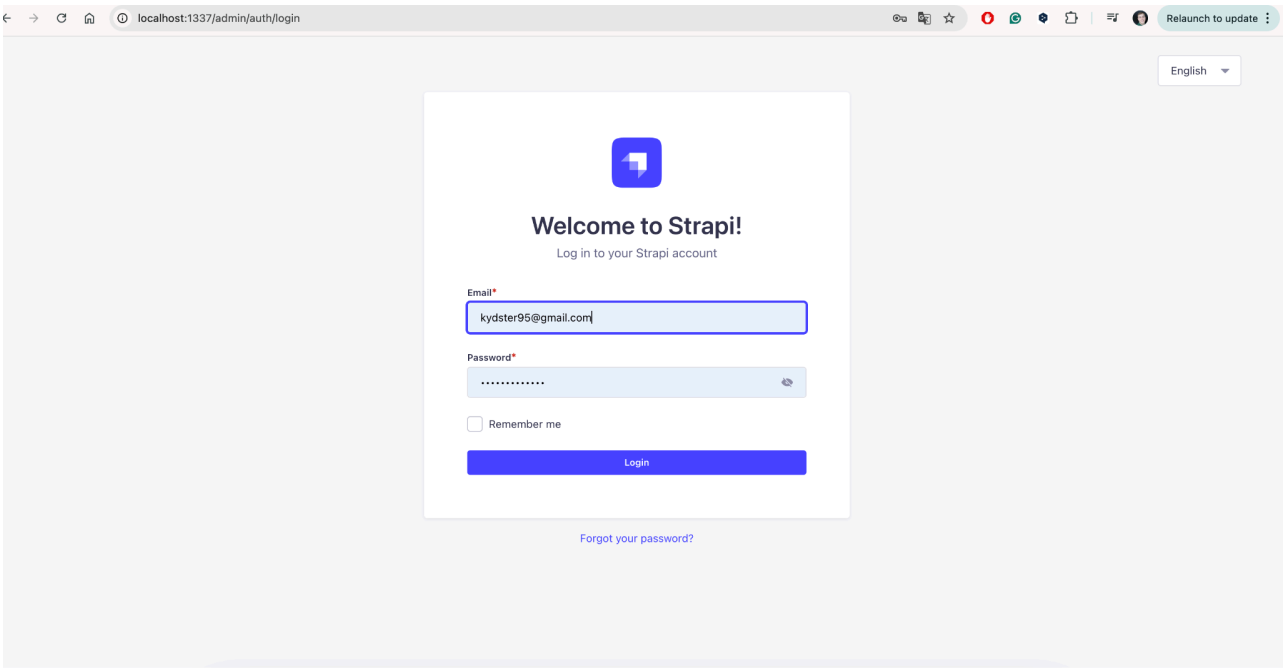


Рис.18 Вікно логіну застосунку

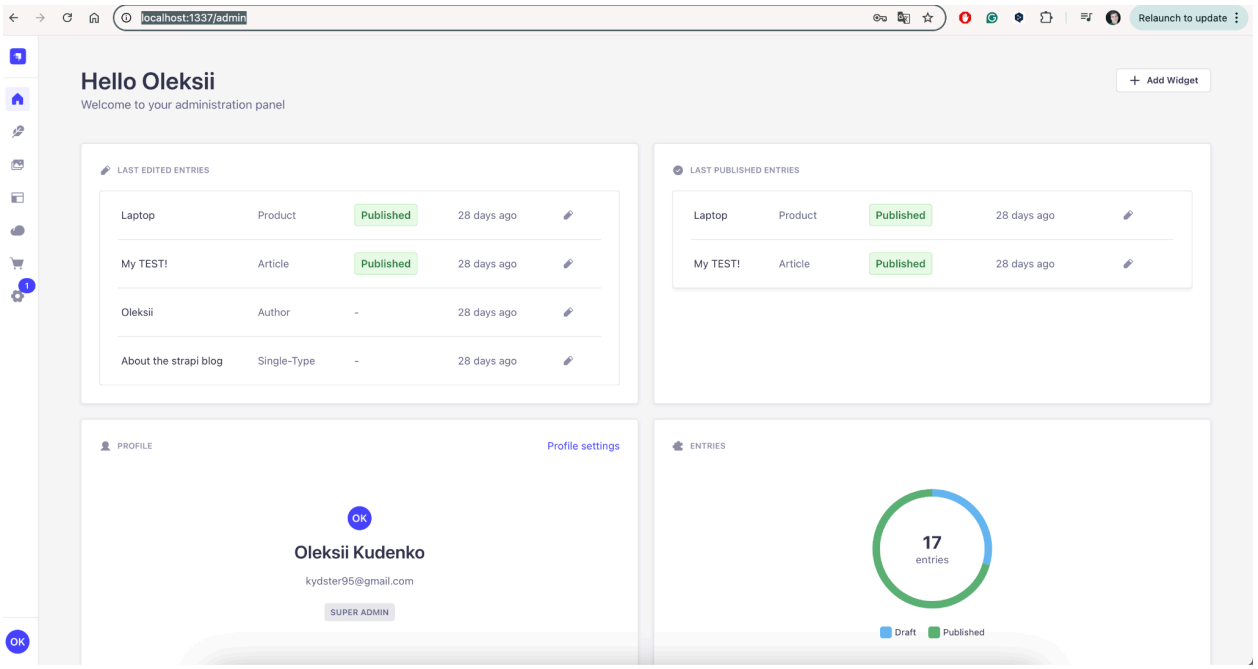


Рис. 18 Приклад головної сторінки адміністративної панелі.

Ключовий функціонал, що підлягає автоматизації:

Для визначення тестового покриття та демонстрації ефективності ФАТ (особливо гібридного підходу) необхідно зосередитися на найбільш критичних та повторюваних функціональних областях Strapi:

Таблиця 2.2 - Визначення функціоналу Адміністрування та Безпеки (Users & Permissions) для покриття автоматизацією.

Функціонал	Рівень тестування	Мета автоматизації
Реєстрація/Аутентифікація	API та UI	Перевірка коректного входу та виходу з системи. API-тест для отримання токена для фікстури authUser.
Керування ролями	UI	Створення, редагування та видалення ролей, перевірка застосування прав доступу (Permissions).
Контроль доступу	API	Перевірка, що користувач із роллю "Guest" не може отримати доступ до адмінських ендпоінтів (наприклад, /content-manager).

Таблиця 2.3 - Визначення функціоналу Контент-менеджмент (Content Management) для покриття автмоатизацією

Тип сценарію	Рівень тестування	Мета автоматизації
Створення колекції (Content Types)	UI	Тестування візуального конструктора схем: створення нової колекції з різними типами полів (Text, Boolean, Date).
CRUD-операці ї над записами	API та UI (E2E)	API: Демонстрація швидкості: створення 50-100 тестових записів. UI: Перевірка редагування та видалення одного запису через адміністративну панель.
Медіатека (Media Library)	UI (E2E)	Демонстрація роботи з файлами: Тестування завантаження файлів (зображень) та їхнє відображення.

Тому, аналіз Strapi як цільового застосунку підтверджує його ідеальну придатність для моделювання ФАТ. Його складна архітектура вимагає впровадження інтегрованого UI+API підходу, що є основною метою даної роботи, а обраний різноманітний функціонал дозволяє ефективно продемонструвати можливості розробленого фреймворку щодо покриття різних рівнів системи.

Висновки до розділу II

У Розділі 2 було виконано моделювання архітектури та структури Фреймворку автоматизованого тестування (ФАТ) на базі Playwright та TypeScript, що є необхідною передумовою для його подальшої ефективної реалізації. Прийняті архітектурні рішення забезпечують масштабованість, надійність та супроводжуваність системи.

1. Смодельовано архітектурну основу ФАТ, яка відповідає принципам ООП та використовує переваги асинхронної моделі Node.js. Визначено

функціональні вимоги до ФАТ, головною з яких є підтримка гібридного тестування UI та API для максимізації швидкості регресійного прогону.

2. Запроєктовано структуру ФАТ із застосуванням ключового патерну Page Object Model (POM). Визначено модульну організацію файлів проєкту, де логіка взаємодії з UI інкапсульована в Класи-Сторінки, а логіка підготовки даних винесена в окремі сервіси (API-хелпери) та фікстури.
3. Розроблено систему керування конфігураціями середовищ на базі файлу playwright.config.ts. Впроваджено механізм динамічного завантаження змінних із файлів .env та створено незалежні Playwright Projects (e2e, api, mobile). Цей підхід гарантує ізоляцію та повторюваність тестів у різних середовищах (Dev, Stage) та забезпечує безпечне керування некритичними змінними.
4. Смодельовано механізм підготовки тестового середовища через Фікстури (Fixtures) Playwright. Обґрунтовано їхню роль як ключового інструменту для Setup/Teardown (створення/очищення) ресурсів та забезпечення ізоляції тестових випадків.
5. Розроблено архітектуру інтегрованого підходу UI+API, де API-хелпери використовуються для виконання всіх I/O-інтенсивних операцій (авторизація, створення тестових даних). Це дозволяє уникнути повільних UI-кроків на етапі підготовки, що кардинально скорочує час виконання наскрізних тестів та підвищує їхню стабільність.
6. Проведено аналіз цільового веб-застосунку Strapi, який підтвердив його придатність для демонстрації інтегрованого підходу. Визначено ключові області функціоналу (Аутентифікація, CRUD-операції, Створення колекцій), які будуть використані для практичної демонстрації можливостей розробленого ФАТ у Розділі 3.

Прийняті архітектурні рішення створюють міцну, гнучку та високоефективну основу для подальшої реалізації та експериментального дослідження впровадженого ФАТ.

РОЗДІЛ 3. ВПРОВАДЖЕННЯ, АВТОМАТИЗАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ

3.1 Реалізація базової функціональності ФАТ з використанням Playwright та TypeScript (конфігурація, керування браузерами)

Першим кроком у реалізації ФАТ є налаштування середовища розробки Node.js та встановлення необхідних пакетів. Для забезпечення статичної типізації, що є архітектурним рішенням даної роботи, використовується TypeScript.

Підготувавши папку у якій будемо робити проект, відкриємо її через редактор коду (Visual Studio Code), та у терміналі введемо команду:

npm init -y

Цією командою ми ініціюємо створення проекту Node.js, після якого у нас додається конфіг файл package.json.Він містить метадані, необхідні для роботи npm (Node Package Manager), а саме:

1. Основна інформація: Назва проєкту, версія, опис.
2. Залежності (dependencies): Список усіх необхідних бібліотек і фреймворків, які будуть використовуватися у ФАТ (наприклад, @playwright/test, typescript).
3. Скрипти (scripts): Визначення користувацьких команд для запуску тестів, компіляції або ініціалізації ("test": "npx playwright test").

Опція -y (yes) вказує npm автоматично прийняти стандартні налаштування без запиту додаткової інформації, що прискорює ініціалізацію. Після цього кроку проєкт готовий до встановлення ключових залежностей, таких як Playwright[1].

Далі встановимо безпосередньо сам Playwright також через термінал. Всі інструкції до встановлення можемо знайти на їх офіційному сайті в розділі з документацією.

Отже, введемо наступну команду у терміналі і відповімо на питання під час встановлення пакетів:

npm init playwright@latest

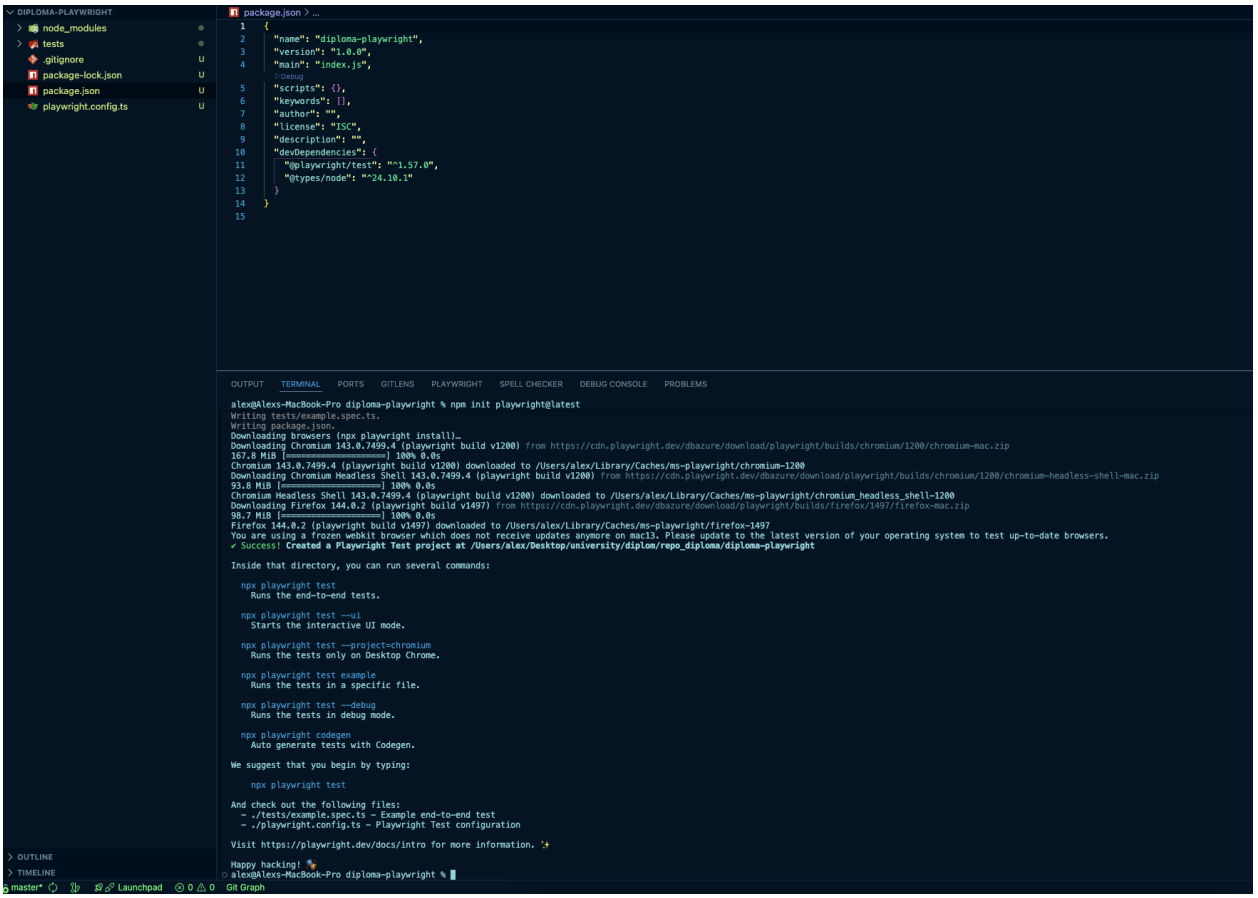


Рис.19 Процес встановлення Playwright через термінал

Як видно на рис.19 у нас з’явились нові файли та папки у нашому проекті, а саме:

1. **node_modules**: папка в якій ми зберігаємо усі модулі та бібліотекі які встановлені у нашому проекті.
2. **tests**: папка в якій є приклад тестів від команди Playwright для ознайомлення. Головна папка у якій будуть зберігатись наші тести.
3. **.gitignore**: файл, який використовується для стеження за файлами та папками які не повинні бути завантажені на віддалений репозиторій. Сюди відносяться будь-які безпекові данні для

нашого оточення (ключі, паролі юзерів та інша інформація яка може відрізнятись в залежності від виду запуску локально чи віддалено на CI, або данні які не повинні бути відбравлені до віддаленого репозиторія з точки зоруу безпеки).

4. **package.json**: головний конфігураційний файл проекту з точки зору Node.js, в якому вказані всі можливі залежності у проекті (бібліотеки, та їх внутрішні залежності каскадно), скрипти запуску, та загальна інформація про проект.
5. package.lock: це файл, що автоматично генерується менеджерами пакетів (як-от npm або Yarn), який фіксує точні версії всіх залежностей проєкту (включно з вкладеними), забезпечуючи відтворюваність збірки: щоб кожен розробник та CI/CD система отримували ідентичний набір пакетів, запобігаючи проблемам через оновлення залежностей в майбутньому[4].

Після цієї команди наш базовий функціонал готовий і тепер ми готові перейти до створення файлової системи, та додаткових налаштувань конфігурації, а також написання перших тестів задля перевірки того, що на данному єтапі ми вже можемо використовувати наш ФАТ[2].

В першу чергу змінимо головний конфіг файл під наші потреби, а саме:

- додамо конфігурацію для запуску наших тестів,
- середовище за замовчуванням у якому будемо виконувати тести (наприклад dev),
- внесемо дані про паралельний запуск,
- повторні запуски тесів при падінні,
- дані про артефакти (запис відео, скріншоти та трейс помилок),
- інформацію про headless режим,
- дані про звіти,
- інформацію про керування браузерами

playwright.config.ts U

playwright.config.ts > default

```ts
// playwright.config.ts
import { defineConfig, devices } from '@playwright/test';
import * as dotenv from 'dotenv';

// 1. Динамічне завантаження .env файлів:
// Використовуємо 'dev' як середовище за замовчуванням
const ENV = process.env.ENV || "dev";
dotenv.config({ path: `.env.${ENV}` });

export default defineConfig({
  // Загальні налаштування тестового раннера
  testDir: './tests',
  fullyParallel: true, // Дозволяє паралельне виконання тестів
  retries: 2, // Кількість повторних запусків при падінні

  // Налаштування для CI/CD
  workers: process.env.CI ? 1 : undefined, // У CI обмежуємо кількість процесів

  // Секція use: глобальні налаштування для всіх тестів
  use: {
    // 2. Використання змінних середовища
    baseURL: process.env.FRONTEND_URL, // Базовий URL з файлу .env

    // Керування артефактами
    trace: 'retain-on-failure',
    video: 'retain-on-failure',
    screenshot: 'only-on-failure',

    // Керування Headless-режимом
    headless: process.env.CI ? true : false, // Headless у CI, false локально
  },

  // Керування звітністю
  reporter: [
    ['list'],
    ['html', { open: 'never' }]
  ],

  // 3. Налаштування проєктів для керування браузерами (кросбраузерність)
  projects: [
    {
      name: 'Chromium',
      use: { ...devices['Desktop Chrome'] },
    },
    {
      name: 'Firefox',
      use: { ...devices['Desktop Firefox'] },
    },
    {
      name: 'Webkit',
      use: { ...devices['Desktop Safari'] },
    },
  ],
});
```

Рис. 20 Зміст конфіг файлу після редагування.

Для початку, напишемо перший простий тест, який перевіре що система працює справно. Напишемо позитивний тест який буде перевіряти що вхід у систему виконано корректно. Відповідно, наш тест буде в папці: /tests/login.spec.ts

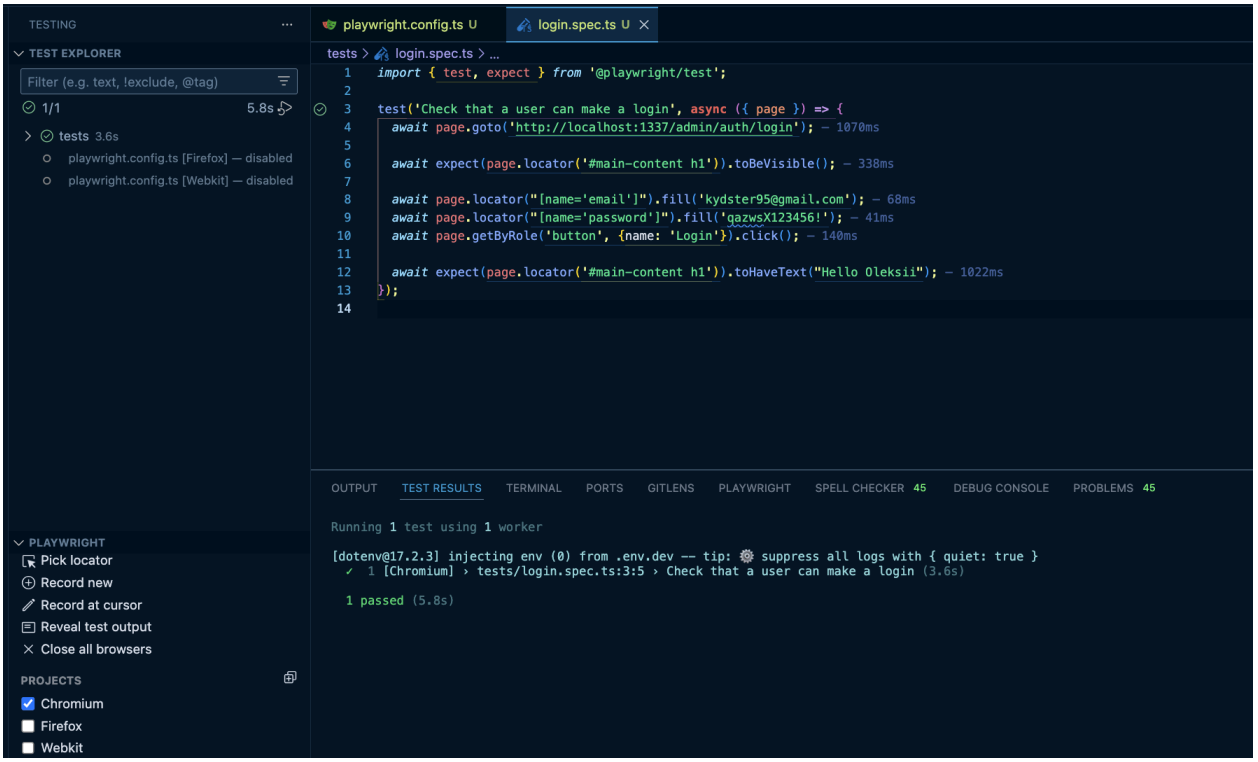


Рис. 21 Код першого тесту який успішно завершений у браузері Google Chrome

Як видно з рис.21 ми написали базовий тест на перевірку того що юзер може зайти у систему. Він успішно виконався за 5.8 секунди, що є доволі швидко. Окрім репорту у консолі можемо переглянути html репорт до відповідного тест запуску командою:

npx playwright show-report

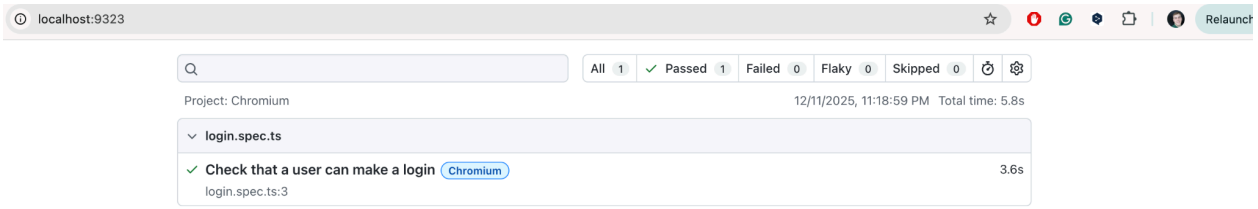


Рис.22 Загальний вигляд html репорту

На рис.22 ми можемо переглянути загальну інформацію про тестовий запуск, а саме:

- Які тести були запущені,
- На яких браузерах (проектах),
- Яка кількість була запущена, успішно пройдена, не пройдена, кількі-сть флакі тестів (нестабільних),
- Дані про дату запуску та час виконання.

Клікнувши на сам тест, ми можемо перейти безпосередньо до його кроків та артефактів які ми зберігли для історії.

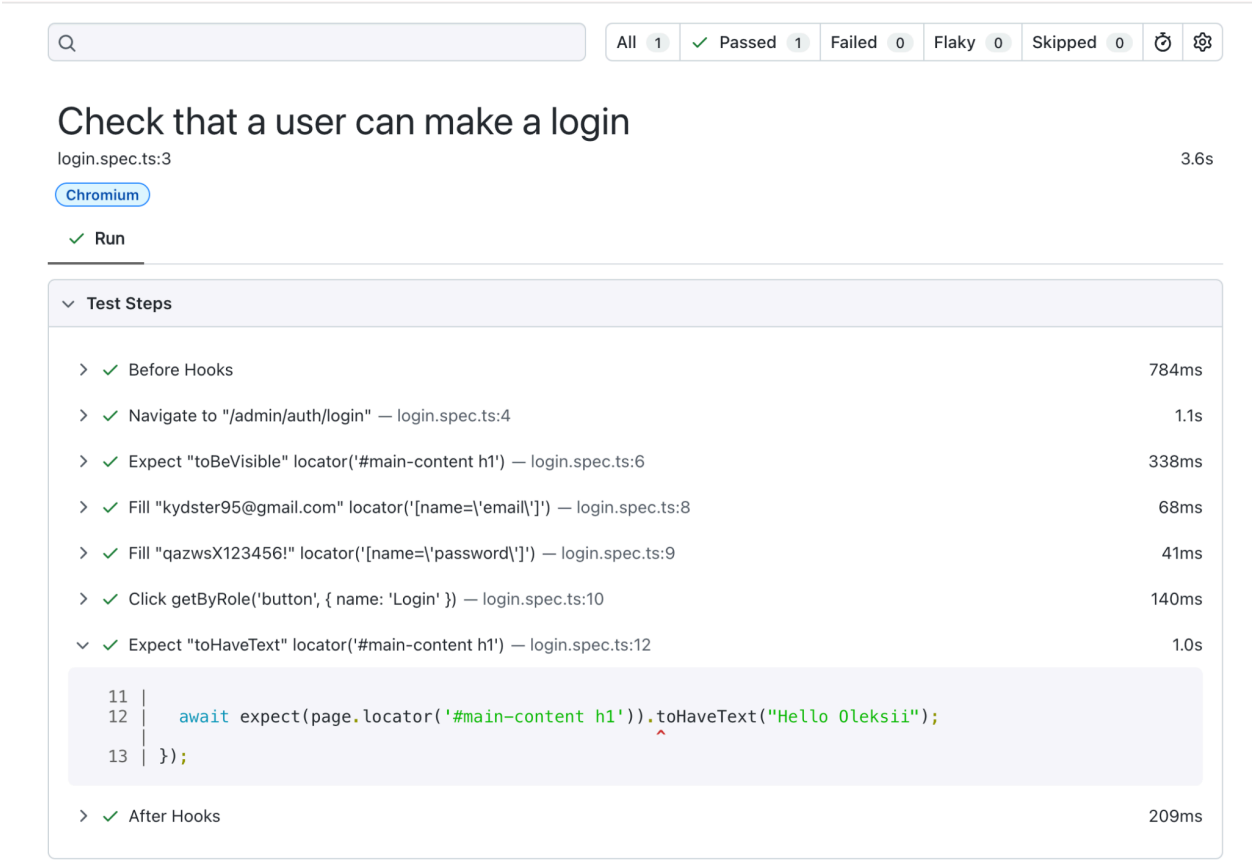


Рис.23 Вікно відкритого тесту у html репорті

Тут ми вже детально можемо переглянути кожен крок тесту, які локатори та функції були задіяні на кожному кроці та скільки часу зайняв кожен окремний крок.

Для чистоти експерименту, спробуємо зайти у систему з неправильними данними та подивимось як це буде відображатись у тесті та репорті. Для цього можемо або змінити очікуваний результат, або ввести некорректні дані для входу в систему.

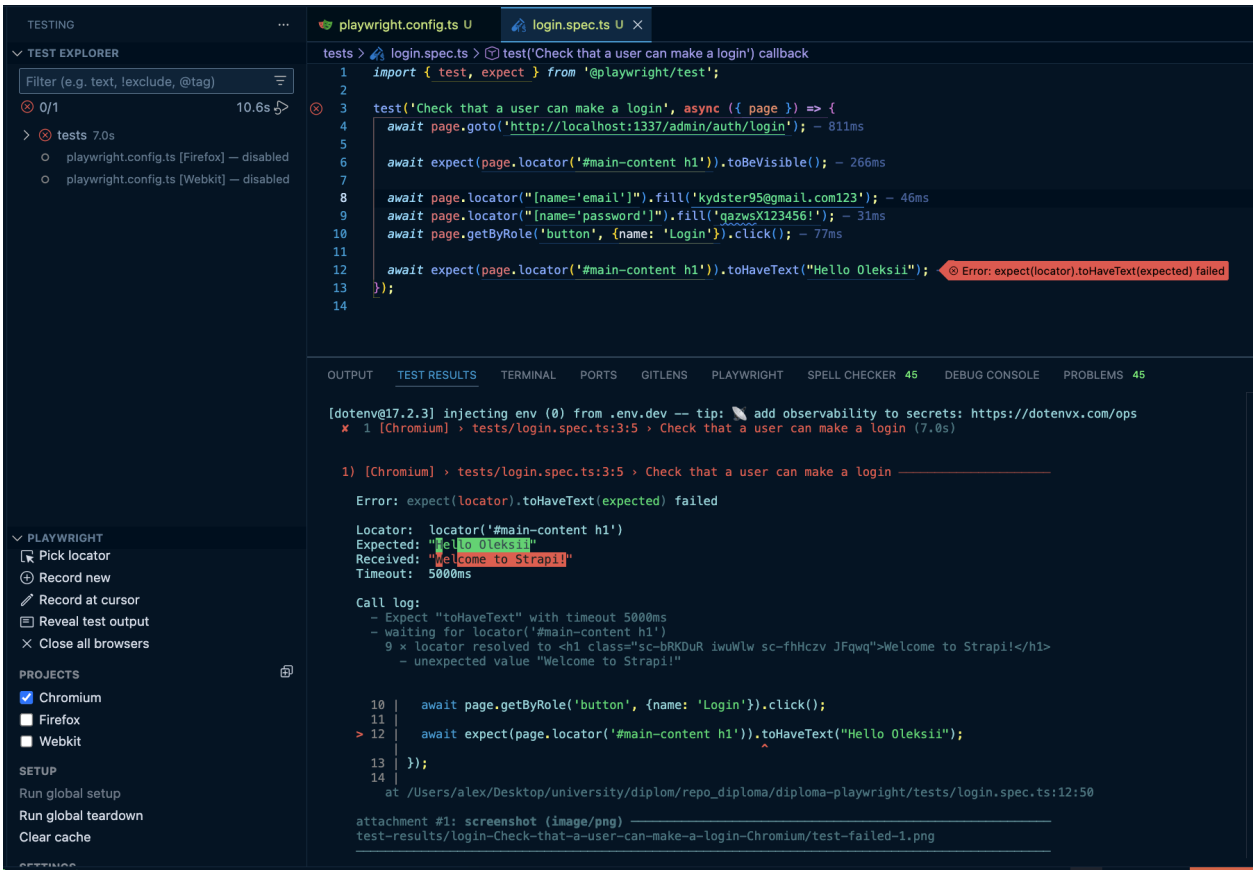


Рис.24 Результат запуску тесту який завершився помилкою

Як бачимо, репорт в терміналі чудово показує місце помилки, та що саме пішло не так, коли ми очікували перевірку того, що загловок головної сторінки матиме текст “Hello Oleksii”, але за цим локатором він знайшов текст “Welcome to Strapi”, що свідчить про невдалу спробу авторизуватися у системі, бо юзер залишився на сторінці логіну і не був пропущений далі.

Також переглянемо детальниій звіт тесту.

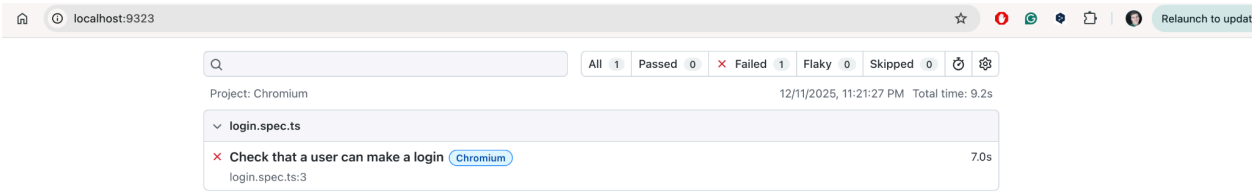


Рис.25 Загальний результат html репорту

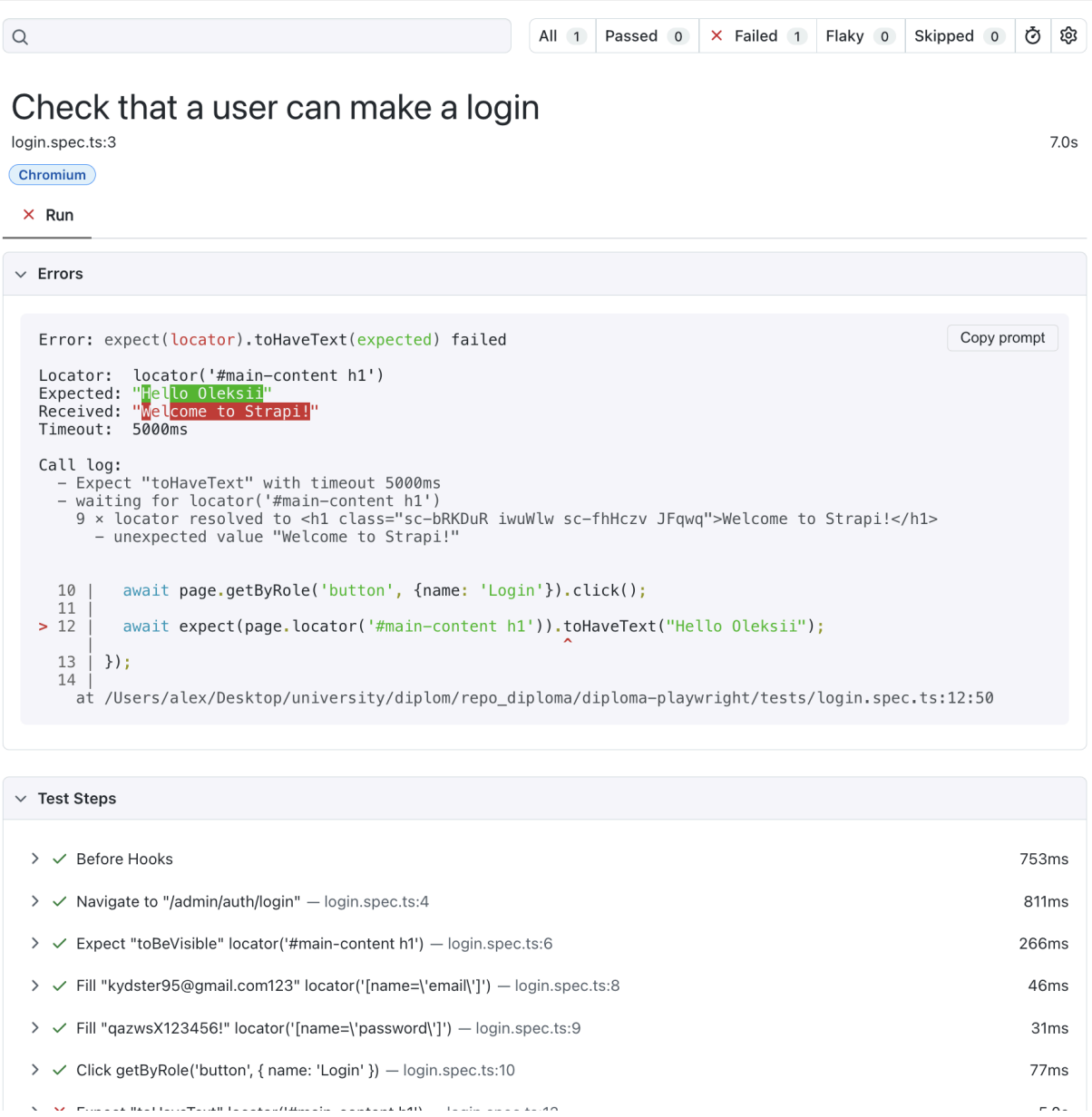


Рис.26 Детальнрий результат html репорту конкретного тестуу

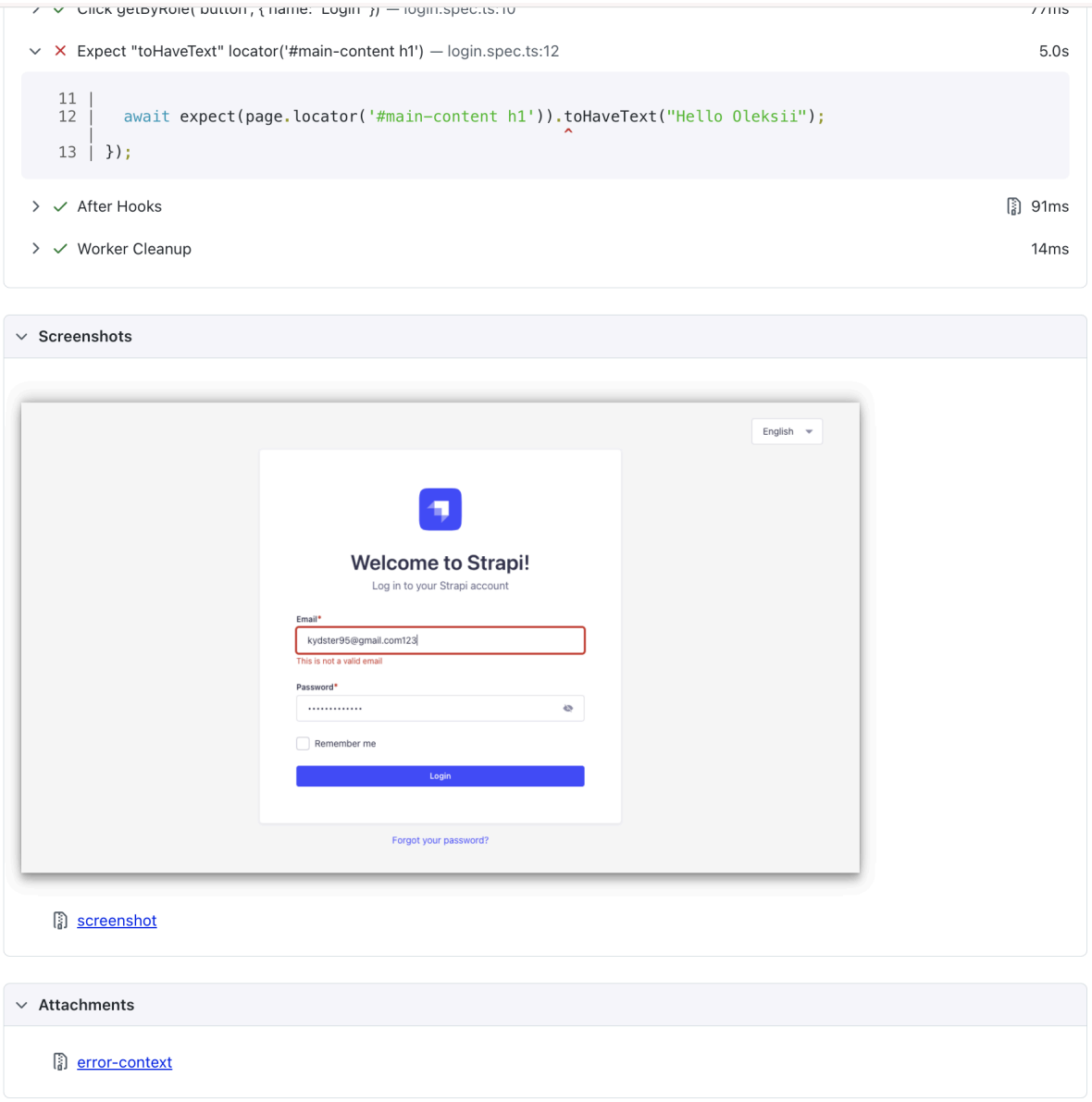


Рис. 27 Приклад прикладеного скріншоту на кроці в якому тест впав.

Як видно з прикладу вище, ми можемо детально прослідкувати місце у якому тест завершився помилкою, а якщо ми увімкнули скріншоти чи запис відео у конфіг файлі, що є одією з головних потреб у сучасних ФАТ.

Далі, можемо покращити наш тест застосуваши page-object підхід, тим самим розділивши логіку тесту від логіку взаємодії зі сторінкою.

Створимо файлову структуру з нашими сторінками:

/app/pages:

- appComponent.ts (де зберігається базовий клас),

- home.page.ts (головна сторіка застосункуу, а саме адмін панель),
- signIn.page.ts (сторінка авторизації)

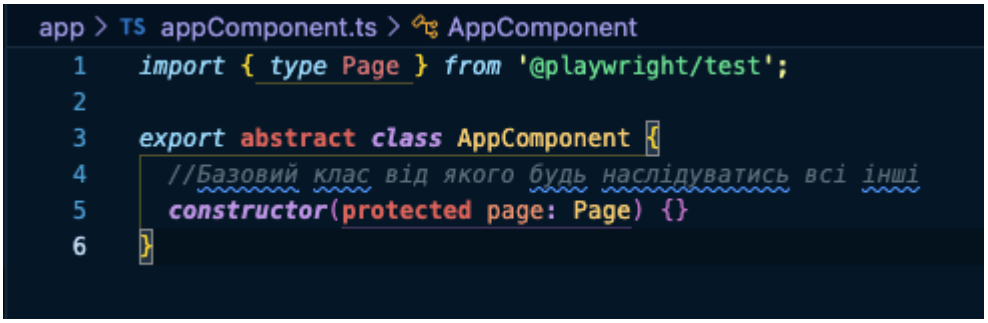


Рис.28 Приклад абстрактного базового класу

Базовий клас ідеально підходить для наших потреб, тут ми будемо просто створювати один раз констркутор класу який всі інші сторінки будуть наслідувати, таким чином ми вже економимо значну частину написання коду, що робить проект чистішим та більш зрозумілим для всіх членів команди.

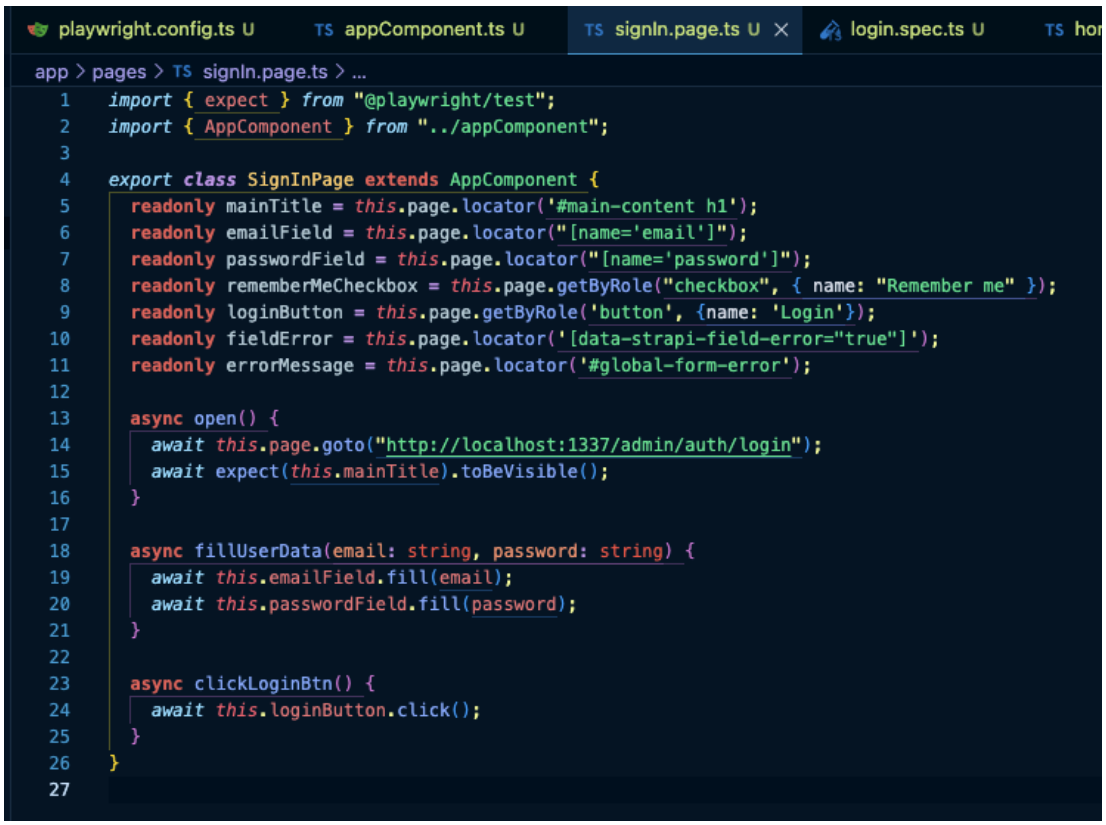


Рис.29 Приклад сторінки логіну

На рис.29 ми описали нашу сторінку авторизації, а саме написали локатори (readonly), які є ідентифікаторами єлементів на веб сторінці, та методи для взаємодії з єлементами на цій сторінці (наприклад open()).

Таким чином, ми застосували page-object патерн, який значно покращує наш код за рахунок:

- Зменшення дублювання коду: елементи (локатори) та методи взаємодії з конкретною сторінкою інкапсульовані в одному класі. Це означає, що вам не потрібно повторно писати ті самі локатори у кожному тесті.

- Покращеної зручності обслуговування: якщо UI елемент на сторінці змінює свій локатор (наприклад, замість [name=’email’] він стає [name=’emailField’], вам потрібно внести зміни лише в одному місці - у відповідному класі Page Object (наприклад, SignInPage). Усі тести, які використовують цей клас, автоматично почнуть працювати з новим локатором[4].
- Кращої читабельності тестів: сам тестовий файл (наприклад, login.test.ts) стає чистим і сфокусованим виключно на логіці тестування (наприклад, "користувач відкриває сторінку -> заповнює дані -> натискає увійти -> перевіряє заголовок"). Деталі реалізації взаємодії з UI приховані всередині Page Objects.

Тепер сам тест виглядає лаконічніше, ми можемо перевикористовувати наші методи та локатори в різних тестах без повторного дублювання коду як на рис.30

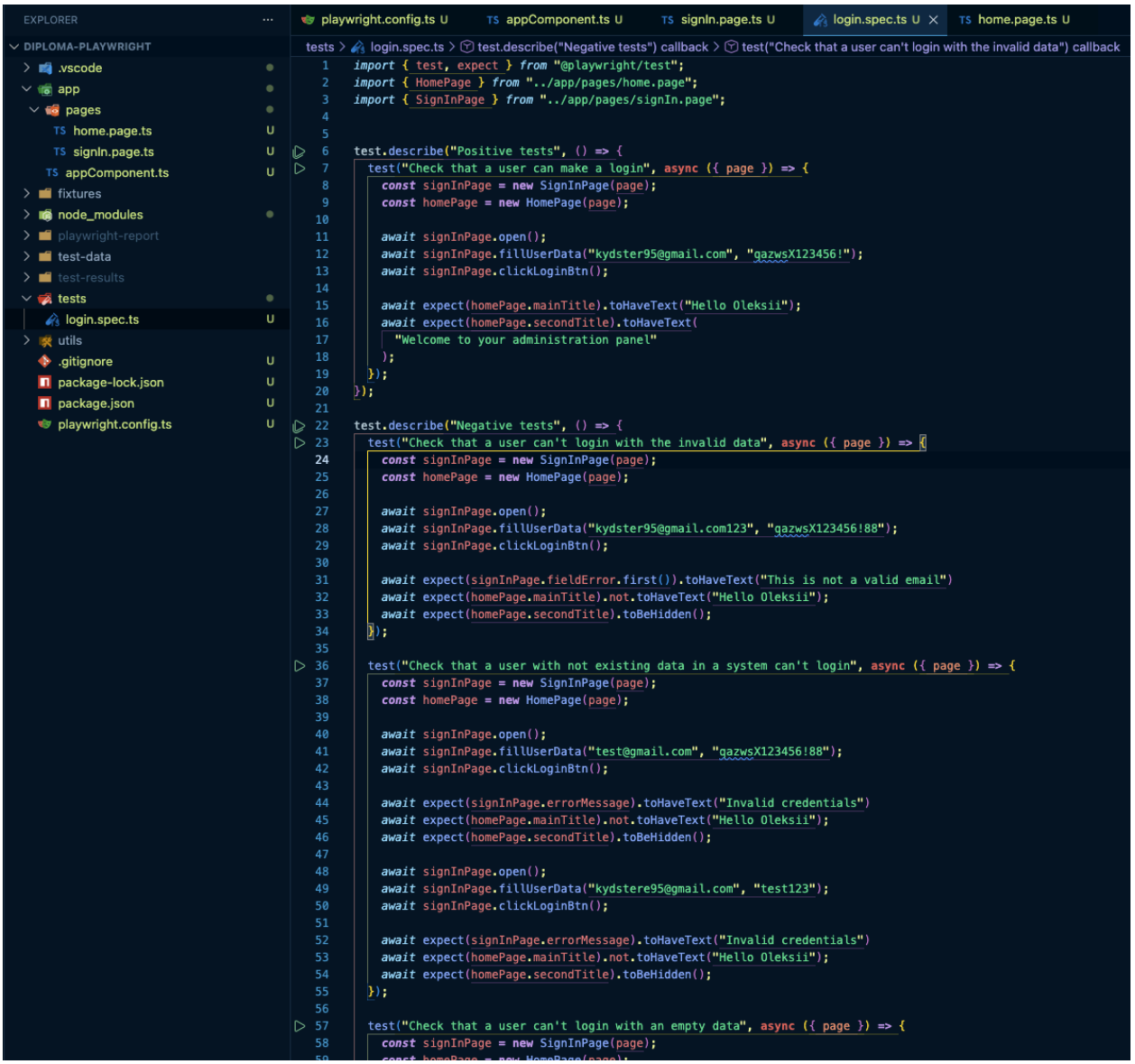


Рис.30 Приклад оновлених тестів до сторінки авторизації.

Також можемо запустити ці тести і подивитись як вони відпрацюють у різних браузерах.

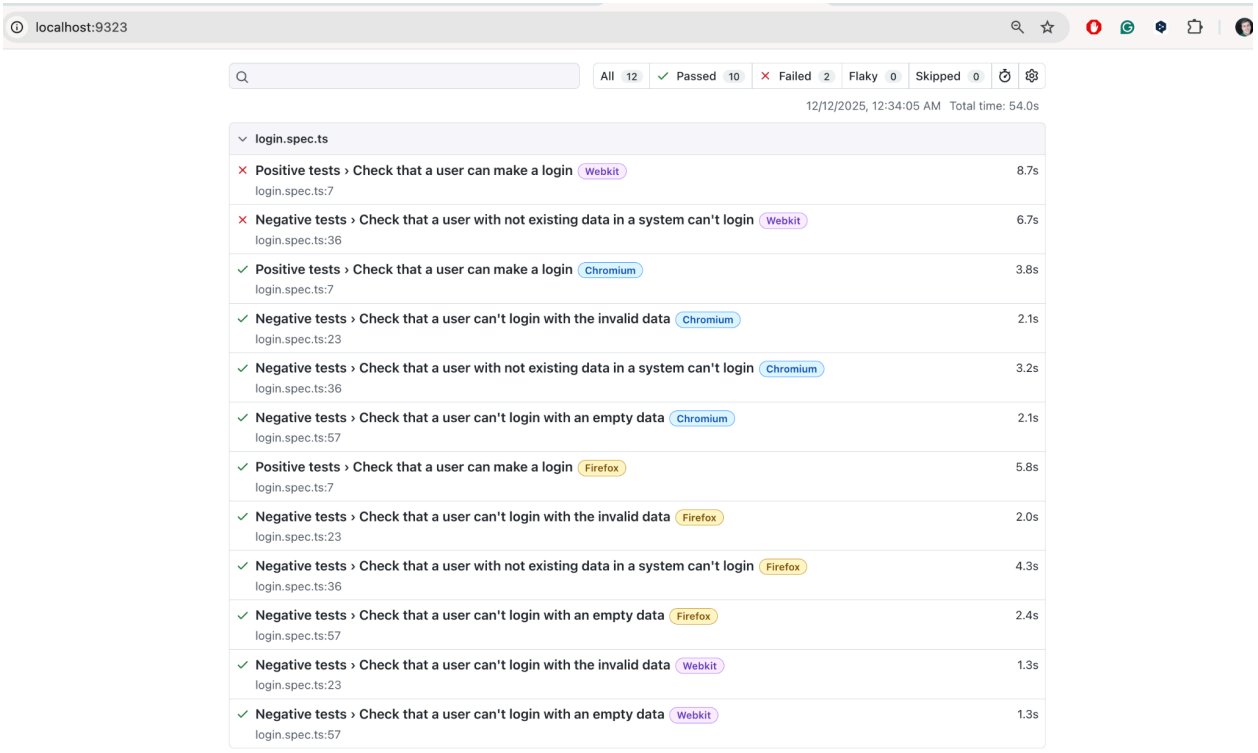


Рис.31 Запуск тестів на декількох браузерах

Як видно з результатів, наші тести відпрацювали на трьох браузерах, а саме Google Chrome, Firefox, Webkit (Safari). Однак два з них завершились помилкою. Проаналізувавши помилку, ми можемо прийти до висновку, що проблема була не в наших тестах, а нашому тестовому застосунку, тому що у нього є механізм від ддос атак, який спрацював саме на наших тестах. Застосунок вважав, що ми є роботом який намагається його зламати через велику кількість одночасних входів у систему. Все це ми зрозуміли завдяки зрозумілому репорту, з описом проблеми (рис.32) та доданим скриншотом (рис.33)[6].

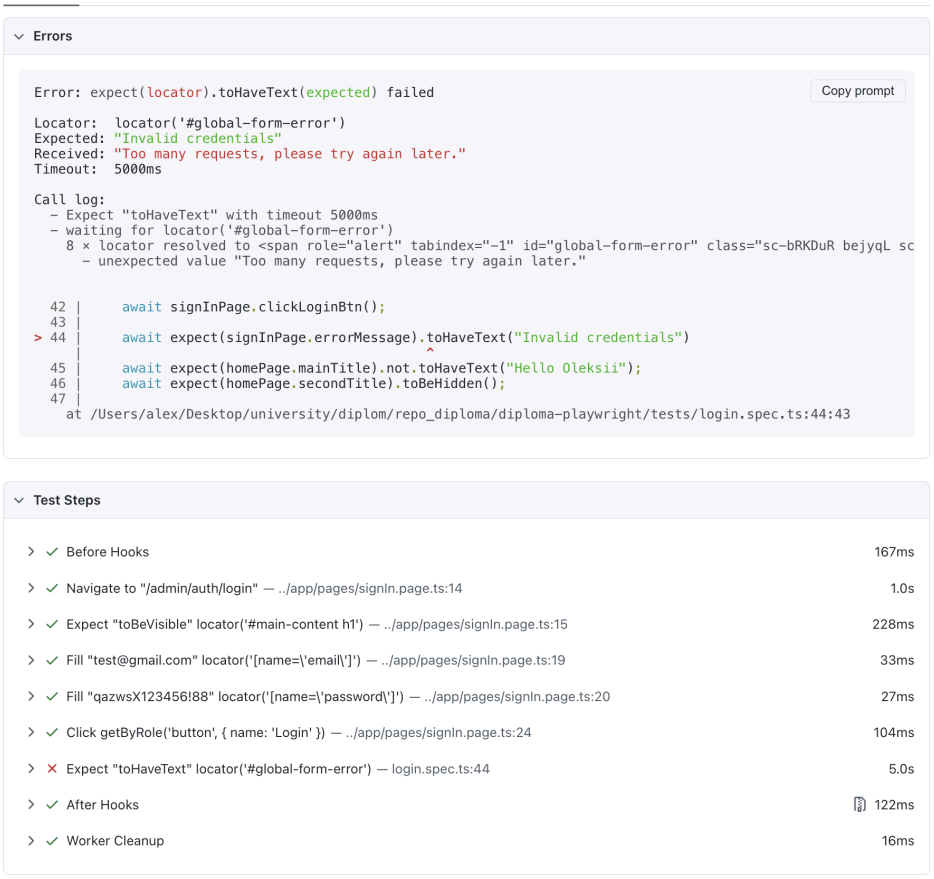


Рис.32 Детальний огляд тесту в Сафарі браузері який завершився помилкою

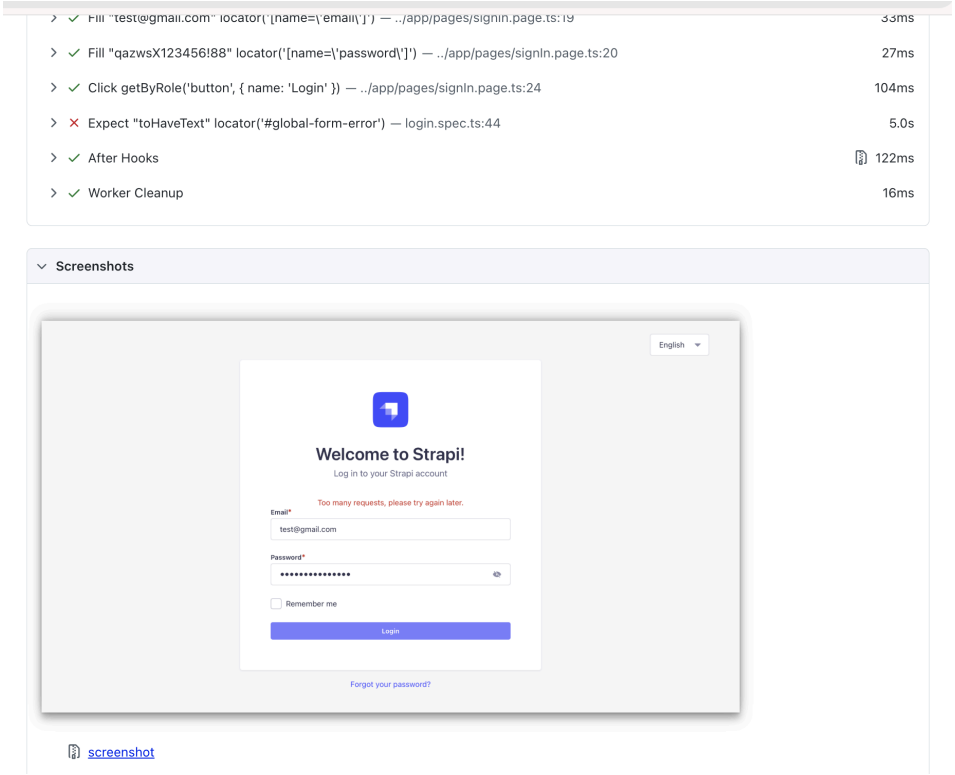


Рис.33 Вигляд доданого скрішноту на кроці з помилкою

3.2 Реалізація модулів API-тестування: перевірка коректності обробки даних та бізнес-логіки

Перед тим як почати роботу з АПІ, поглянемо на новий юай тест та подивимось які в ньому є проблеми.

Рис.34 Новий тест який перевіряє створення продукту у системі

Як показано у наданому прикладі тесту, кожен тестовий випадок (наприклад, Check admin can create new product) вимагає попередньої авторизації, яка виконується через UI у блоці test.beforeEach. Виконання цих трьох UI-кроків перед кожним тестом призводить до: збільшення часу виконання (до 5-10 секунд на тест) та високої крихкості.

Для вирішення цієї проблеми, ми створимо кастомну фікстуру existingUser, яка інкапсулює логіку авторизації, забезпечуючи виконання тесту одразу з авторизованого стану.

Для початку взагалі переконаємось, що ми можемо працювати з АПІ і створимо простий апі тест, який перевірятиме, що ми можемо авторизуватись з даними юзера який вже є в системі.

Рис.35 АПІ тест на перевірку входу у систему

Як бачимо на рис.35 ми можемо здійснити вхід у систему через АПІ, що набагато швидше ніж робити всі кроки через юай, який до того ж може бути не завжди стабільним, що понетційно збільшує ризики флакі тестів.

Тому процес авторизації ми винесемо на АПІ рівень але перед цим, зробимо покращення для використання АПІ як ми робили з page-ogject підходом - підготуємо окремі методи для роботи з АПІ в окремих директоріях[4].

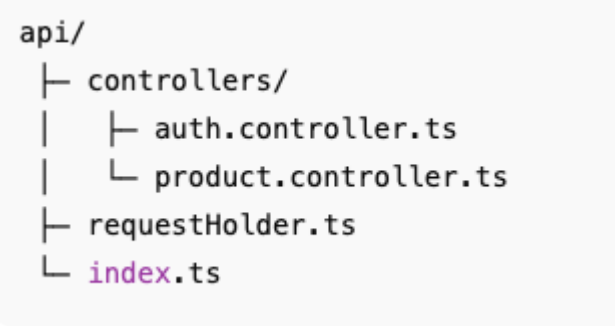


Рис.36 Файлова структура API модуля

У файлі index.ts реалізовано єдину точку доступу до API-контролерів:

```
export class API extends RequestHolder {
    public readonly auth = new AuthController(this.request);
    public readonly product = new ProductController(this.request);
}
```

Абстрактний клас RequestHolder інкапсулює об’єкт APIRequestContext, що надається Playwright, та забезпечує спільний доступ до HTTP-запитів для всіх контролерів:

```
export abstract class RequestHolder {
    constructor(protected request: APIRequestContext) {}
}
```

Такий підхід дозволяє:

- централізувати роботу з HTTP-запитами;
- уникнути дублювання коду;
- легко розширювати API-шар новими контролерами;
- підтримувати єдиний стиль взаємодії з бекендом.

Після підготовки API-шару було реалізовано кастомну фікстуру existingUser, яка інкапсулює логіку автенфікації користувача.

Основне призначення фікстури:

- виконати API-логін;
- отримати токен доступу;
- ініціалізувати браузер у вже авторизованому стані;
- надати токен іншим фікстурам та тестам.

Фікстура виконує авторизацію один раз, після чого тестові сценарії стартують одразу з головної сторінки адміністратора без необхідності проходження UI-логіну.

Таким чином, кожен тест отримує:

- готове середовище;
- валідну сесію користувача;
- стабільний початковий стан.

Для запобігання повторним запитам на авторизацію реалізовано механізм кешування токена в класі Application. Токен зберігається у змінній cachedToken та повторно використовується у межах виконання тестів.

Метод bootstrapAdminAuth виконує ініціалізацію браузерного середовища шляхом встановлення необхідних значень у localStorage, після чого відбувається перехід безпосередньо до адміністративної панелі.

Даний підхід дозволяє:

- зменшити кількість API-запитів;
- уникнути конфліктів сесій;
- забезпечити стабільну авторизацію в UI-тестах.

Рис.37 Вигляд файлу з фікстурами

Рис.38 Вигляд app файлу в якому відбувається ініціалізавція сторінок та базові методи для роботи з авторизацією які використовуються у фікстурі

Додатковою перевагою впровадження API-рівня є можливість підготовки тестових даних без використання UI.

У межах фреймворку реалізовано фікстуру product, яка:

- створює тестовий продукт через API перед виконанням тесту;
- передає ідентифікатор створеної сутності у тест;

- автоматично видаляє тестові дані після завершення сценарію.

Такий підхід забезпечує:

- повну ізольованість тестів;
- відсутність «сміттєвих» даних у системі;
- можливість перевірки бізнес-логіки без впливу UI;
- суттєве скорочення часу виконання тестового набору.

Як результат у нас є приклад запуску 5 тестів без фікстури, де авторизація відбувається на юай рівні і тест займає **49.9** секунд.

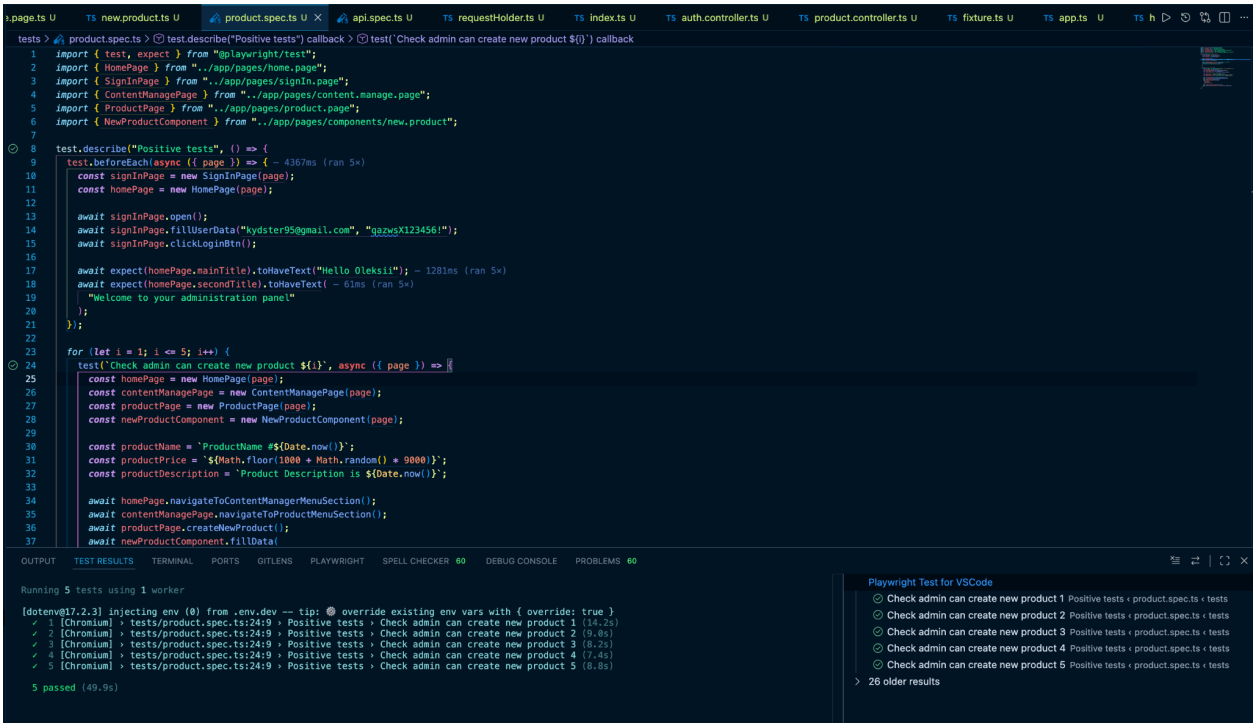


Рис.39 Вигляд і результат запуску тестів без фікстури

Нижче, на рис.40 ми маємо приклад використання тесту з фікстурою. Як результат:

- тести пройшли за 30.5 секунд,
- тест виглядає набагато лаконічніше бо ми прибрали як роботу з beforeEach хуками на рівень фікстури так і роботу з ініціалізацією

сторінок.

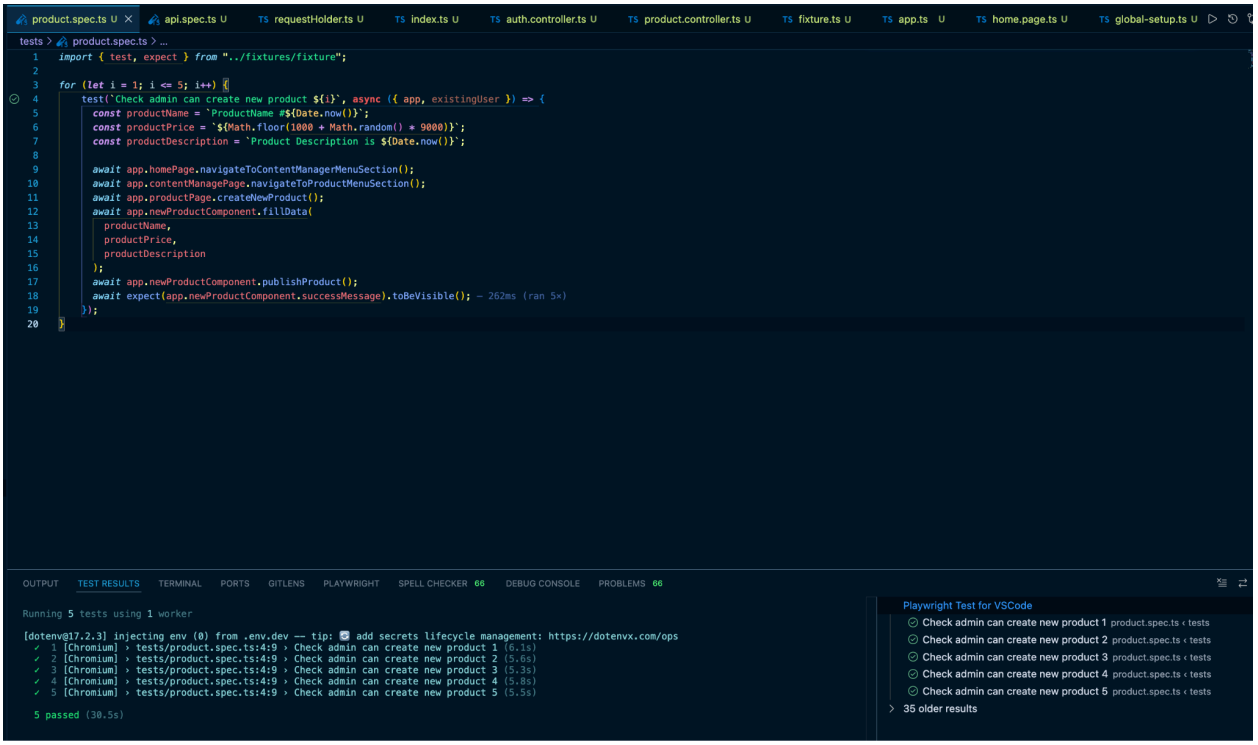


Рис.40 Результат застосування фікстури

Розглянемо інший приклад. У нас є тест який перевіряє що 10 продуктів відображуються на сторінці. Якщо створювати через графічний інтерфейс 10 продуктів, то це знову призведе до звичних проблем - час на створення та потенційна нестабільність.

Винесемо все це в окрему фікстуру product, яка буде створювати по АПІ 10 продуктів. На рис.41 ми бачимо лаконічно написані тести, а на рис.42 видно результат, що перед тестом ми авторизуємось, запускаємо фікстуру створення продуктів, потім відбувається сам тест, а в кінці очищення тестових створених даних, в нашому випадку продуктів.

```
tests > product.spec.ts > ...
1  import { test, expect } from "../fixtures/fixture";
2
3  for (let i = 1; i <= 5; i++) {
4    test(`Check admin can create new product ${i}`, async ({ app, existingUser }) => {
5      await app.homePage.navigateToContentManagerMenuSection();
6      await app.contentManagePage.navigateToProductMenuSection();
7      await app.productPage.createNewProduct();
8      await app.newProductComponent.fillData(
9        `Product ${Date.now()}`,
10       `${Math.floor(1000 + Math.random() * 9000)}`,
11       `Description ${Date.now()}`
12     );
13     await app.newProductComponent.publishProduct();
14     await expect(app.newProductComponent.successMessage).toBeVisible();
15   });
16 }
17
18 test("Check that we can see 10 products on the first page", async ({ app, existingUser, product }) => {
19   await app.homePage.navigateToContentManagerMenuSection();
20   await app.contentManagePage.navigateToProductMenuSection();
21   await expect(app.productPage.publishedCells).toHaveCount(10);
22 });
23
```

Рис.41 Лаконічний вигляд тестів, де вся логіка підготовки та очищення даних винесена у фікстури

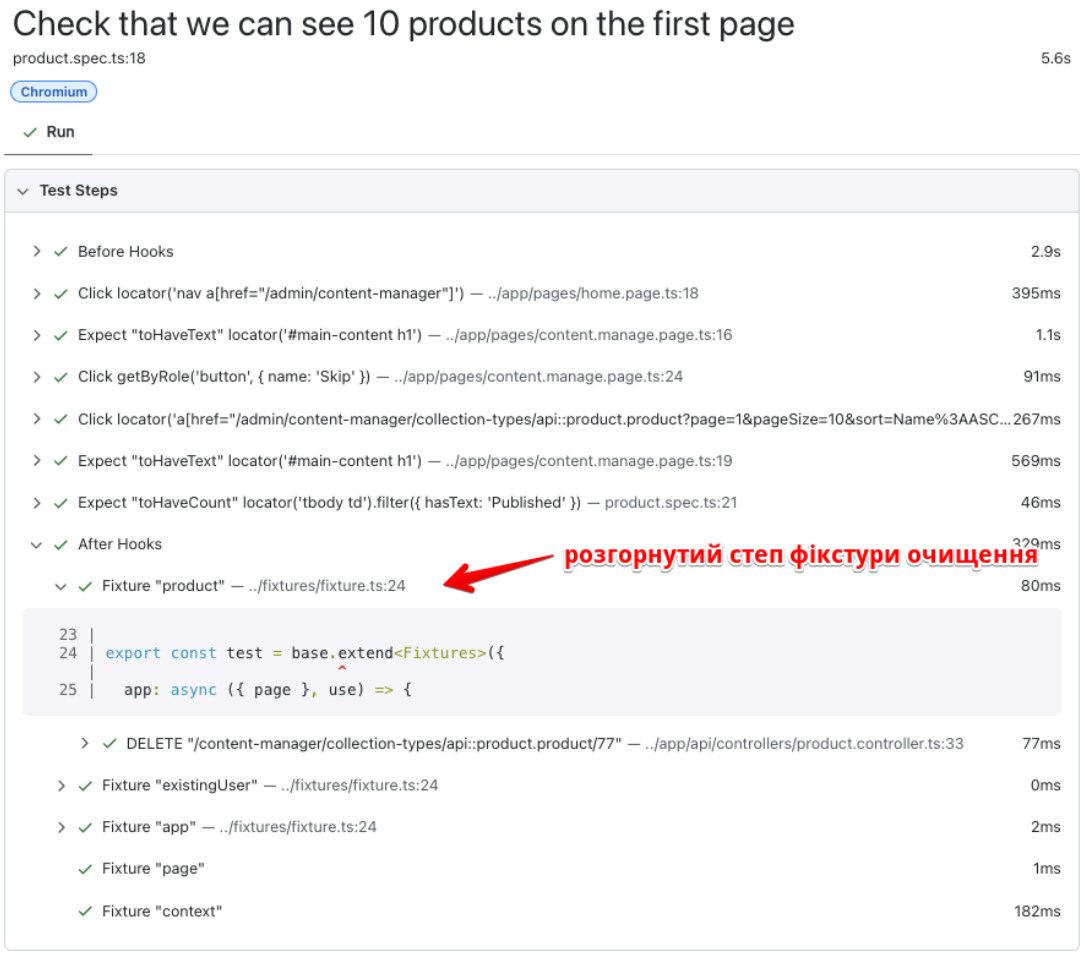


Рис.42 Результат виконання тесту

3.3 Впровадження механізму тегування тестів для фільтрації за середовищами

У процесі розробки та експлуатації автоматизованого фреймворку тестування веб-застосунків виникає необхідність запускати різні набори тестів залежно від середовища виконання, типу перевірок та цілей тестування. Наприклад, у середовищі розробки (dev) доцільно запускати повний набір тестів, тоді як у продуктивному середовищі (prod) - лише обмежений набір smoke- або регресійних тестів.

Для розв’язання цієї задачі у межах розробленого фреймворку було впроваджено механізм тегування тестів із використанням вбудованих можливостей Playwright та системи змінних оточення.

Для забезпечення гнучкої конфігурації середовищ у проєкті застосовано окремі файли змінних середовища:

- .env.dev - для середовища розробки;
- .env.prod - для продуктивного середовища.

У цих файлах визначаються параметри, специфічні для кожного середовища, зокрема:

- облікові дані користувачів (email, пароль);
- базова URL-адреса застосунку;
- параметри доступу до API.

Приклад структури .env.dev:

```
BASE_URL=http://localhost:1337
ADMIN_EMAIL=dev_admin@example.com
ADMIN_PASSWORD=dev_password
```

Аналогічно, у файлі .env.prod визначаються параметри для продуктивного середовища з іншими обліковими даними та адресами.

Завантаження змінних середовища здійснюється перед запуском тестів, що дозволяє одному й тому ж тестовому коду працювати з різними конфігураціями без його модифікації.

Тепер якщо у нас відбудуться якісь зміни в url нашого сайту, нам не потрібно заходити на всі можливі тести чи методи де є урл і все змінювати, достатньо внести зміни у базовй url, і так само, якщо будемо запускати тести для іншого середовища, механізм вводу тестових даних незмінний, будуть змінені лише самі дані для відповідного середовища.

```
test.describe("Positive tests", () => {
  test("Check that a user can make a login", async ({ page }) => {
    const signInPage = new SignInPage(page);
    const homePage = new HomePage(page);

    await signInPage.open();
    await signInPage.fillUserData(process.env.ADMIN_USER!, process.env.ADMIN_PASSWORD!);
    await signInPage.clickLoginBtn();

    await expect(homePage.mainTitle).toHaveText("Hello Oleksii");
    await expect(homePage.secondTitle).toHaveText(
      "Welcome to your administration panel"
    );
  });
});

test.describe("Negative tests", () => {
  test("Check that a user can't login with the invalid data", async ({ page }) => {
    const signInPage = new SignInPage(page);
    const homePage = new HomePage(page);

    await signInPage.open();
    await signInPage.fillUserData(`${process.env.ADMIN_USER!}123`, process.env.ADMIN_PASSWORD!);
    await signInPage.clickLoginBtn();

    await expect(signInPage.fieldError.first()).toHaveText("This is not a valid email")
    await expect(homePage.mainTitle).not.toHaveText("Hello Oleksii");
    await expect(homePage.secondTitle).toBeHidden();
  });

  test("Check that a user with not existing data in a system can't login", async ({ page }) => {
    const signInPage = new SignInPage(page);
    const homePage = new HomePage(page);

    await signInPage.open();
```

Рис.43 Приклад використання базового урл в тесті

Для забезпечення гнучкого запуску тестів у різних середовищах та з різною глибиною перевірок було впроваджено механізм тегування тестів. Кожен тест або група тестів маркуються тегами, що визначають тип

перевірки (API, E2E, smoke, regression) та цільове середовище виконання (dev, prod).

За допомогою параметра --grep у Playwright забезпечується можливість запуску лише необхідної підмножини тестів, що значно підвищує керованість тестового набору та спрощує інтеграцію з CI/CD пайплайнами.

Після додавання тегів наші тести мають додаткову інформацію у назві тесту, які буде корисна далі при фільтрації за допомогою тегів:

```
test("@E2E @smoke See products list", async ({ app, existingUser, product }) =>
{
 await app.homePage.navigateToContentManagerMenuSection();
 await app.contentManagePage.navigateToProductMenuSection();
 await expect(app.productPage.publishedCells.first()).toBeVisible();
});
```

Рис.44 Приклад використання тегу у тесті

На виході ми отримуємо такі переваги:

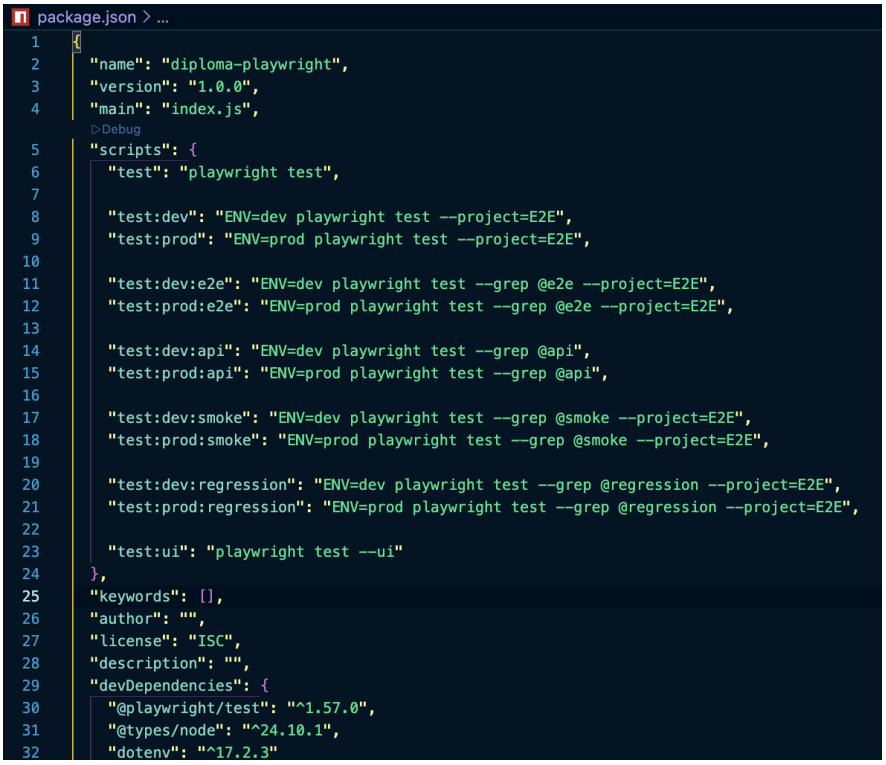
- зменшення часу виконання тестів
- ізоляція критичних smoke-перевірок
- можливість окремого запуску API та UI тестів
- легка інтеграція в CI/CD
- масштабованість тестового фреймворку

Залишилось додати скрипти запуску до package.json файлу і тоді ми матимемо змогу запускати потрібні тести однією командою.

Для прикладу, такий скрипт буде запускатись командою

npm run test:dev:e2e

"test:dev:e2e": "ENV=dev playwright test --grep @e2e --project=E2E",



```
package.json > ...
{
  "name": "diploma-playwright",
  "version": "1.0.0",
  "main": "index.js",
  ▷Debug
  "scripts": {
    "test": "playwright test",

    "test:dev": "ENV=dev playwright test --project=E2E",
    "test:prod": "ENV=prod playwright test --project=E2E",

    "test:dev:e2e": "ENV=dev playwright test --grep @e2e --project=E2E",
    "test:prod:e2e": "ENV=prod playwright test --grep @e2e --project=E2E",

    "test:dev:api": "ENV=dev playwright test --grep @api",
    "test:prod:api": "ENV=prod playwright test --grep @api",

    "test:dev:smoke": "ENV=dev playwright test --grep @smoke --project=E2E",
    "test:prod:smoke": "ENV=prod playwright test --grep @smoke --project=E2E",

    "test:dev:regression": "ENV=dev playwright test --grep @regression --project=E2E",
    "test:prod:regression": "ENV=prod playwright test --grep @regression --project=E2E",

    "test:ui": "playwright test --ui"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "devDependencies": {
    "@playwright/test": "^1.57.0",
    "@types/node": "^24.10.1",
    "dotenv": "^17.2.3"
```

Рис.45 Приклад запису скриптів для запуску тестів відповідно їх тегам у коді

Як видно на рис.46 - запустились самі тести які ми відмітили відповідними тегами.

```
alex@Alexs-MacBook-Pro diploma-playwright % npm run test:dev:smoke

> diploma-playwright@1.0.0 test:dev:smoke
> ENV=dev playwright test --grep @smoke --project=E2E

[dotenv@17.2.3] injecting env (3) from .env.dev -- tip: ⚙ load multiple .env files with { path: ['.env.local', '.env'] }

Running 5 tests using 1 worker

[dotenv@17.2.3] injecting env (0) from .env.dev -- tip: ⚙ override existing env vars with { override: true }
     1 [E2E] › tests/api/api.spec.ts:4:7 › Positive tests › @api @smoke Check that a user can make a login via API
{
  data: {
    token: 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiIxIiwic2Vzc2lvbklkIjoiMjUxYjAxNjE3YWFjMGExOGJiZWNlZjAwNjE2YjFmNmYiLCJ0eXBlIjoiYWNjZXNzIiwiaWF0IjoxNzY1NzE
aWCwbwew78h9lhRsq6ASQIRubek2gwBy8',
    accessToken: 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiIxIiwic2Vzc2lvbklkIjoiMjUxYjAxNjE3YWFjMGExOGJiZWNlZjAwNjE2YjFmNmYiLCJ0eXBlIjoiYWNjZXNzIiwiaWF0IjoxN
b0Dg91aWCwbwew78h9lhRsq6ASQIRubek2gwBy8',
    user: {
      id: 1,
      documentId: 'zilap0l583nuxfvujeicrzoe',
      firstname: 'Oleksii',
      lastname: 'Kudenko',
      username: null,
      email: 'kydster95@gmail.com',
      isActive: true,
      blocked: false,
      preferedLanguage: null,
      createdAt: '2025-11-08T09:51:26.406Z',
      updatedAt: '2025-11-08T09:51:26.406Z',
      publishedAt: '2025-11-08T09:51:26.407Z',
      locale: null
    }
  }
}
  ✓  1 [E2E] › tests/api/api.spec.ts:4:7 › Positive tests › @api @smoke Check that a user can make a login via API (600ms)
  ✓  2 [E2E] › tests/login.spec.ts:6:7 › Login › @E2E @smoke @regression User can login via UI (5.4s)
  ✓  3 [E2E] › tests/login.spec.ts:28:7 › Negative tests › @E2E @smoke Check that a user can't login with the invalid data (3.1s)
  ✓  4 [E2E] › tests/login.spec.ts:53:7 › Negative tests › @E2E @smoke Check that a user with not existing data in a system can't login (5.2s)
  ✓  5 [E2E] › tests/product.spec.ts:23:5 › @E2E @smoke See products list (8.4s)

  5 passed (26.0s)

To open last HTML report run:

  npx playwright show-report
```

Рис.46 Результат виконання запуску смоук тестів для дев середовища

3.4 Інтеграція ФАТ із системою контролю версій (Git) та конвеєром безперервної інтеграції (CI/CD)

Сучасні підходи до забезпечення якості програмного забезпечення неможливі без тісної інтеграції автоматизованого тестування із системами контролю версій та інструментами безперервної інтеграції. Така інтеграція дозволяє автоматично перевіряти коректність роботи системи при кожній зміні коду, мінімізувати ризики регресій та забезпечити стабільність продукту на всіх етапах життєвого циклу розробки.

У межах даної роботи для системи контролю версій використовується Git, а як платформа безперервної інтеграції обрано GitHub Actions, що є нативним CI/CD-рішенням екосистеми GitHub.

Перед інтеграцією з CI/CD необхідно підготувати фреймворк автоматизованого тестування до безголового (headless) запуску в серверному середовищі.

Ключові вимоги до ФАТ

- відсутність жорстко закодованих URL;
- конфігурація через змінні середовища (.env);
- можливість headless-запуску;
- відсутність залежності від локального стану браузера.

У розробленому фреймворку ці вимоги реалізовані за рахунок:

- використання .env.dev та .env.prod;
- параметра headless: true у CI;
- авторизації через API або storageState.

Для автоматизації запуску тестів використовується GitHub Actions, який дозволяє:

- запускати тести при подіях push / pull_request;
- виконувати запуск за розкладом (cron);

- запускати тести вручну (manual dispatch).

У репозиторії ФАТ створюється файл:

.github/workflows/playwright-tests.yml

```yaml
jobs:
  You, 15 minutes ago | 1 author (You)
  run-tests:
    runs-on: ubuntu-latest

    steps:
      You, 19 minutes ago | 1 author (You)
      - name: Checkout repository
        uses: actions/checkout@v4

      You, 20 minutes ago | 1 author (You)
      - name: Setup Node.js
        uses: actions/setup-node@v4
        You, 20 minutes ago | 1 author (You)
        with:
          node-version: 18

      You, 20 minutes ago | 1 author (You)
      - name: Install dependencies
        run: npm ci

      You, 20 minutes ago | 1 author (You)
      - name: Install Playwright browsers
        run: npx playwright install --with-deps

      You, 19 minutes ago | 1 author (You)
      - name: Run selected tests
        run: |
          if [ "${{ github.event.inputs.test_type }}" = "e2e-dev" ]; then
            npm run test:dev:e2e
          elif [ "${{ github.event.inputs.test_type }}" = "api-dev" ]; then
            npm run test:dev:api
          elif [ "${{ github.event.inputs.test_type }}" = "e2e-prod" ]; then
            npm run test:prod:e2e
          fi
        You, 19 minutes ago | 1 author (You)
        env:
          ENV: ${{ github.event.inputs.test_type == 'e2e-prod' && 'prod' || 'dev' }}
          BASE_URL: ${{ secrets.BASE_URL }}
          ADMIN_USER: ${{ secrets.ADMIN_USER }}
          ADMIN_PASSWORD: ${{ secrets.ADMIN_PASSWORD }}

      You, 15 minutes ago | 1 author (You)
      - name: Prepare report for GitHub Pages
        if: always()
        run: |
          rm -rf docs
          mkdir docs
          cp -r playwright-report/* docs/

      You, 15 minutes ago | 1 author (You)
      - name: Upload Pages artifact
        if: always()
        uses: actions/upload-pages-artifact@v3
        You, 15 minutes ago | 1 author (You)
        with:
```

Рис.47 Вигляд воркфлоу файлу

Конфіденційні дані не зберігаються в репозиторії, а передаються через GitHub Secrets:

- BASE_URL;
- ADMIN_USER;
- ADMIN_PASSWORD;

Це відповідає принципам безпеки та best practices CI/CD.

Після того як ми завантажили наш воркфлоу на гітхаб, додали дані у гітхаб сікрет, згенерували гітхаб токет для роботи між репозиторіями, то ми вже можемо переглянути можливості мануального запуску тестів:

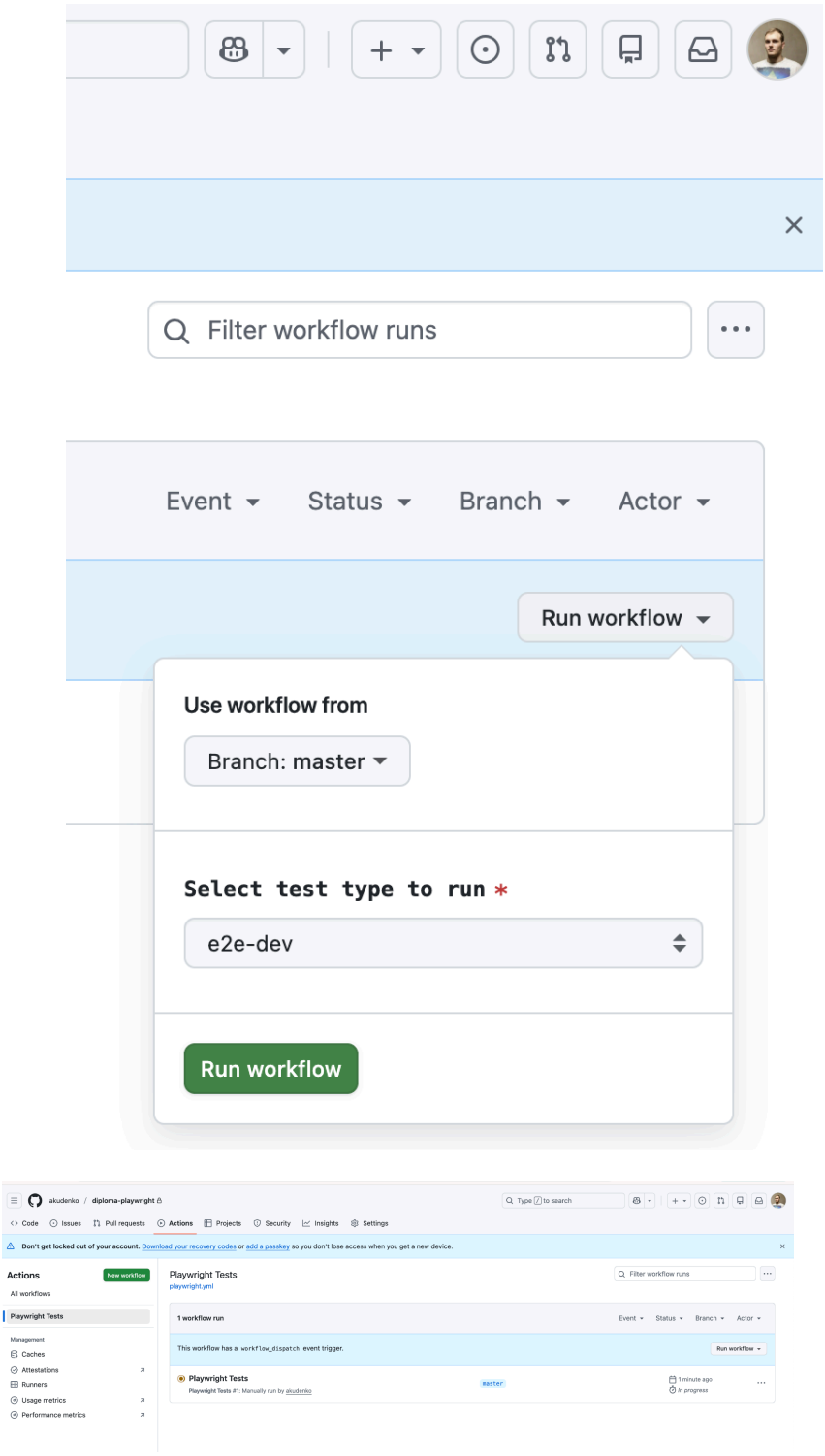


Рис.48 Вигляд мануального запуску

Як бачимо на рис.48, ми маємо зручний варіант ручного запуску. Потрібно вибрати лише необхідний параметр запуску тестів, наприклад e2e регресійні тести, які запустять усі тести в яких зустрічаються ці теги.

Це дозволяє швидко і гнучко запустити тести будь-якому члену команди, а репорт який генерується на гітхаб пейджес, дозволить усім зацікавленим членам команди переглянути виконання тестів, зібрати метрики тощо.

У межах даної роботи було реалізовано повноцінну інтеграцію фреймворку автоматизованого тестування з системою контролю версій Git та платформою безперервної інтеграції GitHub Actions. Запропоноване рішення дозволяє виконувати автоматизовані UI та API тести як локально, так і в CI-середовищі, забезпечуючи стабільність, масштабованість та відповідність сучасним інженерним практикам.

3.5 Налаштування та застосування системи звітності: візуалізація результатів, збір метрик та аналіз логів

Ефективність фреймворку автоматизованого тестування визначається не лише кількістю автоматизованих перевірок, але й можливістю швидко та наочно аналізувати результати їх виконання. У реальних проєктах автоматизоване тестування виконується регулярно (локально, у CI/CD, за розкладом), тому система звітності є критично важливою складовою ФАТ.

Основними вимогами до системи звітності є:

- наочна візуалізація результатів;
- збереження історії запусків;
- можливість аналізу помилок без повторного запуску тестів;
- збір метрик стабільності та продуктивності тестів.

Ефективність фреймворку автоматизованого тестування визначається не лише кількістю автоматизованих перевірок, але й можливістю швидко та наочно аналізувати результати їх виконання. У реальних проєктах автоматизоване тестування виконується регулярно (локально, у CI/CD, за розкладом), тому система звітності є критично важливою складовою ФАТ.

Основними вимогами до системи звітності є:

- наочна візуалізація результатів;
- збереження історії запусків;
- можливість аналізу помилок без повторного запуску тестів;
- збір метрик стабільності та продуктивності тестів.

HTML-репортер Playwright дозволяє збирати та аналізувати такі метрики:

1. Метрики стабільності

- відсоток успішно виконаних тестів;
- кількість флакі-тестів;
- повторні падіння одного сценарію;
- стабільність тестів у різних середовищах (dev / prod).

2. Метрики продуктивності

- середній час виконання тесту;
- загальний час тестового запуску;
- порівняння часу виконання UI та API тестів.

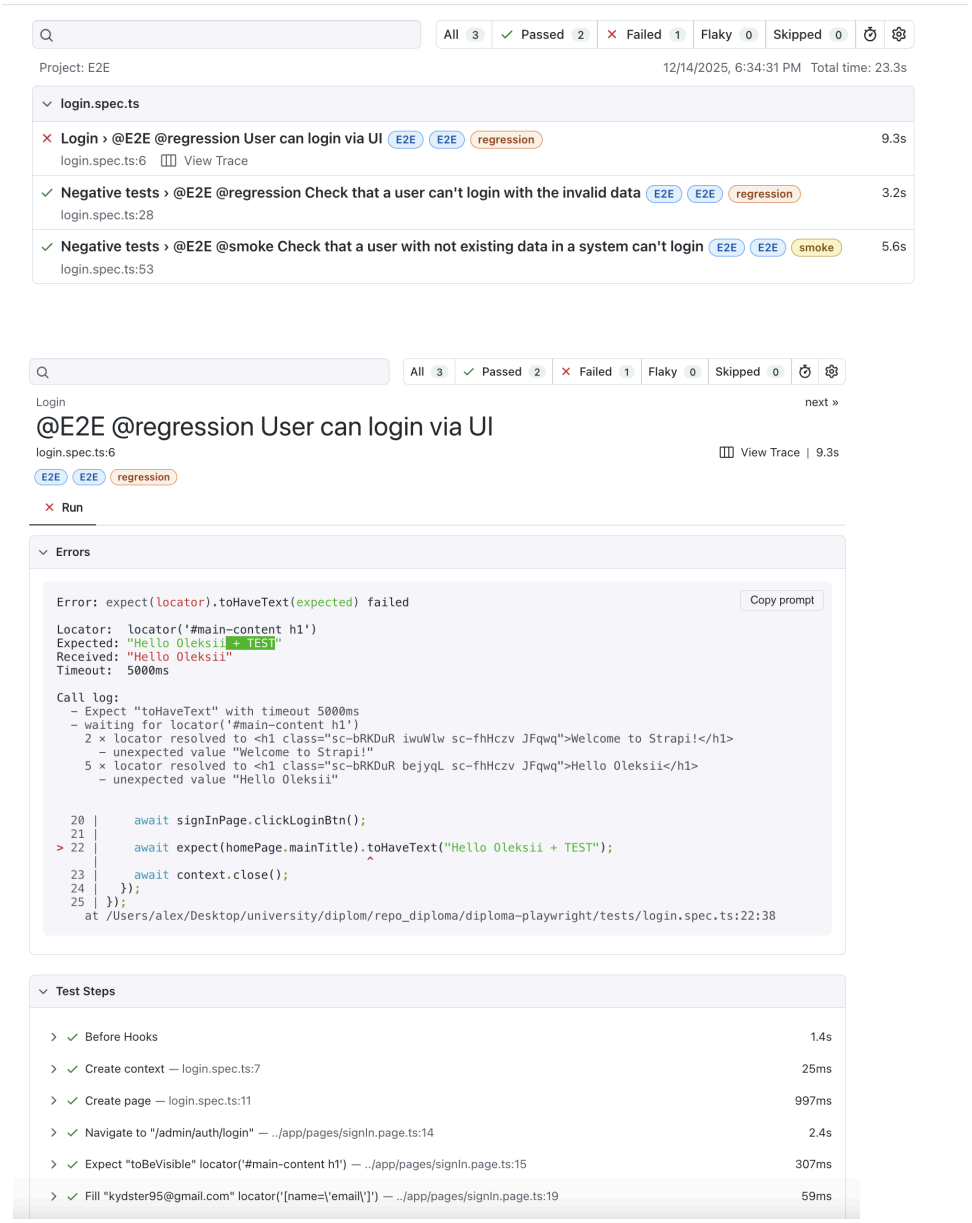
3. Метрики покриття

- кількість перевірених бізнес-сценаріїв;
- співвідношення API та UI тестів;
- кількість допоміжних фікстур (створення/видалення даних).

Завдяки цьому фреймворк дозволяє не лише перевіряти функціональність, а й оцінювати якість самої автоматизації.

Для кожного невдалого тесту Playwright автоматично зберігає:

- screenshot - стан інтерфейсу в момент помилки;
- video - повний запис виконання тесту;
- trace - покрокову хронологію дій, очікувань і асерцій.



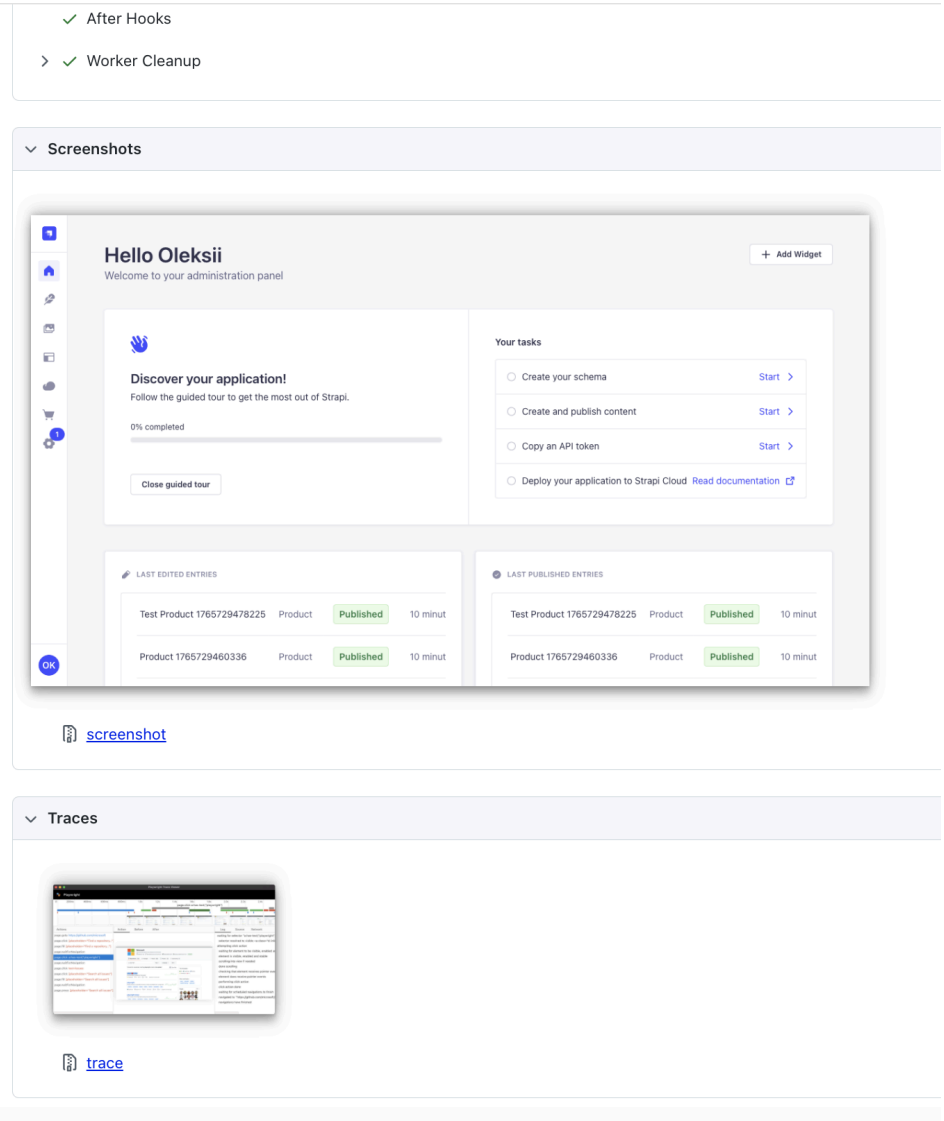


Рис.49 Детальний результат тесту

Trace-файл дозволяє:

- відтворити виконання тесту крок за кроком;
- побачити реальні значення локаторів;
- проаналізувати затримки, очікування та таймінги.

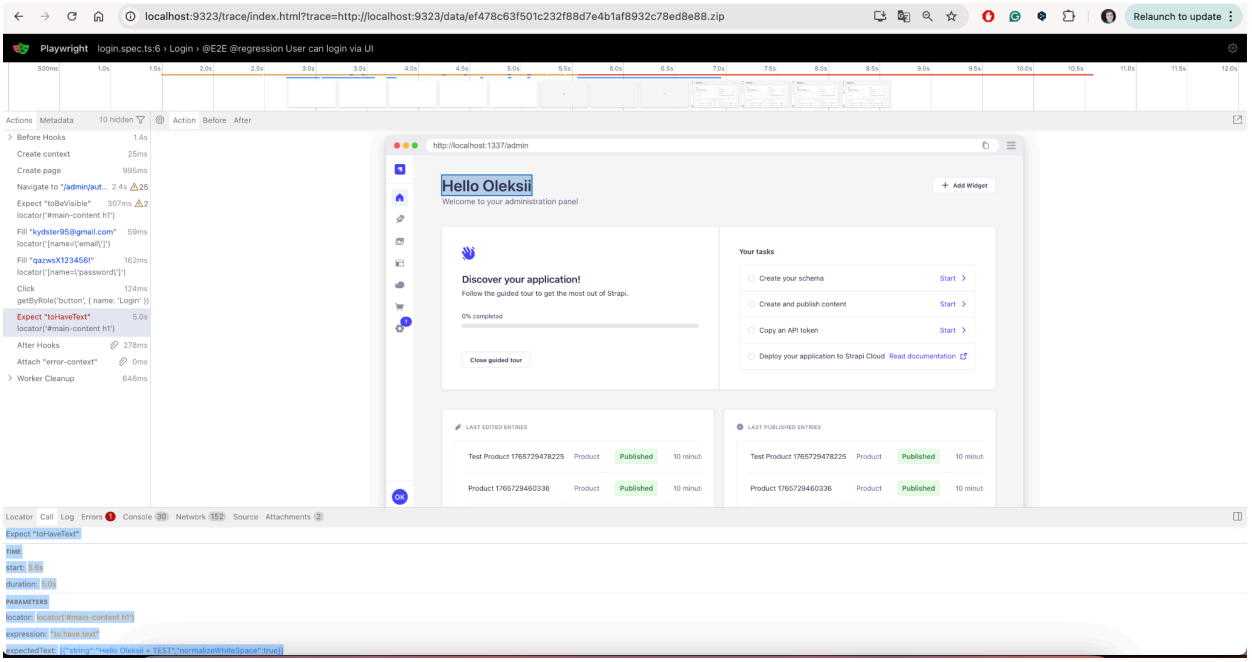


Рис.50 Вікно відкритого трейсу для тесту який завершився помилкою

Це суттєво зменшує час пошуку дефектів і підвищує ефективність роботи з результатами тестування.

Однією з проблем стандартної звітності є надмірна технічність назв кроків, які часто не відображають бізнес-сенс дії. Для вирішення цієї проблеми у фреймворку реалізовано кастомний декоратор для автоматичного формування зрозумілих кроків у звіті.

Рис.51 Реалізація функції декоратору для результатів репорту

Переваги використання step-декоратора:

- кроки у звіті набувають бізнес-орієнтованого вигляду
- параметри методів автоматично підставляються у назву кроку;
- покращується читабельність HTML-репорту;
- зменшується потреба у ручному логуванні.

```
@step('Fill user data: email: ${email}, password: ${password}')
async fillUserData(email: string, password: string) {
  await this.emailField.fill(email);
  await this.passwordField.fill(password);
}
```

Рис.52 Приклад додавання опису кроку в тесті для кращої читаємості

Як результат, замість Navigate to "/admin/auth/login" або Fill "kydster95@gmail.com" locator('[name=\'email\']'), у нас **Open the Home page** та **Fill user data: email: kydster95@gmail.com, password: qazwsX123456!**

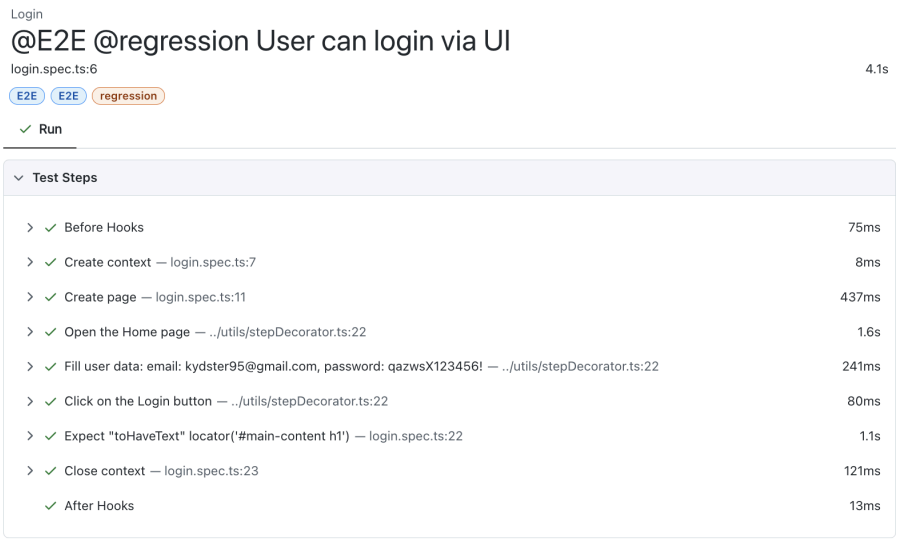


Рис.52 Оновлений вигляд кроків тесту у репорту

Згенерований HTML-репорт автоматично публікується у GitHub Pages після кожного запуску тестів у CI/CD. Це дозволяє:

- переглядати результати через браузер;
- ділитися результатами з командою або викладачем;
- зберігати історію тестових запусків.

Таким чином, система звітності стає повноцінним інструментом контролю якості, а не лише допоміжним логом виконання тестів.

ВИСНОВКИ

У процесі виконання дипломної роботи було здійснено комплексне дослідження теоретичних і практичних аспектів автоматизованого тестування веб-застосунків у сучасних інфокомунікаційних системах. Основною метою роботи було проєктування та реалізація універсального фреймворку автоматизованого тестування (ФАТ), який може бути ефективно використаний для перевірки як UI, так і API-рівнів веб-застосунків незалежно від предметної області.

У першому розділі дипломної роботи було розглянуто фундаментальні засади функціонування веб-застосунків та інфокомунікаційних систем, їх архітектурні особливості та роль у сучасному бізнес-середовищі. Було показано, що зростання складності веб-систем, їх розподіленість, активне використання REST API та мікросервісних підходів суттєво підвищують вимоги до якості програмного забезпечення. У цьому контексті тестування ПЗ виступає не лише інструментом виявлення дефектів, а й невід'ємною складовою процесу забезпечення стабільності, безпеки та надійності систем.

Також у першому розділі було виконано аналіз ручного та автоматизованого тестування, визначено їхні переваги та обмеження. Обґрунтовано доцільність застосування автоматизованого тестування у проєктах з довгим життєвим циклом, частими релізами та високими вимогами до регресійної перевірки. Окрему увагу приділено ситуаціям, у яких автоматизація є економічно або технічно недоцільною, що дозволяє сформувати збалансований підхід до вибору стратегії тестування.

Значну частину першого розділу присвячено історичному огляду та порівняльному аналізу сучасних фреймворків автоматизованого тестування, таких як Selenium, Cypress, Playwright та інші. На основі проведеного аналізу було обґрунтовано вибір Playwright у поєднанні з TypeScript/Node.js як базових технологій для реалізації ФАТ. Ключовими аргументами стали кросбраузерність, висока швидкодія, підтримка паралельного виконання

тестів, зручна робота з API, а також тісна інтеграція з сучасними CI/CD-пайплайнами.

У другому розділі дипломної роботи було зосереджено увагу на моделюванні архітектури фреймворку автоматизованого тестування. Було сформульовано функціональні та нефункціональні вимоги до ФАТ, визначено ключові архітектурні принципи, серед яких модульність, масштабованість, повторне використання коду та незалежність тестів один від одного.

Для організації UI-тестів було застосовано патерн Page Object Model (POM), що дозволило інкапсулювати логіку взаємодії зі сторінками веб-застосунку та зменшити залежність тестів від змін у користувацькому інтерфейсі. Описана структура проєкту демонструє, що фреймворк легко адаптується до нових сторінок і сценаріїв шляхом додавання або розширення відповідних Page Object класів без необхідності зміни існуючих тестів.

Окрему увагу у другому розділі було приділено керуванню конфігураціями середовищ. Реалізований підхід із використанням .env файлів дозволяє запускати тести в різних середовищах (dev, prod) без зміни тестового коду, що є критично важливим для промислового використання та CI/CD інтеграцій. Також було детально розглянуто механізм фікстур (Fixtures) у Playwright та їхню роль у підготовці тестового середовища.

Важливим результатом другого розділу стало проєктування інтегрованого підходу UI та API тестування. Було показано, що API-хелпери можуть використовуватися для швидкої підготовки тестових даних, авторизації та очищення середовища після виконання тестів, що суттєво знижує крихкість UI-тестів і підвищує загальну стабільність тестового набору.

У третьому розділі було реалізовано практичну частину фреймворку та проведено оцінку його ефективності. Реалізовано базову конфігурацію Playwright, керування браузерами, паралельне виконання тестів і підтримку різних проєктів (E2E, API). Значну увагу приділено реалізації модулів

API-тестування, зокрема перевірці авторизації, створення та видалення бізнес-сутностей через REST API.

У межах цього розділу було впроваджено механізм кастомних фікстур, які дозволяють інкапсулювати підготовку стану тестів і повторно використовувати її у різних сценаріях. Такий підхід не лише скорочує час виконання тестів, але й зменшує навантаження на UI та backend, що є критично важливим для великих тестових наборів.

Окремо було реалізовано механізм тегування тестів, що дозволяє гнучко фільтрувати сценарії залежно від середовища та типу перевірки. Це забезпечує можливість запуску лише необхідних тестів у межах CI/CD пайплайнів або під час локальної розробки.

У рамках інтеграції з системою контролю версій Git та конвеєром безперервної інтеграції було показано, що розроблений ФАТ легко інтегрується з GitHub Actions. Реалізовано можливість автоматичного запуску тестів за розкладом, вручну через workflow dispatch, а також публікацію HTML-звітів у GitHub Pages. Це забезпечує прозорість результатів тестування та спрощує аналіз якості продукту для всієї команди.

Значну увагу у третьому розділі приділено системі звітності. Використання HTML-репортера Playwright у поєднанні з трасуванням, скріншотами та відеозаписами дозволяє детально аналізувати причини падіння тестів, збирати метрики стабільності та швидкодії, а також підвищувати якість тестового покриття. Додатково було реалізовано механізм декорування кроків тестів, що значно покращує читабельність логів та звітів.

Узагальнюючи результати дипломної роботи, можна зробити висновок, що розроблений фреймворк автоматизованого тестування є універсальним, масштабованим і придатним до повторного використання. Його архітектура не прив’язана до конкретного веб-застосунку або доменної області, а базові принципи залишаються незмінними при міграції на інші проєкти. Для адаптації фреймворку достатньо реалізувати нові Page Object або API-контролери відповідно до специфіки цільової системи.

Практична цінність цієї дипломної роботи полягає в тому, що отриманий результат може бути безпосередньо використаний у реальних комерційних проєктах для підвищення якості програмного забезпечення, зменшення часу регресійного тестування та зниження ризиків, пов’язаних із частими релізами. Запропонований підхід також може слугувати основою для подальших досліджень у напрямках performance-тестування, security-тестування та інтеграції з іншими інструментами DevOps-екосистеми.

Таким чином, поставлена мета дипломної роботи була досягнута, а всі завдання - успішно виконані. Результати дослідження підтверджують доцільність використання сучасних фреймворків автоматизованого тестування та комплексного підходу до забезпечення якості веб-застосунків у сучасних інфокомунікаційних системах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Playwright. Getting Started [Електронний ресурс]. URL: https://playwright.dev/docs/intro.

2. Playwright. Test Runner [Електронний ресурс]. URL: https://playwright.dev/docs/test-intro.

3. Playwright. Page Object Models [Електронний ресурс]. URL: https://playwright.dev/docs/pom.

4. Playwright. Best Practices [Електронний ресурс]. URL: https://playwright.dev/docs/best-practices.

5. Playwright. API Reference [Електронний ресурс]. URL: https://playwright.dev/docs/api/class-playwright.

6. TypeScript. Handbook [Електронний ресурс]. URL: https://www.typescriptlang.org/docs/.

7. TypeScript. Configuration (tsconfig.json) [Електронний ресурс]. URL: https://www.typescriptlang.org/tsconfig.

8. Node.js. Official Documentation [Електронний ресурс]. URL: https://nodejs.org/en/docs/.

9. Microsoft. End-to-End Testing with Playwright [Електронний ресурс]. URL: https://learn.microsoft.com/en-us/microsoft-edge/playwright/.

10. Gackenheimer C. Introduction to Modern Client-Side Testing. Berkeley : Apress, 2015. 270 p.

11. Horowitz J. Learning Playwright. Birmingham : Packt Publishing, 2023. 31 p.

12. Ghazi A. End-to-End Web Testing with Playwright: Automate and Test Modern Web Apps. Birmingham : Packt Publishing, 2023. 280 p.

13. ISO/IEC/IEEE 29119-1:2013. Software and Systems Engineering – Software Testing - Part 1: Concepts and Definitions. Geneva : ISO, 2013.

14. ISO/IEC 25010:2011. Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models. Geneva : ISO, 2011.

15. Dustin E., Rashka J., Paul J. Automated Software Testing: Introduction, Management, and Performance. Boston : Addison-Wesley, 1999. 350 p.

16. developer.mozilla.org [Електронний ресурс]: https://developer.mozilla.org/en-US/docs/Learn_web_development/Getting_started/Web_standards/How_the_web_works

17. developer.mozilla.org [Електронний ресурс]: https://developer.mozilla.org/en-US/docs/Web/CSS

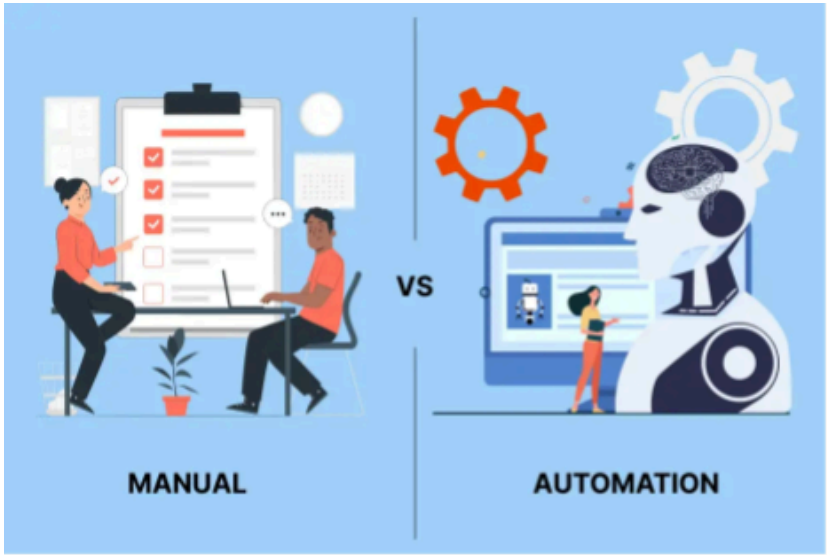
18. developer.mozilla.org [Електронний ресурс]: https://developer.mozilla.org/en-US/docs/Web/XML/XPath

19. developer.mozilla.org [Електронний ресурс]: https://developer.mozilla.org/en-US/docs/Web/API

20. https://www.testrail.com/ [Електронний ресурс]: https://www.testrail.com/blog/test-strategy-approaches/

Додаток А

Міністерство освіти та науки України
Національний університет «Полтавська політехніка імені Юрія Кондратюка»
Кафедра автоматики, електроніки та телекомунікацій
Моделювання та впровадження фреймворку автоматизованного тестування веб-застосунку, як інструменту підвищення якості інфокомунікаційної системи
Кваліфікаційна робота магістра
Виконав:
Студент групи 601ТТ
Куденко О.О.
Керівник:
Професор: д.т.н.
Косенко В.В.



Предмет дослідження: Методи, архітектурні патерни (POM) та інструменти побудови автоматизованого фреймворку.

Об'єкт дослідження: Процес забезпечення якості (QA) веб-застосунку в межах інфокомунікаційної системи.

Актуальність: Необхідність прискорення релізів (CI/CD) та складність сучасних веб-застосунків вимагають заміни ручного тестування на гібридну автоматизацію (UI+API). 1

Мета: Моделювання та впровадження фреймворку на базі бібліотеки Playwright + TypeScript для підвищення якості та швидкості контролю ПЗ.

Наукова новизна: Удосконалено модель інтегрованого тестування, де API-запити прискорюють підготовку даних для UI-тестів, та впроваджено гнучке керування середовищами через Playwright Fixtures.

Практичне значення: Створено готовий до використання ФАТ, що скорочує час регресійного тестування та автоматизує звітність.

Playwright

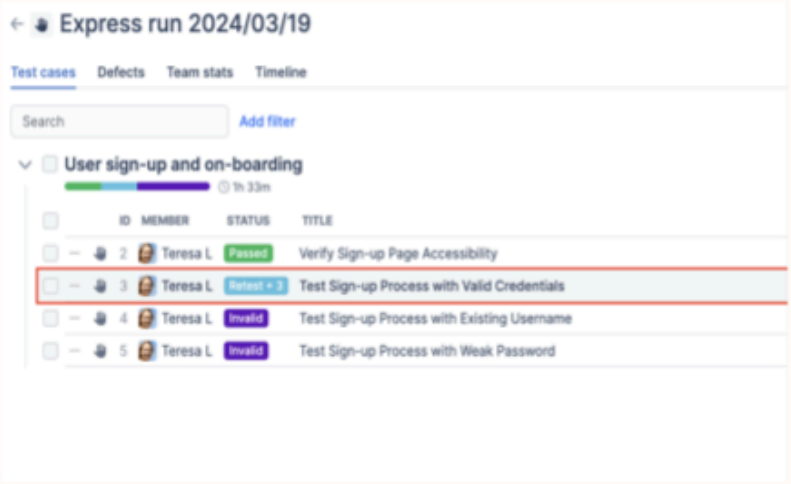
РОЗДІЛ 1. ФУНДАМЕНТАЛЬНІ ЗАСАДИ ТЕСТУВАННЯ ТА АНАЛІЗ ІНСТРУМЕНТАРІЮ

1. Проаналізовано сучасні веб-застосунки та підходи до забезпечення якості програмного забезпечення в умовах складних архітектур.

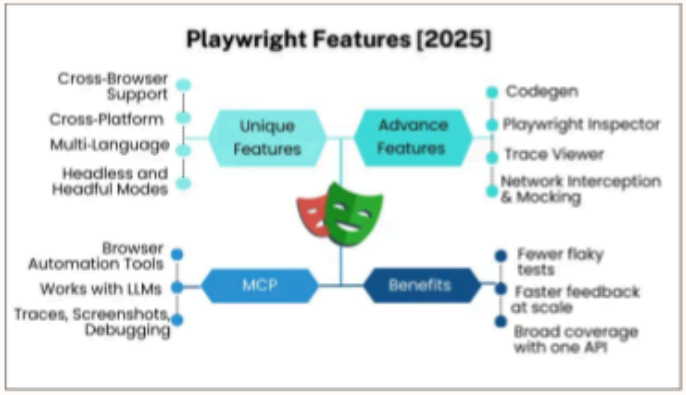
2. Обгрунтовано доцільність використання автоматизованого тестування та вибір **Playwright + TypeScript/Node.js** як базових технологій.

Життєвий цикл тестування ПЗ

Аналіз вимог	Визначення критеріїв входу та виходу, аналіз вимог до тестування
Планування тестування	Розробка стратегії тестування та тест-плану
Розробка тест-кейсів	Створення тест-кейсів, сценаріїв та тестових даних
Налаштування тестового середовища	Підготовка інфраструктури для тестування (середовище, інструменти)
Виконання тестів	Запуск тестів та фіксація результатів
Завершення циклу	Аналіз результатів, підготовка звіту про тестування та закриття циклу.



Чому Playwright?




```
tests > login.spec.ts > ...
1   import { test, expect } from '@playwright/test';
2
3   test('Check that a user can make a login', async ({ page }) => {
4     await page.goto('http://localhost:1337/admin/auth/login'); - 1070ms
5
6     await expect(page.locator('#main-content h1')).toBeVisible(); - 338ms
7
8     await page.locator("[name='email']").fill('kydster95@gmail.com'); - 68ms
9     await page.locator("[name='password']").fill('qazwsX1234567!'); - 41ms
10    await page.getByRole('button', {name: 'Login'}).click(); - 140ms
11
12    await expect(page.locator('#main-content h1')).toHaveText("Hello Oleksii"); - 1022ms
13  });
14
```

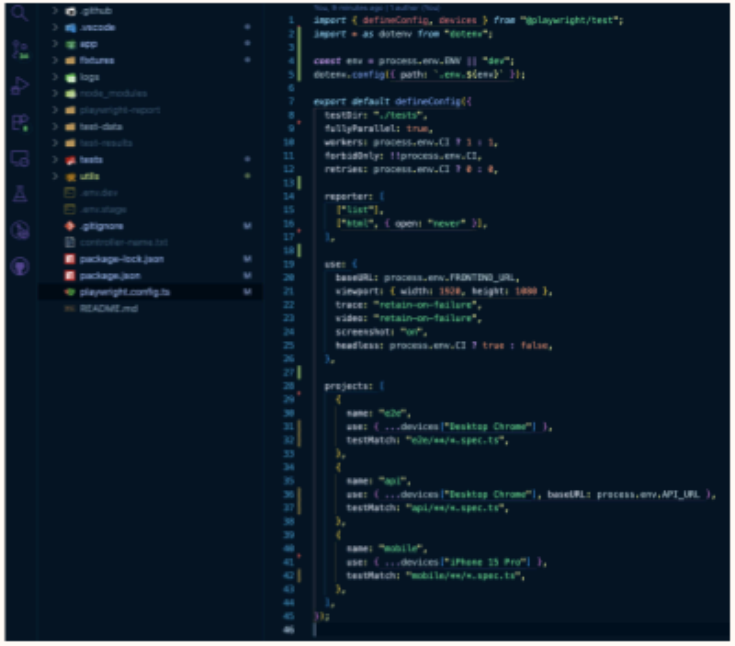
- Підтримка усіх популярних браузерів: ***Chromium, WebKit, and Firefox***;
- Автоматичні очікування за замовчуванням;
- Підтримка усіх популярних рішень: ***iframes, multi-tabs, shadow DOM, mocking, intercepting API calls;***
- Легкий пошук помилок;
- Підтримка різних мов програмування: ***JS/TS, Java, Python, .NET***
- Паралельне виконання тестів;
- Вбудований тест ранер з репортом;
- Microsoft підтримка;
- Безкоштовний.

Порівняльна характеристика сучасних бібліотек для тестування

Feature	Selenium	Cypress	Playwright
Cross-Browser Support	Yes (Chrome, Firefox, Safari, Edge, IE)	Limited (Chrome, Edge)	Yes (Chrome, Firefox, Safari, Edge)
Cross-Platform Testing	Yes (Windows, macOS, Linux)	No (limited Linux/macOS)	Yes (Windows, macOS, Linux)
Supports Headless Mode	Yes	Yes	Yes
API for Interacting with Network	Limited (requires plugins)	Yes (basic control)	Yes (full network control)
Automated Waiting	No (requires explicit waits)	Yes (automatic waits)	Yes (automatic waits)
Parallel Test Execution	Limited (with 3rd-party tools)	Yes (limited)	Yes (built-in support)
Frames and iFrames Handling	Yes	Partial	Yes
Shadow DOM Support	Partial	No	Yes
Multi-Tab/Window Support	Yes	No	Yes
Native Mobile Support	Yes (Appium integration)	No	No
Network Interception & Mocking	Limited (plugins)	Yes	Yes
Screenshot and Video Capture	Yes (plugins)	Yes	Yes
Built-in Test Runner	No (needs external runner)	Yes (Cypress Runner)	Yes (Playwright Test)
Locator Strategies (CSS, XPath)	Yes	CSS only	Yes
Locator Strategies (CSS, XPath)	Yes	CSS only	Yes
TypeScript Support	Partial (with setup)	Yes	Yes
Real Browser Events	No (simulated events)	Yes	Yes
Automatic Retry on Failure	No (plugins or manual)	Yes	Yes
Community & Documentation	Large, extensive	Large, active	Growing, actively maintained
CI/CD Integration	Yes	Yes	Yes
File Upload and Download	Partial (plugins)	Limited	Yes
Support for WebSockets	Partial	Yes	Yes
Debugging Tools (Tracing, etc.)	No (basic debugging)	Yes (limited)	Yes (tracing, codegen)

РОЗДІЛ 2. МОДЕЛЮВАННЯ АРХІТЕКТУРИ ТА СТРУКТУРИ ФРЕЙМВОРКУ

1. Спроєктовано архітектуру масштабованого ФАТ на базі Playwright.
2. Реалізовано гібридний підхід UI + API з використанням фікстур.
3. Підготовлено основу для практичної реалізації та експериментів.



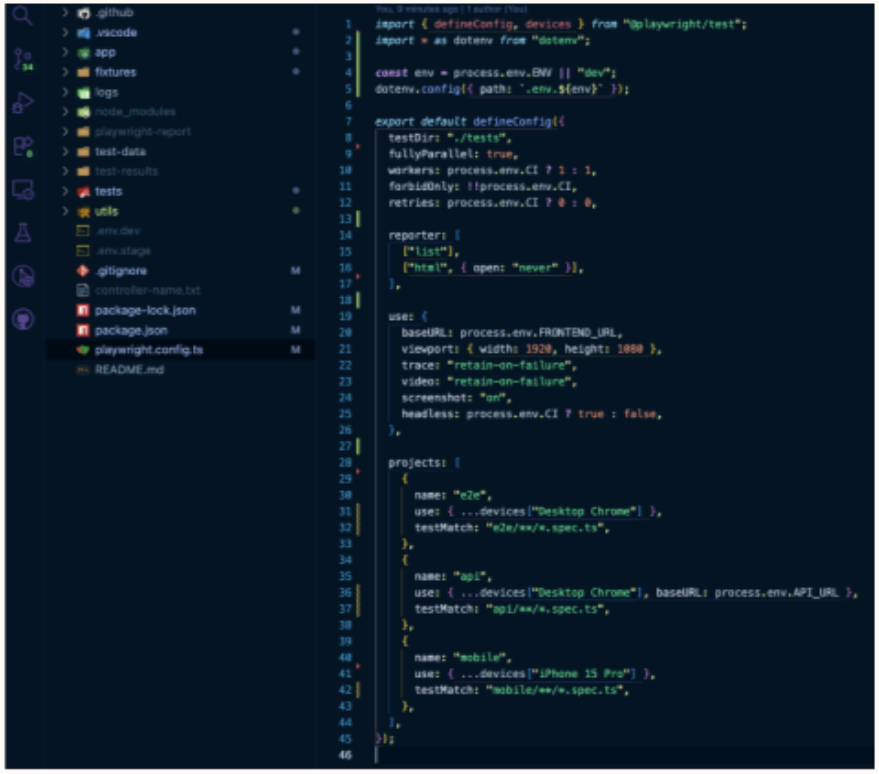
```ts
import { defineConfig, devices } from "@playwright/test";
import * as dotenv from "dotenv";

const env = process.env.ENV || "dev";
dotenv.config({ path: `.env.${env}` });

export default defineConfig({
  testDir: "./tests",
  fullyParallel: true,
  workers: process.env.CI ? 1 : 5,
  forbidOnly: !!process.env.CI,
  retries: process.env.CI ? 0 : 0,

  reporter: [
    ["list"],
    ["html", { open: "never" }],
  ],

  use: {
    baseURL: process.env.FRONTEND_URL,
    viewport: { width: 1920, height: 1080 },
    trace: "retain-on-failure",
    video: "retain-on-failure",
    screenshot: "on",
    headless: process.env.CI ? true : false,
  },

  projects: [
    {
      name: "e2e",
      use: { ...devices["Desktop Chrome"] },
      testMatch: "e2e/**/*.spec.ts",
    },
    {
      name: "api",
      use: { ...devices["Desktop Chrome"], baseURL: process.env.API_URL },
      testMatch: "api/**/*.spec.ts",
    },
    {
      name: "mobile",
      use: { ...devices["iPhone 15 Pro"] },
      testMatch: "mobile/**/*.spec.ts",
    },
  ],
});
```

```ts
import { expect } from "@playwright/test";
import { AppComponent } from "../appComponent";
import { step } from "../../utils/stepDecorator";

export class SignInPage extends AppComponent {
  readonly mainTitle = this.page.locator('#main-content h1');
  readonly emailField = this.page.locator("[name='email']");
  readonly passwordField = this.page.locator("[name='password']");
  readonly rememberMeCheckbox = this.page.getByRole("checkbox", { name: "Remember me" });
  readonly loginButton = this.page.getByRole('button', {name: 'Login'});
  readonly fieldError = this.page.locator('[data-strapi-field-error="true"]');
  readonly errorMessage = this.page.locator('#global-form-error');

  @step('Open the Home page')
  async open() {
    await this.page.goto("http://localhost:1337/admin/auth/login");
    await expect(this.mainTitle).toBeVisible();
  }

  @step('Fill user data: email: ${email}, password: ${password}')
  async fillUserData(email: string, password: string) {
    await this.emailField.fill(email);
    await this.passwordField.fill(password);
  }

  @step('Click on the Login button')
  async clickLoginBtn() {
    await this.loginButton.click();
  }
}
```

Проектування ФАТ

Компонент	Призначення та інструментальна реалізація
Організація та запуск	Керування порядком виконання тестів, паралелізація та фільтрація за тегами. Реалізовано через **Playwright Test Runner**.
Налаштування середовища	Глобальна конфігурація (playwright.config.ts), керування URL, тайм-аутами та оточеннями (dev, stage, prod).
Керування залежностями	Встановлення та контроль версій зовнішніх бібліотек і драйверів браузерів. Використовується **NPM (package.json)**.
Взаємодія з системою	Механізми взаємодії з UI (локатори, селектори) та API (HTTP-запити). Реалізовано через **Playwright APIRequestContext** та **Locators**.
Логіка та патерни	Структурування коду для повторного використання та легкої підтримки. Використано патерн **Page Object Model (POM)**.
Тестові дані та стан	Підготовка середовища, авторизація та очищення даних. Реалізовано через **Playwright Fixtures**.
Звітність (Reporting)	Збір результатів виконання, скріншотів, відео та логів. Реалізовано через **HTML Reporter** та **Allure**.
CI/CD Інтеграція	Автоматизація запуску в загальному конвеєрі розробки. Використовується **GitHub Actions** або **GitLab CI**.

Механізм глобальної конфігурації ФАТ



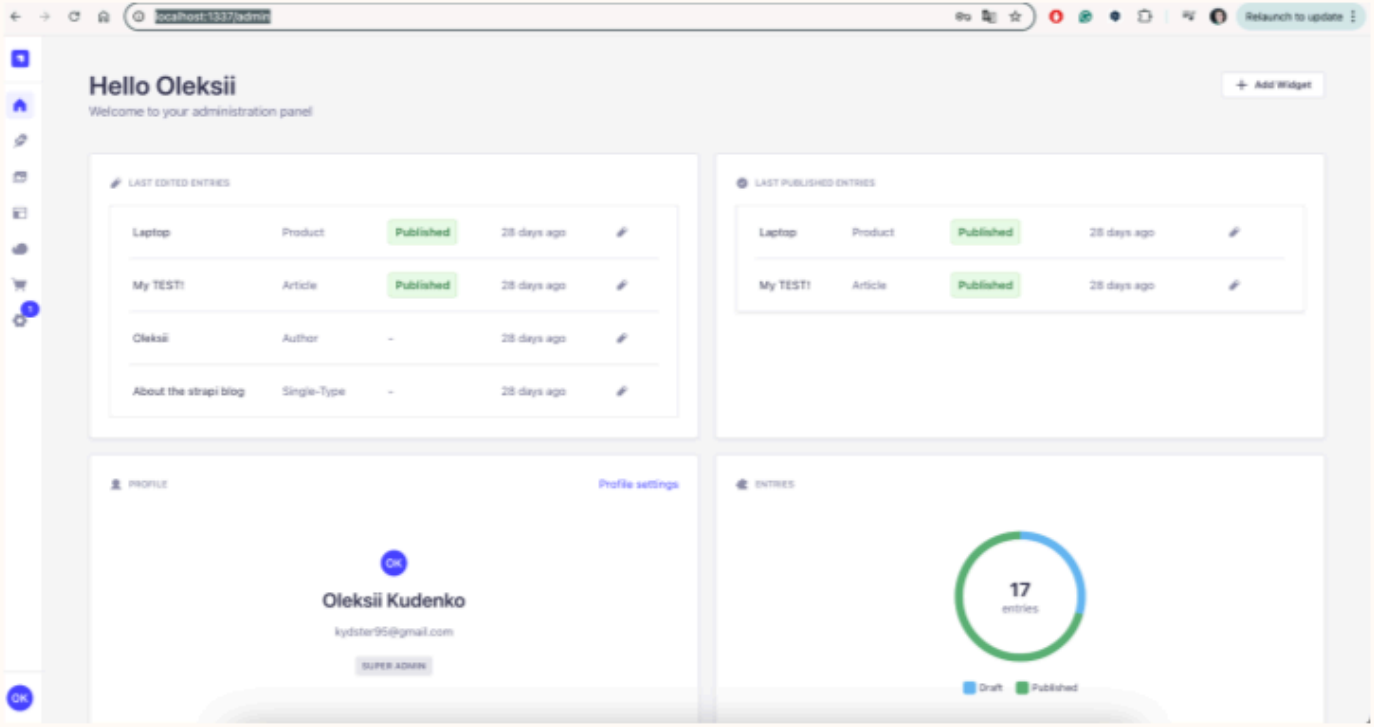

```
# Приклад .env.dev
FRONTEND_URL=https://dev.apitable.com
API_URL=https://dev.apitable.com/api/v1
API_KEY=dev-123-api-key
```

Аналіз для застосування механізму фікстур

Перевага	Опис
Ізоляція стану	Кожен тест отримує чисте середовище, що виключає вплив тестів один на одного.
Автоматичний Setup/Teardown	Самостійно створюють дані перед тестом та видаляють їх після, навіть якщо тест впав.
Перевикористання коду	Логіка (наприклад, авторизація) пишеться один раз і автоматично "підмішується" у потрібні тести.
Лінива ініціалізація	Фікстура створюється лише тоді, коли тест її явно запитує, заощаджуючи ресурси.

```
test.describe('todo tests', () => {
  let todoPage;

  test.beforeEach(async ({ page }) => {
    todoPage = new TodoPage(page);
    await todoPage.goto();
    await todoPage.addToDo('item1');
    await todoPage.addToDo('item2');
  });

  test.afterEach(async () => {
    await todoPage.removeAll();
  });

  test('should add an item', async () => {
    await todoPage.addToDo('my item');
    // ...
  });
```

Аналіз цільового веб-застосунку

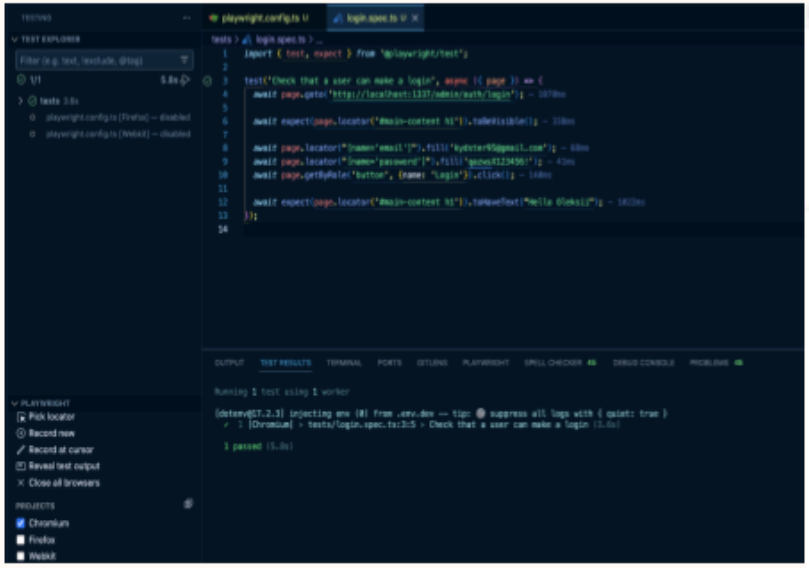


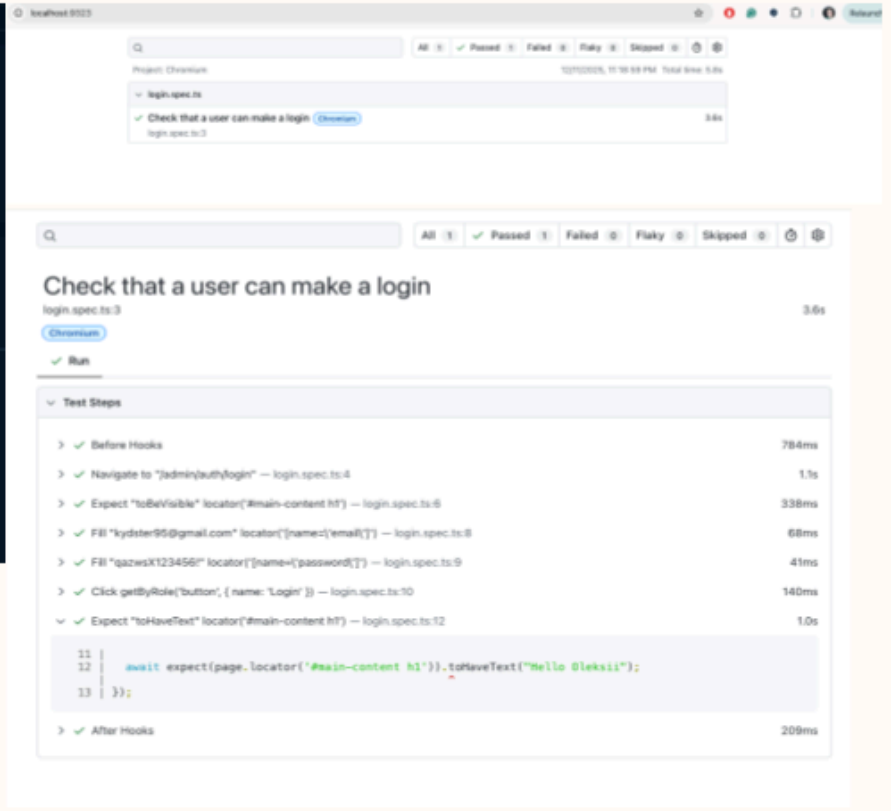
РОЗДІЛ 3. ВПРОВАДЖЕННЯ, АВТОМАТИЗАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ

Спроєктовано та зроблено архітектуру масштабованого ФАТ на Playwright, готового до використання на будь-якому проекті зі складною бізнес-логікою

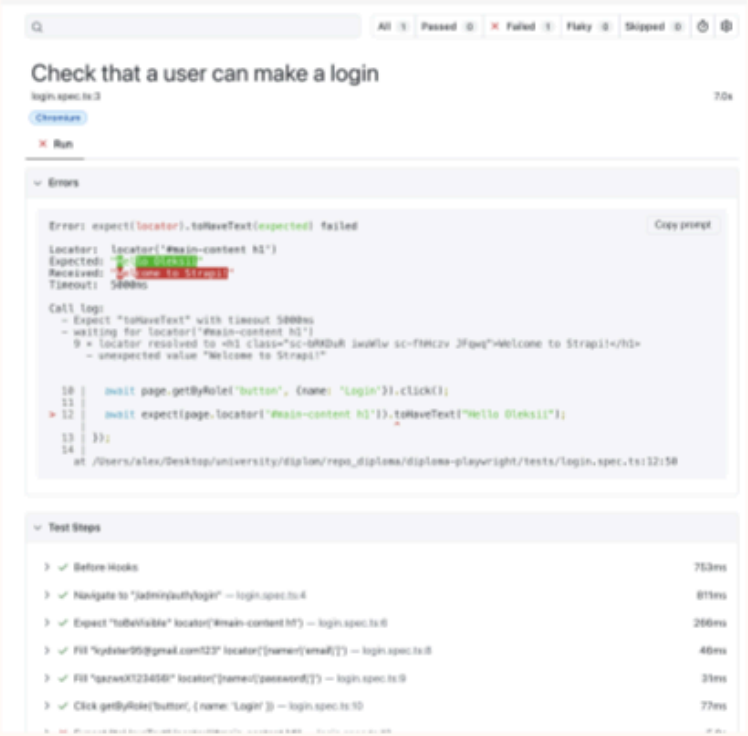
Реалізація базової функціональності ФАТ з використанням Playwright та TypeScript (конфігурація, керування браузерами)

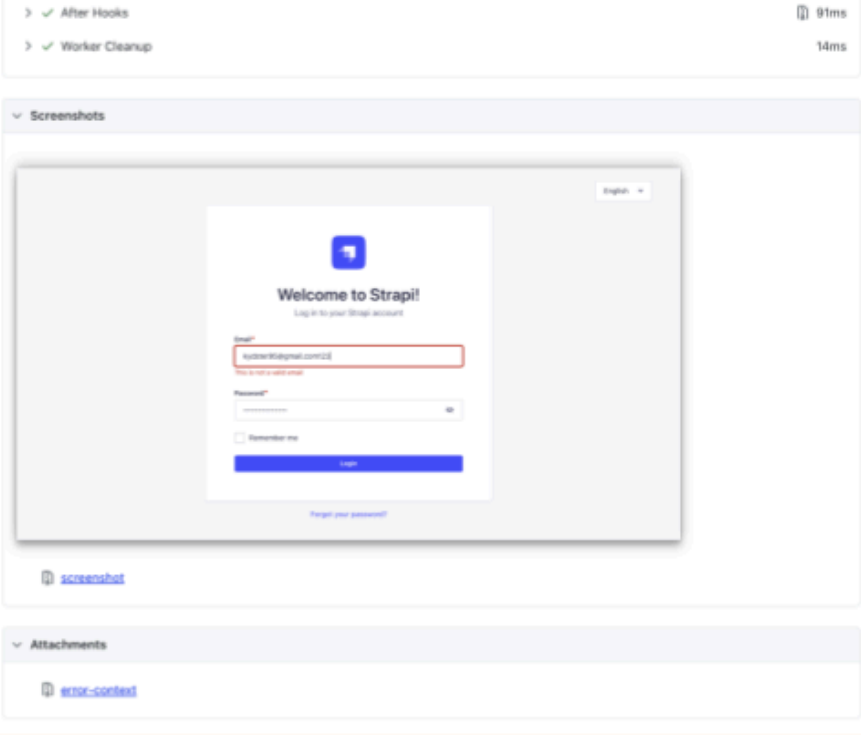
Реалізація першого тесту





Перевірка роботи тесту під час помилки

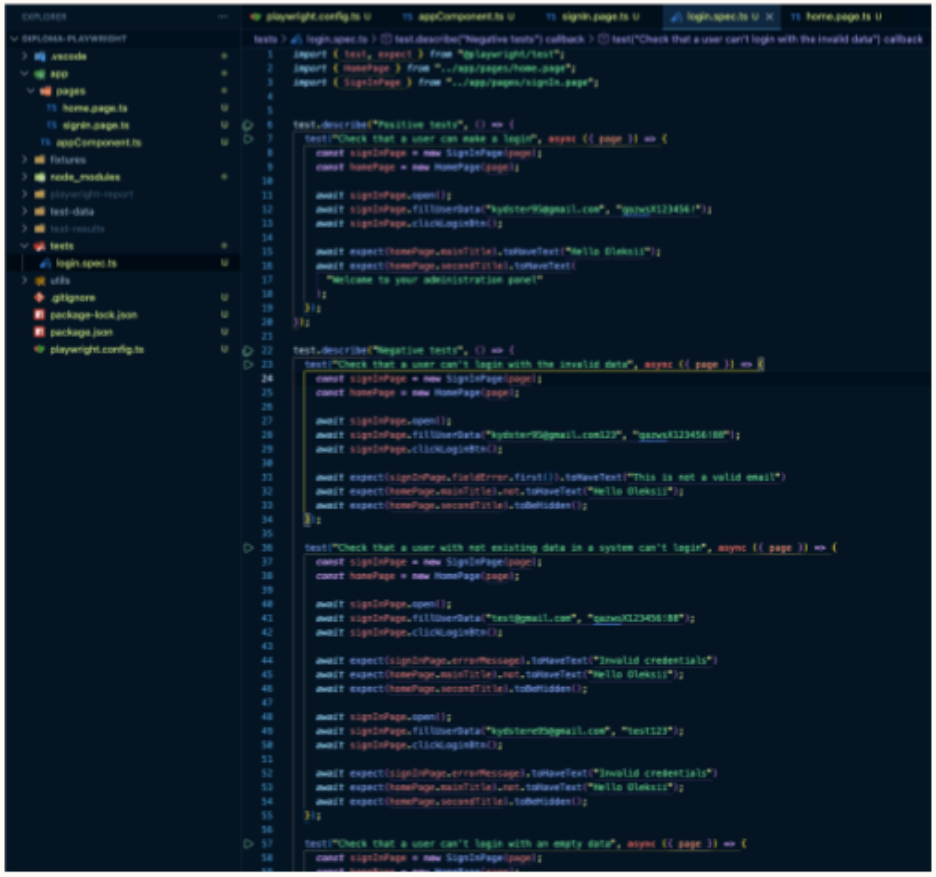




Впровадження Page Object паттерну

```
app > TS appComponent.ts > AppComponent
1   import { type Page } from '@playwright/test';
2
3   export abstract class AppComponent {
4     //Базовий клас від якого будь наслідуватись всі інші
5     constructor(protected page: Page) {}
6   }
```

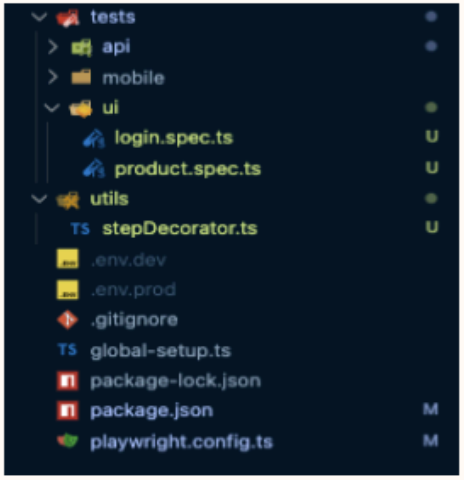
```
playwright.config.ts U   TS appComponent.ts U   TS signIn.page.ts U ×   login.spec.ts U   TS ho
app > pages > TS signIn.page.ts > ...
1   import { expect } from "@playwright/test";
2   import { AppComponent } from "../appComponent";
3
4   export class SignInPage extends AppComponent {
5     readonly mainTitle = this.page.locator('#main-content h1');
6     readonly emailField = this.page.locator('[name="email"]');
7     readonly passwordField = this.page.locator('[name="password"]');
8     readonly rememberMeCheckbox = this.page.getByRole("checkbox", { name: "Remember me" });
9     readonly loginButton = this.page.getByRole('button', {name: 'Login'});
10    readonly fieldError = this.page.locator('[data-strapi-field-error="true"]');
11    readonly errorMessage = this.page.locator('#global-form-error');
12
13    async open() {
14      await this.page.goto("http://localhost:1337/admin/auth/login");
15      await expect(this.mainTitle).toBeVisible();
16    }
17
18    async fillUserData(email: string, password: string) {
19      await this.emailField.fill(email);
20      await this.passwordField.fill(password);
21    }
22
23    async clickLoginBtn() {
24      await this.loginButton.click();
25    }
26  }
27
```



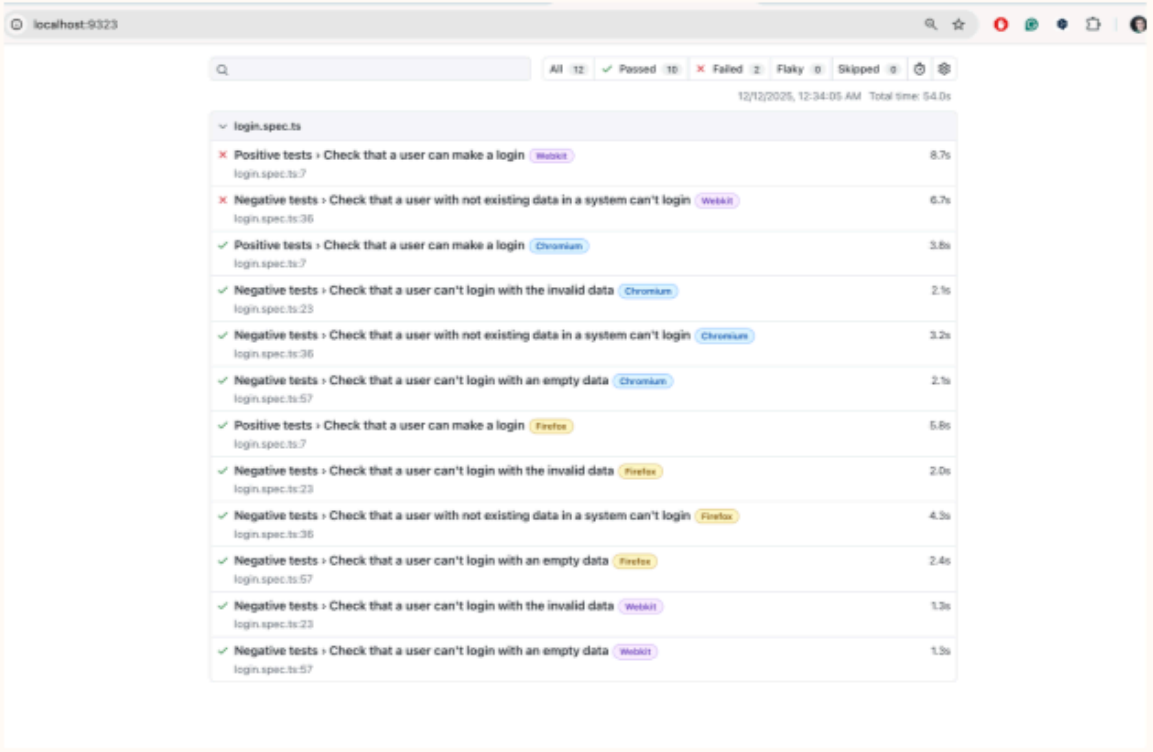
Впровадження механізму тегування тестів для фільтрації за середовищами

```
test.describe("Negative tests", () => {
  test("@E2E @regression Check that a user can't login with the invalid data", async ({ browser }) => {
    const context = await browser.newContext({
      storageState: undefined,
    });

    const page = await context.newPage();
    const signInPage = new SignInPage(page);
    const homePage = new HomePage(page);

    await signInPage.open();
    await signInPage.fillUserData(
      `${process.env.ADMIN_USER!}123`,
      process.env.ADMIN_PASSWORD!
    );
    await signInPage.clickLoginBtn();

    await expect(signInPage.fieldError.first()).toHaveText(
      "This is not a valid email"
    );
    await expect(homePage.mainTitle).not.toHaveText("Hello Oleksii");
    await expect(homePage.secondTitle).toBeHidden();
  });
```

```
# Приклад .env.dev
FRONTEND_URL=https://dev.apitable.com
API_URL=https://dev.apitable.com/api/v1
API_KEY=dev-123-api-key
```



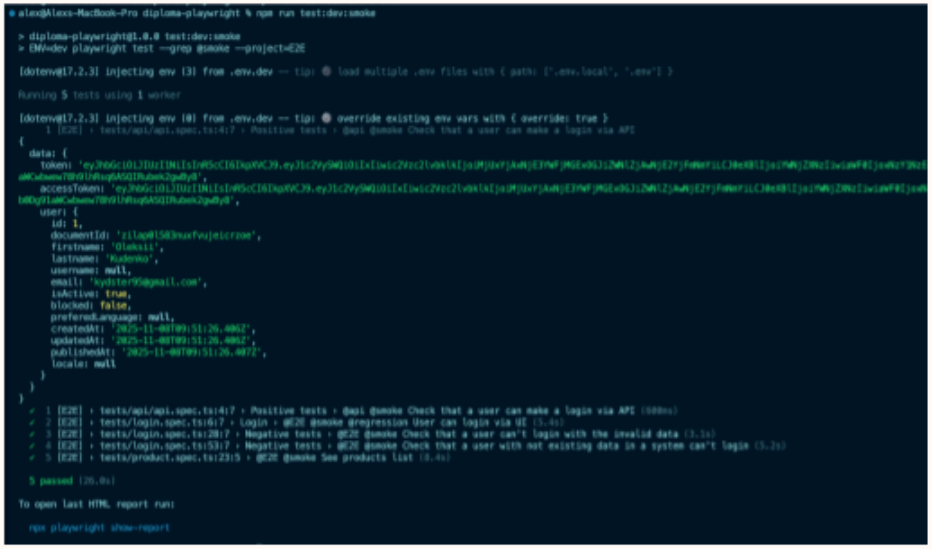
Перевірка запуску тестів у різних браузерах



Створення скриптів для запуску тестів відповідно тегам

package.json > ...

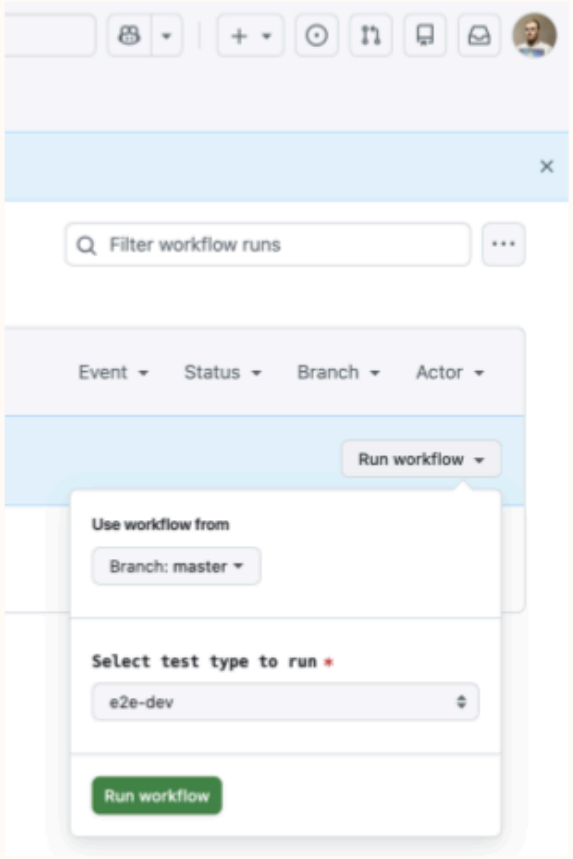
```
{
  "name": "diploma-playwright",
  "version": "1.0.0",
  "main": "index.js",
  Debug
  "scripts": {
    "test": "playwright test",

    "test:dev": "ENV=dev playwright test --project=E2E",
    "test:prod": "ENV=prod playwright test --project=E2E",

    "test:dev:e2e": "ENV=dev playwright test --grep @e2e --project=E2E",
    "test:prod:e2e": "ENV=prod playwright test --grep @e2e --project=E2E",

    "test:dev:api": "ENV=dev playwright test --grep @api",
    "test:prod:api": "ENV=prod playwright test --grep @api",

    "test:dev:smoke": "ENV=dev playwright test --grep @smoke --project=E2E",
    "test:prod:smoke": "ENV=prod playwright test --grep @smoke --project=E2E",

    "test:dev:regression": "ENV=dev playwright test --grep @regression --project=E2E",
    "test:prod:regression": "ENV=prod playwright test --grep @regression --project=E2E",

    "test:ui": "playwright test --ui"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "devDependencies": {
    "@playwright/test": "^1.57.0",
    "@types/node": "^24.10.1",
    "dotenv": "^17.2.3"
```

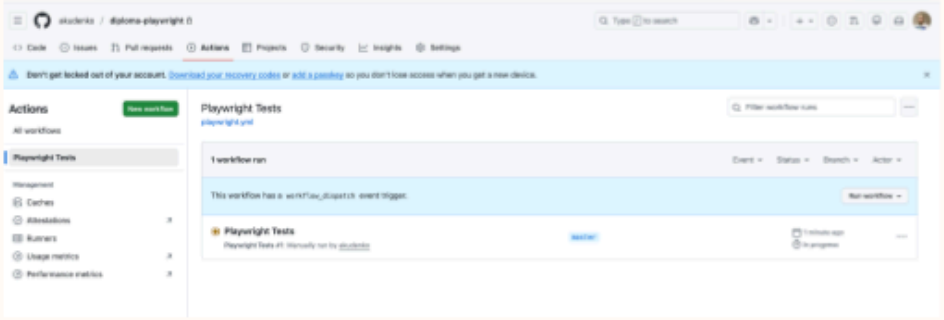


Аналіз швидкодії системи з використанням фікстур

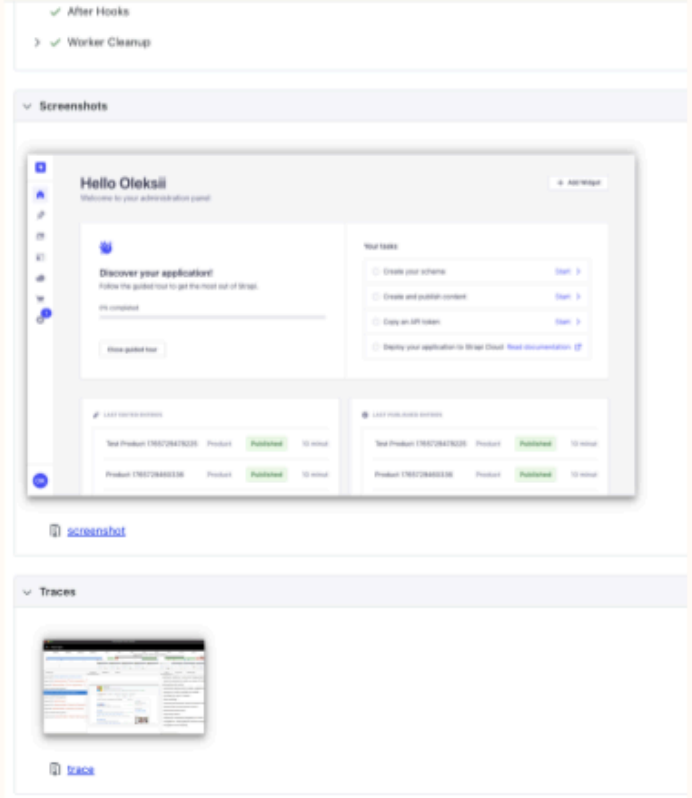
- тести пройшли за 30.5 секунд,
- тест виглядає набагато лаконічніше бо ми прибрали як роботу з beforeEach хуками на рівень фікстури так і роботу з ініціалізацією сторінок.

Інтеграція ФАТ з системою безперервної інтеграції





Налаштування та застосування системи звітності: візуалізація результатів, збір метрик та аналіз логів



Додаток Б

Код програми

1. playwright.config.ts - Файл з описом Глобальної конфігурації плеврайту

```
// playwright.config.ts
import { defineConfig, devices } from "@playwright/test";
import * as dotenv from "dotenv";

// 1. Динамічне завантаження .env файлів:
// Використовуємо 'dev' як середовище за замовчуванням
dotenv.config({
 path: process.env.ENV === "prod" ? ".env.prod" : ".env.dev",
});

export default defineConfig({
 // Загальні налаштування тестового раннера
 // globalSetup: "./global-setup.ts",
 testDir: "./tests",
 fullyParallel: true, // Дозволяє паралельне виконання тестів
 retries: 0, // Кількість повторних запусків при падінні

 // Налаштування для CI/CD
 workers: 1, // У CI обмежуємо кількість процесів

 // Секція use: глобальні налаштування для всіх тестів
 use: {
   // 2. Використання змінних середовища
   baseURL: process.env.BASE_URL,
   storageState: "storageState/admin.json", // Базовий URL з файлу .env

   // Керування артефактами
   trace: "retain-on-failure",
   screenshot: "only-on-failure",

   // Керування Headless-режимом
   headless: process.env.CI ? true : false, // Headless у CI, false
локально
 },

 // Керування звітністю
 reporter: [["list"], ["html", { open: "never" }]],

 // 3. Налаштування проєктів для керування браузерами (кросбраузерність)
 projects: [
   {
     name: "E2E",
     use: {
```

```
            ...devices["Desktop Chrome"],
            trace: "retain-on-failure",
            video: "retain-on-failure",
            screenshot: "only-on-failure",
          },
        },
        {
          name: "Firefox",
          use: { ...devices["Desktop Firefox"] },
        },
        {
          name: "Webkit",
          use: { ...devices["Desktop Safari"] },
        },
        {
          name: "API",
          use: {},
        },
      ],
    });
```

2. Package.json - Файл з данними про проект, залежностями та скриптами запуску тестів у різній конфігурації через термінал

```
    {
      "name": "diploma-playwright",
      "version": "1.0.0",
      "main": "index.js",
      "scripts": {
        "test": "playwright test",
        "test:dev": "ENV=dev playwright test --project=E2E",
        "test:prod": "ENV=prod playwright test --project=E2E",
        "test:dev:e2e": "ENV=dev playwright test --grep @e2e --project=E2E",
        "test:prod:e2e": "ENV=prod playwright test --grep @e2e --project=E2E",
        "test:dev:api": "ENV=dev playwright test --grep @api",
        "test:prod:api": "ENV=prod playwright test --grep @api",
        "test:dev:smoke": "ENV=dev playwright test --grep @smoke
--project=E2E",
        "test:prod:smoke": "ENV=prod playwright test --grep @smoke
--project=E2E",
        "test:dev:regression": "ENV=dev playwright test --grep @regression
--project=E2E",
        "test:prod:regression": "ENV=prod playwright test --grep @regression
--project=E2E",
        "test:ui": "playwright test --ui"
      },
```

```
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "devDependencies": {
    "@playwright/test": "^1.57.0",
    "@types/node": "^24.10.1",
    "dotenv": "^17.2.3"
  }
}
```

3. global-setup.ts - Файл який описує глобальне налаштування яке буде запускатись перед усіма тестами

```
export default async () => {
  const browser = await chromium.launch();
  const page = await browser.newPage();

  await page.goto("http://localhost:1337/admin/auth/login");

  await page.fill('[name="email"]', "kydster95@gmail.com");
  await page.fill('[name="password"]', "qazwsX123456!");
  await page.click('button[type="submit"]');

  await page.waitForURL("**/admin");

  await page.context().storageState({
    path: "storageState/admin.json",
  });

  await browser.close();
};
```

4. .env.dev - приклад файлу конфігурації для девелопмент оточення

```
BASE_URL=http://localhost:1337
ADMIN_USER=kydster95@gmail.com
ADMIN_PASSWORD=qazwsX123456!
```

5. /tests/ui/login.spec.ts - Приклад UI тестів для логіну

```
import { HomePage } from "../../app/pages/home.page";
import { SignInPage } from "../../app/pages/signIn.page";

test.describe("Login", () => {
 test("@E2E @regression User can login via UI", async ({ browser }) => {
   const context = await browser.newContext({
     storageState: undefined,
   });

   const page = await context.newPage();
   const signInPage = new SignInPage(page);
   const homePage = new HomePage(page);

   await signInPage.open();
   await signInPage.fillUserData(
     process.env.ADMIN_USER!,
     process.env.ADMIN_PASSWORD!
   );
   await signInPage.clickLoginBtn();

   await expect(homePage.mainTitle).toHaveText("Hello Oleksii");
   await context.close();
 });
});

test.describe("Negative tests", () => {
 test("@E2E @regression Check that a user can't login with the invalid data", async ({
browser }) => {
   const context = await browser.newContext({
     storageState: undefined,
   });

   const page = await context.newPage();
   const signInPage = new SignInPage(page);
   const homePage = new HomePage(page);

   await signInPage.open();
   await signInPage.fillUserData(
     `${process.env.ADMIN_USER!}123`,
     process.env.ADMIN_PASSWORD!
```

```
    );
    await signInPage.clickLoginBtn();

    await expect(signInPage.fieldError.first()).toHaveText(
      "This is not a valid email"
    );
    await expect(homePage.mainTitle).not.toHaveText("Hello Oleksii");
    await expect(homePage.secondTitle).toBeHidden();
  });

  test("@E2E @smoke Check that a user with not existing data in a system can't login",
async ({
    browser,
  }) => {
    const context = await browser.newContext({
      storageState: undefined,
    });

    const page = await context.newPage();
    const signInPage = new SignInPage(page);
    const homePage = new HomePage(page);

    await signInPage.open();
    await signInPage.fillUserData(
      "test@gmail.com",
      process.env.ADMIN_PASSWORD!
    );
    await signInPage.clickLoginBtn();

    await expect(signInPage.errorMessage).toHaveText("Invalid credentials");
    await expect(homePage.mainTitle).not.toHaveText("Hello Oleksii");
    await expect(homePage.secondTitle).toBeHidden();

    await signInPage.open();
    await signInPage.fillUserData(process.env.ADMIN_USER!, "test123");
    await signInPage.clickLoginBtn();

    await expect(signInPage.errorMessage).toHaveText("Invalid credentials");
    await expect(homePage.mainTitle).not.toHaveText("Hello Oleksii");
    await expect(homePage.secondTitle).toBeHidden();
  });

  test("Check that a user can't login with an empty data", async ({
    browser,
  }) => {
    const context = await browser.newContext({
      storageState: undefined,
```

```
  });

  const page = await context.newPage();
  const signInPage = new SignInPage(page);
  const homePage = new HomePage(page);

  await signInPage.open();
  await signInPage.fillUserData("", "");
  await signInPage.clickLoginBtn();

  await expect(signInPage.fieldError.nth(0)).toHaveText(
    "This value is required."
  );
  await expect(signInPage.fieldError.nth(1)).toHaveText(
    "This value is required."
  );
  await expect(homePage.mainTitle).not.toHaveText("Hello Oleksii");
  await expect(homePage.secondTitle).toBeHidden();
 });
});
```

6. /tests/ui/product.spec.ts - Приклад UI тестів для створення продукту

```
import { test, expect } from "../../fixtures/fixture";

test(`@E2E @smoke @regression Create product`, async ({ app, existingUser }) => {
  const productName = `Product ${Date.now()}`;
  const productPrice = `${Math.floor(1000 + Math.random() * 9000)}`;
  const productDescription = `Description ${Date.now()}`;

  await app.homePage.navigateToContentManagerMenuSection();
  await app.contentManagePage.navigateToProductMenuSection();
  await app.productPage.createNewProduct();
  await app.newProductComponent.fillData(
    productName,
    productPrice,
    productDescription
  );
  await app.newProductComponent.publishProduct();

  await expect(app.newProductComponent.successMessage).toHaveText('Custom Success
Message !');
 });

test("@E2E @regression See products list", async ({ app, existingUser, product }) => {
 await app.homePage.navigateToContentManagerMenuSection();
```

```
 await app.contentManagePage.navigateToProductMenuSection();
 await expect(app.productPage.publishedCells.first()).toBeVisible();
});
```

7. /tests/api/login.spec.ts - Приклад API тесту для логіну

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

test.describe("Positive tests", () => {
 test("@api Check that a user can make a login via API", async ({ request }) => {


   const response = request.post("http://localhost:1337/admin/login", {
     data: {
       email: "kydster95@gmail.com",
       password: "qazwsX123456!",
       deviceId: "a2a385d3-aa60-45f0-be36-928c865d200c",
       rememberMe: false,
     },
   });

   expect((await response).status()).toBe(200);
   const responseBody = await (await response).json();
   expect(responseBody).toHaveProperty("data");

   console.log(responseBody);

   const data = responseBody.data;
   expect(data.token).not.toBeNull();
   expect(data.accessToken).not.toBeNull();

   expect(data.user.firstname).toBe("Oleksii");
   expect(data.user.lastname).toBe("Kudenko");
 });
});
```

8. /tests/api/product.spec.ts - Приклад API тесту для створення продукту

```
import { test, expect } from "../../fixtures/fixture";
import { ProductPayload } from "../../app/api/controllers/product.controller";

test.describe("@api @regression Product API", () => {
 test("Admin can create product via API", async ({ app, existingUser }) => {
   const payload: ProductPayload = {
     Name: `Test Product ${Date.now()}`,
```

```
      Price: 999,
      Image: null,
      Description: [
        {
          type: "paragraph",
          children: [{ type: "text", text: "API Product" }],
        },
      ],
    };

    const response = await app.api.product.createProduct(
      payload,
      existingUser.token
    );

    expect(response.ok()).toBeTruthy();

    const body = await response.json();
    expect(body.data.id).toBeDefined();
  });
});
```

9. /app/app.Component.ts - Базовий клас який для Page Object патерну

```
import { type Page } from '@playwright/test';

export abstract class AppComponent {
  //Базовий клас від якого будь наслідуватись всі інші
  constructor(protected page: Page) {}
}
```

10. app/app.ts - Базовий клас нашого тест фреймворку де є глобальні методи та ініалізація наших сторінок та компонентів

```
import { Page } from "@playwright/test";
import { API } from "./api";
import { AppComponent } from "./appComponent";
import { HomePage } from "./pages/home.page";
import { ProductPage } from "./pages/product.page";
import { ContentManagePage } from "./pages/content.manage.page";
import { NewProductComponent } from "./pages/components/new.product";
import { SignInPage } from "./pages/signIn.page";

type LoginPayload = {
  email: string;
```

```
 password: string;
 deviceId: string;
 rememberMe: boolean;
};

let cachedToken: string | undefined;

export class Application extends AppComponent {
 public api: API;

 signInPage: SignInPage;
 homePage: HomePage;
 productPage: ProductPage;
 contentManagePage: ContentManagePage;
 newProductComponent: NewProductComponent;

 constructor(page: Page) {
   super(page);
   this.api = new API(page.request);

   this.signInPage = new SignInPage(page);
   this.homePage = new HomePage(page);
   this.productPage = new ProductPage(page);
   this.contentManagePage = new ContentManagePage(page);
   this.newProductComponent = new NewProductComponent(page);
 }

 async loginViaApi(data: LoginPayload): Promise<string> {
   if (cachedToken) {
     return cachedToken;
   }

   const response = await this.api.auth.login(data);

   if (!response.ok()) {
     throw new Error(`Login failed: ${response.status()}`);
   }

   const body = await response.json();
   const token = body?.data?.token;

   if (!token) {
     throw new Error("Login response does not contain token");
   }

   cachedToken = token;
   return token;
```

```
  }

  async bootstrapAdminAuth(token: string, deviceId: string) {
    await this.page.goto("/admin", { waitUntil: "commit" });

    await this.page.evaluate(
      ({ token, deviceId }) => {
        localStorage.setItem("isLoggedIn", "true");
        localStorage.setItem("token", token);
        localStorage.setItem("strapi.admin.deviceId", deviceId);
        localStorage.setItem(
          "STRAPI_FREE_TRIAL_ENDED_MODAL:16b079ba-f282-46a3-9871-401c35f4ecee",
          "false"
        );
      },
      { token, deviceId }
    );

    await this.page.reload();
  }
}
```

11. /app/pages/components/new.product.ts - Файл компоненту Новий Продукт

```
import { expect } from "@playwright/test";
import { AppComponent } from "../../appComponent";

export class NewProductComponent extends AppComponent {
  readonly mainTitle = this.page.locator('#main-content h1');
  readonly nameInput = this.page.locator('[name="Name"]');
  readonly priceInput = this.page.locator('[name="Price"]');
  readonly descriptionInput = this.page.locator('p[data-slate-node="element"]');
  readonly publishButton = this.page.getByRole('button', {name: 'Publish'});
  readonly successMessage = this.page.locator('[role="status"] p');


  async open() {
    await
this.page.goto("http://localhost:1337/admin/content-manager/single-types/api::about.ab
out");
    await expect(this.mainTitle).toBeVisible();
  }

  async fillData(name: string, price: string, description: string) {
    await this.nameInput.fill(name);
    await this.priceInput.fill(price);
```

```
    await this.descriptionInput.fill(description);
  }

  async publishProduct(){
    await this.publishButton.click();
  }
}
```

12. /app/pages/content.manage.page.ts - Файл управління контентом

```
import { expect, Locator } from "@playwright/test";
import { AppComponent } from "../appComponent";

export class ContentManagePage extends AppComponent {
  readonly mainTitle = this.page.locator('#main-content h1');
  readonly productCollectionType =
this.page.locator('a[href="/admin/content-manager/collection-types/api::product.produc
t?page=1&pageSize=10&sort=Name%3AASC"]');
  readonly guideWrapper = this.page.locator('[aria-labelledby="guided-tour-title"]');
  readonly guideSkipButton = this.page.getByRole('button', {'name': 'Skip'});

  async open() {
    await
this.page.goto("http://localhost:1337/admin/content-manager/single-types/api::about.ab
out");
    await expect(this.mainTitle).toBeVisible();
  }

  async navigateToProductMenuSection(){
    await expect(this.mainTitle).toHaveText('About the strapi blog');
    await this.skipGuidePopup();
    await this.productCollectionType.click();
    await expect(this.mainTitle).toHaveText('Product');
  }

  async skipGuidePopup(){
    if(await this.guideWrapper.isVisible()) {
        await this.guideSkipButton.click();
    }
  }
}
```

13. /app/pages/home.page.ts - Файл з описом головної сторінки

```
import { expect } from "@playwright/test";
```

```
import { AppComponent } from "../appComponent";

export class HomePage extends AppComponent {
 readonly mainTitle = this.page.locator('#main-content h1');
 readonly secondTitle = this.page.locator('[data-strapi-header="true"] p');
 readonly homeMenuSection = this.page.locator('a[href="/admin"]');
 readonly mediaLibraryMenuSection =
this.page.locator('a[href="/admin/plugins/upload"]');
 readonly contentTypeBuilderMenuSection =
this.page.locator('a[href="/admin/plugins/content-type-builder"]');
 readonly contentManagerMenuSection = this.page.locator('nav
a[href="/admin/content-manager"]');

 async open() {
   await this.page.goto("http://localhost:1337/admin");
   await expect(this.mainTitle).toBeVisible();
 }

 async navigateToContentManagerMenuSection(){
   await this.contentManagerMenuSection.click();
 }
}
```

14. /app/pages/product.page.ts - Файл з описом сторінки продукту

```
import { expect, Locator } from "@playwright/test";
import { AppComponent } from "../appComponent";

export class ProductPage extends AppComponent {
 readonly mainTitle = this.page.locator('#main-content h1');
 readonly createNewEntryButton =
this.page.locator('a[href="/admin/content-manager/collection-types/api::product.produc
t/create"]');
 readonly publishedCells = this.page.locator('tbody td', { hasText: 'Published' });

 async open() {
   await
this.page.goto("http://localhost:1337/admin/content-manager/collection-types/api::prod
uct.product?page=1&pageSize=10&sort=Name%3AASC");
   await expect(this.mainTitle).toBeVisible();
 }

 async createNewProduct(){
   await this.createNewEntryButton.click();
   await expect(this.mainTitle).toHaveText("Create an entry");
 }
```

```
}
```

15. /app/pages/signIn.page.ts - файл з описом сторінки авторизації

```
import { expect } from "@playwright/test";
import { AppComponent } from "../appComponent";
import { step } from "../../utils/stepDecorator";

export class SignInPage extends AppComponent {
 readonly mainTitle = this.page.locator('#main-content h1');
 readonly emailField = this.page.locator("[name='email']");
 readonly passwordField = this.page.locator("[name='password']");
 readonly rememberMeCheckbox = this.page.getByRole("checkbox", { name: "Remember me"
});
 readonly loginButton = this.page.getByRole('button', {name: 'Login'});
 readonly fieldError = this.page.locator('[data-strapi-field-error="true"]');
 readonly errorMessage = this.page.locator('#global-form-error');

 @step('Open the Home page')
 async open() {
   await this.page.goto("http://localhost:1337/admin/auth/login");
   await expect(this.mainTitle).toBeVisible();
 }

 @step('Fill user data: email: ${email}, password: ${password}')
 async fillUserData(email: string, password: string) {
   await this.emailField.fill(email);
   await this.passwordField.fill(password);
 }

 @step('Click on the Login button')
 async clickLoginBtn() {
   await this.loginButton.click();
 }
}
```

16. /api/index.ts - Клас з ініціалізацією апі реквестів

```
import { AuthController } from "./controllers/auth.controller";
import { ProductController } from "./controllers/product.controller";
import { RequestHolder } from "./requestHolder";

export class API extends RequestHolder {
 public readonly auth = new AuthController(this.request);
 public readonly product = new ProductController(this.request);
}
```

17. /api/requestHolder.ts - Базовий клас для роботи з АПІ

```ts
import { APIRequestContext } from '@playwright/test';

export abstract class RequestHolder {
 constructor(protected request: APIRequestContext) {}
}
```

18. /api/controllers/auth.controller.ts - Файл з описом контроллеру авторизації

```ts
import { RequestHolder } from "../requestHolder";
import { APIResponse } from "@playwright/test";

type LoginPayload = {
   email: string;
   password: string;
   deviceId: string;
   rememberMe: boolean;
};

export class AuthController extends RequestHolder {
 async login(data: LoginPayload): Promise<APIResponse> {
   const response = await this.request.post(`http://localhost:1337/admin/login`, {
     data: {
       email: data.email,
       password: data.password,
     },
   });

   return response;
 }
}
```

19. /api/controllers/product.controller.ts - Файл з описом контроллеру продукта

```ts
import { APIRequestContext } from "@playwright/test";

export type ProductPayload = {
 Name: string;
 Price: number;
 Description: ProductDescription[];
 Image: string | null;
};

type TextNode = {
 type: "text";
```

```
 text: string;
};

type ProductDescription = {
 type: "paragraph";
 children: TextNode[];
};

export class ProductController {
 constructor(private request: APIRequestContext) {}

 createProduct(payload: ProductPayload, token: string) {
   return
this.request.post("http://localhost:1337/content-manager/collection-types/api::product
.product/actions/publish?", {
     headers: {
       Authorization: `Bearer ${token}`,
     },
     data: payload,
   });
 }

 deleteProduct(id: number, token: string) {
   return
this.request.delete(`http://localhost:1337/content-manager/collection-types/api::produ
ct.product/${id}`, {
     headers: {
       Authorization: `Bearer ${token}`,
     },
   });
 }
}
```

20. /fixtures/fixture.ts - Файл з описом фікстури

```
import { test as base, expect } from "@playwright/test";
import { Application } from "../app/app";
import { ProductPayload } from "../app/api/controllers/product.controller";

type UserModel = {
 email: string;
 password: string;
 deviceId: string;
 rememberMe: boolean;
};
```

```
type Fixtures = {
 app: Application;
 existingUser: {
   userModel: UserModel;
   token: string;
 };
 product: {
   id: number;
   payload: ProductPayload;
 };
};

export const test = base.extend<Fixtures>({
 app: async ({ page }, use) => {
   await use(new Application(page));
 },

 existingUser: async ({ app }, use) => {
   const userModel: UserModel = {
     email: process.env.ADMIN_USER!,
     password: process.env.ADMIN_PASSWORD!,
     deviceId: "a2a385d3-aa60-45f0-be36-928c865d200c",
     rememberMe: false,
   };

   const token = await app.loginViaApi(userModel);
   await app.bootstrapAdminAuth(token, userModel.deviceId);

   await expect(app.homePage.mainTitle).toHaveText("Hello Oleksii");

   await use({ userModel, token });
 },

 product: async ({ app, existingUser }, use) => {
   const payload: ProductPayload = {
     Name: `Test Product ${Date.now()}`,
     Price: 999,
     Image: null,
     Description: [
       {
         type: "paragraph",
         children: [{ type: "text", text: "API Product" }],
       },
     ],
   };

   const createResponse = await app.api.product.createProduct(
```

```
      payload,
      existingUser.token
    );

    const body = await createResponse.json();
    const productId = body.data.id;

    await use({ id: productId, payload });

    await app.api.product.deleteProduct(productId, existingUser.token);
  },
});

export { expect };
```

УДК 621.34
***О.В. Шефер**, д.т.н., професор,*
***О.О. Куденко**, магістрант*
Національний університет «Полтавська політехніка імені Юрія Кондратюка»

РОЗРОБКА ФРЕЙМВОРКУ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ВЕБ-ЗАСТОСУНКІВ НА ОСНОВІ PLAYWRIGHT ТА TYPESCRIPT

Сучасні веб-застосунки характеризуються високою динамічністю, багаторівневою архітектурою та розподіленою моделлю надання послуг, що суттєво ускладнює процес їх тестування. Зростаючі вимоги до надійності, стабільності та швидкості розгортання оновлень вимагають впровадження ефективних інструментів автоматизації UI/API. Саме тому розробка фреймворку автоматизованого тестування (ФАТ), який відповідає принципам масштабованості, гнучкості та розширюваності, є актуальним завданням інженерії програмного забезпечення.

У роботі проведено аналіз сучасних засобів автоматизації, зокрема Selenium WebDriver, Selenide, Cypress, WebdriverIO та Robot Framework. Показано, що на відміну від драйвер-орієнтованих рішень, Playwright (розробка Microsoft) використовує прямий доступ до браузерних протоколів Chromium, WebKit та Firefox, що забезпечує високу швидкість, детермінованість, стабільність тестів та зменшує кількість flaky-випадків. Додатковою перевагою є вбудовані механізми автоматичного очікування (Actionability Checks) та підтримка повноцінної ізоляції тестів через окремі Browser Context [1].

Типізація та об’єктно-орієнтоване моделювання за допомогою TypeScript (також розробка Microsoft) дозволили реалізувати фреймворк зі строгою структурою, передбачуваними залежностями та високою читабельністю коду. На основі аналізу вимог розроблено архітектуру ФАТ, що включає [2-5]:

23. реалізацію Page Object Model (POM) для UI-рівня;
24. створення компонентної моделі (Component Objects) для повторюваних UI-елементів;
25. використання фікстур Playwright для ізоляції тестів, підготовки даних та попередніх умов;
26. модулі для API-взаємодії (HTTP-клієнт Playwright);
27. централізовану систему конфігурацій середовищ (dev/stage/prod) на базі .env;
28. генератори тестових даних, утиліти та допоміжні сервіси;
29. інтеграцію з CI/CD (GitHub Actions) та формування HTML-/trace-репортів.

Структура створеного фреймворку є модульною, відповідає принципам Clean Architecture та забезпечує повне розділення відповідальностей.

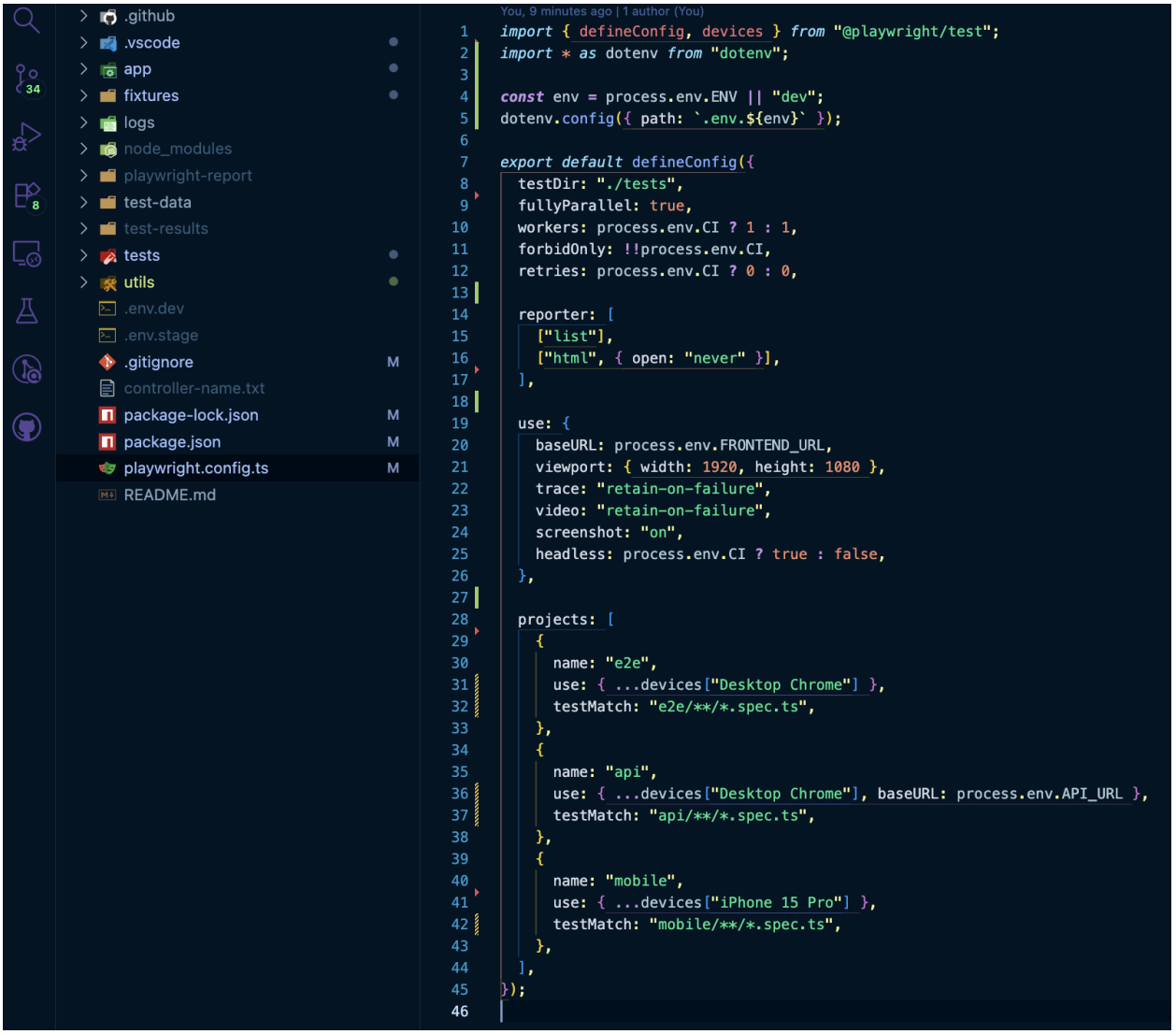


Рис.1 Приклад файлової структури створенного фреймворку

Тестові сценарії реалізовано у вигляді бізнес-орієнтованих кроків, що спираються на POM та фікстури. У результаті проведених експериментів встановлено, що використання Playwright з TypeScript дозволяє досягти:

- підвищення швидкості тестів у 2–5 разів порівняно з WebDriver-орієнтованими інструментами;
- зниження кількості нестабільних тестів (flaky) завдяки автоматичному очікуванню станів елементів;
- зменшення часу розробки сценаріїв та спрощення їх підтримки за рахунок чіткої архітектурної моделі;
- можливості горизонтального масштабування тестування у рамках CI/CD.

Отже, запропонований ФАТ на основі Playwright та TypeScript може бути рекомендований як сучасне, ефективне та придатне до промислового використання рішення для автоматизації UI/API тестування веб-застосунків.

ЛІТЕРАТУРА:

5. Playwright. Getting Started [Електронний ресурс]. URL: https://playwright.dev/docs/intro (дата звернення: 25.09.2025).
6. Playwright. Test Runner [Електронний ресурс]. URL: https://playwright.dev/docs/test-intro (дата звернення: 25.09.2025).
7. Playwright. Page Object Models [Електронний ресурс]. URL: https://playwright.dev/docs/pom (дата звернення: 25.09.2025).
8. Playwright. Best Practices [Електронний ресурс]. URL: https://playwright.dev/docs/best-practices (дата звернення: 25.09.2025).
9. Playwright. API Reference [Електронний ресурс]. URL: https://playwright.dev/docs/api/class-playwright (дата звернення: 25.09.2025).
10. TypeScript. Handbook [Електронний ресурс]. URL: https://www.typescriptlang.org/docs/ (дата звернення: 25.09.2025).

DEVELOPMENT OF AN AUTOMATED TESTING FRAMEWORK FOR WEB APPLICATIONS BASED ON PLAYWRIGHT AND TYPESCRIPT

O. Shefer, *Doctor of Science, professor,*
O. Kudenko, *undergraduate*
National University «Yuri Kondratyuk Poltava Polytechnic»