

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**

КВАЛІФІКАЦІЙНА РОБОТА
на тему: «Розробка фреймворку автоматизованого
тестування для web-застосунку AgroManagement»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

*Кваліфікаційна робота містить результати власних досліджень.
Використання ідей, результатів і текстів інших авторів мають посилання
на відповідне джерело*

_____ Павло АРАБАДЖИ
(підпис)

Виконав: здобувач вищої освіти групи ПД-41

_____ Павло АРАБАДЖИ

Керівник: _____
викладач кафедри ІПЗ

_____ Юрій БАЖАН

Рецензент: _____

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

Інженерії програмного забезпечення

_____ Ірина ЗАМРІЙ

« ____ » _____ 2026 р.

**ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

Арабаджи Павлу Олександровичу

1. Тема кваліфікаційної роботи: «Розробка фреймворку автоматизованого тестування для web-застосунку AgroManagement»

керівник кваліфікаційної роботи викладач кафедри ІІЗ Юрій БАЖАН,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «29» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості з методів і видів автоматизованого тестування web-застосунків (функціональне, API/UI, інтеграційне, наскрізне та безпекове тестування); нормативні та методичні матеріали щодо якості програмного забезпечення; технічна документація до платформи Playwright, мови TypeScript, системи контролю версій Git та сервісу GitHub Actions.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Дослідження предметної області та функціональних можливостей вебзастосунку AgroManagement.

2. Порівняльний аналіз інструментів автоматизованого тестування вебзастосунків та обрання інструмента реалізації фреймворку.

3. Визначення функціональних та нефункціональних вимог до тестового фреймворку.
4. Визначення архітектури фреймворку автоматизованого тестування та моделі організації тестового коду.
5. Реалізація фреймворку автоматизованого тестування для перевірки основних функціональних сценаріїв AgroManagement.
6. Запуски автоматизованих тестів та оцінка результатів застосування фреймворку.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Багатошарова архітектура.
5. Діаграма послідовностей.
6. Екранні форми.
7. Апробація результатів дослідження

6. Дата видачі завдання «23» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	23.02-28.02.2026	
2	Аналіз та дослідження існуючих аналогів	01.03-07.03.2026	
3	Деталізація предметної області AgroManagement, вимог і обґрунтування стеку автотестування	01.04-14.04.2026	
4	Проектування архітектури фреймворку та моделі тестових сценаріїв	15.04-24.04.2026	
5	Програмна реалізація фреймворку (тести, конфігурація, CI)	25.04-05.05.2026	
6	Прогони тестів, аналіз результатів, виправлення нестабільностей / регресія	06.05-10.05.2026	
7	Оформлення роботи: вступ, висновки, реферат	06.05-10.05.2026	
8	Розробка демонстраційних матеріалів	08.05-10.05.2026	
9	Попередній захист роботи	11.05-22.05.2026	

Здобувач вищої освіти _____
(підпис)

Павло АРАБАДЖИ

Керівник кваліфікаційної роботи _____
(підпис)

Юрій БАЖАН

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 59 стор., 2 табл., 21 рис., 16 джерел.

Мета роботи – автоматизація тестування основних функціональних сценаріїв вебзастосунку AgroManagement для зменшення кількості помилок та часу їх виявлення.

Об'єкт дослідження – процес автоматизації тестування вебзастосунку AgroManagement.

Предмет дослідження – методи та засоби автоматизації тестування.

Короткий зміст роботи: У роботі розглянуто підходи до автоматизації тестування вебзастосунків, узагальнено переваги та обмеження типових стеків (Selenium WebDriver, Cypress, Playwright). Розроблено архітектуру та програмну реалізацію фреймворку на TypeScript для вебзастосунку AgroManagement: окремі проекти Playwright для UI та API, шар сторінок за принципом Page Object, контролери для звернень до API, фікстури автентифікації й доменних передумов, фабрики тестових даних, набори сценаріїв для UI та API, наскрізних бізнес-потоків і перевірок безпеки доступу. Проведено автоматизовані функціональні та інтеграційні тести зі звітністю Playwright. Налаштовано workflow GitHub Actions для виконання наборів у середовищі безперервної інтеграції. У роботі використано Playwright Test, TypeScript, Node.js, dotenv для параметризації середовищ та GitHub Actions для автоматизації конвеєра перевірок.

Сферою використання застосунку є підтримка регресійного та приймального контролю функціональності вебсистеми AgroManagement у циклі розробки програмного забезпечення для агрономічного обліку та керування даними.

КЛЮЧОВІ СЛОВА: АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ, PLAYWRIGHT, TYPESCRIPT, PAGE OBJECT, API-ТЕСТИ, UI-ТЕСТИ.

ЗМІСТ

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ	12
1.1 Характеристика предметної галузі	12
1.2 Аналіз існуючих аналогів інструментів автоматизації тестування	14
1.2.1 Аналіз Selenium WebDriver	15
1.2.2 Аналіз Cypress	18
1.2.3 Аналіз Playwright	22
1.3 Визначення проблем предметної галузі	26
1.4 Постановка завдання на розробку тестового фреймворку	28
1.5 Визначення вимог до тестового фреймворку	30
2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ФРЕЙМВОРКУ	33
2.1 Загальна архітектура та структура програмного комплексу автоматизації	33
2.2 Інструментальний стек і конфігурація середовища виконання тестів	35
2.3 Проєктування UI-рівня на основі Page Object Model	36
2.4 Проєктування підготовки тестових даних і композиції фікстур	39
2.5 Інтеграція CI/CD та формування тестових звітів	40
3 РЕАЛІЗАЦІЯ ТА РОЗРОБКА ТЕСТІВ	42
3.1 Загальна характеристика реалізації та організація тестового коду	42
3.2 Реалізація UI-автотестів (Page Object і специфікації)	44
3.3 Реалізація API-автотестів (HTTP-клієнт і контролери доступу)	47
3.4 Сквозні E2E-сценарії та перевірки безпеки	50
3.5 Звітність Playwright, артефакти діагностики та інтерпретація результатів прогонів	53
4 ОЦІНКА ЕФЕКТИВНОСТІ	60
4.1 Аналіз швидкості виконання тестів	60
4.2 Охоплення системи автоматизованими тестами	63
4.3 Доцільність впровадження автоматизації тестування	65

ВИСНОВКИ	68
ПЕРЕЛІК ПОСИЛАНЬ	70
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ	72
ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ	79

ВСТУП

Актуальність: у сучасних умовах цифровізації аграрного сектору та ускладнення вебзастосунків забезпечення стабільної якості програмного забезпечення стає критично важливим завданням. Для системи AgroManagement помилки у сценаріях авторизації, керування доменними сутностями (ферми, поля, культури, посіви, врожаї) і контролю прав доступу можуть призводити до порушення цілісності даних і зниження надійності експлуатації. Ручна регресійна перевірка таких сценаріїв є трудомісткою, повільною та залежною від людського чинника. Тому розробка фреймворку автоматизованого тестування є актуальним практичним напрямом, що забезпечує регулярний контроль якості, швидший зворотний зв'язок для команди розробки та підвищення передбачуваності релізного процесу.

Мета роботи: автоматизація тестування основних функціональних сценаріїв вебзастосунку AgroManagement для зменшення кількості помилок та часу їх виявлення.

Об'єкт дослідження: процес автоматизації тестування вебзастосунку AgroManagement.

Предмет дослідження: методи та засоби автоматизації тестування.

Для досягнення поставленої мети у кваліфікаційній роботі сформульовано такі завдання:

1. Дослідити предметну область та функціональні можливості вебзастосунку AgroManagement з метою визначення критичних зон перевірок і обґрунтування вимог до автоматизації тестування;
2. Провести порівняльний аналіз існуючих інструментів автоматизованого тестування вебзастосунків для обрання найбільш доцільного інструмента реалізації фреймворку;

3. Визначити функціональні та нефункціональні вимоги до тестового фреймворку задля забезпечення повноти та відтворюваності автоматизованих перевірок;

4. Визначити архітектуру фреймворку автоматизованого тестування та модель організації тестового коду для підвищення підтримуваності й масштабованості набору тестів;

5. Реалізувати фреймворк автоматизованого тестування для перевірки основних функціональних сценаріїв вебзастосунку AgroManagement;

6. Провести запуски автоматизованих тестів та оцінити результати його застосування з метою підтвердження практичної доцільності автоматизації та ефективності запропонованого рішення.

Методи дослідження: аналіз і синтез технічних рішень у сфері test automation; порівняльний аналіз інструментів Selenium, Cypress, Playwright; методи проєктування програмних систем; методи функціонального, інтеграційного, безпекового та наскрізного тестування; емпіричний аналіз результатів виконання тестових наборів.

Практичне значення одержаних результатів полягає у створенні відтворюваного фреймворку автоматизованого тестування для AgroManagement, який забезпечує структурований контроль якості за функціональними зонами, підтримує CI/CD-процес і може бути використаний як база для подальшого розширення тестового покриття в умовах командної розробки.

Наукова новизна роботи полягає у формуванні цілісної моделі автоматизованого тестування для вебзастосунку AgroManagement, яка об'єднує функціональні UI- та API-перевірки, безпекові ownership/direct-link сценарії та наскрізні E2E-потоки в єдиному керованому контурі на базі Playwright. На відміну від фрагментарного підходу, у роботі запропоновано трасованість «функціональна вимога — тестовий сценарій», feature-орієнтовану модель оцінювання покриття та практику оцінки ефективності

без підміни результатів штучними метриками, що не підтверджені артефактами проєкту.

Практична значущість результатів полягає у створенні та апробації відтворюваного фреймворку автоматизованого тестування, який може застосовуватися в процесі супроводу AgroManagement для регулярної перевірки критичних бізнес-сценаріїв і швидкого виявлення регресій. Запропоноване рішення інтегрується з CI/CD-процесом, підтримує діагностичні артефакти (звітність, trace, screenshot/video) і забезпечує прозоре функціональне покриття за функціональними можливостями, що підвищує керованість контролю якості та зменшує ризик пропуску критичних дефектів під час розвитку системи.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Характеристика предметної галузі

Цифровізація аграрного сектору супроводжується переходом від локального обліку до інтегрованих вебзастосунків, у яких агрегуються та узгоджуються дані про виробничі одиниці та операції [1]. Для таких систем критичними атрибутами якості програмного забезпечення є надійність функціонування, безпека та придатність до супроводу, оскільки помилки обліку та доступу безпосередньо впливають на довіру до даних і прийняття управлінських рішень [1].

Предметна область цієї кваліфікаційної роботи пов'язана з автоматизованим тестуванням вебзастосунку AgroManagement, у якому реалізовано керування взаємопов'язаними доменними сутностями: ферма (Farm), поле (Field), культура (Crop), посів (Planting) і врожай (Harvest). Наявність ієрархії та залежностей між сутностями означає, що система належить до класу доменно-орієнтованих інформаційних рішень, де коректність бізнес-правил визначається не окремими екранними формами, а цілісністю ланцюжка операцій.

З інженерної точки зору предметна галузь характеризується високою чутливістю до цілісності зв'язків: запис про врожай (Harvest) має сенс лише за наявності валідного посіву (Planting), а посів — за коректної прив'язки до поля (Field) та культури (Crop). Тому забезпечення якості не може обмежуватися точковою перевіркою окремих CRUD-операцій; необхідні наскрізні сценарії, що підтверджують узгодженість даних і поведінки системи на різних рівнях представлення (інтерфейс користувача та програмні інтерфейси доступу до даних).

Другою визначальною ознакою є безпекова модель та розмежування доступу між ролями користувача. Для web-застосунків типовими ризиками є

порушення авторизації, некоректна маршрутизація після автентифікації та можливість несанкціонованого доступу до ресурсів іншого користувача [4]. У контексті AgroManagement це формулюється як вимога системно перевіряти не лише позитивні сценарії, а й негативні випадки (відмова в доступі, коректність редиректів, ownership-обмеження).

Третьою характеристикою є процесна необхідність регулярних і відтворюваних регресійних перевірок у середовищі безперервної інтеграції. Дослідження нестабільних (flaky) тестів показують, що вони знижують ефективність процесу тестування, уповільнюють інтеграцію змін і підбивають довіру до автоматичних перевірок [2]. Окремі емпіричні роботи підкреслюють значні витрати ресурсів безперервної інтеграції, пов'язані з виявленням та обробкою нестабільних тестів [3]. Отже, предметна галузь вимагає не лише наявності тестів, а й інженерно стабільного тестового контуру.

Узагальнюючи, предметна область охоплює взаємопов'язані компоненти: доменну бізнес-логіку (сутності Farm, Field, Crop, Planting, Harvest), інтерфейсні сценарії, політики безпеки та інфраструктурний контур виконання перевірок. Саме ця структура визначає постановку задачі щодо розробки фреймворку автоматизованого тестування, який забезпечує узгоджений контроль якості на всіх зазначених рівнях.

Структуру основних доменних сутностей і зв'язки між ними наведено на рисунку 1.1.

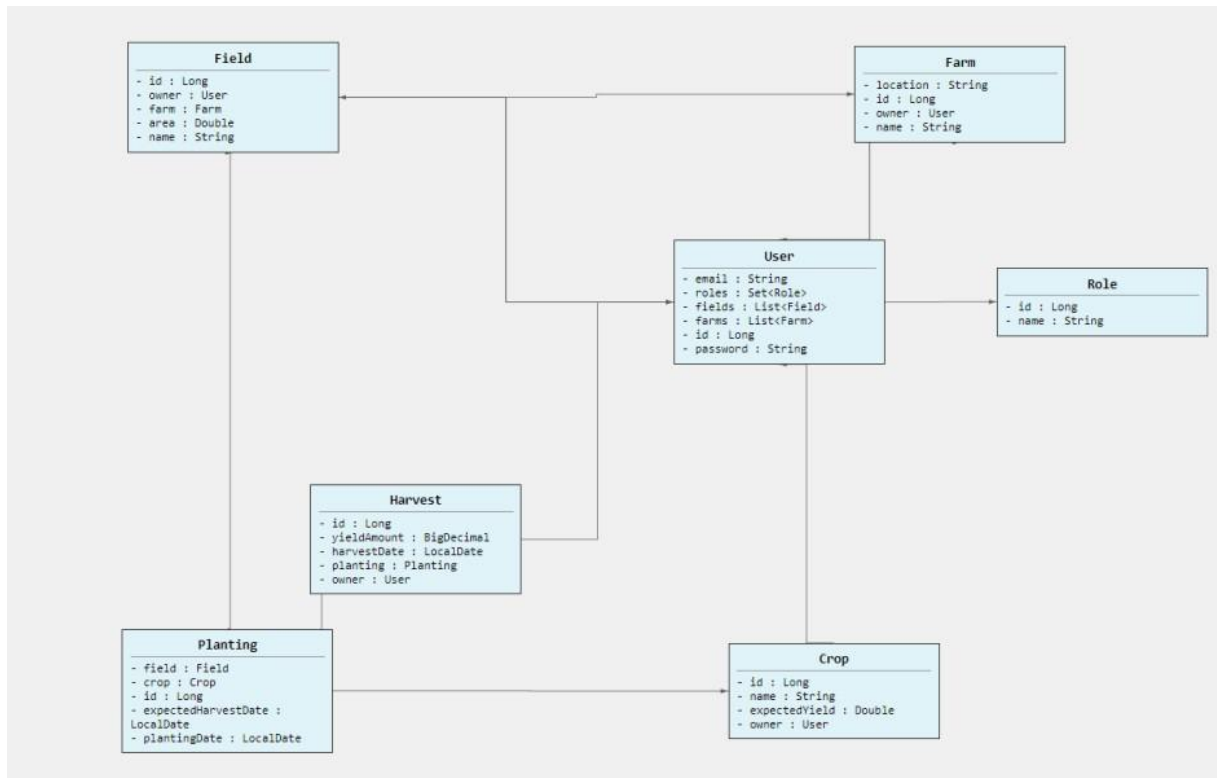


Рис. 1.1 Доменна модель сутностей AgroManagement та їх взаємозв'язки

1.2 Аналіз існуючих аналогів інструментів автоматизації тестування

Ринок інструментів end-to-end автоматизації для вебзастосунків характеризується наявністю кількох зрілих платформ, які дозволяють імітувати дії користувача в браузері та інтегрувати перевірки в процеси безперервної інтеграції. Для задачі побудови фреймворку тестування AgroManagement доцільно розглянути три найпоширеніші аналоги: Selenium WebDriver [5], Cypress [6] та Playwright [7; 8], оскільки саме вони найчастіше використовуються в інженерній практиці для UI-рівня та суміжних інтеграційних сценаріїв.

Мета цього підрозділу — виконати порівняльний аналіз зазначених рішень з урахуванням характеристик предметної галузі, описаних у пункті 1.1: багаторівневий тестовий контур (інтерфейс, API, безпека, наскрізні сценарії), стабільність повторних прогонів у CI та діагностичність

результатів. Окремо враховуються висновки досліджень щодо нестабільності автоматичних тестів і витрат ресурсів CI, оскільки вони безпосередньо впливають на ефективність обраного інструмента [2; 3].

1.2.1 Аналіз Selenium WebDriver

Selenium WebDriver є одним із найвідоміших і найтриваліше використовуваних інструментів автоматизації web-інтерфейсу. Платформа має зрілу екосистему, підтримує кілька мов програмування та сумісна з основними браузером, що забезпечує її широке застосування в корпоративних і навчальних проєктах [5; 10]. Для задачі кваліфікаційної роботи Selenium доцільно розглядати як базовий технологічний аналог, оскільки він представляє класичний підхід до UI-автоматизації через WebDriver-протокол.

З позиції архітектури тестового фреймворку ключовою перевагою Selenium є гнучкість композиції: команда може самостійно обирати тестовий раннер, структуру сторінкових об'єктів, репортинг і правила запуску в CI. Саме ця гнучкість історично зробила Selenium де-факто стандартом для багатьох організацій. Водночас така модель означає, що значна частина інженерних рішень переноситься на рівень проєктної команди: необхідно окремо узгоджувати бібліотеки, стабілізувати очікування елементів, підтримувати єдність стилю тестового коду та окремо розв'язувати питання діагностики падінь.

Для контексту AgroManagement це має практичне значення, оскільки тестовий контур повинен охоплювати не лише позитивні UI-перевірки, а також рольові та ownership-обмеження, коректність навігації між доменними модулями та відтворюваність повторних запусків у CI. У Selenium такі вимоги реалізуються, але здебільшого через додаткові інженерні шари. Наприклад, контроль синхронізації динамічного інтерфейсу часто потребує ретельно побудованих explicit waits, а помилки конфігурації очікувань призводять до нестабільних результатів прогонів [5].

Окремо слід відзначити, що в офіційній документації Selenium питання очікувань винесено в самостійний практичний блок, де підкреслено ризики некоректного поєднання стратегій wait та їх вплив на стабільність автоматичних сценаріїв [5]. Для досліджуваного фреймворку це критично, оскільки метою є не просто виконання тестів, а отримання надійного інструмента контролю якості, якому довіряє команда під час щоденної розробки.

Загалом можна зафіксувати узагальнену оцінку Selenium WebDriver за вісьма практичними критеріями порівняння. Швидкість виконання тестів зазвичай оцінюється як середня, оскільки вона залежить від обраного раннера, мережових затримок WebDriver та обсягу ручних очікувань. Підтримка поширених браузерів (Chrome, Firefox, Edge) забезпечується відповідними драйверами. API-тестування не є нативною частиною одного стеку з WebDriver і тому залишається обмеженим без додаткових бібліотек і окремого узгодження з UI-раннером. Паралельний запуск тестів зазвичай реалізується зовнішньою оркестрацією та додатковими налаштуваннями (складний у сенсі інженерної повноти). Складність початкового налаштування типово висока через необхідність зібрати узгоджений стек бібліотек і правил стабілізації. Стабільність тестів у динамічному UI зазвичай середня за умови дисципліни wait-стратегій. Підтримка інтеграції в CI/CD досягається усталеними pipeline-патернами. Робота з асинхронним оновленням інтерфейсу переважно складна й потребує явних explicit/implicit стратегій очікування [5].

До переваг Selenium WebDriver у межах аналізованої задачі можна віднести:

- технологічну зрілість та велику спільноту, що спрощує пошук практик і прикладів впровадження;
- підтримку багатьох мов програмування та браузерів, що полегшує інтеграцію в heterogeneous-середовища;

- можливість побудови індивідуальної архітектури тестового фреймворку відповідно до стандартів команди;
- сумісність із класичними підходами CI/CD за умови належного налаштування інфраструктури запусків.

Суттєві обмеження Selenium для задач цієї бакалаврської роботи:

- підвищена вартість супроводу через необхідність окремо проєктувати значну частину інфраструктурних компонентів;
- вища чутливість до помилок синхронізації в динамічному UI порівняно з сучасними інструментами з нативним auto-waiting;
- фрагментація діагностичного контуру (звіти, артефакти, відлагодження), яку часто потрібно збирати з кількох рішень;
- складніше забезпечити уніфікований підхід для змішаного контуру UI+API+security в одному тестовому середовищі.

Отже, Selenium WebDriver є релевантним і технічно обґрунтованим аналогом для порівняння, однак для цілей створення відтворюваного багаторівневого фреймворку AgroManagement його застосування може потребувати більшого обсягу інженерної підтримки, ніж сучасні інтегровані платформи. Приклад головної сторінки офіційного ресурсу Selenium та приклад документації з керування очікуваннями наведено на рисунках 1.2 та 1.3 відповідно.

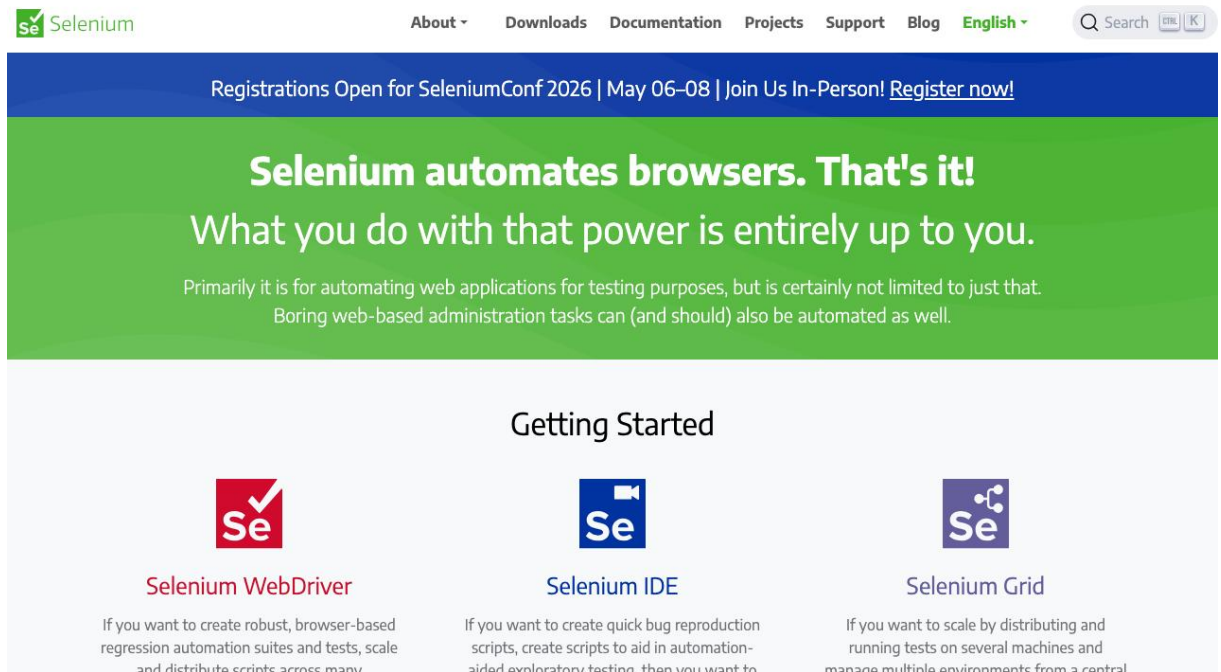


Рис. 1.2 Головна сторінка офіційного сайту Selenium

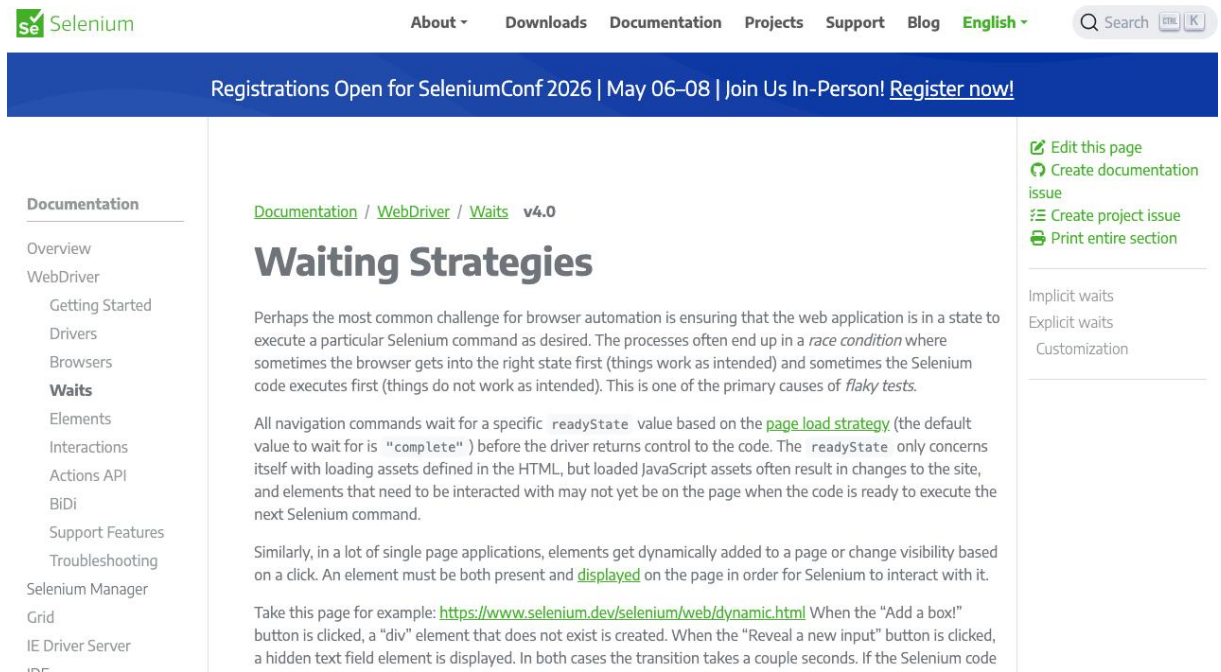


Рис. 1.3 Сторінка документації Selenium WebDriver Waits

1.2.2 Аналіз Cypress

Cypress — це спеціалізований фреймворк для автоматизації тестування вебзастосунків, орієнтований на швидкий цикл розробки та зручний досвід інженера в JavaScript/TypeScript-екосистемі [6]. На офіційному ресурсі інструмент позиціонується як рішення для надійного тестування сучасних

web-додатків із акцентом на простоту запуску та інтерактивність під час налагодження [6]. У контексті кваліфікаційної роботи Cypress доцільно розглядати як конкурентний аналог для UI-рівня, який часто порівнюють із класичними WebDriver-підходами та сучасними інтегрованими раннерами.

Архітектурно Cypress будується навколо моделі, у якій тестовий код тісно пов'язаний із браузерним контекстом і інструментами розробника, що сприяє швидкому зворотному зв'язку під час локальної роботи [6; 11]. Для команд, що переважно фокусуються на регресії інтерфейсу та стабільних користувацьких сценаріях, такий підхід може суттєво скоротити час ітерації «написання тесту — перевірка результату». Водночас для доменно-орієнтованих систем на кшталт AgroManagement важливо оцінювати не лише швидкість локального циклу, а й здатність інструмента узгоджено підтримувати змішаний контур якості: UI, API, перевірки безпеки та наскрізні E2E-ланцюжки з урахуванням ролей і ownership-обмежень.

У практичному плані Cypress добре пристосований до сценаріїв, де основний акцент робиться на взаємодії з одним додатком у межах типової моделі навігації та де тести будуються як послідовність дій користувача в інтерфейсі. Для AgroManagement це відповідає значній частині регресійних перевірок форм, таблиць і переходів між модулями. Проте саме «розширення» цього контуру до складних багатоконтекстних сценаріїв (наприклад, кілька незалежних сесій, специфічні моделі ізоляції, комбіновані UI+API перевірки в одному технічному артефакті) часто стає точкою, де інженерні компроміси проявляються найбільш виразно.

Окремо варто зазначити, що офіційна документація Cypress містить вступні матеріали, які пояснюють призначення інструмента та базові принципи його застосування у сучасному циклі розробки [11]. Для академічного порівняльного аналізу це важливе джерело, оскільки дозволяє узгодити опис можливостей із первинною інформацією виробника, а не лише з вторинними оглядами.

Узагальнено для Cypress характерно наступне. Швидкість виконання тестів у повному регресійному наборі часто оцінюється як середня, натомість локальний цикл зворотного зв'язку для типових UI-сценаріїв може бути швидким [6; 11]. Підтримка браузерів цього інструменту віднесена до Chrome-базованих рішень. API-тестування можливе, проте в конфігурації «єдиного стеку» часто виступає додатковим шаром порівняно з UI-фокусом (часткова підтримка). Паралельний запуск обмежений моделлю виконання та конфігурацією наборів. Складність налаштування для стандартних UI-проектів зазвичай середня. Стабільність тестів для типових UI-потоків зазвичай висока. Підтримка CI/CD забезпечується практикою інтеграції у pipeline. Робота з асинхронністю в інтерфейсі здебільшого автоматизується вбудованими механізмами інструмента [6; 11].

До переваг Cypress у межах аналізованої задачі можна віднести:

- високу продуктивність локального циклу розробки тестів і зручність інтерактивного налагодження для типових UI-сценаріїв;
- чітку орієнтацію на JavaScript/TypeScript, що спрощує інтеграцію в frontend-орієнтовані команди;
- наявність розвиненої документації та екосистеми навколо типових практик E2E-тестування web-інтерфейсу [6; 11];
- придатність для швидкого нарощування набору регресійних перевірок інтерфейсу за умови дисципліни архітектури тестів.

Суттєві обмеження Cypress для задач цієї бакалаврської роботи:

- підвищена ймовірність інженерних ускладнень при реалізації складних багатоконтекстних сценаріїв порівняно з інструментами, де такі моделі є більш природними;
- потенційно вищі витрати на уніфікацію змішаного контуру UI+API+security у межах одного проєкту без додаткових технічних шарів;
- залежність від обраної мовної екосистеми, що може бути критичним для команд із іншим стеком або для уніфікації з існуючим фреймворком на базі іншого раннера;

– необхідність ретельного планування структури тестів, щоб уникнути накопичення технічного боргу при зростанні доменної складності.

Отже, Cypress є сильним аналогом для UI-фокусованої автоматизації та заслуговує на окреме місце в порівняльному аналізі, проте для AgroManagement його доцільно оцінювати саме через призму повноти багаторівневого тестового контуру та передбачуваності супроводу при масштабуванні. Приклад головної сторінки офіційного ресурсу Cypress та приклад сторінки документації наведено на рисунках 1.4 та 1.5 відповідно.

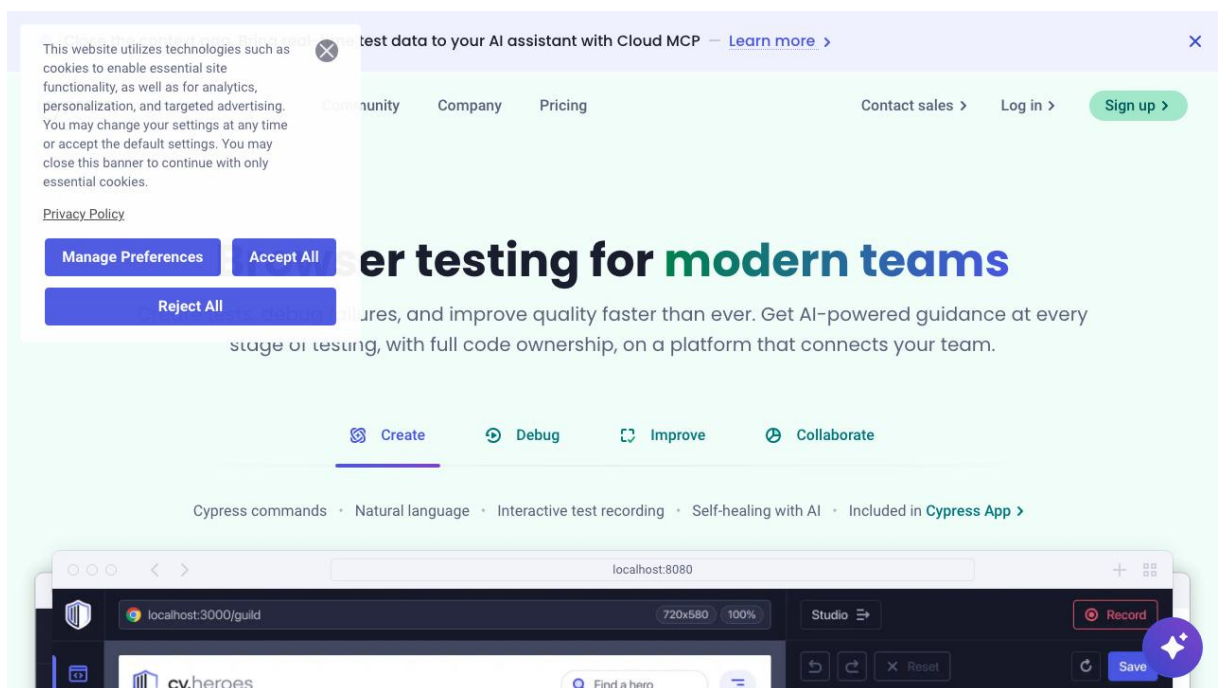


Рис. 1.4 Головна сторінка офіційного сайту Cypress

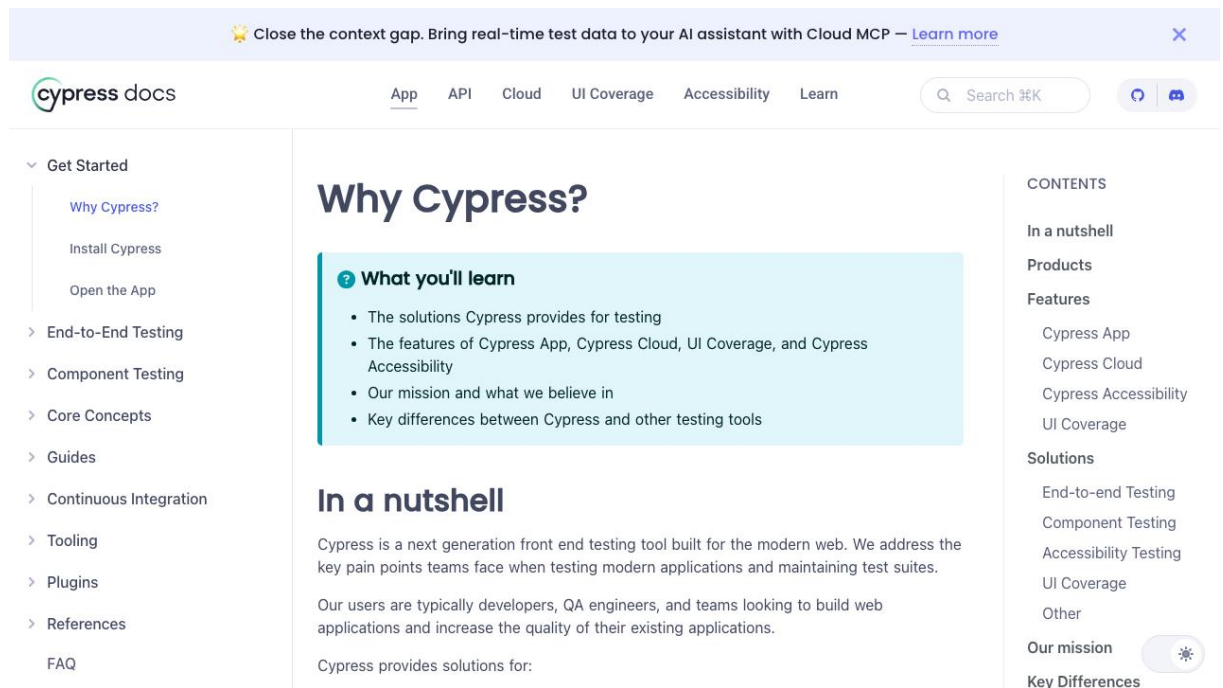


Рис. 1.5 Фрагмент офіційної документації Cypress (розділ Why Cypress)

1.2.3 Аналіз Playwright

Playwright — сучасний фреймворк автоматизації тестування для web-застосунків, розроблений і підтримуваний Microsoft, який поєднує керування браузером, стабільні очікування дій користувача та інтегрований тестовий раннер `@playwright/test` [7; 8]. На офіційному ресурсі інструмент позиціонується як рішення для швидкого та надійного end-to-end тестування, що безпосередньо відповідає інженерним пріоритетам кваліфікаційної роботи: відтворюваність прогонів, масштабованість наборів тестів і зручність інтеграції в CI/CD [7].

З архітектурної точки зору Playwright орієнтований на «actionability»: перед виконанням дії фреймворк перевіряє, чи елемент придатний для взаємодії, що зменшує частку хибних падінь, типових для динамічних інтерфейсів без коректної синхронізації [8]. Для AgroManagement, де сценарії охоплюють форми, табличні подання та переходи між модулями, така модель знижує ризик нестабільності, яка в інших стеках часто компенсується ручними очікуваннями або додатковими обгортками.

Важливою перевагою для задачі багаторівневого контуру якості є можливість у межах одного проєкту поєднувати UI-перевірки з API-

взаємодією та використовувати єдину конфігурацію запуску, зокрема налаштування паралельного виконання через `workers` і проекти тестів [9]. Це узгоджується з характеристиками предметної галузі, описаними у пункті 1.1, де наголошується на необхідності наскрізних сценаріїв, рольових обмежень і відтворюваних регресій у середовищі безперервної інтеграції.

Практична реалізація фреймворку в цій бакалаврській роботі виконана саме на базі Playwright: тестові сценарії організовані як `srec`-файли, для інкапсуляції UI-логіки використовуються сторінкові класи (наприклад, у каталозі ``pages/``), а для підготовки контексту автентифікації та доменних даних — `fixtures` у каталозі ``fixtures/``. API-рівень представлено контролерами HTTP-запитів у ``api/controllers/``, що дозволяє узгоджено підтримувати змішані перевірки без розриву між різними раннерами.

Окремо слід зазначити, що офіційна документація містить окремі розділи щодо написання тестів і конфігурації прогонів, які використовуються як первинні джерела при описі інженерних можливостей інструмента [8; 9]. Це підвищує прозорість академічного викладу: твердження про можливості Playwright можна безпосередньо звіряти з матеріалами виробника.

Отже для Playwright характерно наступне. Швидкість виконання тестів зазвичай висока завдяки оптимізованому раннеру та можливостям паралелізації [7; 9]. Підтримка браузерів охоплює Chrome, Firefox, Edge та WebKit [7]. Підтримка API-тестування є повноцінною в межах ``@playwright/test`` через мережевий контекст HTTP-запитів [8]. Паралельний запуск тестів вбудований (`workers`, `projects`, конфігурація прогонів) [9]. Складність базового налаштування стандартного проекту зазвичай низька порівняно зі «збиранням» класичного Selenium-стеку [7; 8]. Стабільність тестів для динамічного UI зазвичай висока завдяки `actionability` та `auto-waiting` [8]. Підтримка інтеграції в CI/CD забезпечується типовими сценаріями `pipeline` та штатними артефактами діагностики [8]. Робота з асинхронністю інтерфейсу здебільшого автоматизується на рівні моделі дій без ручних пауз [8].

До переваг Playwright у межах аналізованої задачі можна віднести:

- інтегровану модель E2E-тестування з орієнтацією на стабільність взаємодії з динамічним UI;
- зручне масштабування виконання тестів і керування профілями запуску через конфігурацію проєкту [9];
- можливість уніфікації UI- та API-перевірок у межах одного тестового стеку, що знижує вартість супроводу порівняно з фрагментованими рішеннями;
- штатні механізми діагностики (trace, screenshot, video), які підвищують аналізованість інцидентів у pipeline [8].

Суттєві обмеження та ризики, які необхідно враховувати:

- необхідність дисципліни версійності залежностей і узгодженості середовища CI з обраними браузерерами;
- зростання складності супроводу при відсутності модульних правил організації тестового коду (сторінкові об'єкти, повторно використовувані фікстури, контролери API);
- потреба системного підходу до стабільності тестів загалом, оскільки жоден інструмент не усуває логічні помилки у тест-кейсах або некоректні тестові дані.

З огляду на зазначені властивості — Playwright найкраще відповідає цілі кваліфікаційної роботи, а саме створити відтворюваний багаторівневий фреймворк автоматизованого тестування для AgroManagement. Саме тому в подальших розділах як базова інструментальна платформа обрано Playwright [7; 8; 9]. Приклад головної сторінки офіційного ресурсу Playwright та приклад розділу документації щодо написання тестів наведено на рисунках 1.6 та 1.7 відповідно.

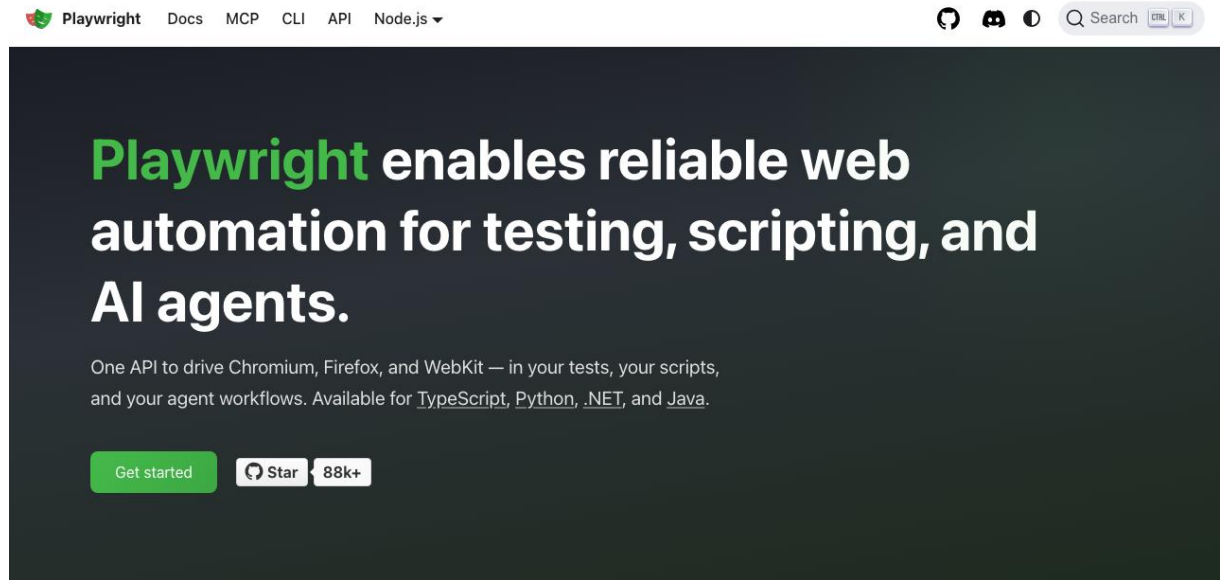


Рис. 1.6 Головна сторінка офіційного сайту Playwright

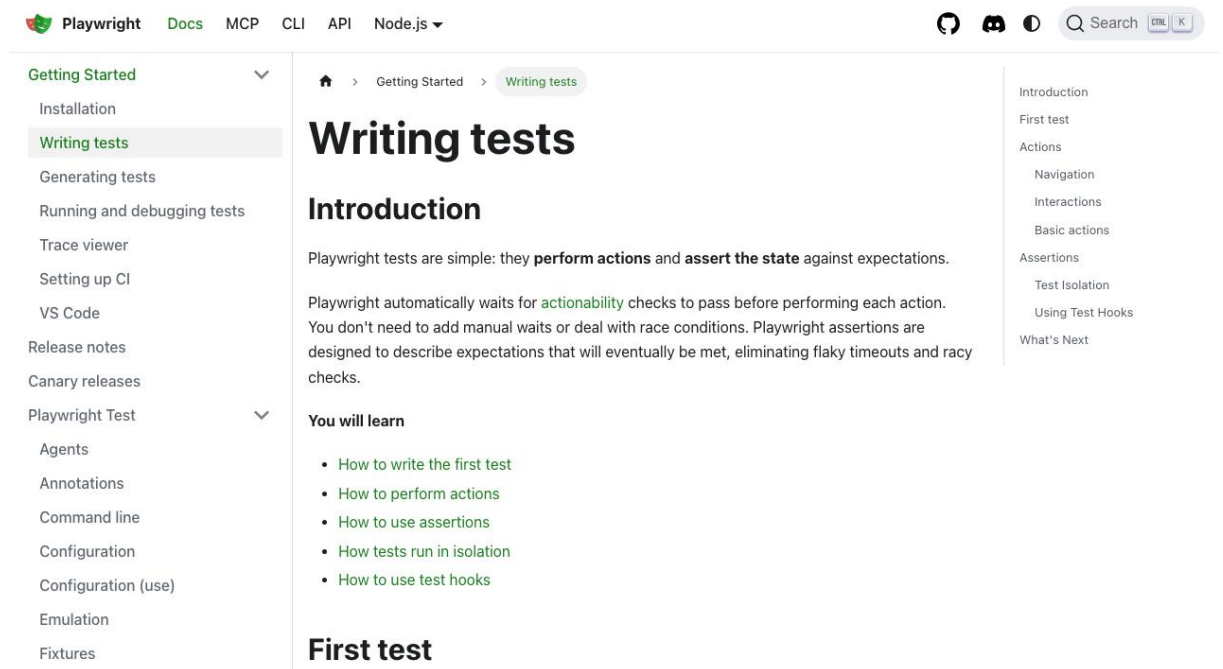


Рис. 1.7 Фрагмент офіційної документації Playwright (написання тестів)

Зведене порівняння інструментів автоматизації тестування за вісьма критеріями: швидкість виконання тестів; підтримка браузерів; підтримка API-тестування; паралельний запуск тестів; складність налаштування; стабільність тестів; підтримка CI/CD; робота з асинхронністю інтерфейсу —

наведено у таблиці 1.1. Деталізований аналіз особливостей кожного інструмента з урахуванням вимог багаторівневого контуру AgroManagement (інтерфейс, API, безпека, наскрізні сценарії), стабільності прогонів у CI та діагностики результатів наведено у підрозділах 1.2.1–1.2.3, включно з додатковими узгоджувальними узагальненнями перед переліками переваг і обмежень.

Таблиця 1.1

Порівняльна характеристика Selenium WebDriver, Cypress та Playwright

Критерій	Selenium	Cypress	Playwright
Швидкість виконання тестів	середня	середня	висока
Підтримка браузерів	Chrome, Firefox, Edge	Chrome-базовані браузери	Chrome, Firefox, Edge, Safari
Підтримка API тестування	обмежена	часткова	повна
Паралельний запуск тестів	складний	обмежений	вбудований
Складність налаштування	висока	середня	низька
Стабільність тестів	середня	висока	висока
Підтримка CI/CD	підтримується	підтримується	підтримується
Робота з асинхронністю	складна	автоматична	автоматична

1.3 Визначення проблем предметної галузі

Цей підрозділ узагальнює типові проблеми забезпечення якості саме як інженерно-організаційний виклик для вебзастосунку AgroManagement. Доменні особливості та ланцюжки узгодженості даних уже розкрито в п. 1.1;

інструментальні альтернативи та їх порівняння — у п. 1.2 (та табл. 1.1). Нижче сформульовано прогалини та ризики, які безпосередньо обґрунтовують розробку цілісного тестового фреймворку.

Перша група проблем пов'язана з розривом між «точковими» перевітками окремих функцій і необхідністю підтвердження коректності наскрізних операцій, коли зміна стану на одному рівні предметної моделі має узгоджено відображатися на пов'язаних сутностях (див. п. 1.1). Якщо регресія не охоплює такі ланцюжки, зростає ризик прихованих дефектів обліку та неконсистентності даних, які проявляються лише в інтеграційних сценаріях або після накопичення змін.

Друга група проблем стосується недостатньої репрезентативності security-контур у регулярних автоматичних прогонах: для застосунків із розмежуванням ролей критично системно перевіряти не лише «щасливі» шляхи, а й негативні кейси доступу та ownership-обмежень [4]. Відсутність таких перевірок у стандартному регресійному наборі створює ілюзію контролю якості за рахунок UI-позитиву.

Третя група проблем впливає з вимоги відтворюваних прогонів у СІ: нестабільні тести підривають довіру до pipeline, сповільнюють інтеграцію змін і збільшують операційні витрати на повторні запуски та локалізацію хибних спрацьовувань [2; 3]. Для інтерфейсів із динамічним оновленням стану це особливо чутливо без єдиних правил синхронізації, діагностики та звітності.

Додатково до зазначеного доцільно виділити такі типові дефіцити тестового контуру:

- відсутність єдиної архітектури автоматизації для узгодженого поєднання UI-, API- та security-перевірок;
- фрагментація інструментів і правил запуску, що ускладнює супровід і масштабування набору тестів;
- обмежена відтворюваність результатів у віддаленому середовищі СІ через недостатність інженерних артефактів аналізу падінь;

– накопичення технічного боргу в тестах при розширенні функціональності без модульних угод про структуру коду тестів.

Системне впровадження багаторівневого автоматизованого тестування узгоджується з орієнтирами оцінювання якості програмних систем [1] і дозволяє зменшити втрати, пов'язані з нестабільністю та низькою прогнозованістю результатів перевірок [2; 3]. При цьому акцент робиться не на формальному збільшенні кількості тестів, а на підвищенні довіри до сигналу якості під час постачання змін.

Отже, основні напрями подолання вказаних проблем у межах кваліфікаційної роботи такі:

- сформувані узгоджений набір рівнів перевірок (UI, API, security, наскрізні E2E) у межах одного фреймворку;
- забезпечити відтворювані прогони в CI з прозорою діагностикою інцидентів;
- інституціоналізувати негативні сценарії доступу як обов'язкову частину регресії [4];
- закріпити інженерні практики організації коду тестів (сторінкові об'єкти, фікстури, контролери API) для стримування вартості супроводу.

Вказані прогалини обґрунтовують практичну доцільність розробки фреймворку автоматизованого тестування для AgroManagement; вибір базової платформи Playwright детально обґрунтовано в п. 1.2.3 і узгоджений з результатами порівняння у табл. 1.1.

1.4 Постановка завдання на розробку тестового фреймворку

На основі узагальнення характеристик предметної галузі (п. 1.1), результатів порівняння інструментів автоматизації (п. 1.2, табл. 1.1), а також виявлених прогалин тестового контуру (п. 1.3) та з опорою на джерела щодо якості програмного забезпечення, нестабільності тестів і вимог безпеки web-застосунків [1–4] формулюється інженерне завдання кваліфікаційної роботи.

Повторний перелік проблем тут навмисно не дублюється: він концентровано представлений у п. 1.3.

У межах кваліфікаційної роботи ставиться завдання розробити та апробувати фреймворк автоматизованого тестування для веб-застосунку AgroManagement, який забезпечує відтворювані багаторівневі перевірки: інтерфейс користувача, програмні інтерфейси доступу до даних (API), сценарії безпеки та наскрізні E2E-ланцюжки доменної логіки. Первинним об'єктом розробки є програмний комплекс автоматизації, а не розширення функціональності самого AgroManagement поза межами необхідного для демонстрації роботи тестів.

Для реалізації поставленого завдання передбачається:

- використання мови TypeScript та тестового раннера `@playwright/test` як базової інструментальної платформи [7; 8];
- організація UI-автоматизації за підходом Page Object у каталозі ``pages/`` та підготовка тестового контексту через Playwright fixtures у каталозі ``fixtures/``;
- реалізація API-перевірок через шар HTTP-контролерів у ``api/controllers/`` та відповідні набори спец-файлів у ``tests/api/``;
- формування набору security-перевірок (зокрема ownership та сценарії прямого доступу за посиланням) у структурі ``tests/security/`` з можливістю таргетованого запуску відповідно до прийнятих у проєкті правил маркування тестів;
- розробка наскрізних E2E-сценаріїв доменних ланцюжків у ``tests/e2e/business/`` для підтвердження узгодженості критичних операцій;
- підтримка відтворюваних тестових даних за допомогою фабрик і допоміжних утиліт у ``data/factories/`` та узгоджених механізмів очищення/підготовки даних, передбачених проєктом;
- конфігурація профілів виконання Playwright (зокрема розділення контурів UI та API) у ``playwright.config.ts`` та узгодження команд запуску з ``package.json``;

– інтеграція прогонів автоматичних тестів у контур безперервної інтеграції на базі GitHub Actions workflow репозиторію з підтримкою вибору наборів тестів (наприклад, smoke, ui, api, security, regression, full).

Поставлене завдання передбачає комплексний підхід до побудови інженерного рішення, де узгоджені рівні перевірок підсилюють один одного: API- та UI-сценарії прискорюють локалізацію дефектів, security-перевірки знижують ризики помилок авторизації та доступу [4], а E2E-ланцюжки підтверджують коректність бізнес-операцій у цілому. Окрема увага приділяється стабільності прогонів і прозорості діагностики результатів, оскільки саме ці властивості визначають практичну придатність фреймворку для регулярної експлуатації командою розробки [2; 3].

Очікуваним результатом є відтворюваний фреймворк автоматизованого тестування для AgroManagement, який може використовуватися для регресійного контролю якості під час розвитку застосунку, а також як база для подальшого розширення покриття без принципової зміни архітектури рішення.

1.5 Визначення вимог до тестового фреймворку

На основі аналізу предметної галузі (п. 1.1), порівняння інструментів (п. 1.2, табл. 1.1), узагальнення прогалин тестового контуру (п. 1.3) та постановки завдання (п. 1.4) сформульовано вимоги до функціональності та якості розроблюваного фреймворку автоматизованого тестування для вебзастосунку AgroManagement. Під «застосунком» у контексті предметної області розуміється система AgroManagement, тоді як об'єктом вимог нижче є саме програмний комплекс автоматизації (фреймворк автоматизованого тестування), що забезпечує її верифікацію. Вимоги поділено на функціональні та нефункціональні.

Функціональні вимоги:

1. Автоматизована перевірка сценаріїв автентифікації користувачів із використанням валідних та невалідних облікових даних (позитивні та негативні кейси).

2. Автоматизована перевірка CRUD-операцій для доменних сутностей Farm, Field, Crop, Planting та Harvest узгоджено з предметною моделлю, описаною в п. 1.1.

3. Виконання UI-тестів для перевірки коректності роботи інтерфейсу користувача (навігація, форми, відображення даних) у межах обраного інструментального стеку Playwright.

4. Виконання API-тестів для перевірки бізнес-логіки та контрактів доступу до даних на рівні HTTP-інтерфейсу застосунку.

5. Формування звітів про результати виконання тестів із відображенням статусу кожного тесту (у тому числі з використанням штатних можливостей Playwright для звітності).

6. Логування та збереження артефактів діагностики результатів виконання тестів із фіксацією причин падінь (наприклад, trace, screenshot, журнали прогону) для аналізу інцидентів у CI/CD.

Нефункціональні вимоги:

1. Час виконання повного набору автоматизованих тестів у базовій конфігурації середовища виконання не повинен перевищувати 5 хвилин (цільовий орієнтир для оцінки придатності набору до регулярних прогонів).

2. Модульна архітектура фреймворку (розділення рівнів UI, API, security, E2E, підготовки даних) для спрощення підтримки та розширення без неконтрольованого зростання зв'язності компонентів.

3. Стабільність виконання тестів: за однакових умов конфігурації та тестових даних очікується отримання узгоджених результатів принаймні на трьох послідовних повторних запусках повного регресійного набору (за відсутності змін коду та нестабільності середовища).

4. Підтримка інтеграції з системою безперервної інтеграції та доставки змін на базі GitHub Actions з можливістю автоматизованого запуску наборів тестів і публікації звітності.

Визначені вимоги ляжуть в основу подальшого етапу проєктування архітектури фреймворку, структури каталогів, наборів тестів і правил їх виконання в CI/CD. Вони спрямовані на забезпечення прогнозованості регресійного контролю якості, прозорості результатів прогонів і придатності рішення до супроводу в умовах розвитку застосунку AgroManagement.

2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ФРЕЙМВОРКУ

2.1 Загальна архітектура та структура програмного комплексу автоматизації

Проектування архітектури фреймворку базується на вимогах, сформульованих у п. 1.5, та на обраній у п. 1.2.3 інструментальній платформі Playwright [7; 8]. На етапі проектування програмний комплекс планується як багаторівнева система перевірок, де кожен рівень відповідає окремому класу ризиків якості вебзастосунку AgroManagement: коректність інтерфейсу, коректність бізнес-логіки через HTTP-інтерфейси, коректність політик доступу та наскрізну узгодженість доменних операцій.

У логічній моделі передбачається виділити такі шари: (1) шар специфікацій тестів, згрупованих за типом перевірки; (2) шар абстракцій взаємодії з системою (сторінкові об'єкти та API-контролери); (3) шар підготовки тестового контексту та даних; (4) шар допоміжних утиліт і конфігурації середовища. Таке розділення проєктується для зниження зв'язності компонентів: зміни в представленні UI мають локалізуватися в межах сторінкових класів, зміни в контракті API — у межах контролерів і типів, а сценарії тестів залишатимуться відносно стабільними на рівні бізнес-кроків.

Графічне узагальнення запланованої багатошарової структури, смуг контексту (об'єкт тестування та CI/CD) та інструментальної основи наведено на рисунку 2.1; принципи організації автоматизованих workflow у GitHub Actions узгоджено з офіційною документацією платформи [12]. Схема відображає проєктну відповідність логічних шарів майбутнім каталогам фреймворку без деталізації окремих тест-кейсів, що належать до етапу реалізації.

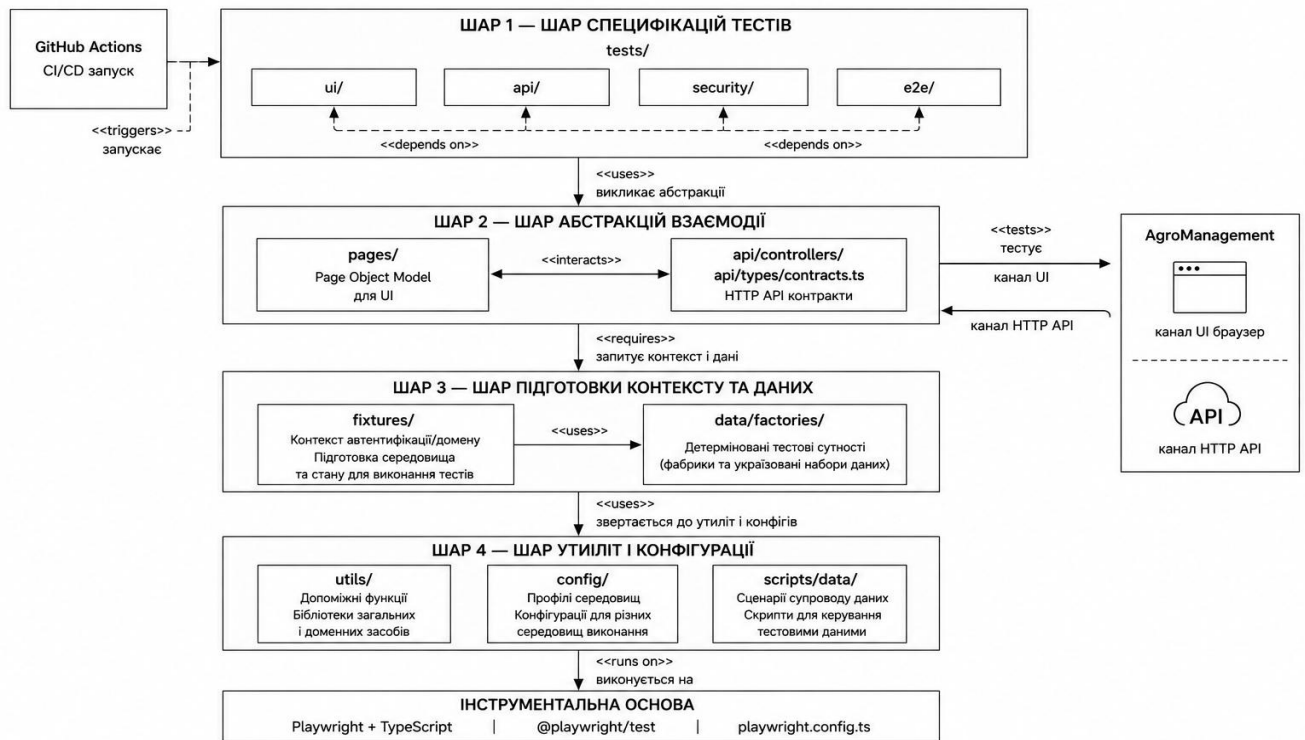


Рис. 2.1 Логічна багатошарова структура фреймворку автоматизованого тестування AgroManagement та інструментальна основа

Фізичну структуру репозиторію передбачається спроектувати як відображення зазначеного розділення відповідальності. Каталог ``tests/`` має містити специфікації, згруповані за рівнем перевірки (``ui``, ``api``, ``security``, ``e2e``), що відповідає практиці розділення регресійних контурів за ризиком і вартістю виконання [8]. Каталог ``pages/`` призначається для Page Object Model і інкапсуляції локаторів та дій користувача [13]. Каталог ``api/controllers/`` формує шар доступу до HTTP-інтерфейсу, а модуль ``api/types/contracts.ts`` — узгодженість структур даних між API-перевірками та генераторами тестових сутностей.

Підготовку тестового середовища заплановано реалізувати через `fixtures Playwright` у каталозі ``fixtures/`` (базові, автентифікаційні та доменні профілі) [14] і через детерміновані будівники сутностей у ``data/factories/``. Допоміжні функції концентруються в ``utils/`` (автентифікація, робота з API-відповідями, реєстри тестових даних), параметри середовища — у ``config/``.

Окремо передбачаються виконувати сценарії супроводу даних у ``scripts/data/``, зокрема для очищення тестових артефактів після прогонів.

Конфігурація виконання тестів у файлі ``playwright.config.ts`` проєктує архітектурне розділення UI та API на рівні проєктів Playwright [9]: для UI-контур передбачається профіль браузера Desktop Chrome, для API-контур — окремий проєкт із базовим URL API. Паралельне виконання, політика повторів у CI та збір діагностичних артефактів (trace, знімки екрана, відео) закладаються як інженерні налаштування стабільності прогонів [8; 9] і підтримують вимоги п. 1.5 щодо відтворюваності та прозорості результатів.

2.2 Інструментальний стек і конфігурація середовища виконання тестів

У цьому підрозділі буде формалізовано проєктні рішення щодо інструментального стеку та правил конфігурації, на яких спирається архітектура п. 2.1. Акцент — на відтворюваності середовища, типізації коду та узгодженості параметрів запуску з вимогами п. 1.5; програмна реалізація та приклади сценаріїв наводяться у розділі 3.

Мовою реалізації фреймворку на етапі проєктування обрано TypeScript як типізований надмножинний JavaScript, що підвищує надійність супроводу великого набору тестів і узгоджується з екосистемою Playwright [15]. У ``tsconfig.json`` передбачається цільова версія ECMAScript (ES2022), режим ``strict``, підключення типів Node та ``@playwright/test``, а також шляхові псевдоніми (``@api/*``, ``@pages/*``, ``@fixtures/*``, ``@data/*``, ``@config/*``, ``@utils/*``) для підтримки модульної структури каталогів п. 2.1 [15].

Тестовий раннер і автоматизацію браузера заплановано базувати на пакеті ``@playwright/test`` (цільова лінійка версій 1.54.x у залежностях проєкту згідно з ``package.json``) [7; 8]. Це проєктне рішення забезпечує єдиний інструмент для UI- та API-прогонів, інтегровані асerti та звітність у зв'язці з ``playwright.config.ts`` [9]. Для локального запуску передбачається

встановлення браузерного рушія Chromium відповідно до інструкцій Playwright; вимоги до середовища мають бути зафіксовані в документації репозиторію (`README.md`) [8].

Керування залежностями та сценаріями запуску проєктується через `npm`-скрипти, визначені в `package.json`: повний прогін, окремі профілі `test:ui` та `test:api`, таргетовані набори `test:smoke` і `test:security`, а також допоміжні команди `typecheck`, `report:open` і сценарій `test:clean` з пост-очищенням даних [8]. Наявність окремих команд відображає проєктне рішення щодо декомпозиції регресійного навантаження без зміни вихідного коду тестів при перемиканні режиму прогону.

Параметри середовища виконання передбачається централізувати в модулі `config/env.ts`: підвантаження значень з `.env` здійснюється бібліотекою `dotenv` [16], вибір профілю `dev` або `stage` визначається змінною `TEST_ENV`, а базові URL для інтерфейсу та API беруться з `config/env/dev.env.ts` та `config/env/stage.env.ts` з можливістю перевизначення через `BASE_URL` та `API_BASE_URL`. Префікс тестових даних `TEST_DATA_PREFIX` (за замовчуванням `qaauto`) та облікові дані користувача й адміністратора зчитуються з відповідних змінних середовища; функція `assertRequiredEnv()` проєктується для явної перевірки повноти обов'язкових секретів перед прогоном, що критично для CI/CD [8].

Для онбордингу розробників та тестувальників передбачається шаблон `.env.example` з переліком змінних і прикладами для локального профілю (`TEST_ENV`, `BASE_URL`, `API_BASE_URL`, `TEST_DATA_PREFIX`, плейсхолдери облікових записів). Такий підхід знижує ризик «немовчущих» помилок конфігурації і узгоджується з практикою винесення налаштувань у змінні середовища [16].

2.3 Проєктування UI-рівня на основі Page Object Model

Спроектовано наступний підхід до моделювання взаємодії з вебінтерфейсом AgroManagement у тестах: інкапсуляція локаторів і дій

користувача в окремих класах сторінок, а в специфікаціях — опис сценаріїв перевірки без дублювання низькорівневих селекторів. Таке розділення відповідає рекомендаціям Playwright щодо Page Object Model [13] і підтримує вимоги до супроводжуваності, сформульовані у п. 1.5.

Каталог ``pages/`` проектується зі структуруванням за предметними областями застосунку: ``auth`` (автентифікація та реєстрація), ``home`` (головна після входу), ``farm``, ``field``, ``crop``, ``planting``, ``harvest`` (основні бізнес-сутності), ``admin`` (адміністративні екрани). Такий поділ відображає навігаційну та семантичну логіку UI й полегшує пошук відповідних абстракцій при розширенні покриття.

Структурно-логічну організацію шару Page Object, звернення зі специфікацій ``tests/ui/**`` до методів сторінок, поділ за доменними підкаталогами та опору на ``Page`/`Locator`` із ``baseURL`` у ``playwright.config.ts`` наведено на рисунку 2.2.

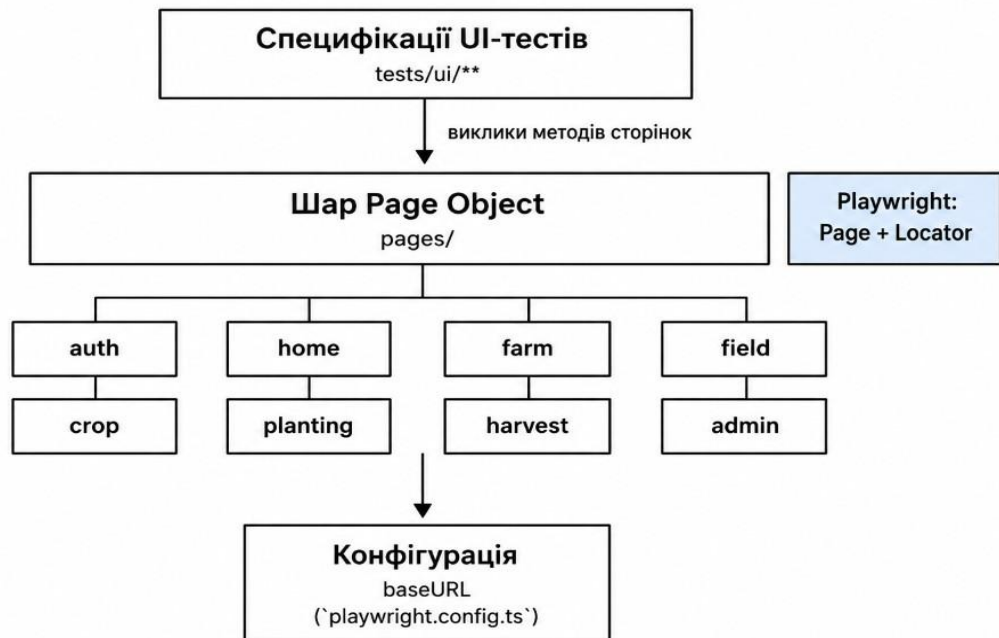


Рис. 2.2 Структура шару Page Object і зв'язок UI-специфікацій із каталогом ``pages/`` та конфігурацією ``baseURL``

Для кожної сторінки проєктується клас із приватними полями типу ``Locator``, ініціалізованими в конструкторі на основі екземпляра ``Page``. Публічні методи групуються за змістом: відкриття маршруту (``open`` з відносним шляхом, узгодженим із ``baseURL`` у ``playwright.config.ts``), введення даних у форми, натискання кнопок, перехід за посиланнями, а також перевірки видимості та стану елементів через ``expect``. Прикладами проєктних абстракцій є ``LoginPage`` (поля входу, кнопка входу, індикатор помилки) та ``HomePage`` (елементи панелі після успішної автентифікації).

Стратегія локаторів проєктується з поєднанням доступності й стійкості: пріоритет надається семантичним прив'язкам (``getByRole`` з ім'ям кнопки або посилання), за потреби використовуються стабільні ідентифікатори (``#username``, ``#password``) та класи для повідомлень про помилки. Це має зменшити залежність UI-тестів від косметичних змін розмітки й підвищити стійкість сценаріїв до регресій інтерфейсу.

З точки зору супроводу тестового коду сценарії у файлах ``tests/ui/**`` проєктуються лінійними: «відкрити сторінку → виконати дію → перевірити результат», тоді як деталі DOM приховуються в Page Object. Для сценаріїв із спільними передумовами (наприклад, автентифікований контекст) передбачається розширення базового ``test`` через фікстури; їхня роль у композиції кроків деталізується в п. 2.4 [14].

Загалом цей підрозділ визначає гранулювання Page Object за доменами AgroManagement, уніфікований інтерфейс методів сторінок і правила вибору локаторів, що разом забезпечують передбачувану структуру UI-автотестів і підготовку до реалізації.

Проєктні рішення п. 2.3 узгоджуються з предметною моделлю AgroManagement (п. 1.1): кожному критичному екрану відповідає окрема абстракція сторінки, а UI-специфікації мають оперувати бізнес-діями, а не селекторами. На рисунку 2.2 подано цільову схему взаємодії шару специфікацій ``tests/ui/**``, каталогу ``pages/`` та конфігурації ``baseURL``; програмна реалізація класів сторінок, приклади методів

``open`/`login`/`expect...`` і відносних переходів на кшталт ``page.goto("/login")`` наводяться у розділі 3.

2.4 Проєктування підготовки тестових даних і композиції фікстур

У цьому підрозділі буде визначено проєктні механізми формування вхідних даних, передачі залежностей у тестові сценарії та керованого прибирання створених артефактів після виконання. Це забезпечує відтворюваність перевірок та мінімізує вплив попередніх запусків на поточний результат.

Генерацію доменних наборів полів для API-запитів заплановано через фабрики в каталозі ``data/factories/``. Для кожної сутності передбачається клас `*Factory*` з методом ``valid()``, що повертає об'єкт контракту з унікальними значеннями, сформованими за допомогою ``randomSuffix()``. Базовий префікс тестових даних задається середовищними змінними (``TEST_DATA_PREFIX``) в модулі конфігурації.

Для керування зв'язками між сутностями та контрольованого teardown проєктується реєстр ``TestDataRegistry`` (``utils/data/TestDataRegistry.ts``). Після створення об'єктів у тесті ідентифікатори реєструються в залежності від бізнес-ланцюга, а у фікстурі забезпечується видалення сутностей у зворотному порядку їх створення.

Композицію тестового рантайму заплановано реалізувати ланцюгом розширення ``test`` у Playwright: ``fixtures/base.fixture.ts`` має надавати об'єкт ``ENV``, ``fixtures/auth.fixture.ts`` — формувати автентифіковані ``APIRequestContext`` з переносом form-session та додаванням CSRF-заголовка ``X-CSRF-TOKEN``, а ``fixtures/domain.fixture.ts`` — додавати екземпляри API-контролерів і фікстуру ``dataRegistry`` з фазою очищення [14].

Повторювані сценарії підготовки доменного стану передбачається винести у функції ``setupFarmAndField``, ``setupFieldAndCrop``, ``setupPlanting``, ``setupHarvest`` у межах ``fixtures/domain.fixture.ts``. Це має знизити дублювання

кроків у специфікаціях та спростити підтримку при зміні контрактів або бізнес-логіки.

Окремо передбачається адміністративне очищення залишків через скрипт ``scripts/data/cleanupTestData.ts``, який має виконувати масове видалення сутностей за префіксом тестових даних. У ``package.json`` ця функція інтегрується в команди ``cleanup:data`` і ``test:clean``, що дозволяє поєднати виконання тестів із прибиранням залишків у контрольований спосіб.

2.5 Інтеграція CI/CD та формування тестових звітів

Цей підрозділ описує проєктну архітектуру інтеграції фреймворку автотестування з CI/CD середовищем на базі GitHub Actions, що має забезпечувати повторювані перевірки на подіях ``pull_request``, ``push`` до гілки ``main``, нічні запуски за ``schedule``, а також ручний запуск через ``workflow_dispatch`` із вибором тестового набору. Такий підхід відповідає загальним принципам організації CI в GitHub Actions [12].

Керування вибором suite проєктується змінною ``SELECTED_SUITE``: для ``pull_request`` і ``push`` передбачається режим ``smoke``, а для нічного ``schedule`` — ``full``; для ``workflow_dispatch`` застосовується параметр ``suite`` (опції: ``smoke``, ``ui``, ``api``, ``regression``, ``security``, ``smoke+security``, ``full``). Додатково проєктується змінна ``STRICT_MODE``, яка активує strict-режим для PR/push і ручного smoke, тоді як нічний повний запуск може трактуватися як informational (відомі продуктні відмови не блокують процес розробки).

Перед виконанням тестів у pipeline передбачаються крок ``npm run typecheck`` (TypeScript), а також встановлення браузерів Playwright через ``npx playwright install --with-deps chromium``, що має забезпечити відповідність середовища очікуванням тестової інфраструктури.

Середовище тестування в CI проєктується контейнерним стеком у каталозі ``agro-management``: передбачається ``docker compose up -d --build``

mysql keycloak`, очікування готовності MySQL та Keycloak, далі старт контейнера застосунку (`docker compose up -d app`) і циклічна перевірка доступності endpoint `/login` за HTTP-статусом 200/302. Після готовності інфраструктури у БД мають бути задані ролі (`ROLE_USER`, `ROLE_ADMIN`) та зареєстровані smoke-користувачі (`qa_user`, `qa_admin`) із використанням секретів `USER_PASSWORD` та `ADMIN_PASSWORD`, що узгоджує тестові передумови з механізмами автентифікації у фреймворку.

Виконання тестового набору проєктується в кроці «Run selected suite» як `case`-розгалуження залежно від значення `SELECTED_SUITE`. Для відбірних режимів передбачаються `npm run test:smoke`, `npm run test:ui`, `npm run test:api`, а для `regression` — фільтр за тегом через `npx playwright test --grep @regression`, для `security` — `npm run test:security`, для `smoke+security` — послідовний запуск `smoke` та `security`. У режимі `full` — `npm test`.

Для коректного збирання інформації при часткових відмовах workflow проєктується з `set +e` і фіксацією `exit_code`. Якщо `STRICT_MODE` увімкнено і `exit_code` не дорівнює нулю, workflow має завершуватися з помилкою; в іншому випадку падіння трактується як non-blocking і публікується інформаційний підсумок у `$GITHUB_STEP_SUMMARY`. Завжди (з `if: always()`) передбачається прибирання тестових даних через `npm run cleanup:data`.

Систему звітності та діагностики проєктують на артефактах Playwright: `actions/upload-artifact@v4` має завантажувати каталоги `test-automation-framework/test-results` і `test-automation-framework/playwright-report` під ім'ям ``${env.SELECTED_SUITE}`-artifacts`. Для сценаріїв, де допускається публікація, додатково передбачається `actions/upload-pages-artifact@v3` з шляхом `test-automation-framework/playwright-report`, а окремий job `deploy-report` — публікацію звіту в GitHub Pages через `actions/deploy-pages@v4` (з URL, що формується через `steps.deployment.outputs.page_url`).

3 РЕАЛІЗАЦІЯ ТА РОЗРОБКА ТЕСТІВ

3.1 Загальна характеристика реалізації та організація тестового коду

Цей розділ фіксує програмну реалізацію фреймворку автоматизованого тестування у репозиторії ``test-automation-framework``: від структури специфікацій і проєктів Playwright до ключових пакетів підтримки сценаріїв. Позиція узгоджується з архітектурними рішеннями, сформульованими у розділі 2, проте акцент переноситься на фактичну організацію коду, відбір тестів і інженерні механізми запуску, видимі безпосередньо у файлах та конфігураціях [8; 9].

Специфікації зосереджено в каталозі ``tests/``, де реалізовано розділення за рівнем перевірки: ``ui`` (перевірки через браузерний контур), ``api`` (перевірки через HTTP-інтерфейс), ``e2e`` (наскрізні бізнес-сценарії) та ``security`` (перевірки політик доступу та небажаних шляхів взаємодії). У поточній версії проєкту наявні 34 файли ``*.spec.ts``, що відображає масштаб покриття та потребу в дисциплінованій структурі каталогів.

У ``playwright.config.ts`` директорія тестів задана як ``testDir: "/tests/"``, увімкнено ``fullyParallel: true``. У CI встановлено ``retries: 1`` та ``workers: 2``, локально кількість воркерів визначається типовою логікою Playwright. Глобальні опції ``use`` включають ``trace: on-first-retry``, ``screenshot: only-on-failure``, ``video: retain-on-failure``, що підвищує діагностичність при нестабільних або відхилених кейсах; репортери налаштовані як ``html`` (без автозапуску браузера) та ``list``.

Кількісно каталог ``tests/`` у поточній версії репозиторію містить 34 файли ``*.spec.ts``, зокрема 11 у підкаталозі ``ui``, 9 у ``api``, 2 у ``e2e`` та 12 у ``security``. Такий розподіл відображає пріоритет перевірок інтерфейсу та безпеки при збереженні окремого шару контрактних API-перевірок і

сквозних сценаріїв.

Розділення UI- та API-прогонів реалізовано на рівні `playwright.config.ts` через два проєкти Playwright: `ui` (профіль `Desktop Chrome`, виключення тестів із маркером `@api` за допомогою `grepInvert`) та `api` (фільтрація тестів за `@api` через `grep`, із базовим URL API-середовища). Такий поділ дозволяє незалежно масштабувати виконання та спрощує інтерпретацію результатів прогону [9].

Для керованого відбору піднаборів тестів у назвах тестів використовуються текстові маркери (на кшталт `@smoke`, `@regression`, `@security`, `@ui`, `@api`, `@ownership`), що узгоджується з практикою фільтрації Playwright через `--grep` і npm-скрипти (`test:smoke`, `test:security`) у `package.json`. Зокрема, `@smoke` застосовується для швидких інваріантних перевірок, `@regression` — для ширшого набору доменних сценаріїв, а `@security/@ownership` — для перевірок ізоляції доступу до сутностей.

Реалізація шару абстракцій відображена в каталогах `pages/` (класи сторінок для UI), `api/controllers/` (контролери `FarmController`, `FieldController`, `CropController`, `PlantingController`, `HarvestController`, `AuthController` та базовий `BaseApiController`), `fixtures/` (ланцюг розширення `test`), `data/factories/` (генерація валідних payload-ів) і `utils/data/` (зокрема `TestDataRegistry` та допоміжні функції для унікальних імен). Це безпосередньо підтримує сценарії з `tests/**` і відповідає моделі Page Object та API-обгортки, окреслених у п. 2.3–2.4 [13; 14].

У межах `tests/ui` специфікації згруповано за предметними підкаталогами: `auth`, `home`, `farm`, `field`, `crop`, `planting`, `harvest`, `validation`, `admin`. Таке групування полегшує навігацію в коді й узгоджується з поділом сторінкових класів у `pages/**`, що зменшує ризик «розмиття» відповідальності між файлами тестів і шаром Page Object.

Структурно-об'єктну взаємодію основних класів реалізації наведено на рисунку 3.1.

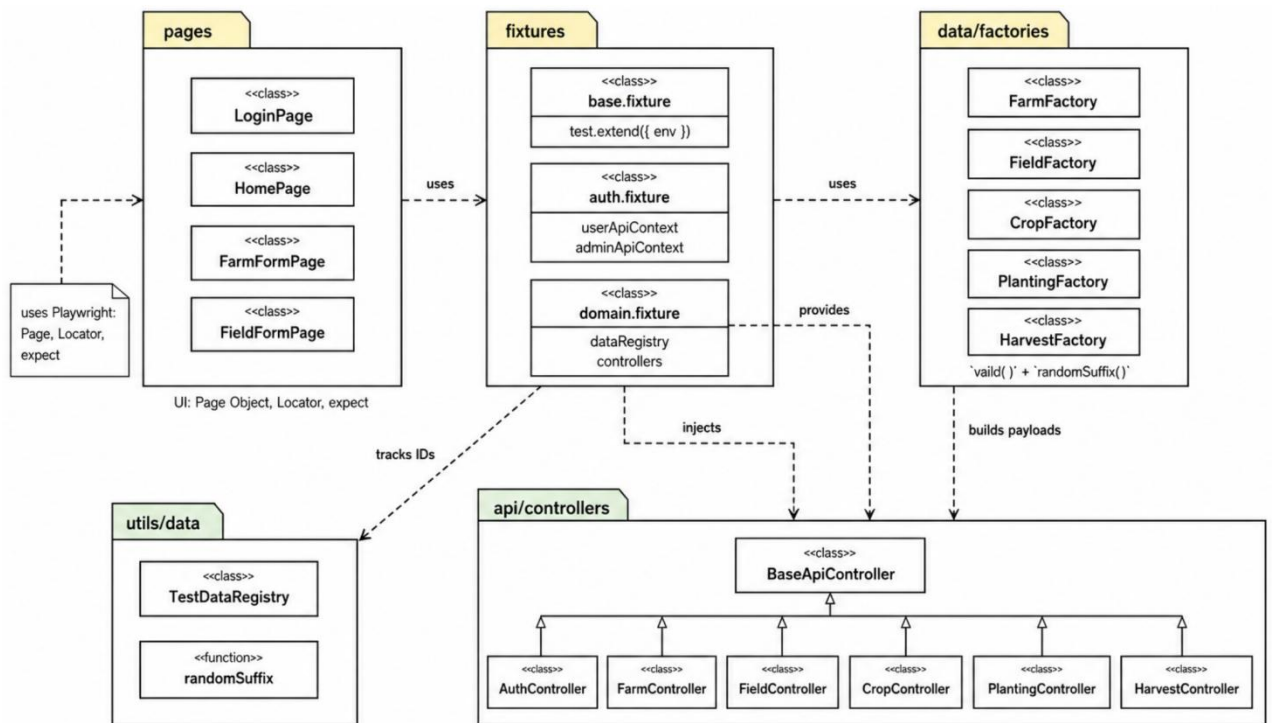


Рис. 3.1 Діаграма класів основних пакетів реалізації фреймворку (`pages/`, `api/controllers/`, `fixtures/`, `data/factories/`, `utils/data/`)

Таким чином, у підрозділі 3.1 встановлено, що фактичну структуру тестового коду та інженерні механізми відбору сценаріїв (каталоги, проекти Playwright, маркери), а також визначає місце графічної діаграми класів як узагальнення реалізації перед деталізацією UI-, API-, E2E- та security-рівнів у наступних підрозділах.

3.2 Реалізація UI-автотестів (Page Object і специфікації)

У межах цього підрозділу було розкрито реалізацію UI-автотестів як поєднання специфікацій Playwright (`\*.spec.ts`) та класів Page Object у каталозі `pages/`. На відміну від проєктних узагальнень розділу 2, тут наведено конкретні файли, маркери відбору та ланцюги викликів методів

сторінок, що безпосередньо відтворюють сценарії користувача в браузері [8; 13].

Мінімальний приклад smoke-перевірки автентифікації реалізовано у ``tests/ui/auth/login.spec.ts``: імпортується ``test`/`expect`` з ``fixtures/base.fixture.ts``, у тілі тесту створюються ``new LoginPage(page)`` та ``new HomePage(page)``, виконується ``await loginPage.open()`` (перехід на ``/login`` з очікуванням видимості полів), далі ``await loginPage.login(env.userName, env.userPassword)`` і перевірка ``await homePage.expectAuthenticatedDashboard()``. Негативний сценарій використовує ``expectInvalidLoginMessage()`` і додатково ``expect(page).toHaveURL(/\/login/)`` після введення некоректного пароля. Назви тестів містять маркери ``@smoke`` та ``@ui``, що узгоджується з прм-скриптом ``test:smoke`` і проєктом ``ui`` [9].

Доменні сценарії для сутності «ферма» зосереджені у ``tests/ui/farm/farm.spec.ts``, де ``test`` імпортується з ``fixtures/domain.fixture.ts`` (доступні ``page``, ``env``, ``dataRegistry``, ``farmController``). Підготовка автентифікованої сесії винесена у функцію ``loginViaUi(page, username, password)`` з ``utils/auth/uiAuth.ts``, яка всередині створює ``LoginPage`` і виконує ``open()`/`login()``, що зменшує дублювання коду між тестами списку, створення, оновлення та видалення ферми.

Взаємодія з формами та списками реалізована класами ``FarmListPage`` та ``FarmFormPage`` (``pages/farm/``): список відкривається через ``goto("/farms")`` з перевіркою видимості посилання «додати ферму»/«add farm», підтримуються операції ``clickAddFarm()``, ``expectFarmVisible(name)``, навігація до редагування/видалення через локатор рядка ``tr`` з текстом імені та посилання з ``href`` або рольовими назвами; форма перевіряє видимість полів ``name`/`location`` і виконує ``submit(name, location)`` з кліком по кнопці «зберегти»/«save». У тесті створення використовується ``FarmFactory.valid()`` для генерації унікальних значень, після успіху ідентифікатор ферми витягується зі списку та реєструється в ``dataRegistry`` для подальшого

teardown.

Ключові методи якірних класів зведено так: `LoginPage` — `open()`, `login(username, password)`, `expectInvalidLoginMessage()`, `HomePage` — `expectAuthenticatedDashboard()`, `FarmListPage` — `open()`, `clickAddFarm()`, `expectFarmVisible(name)`, `getFarmIdByName(name)` (витяг ідентифікатора з `href` посилання редагування в рядку таблиці); `FarmFormPage` — `expectFormVisible()`, `submit(name, location)`. Перевірки базуються на `expect` Playwright і семантичних локаторів `getByRole`, а для полів форми — стабільних `input[name=...]`.

Окрім наведених «якірних» прикладів входу та CRUD для ферми, у `tests/ui/auth/session.spec.ts` перевіряються вихід із сесії (форма з `action="/logout"`) та відображення посилань входу й реєстрації після лог-ауту, а також редирект анонімного користувача з маршруту `/farms` на сторінку автентифікації (маркування `@ui @security`). У `tests/ui/auth/register.spec.ts` використовуються `RegisterPage` та `LoginPage`, унікальне ім'я користувача формується через `randomSuffix`, сценарій позначено як `@ui @critical` через критичність реєстрації та подальшого входу в систему. Файл `tests/ui/validation/forms.validation.spec.ts` демонструє перевірку обов'язкових полів через нативний виклик `HTMLFormElement.reportValidity()` для маршрутів створення сутностей, зокрема `/farms/add`, `/fields/add`, `/crops/add` та інших форм домену. У `tests/ui/admin/admin.spec.ts` поєднано сценарії адміністративного доступу (наприклад, очікуваний перехід з `/admin` на `/admin/users`) з перевіркою заборони доступу звичайного користувача до адміністративної статистики.

Незалежно від предметного домену в UI-специфікаціях зберігається єдиний шаблон підготовка дії перевірка: фікстури та/або сторінкові об'єкти формують контекст, дії користувача або перехід за URL виконують сценарій, а `expect` фіксує стан URL, видимість локаторів або значення, отримані з DOM. Коментарі `// arrange`, `// act`, `// assert` у коді лише маркують етапи й не впливають на виконання тесту.

Послідовність кроків smoke-сценарію входу (spec → сторінкові об'єкти → очікування) доцільно подати графічно; відповідна UML-діаграма послідовностей наведена на рисунку 3.2.

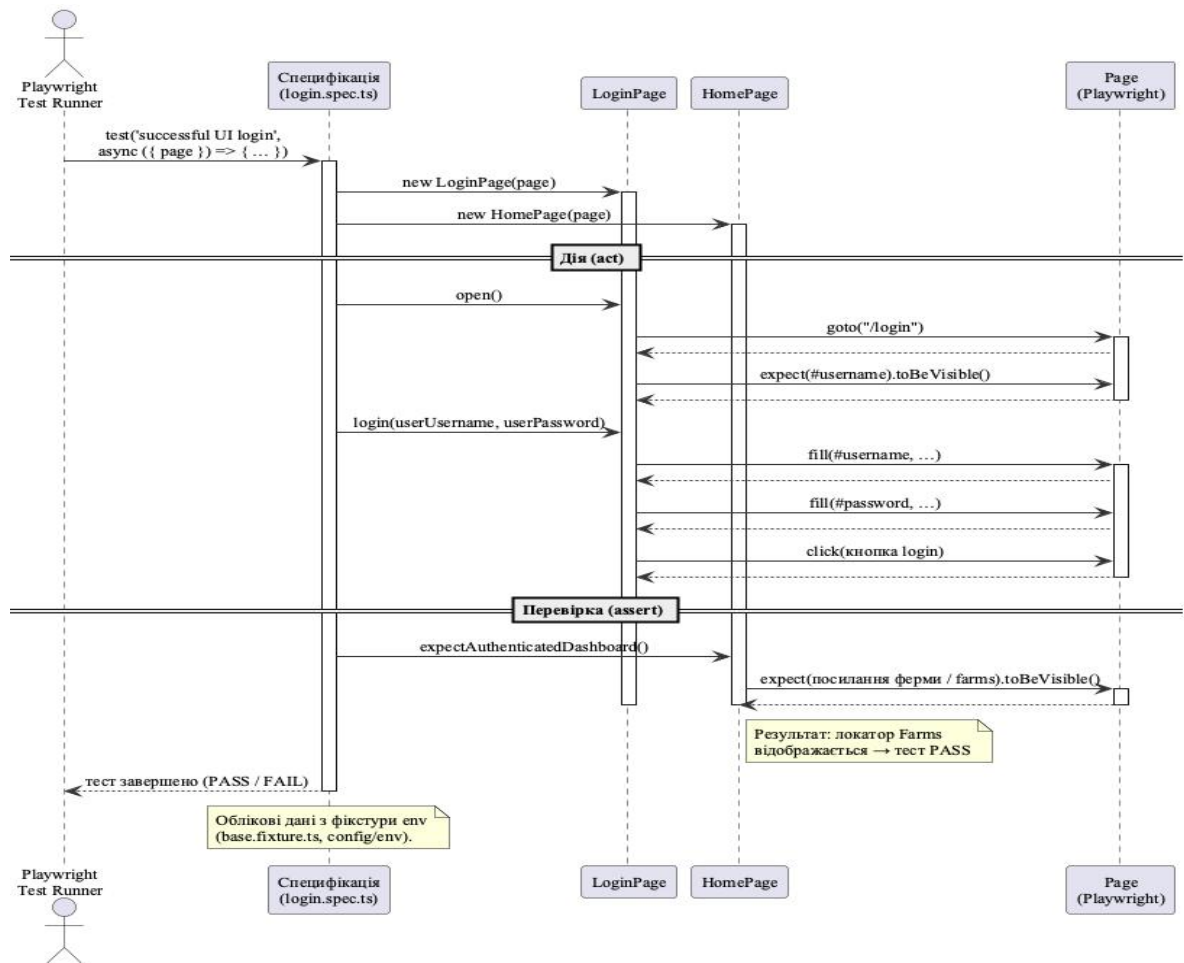


Рис. 3.2 Діаграма послідовностей для сценарію входу в систему
(`tests/ui/auth/login.spec.ts`, `LoginPage`, `HomePage`)

3.3 Реалізація API-автотестів (HTTP-клієнт і контролери доступу)

Цей підрозділ описує реалізацію автоматизованих перевірок доступу до даних на рівні HTTP-клієнта Playwright (`APIRequestContext`) без відтворення DOM. Специфікації зосереджено в каталозі `tests/api/`, де згруповано дев'ять файлів `\*.spec.ts` за доменами `auth`, `health`, `farm`, `field`, `crop`, `planting`, `harvest`, `validation`, а також окремий набір `contracts` для узагальнених контрактних перевірок.

У `playwright.config.ts` для проєкту `api` базовою адресою виступає `ENV.apiUrl` (див. `config/env.ts`), а відбір тестів здійснюється за шаблоном `grep: /@api/`. Проєкт `ui`, навпаки, використовує `grepInvert: /@api/`, що унеможливорює змішування UI- та API-специфікацій в одному прогоні без зміни коду тестів.

Мінімальна перевірка доступності та базових інваріантів безпеки реалізована у `tests/api/health/health.spec.ts` (`@api @smoke`): анонімний запит `GET` до `${env.apiUrl}/login` має статус 200 і містити підрядок `username`, тоді як звернення до `${env.apiUrl}/api/farms` без автентифікації з `maxRedirects: 0` має завершуватися одним із кодів 302, 401 або 403.

Сценарії реєстрації на рівні API зосереджені у `tests/api/auth/register.spec.ts` (`@api @regression`): `POST` на `/api/auth/register` для нового користувача та для дублікату імені повертає 302, а заголовок `Location` містить шлях до `/login`, що відповідає серверній політиці перенаправлення.

Аутентифікований контекст для подальших JSON-операцій формується у `fixtures/auth.fixture.ts`. Функція `createAuthenticatedApiContext` створює початковий `APIRequestContext`, виконує `AuthController.loginWithFormSession` (отримання маркера `_csrf` з HTML-сторінки `/login` і відправлення форми на `/login` з очікуванням 302), після чого зчитує маркер із відповіді `GET /farms/add` і створює новий контекст із `storageState` та заголовком `X-CSRF-TOKEN`. Для ролей користувача й адміністратора надано фікстури `userApiContext` та `adminApiContext`.

Багатоетапне формування автентифікованого HTTP-клієнта з передачею маркера CSRF у заголовку для подальших JSON-запитів доцільно ілюструвати послідовністю взаємодій компонентів; відповідна UML-діаграма послідовностей наведена на рисунку 3.3.

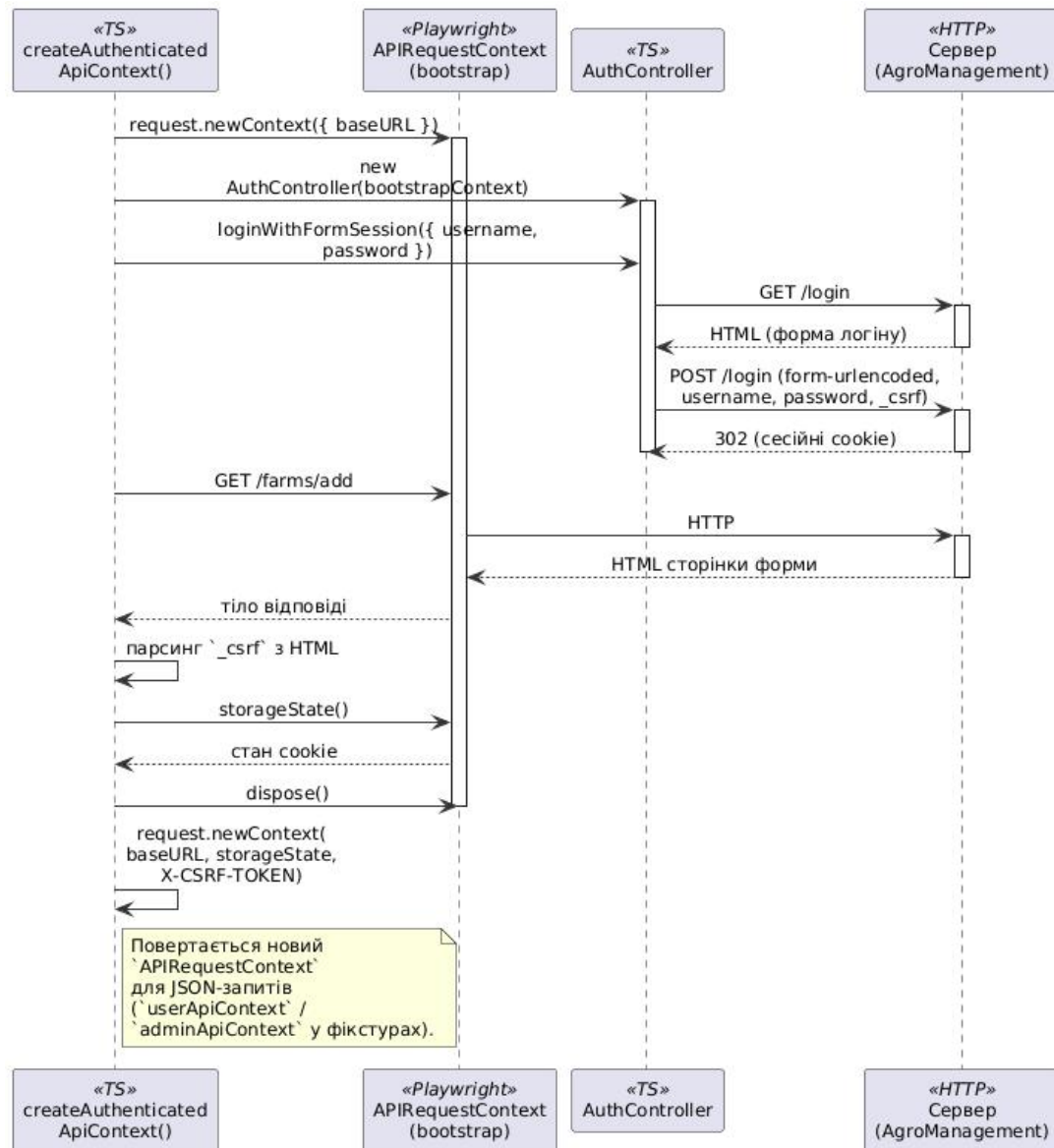


Рис. 3.3 Діаграма послідовностей підготовки `APIRequestContext` для API-тестів (`fixtures/auth.fixture.ts`, функція `createAuthenticatedApiContext`)

Файл `fixtures/domain.fixture.ts` розширює `auth.fixture` контролерами `FarmController`, `FieldController`, `CropController`, `PlantingController`, `HarvestController` та об'єктом `TestDataRegistry`. Після завершення тесту виконується послідовне видалення зареєстрованих сутностей (збори, посіви, поля, культури, ферми) у зворотному порядку з обгорткою `try/catch` як best-effort cleanup.

Шар `api/controllers/` інкапсулює виклики REST-подібних маршрутів. Наприклад, `FarmController.createFarm` виконує `POST /api/farms` з

очікуванням 201, `getFarmById` повертає тіло при 200 або код без тіла, `updateFarm` використовує `PUT`, `deleteFarm` — `DELETE` з очікуванням 204. Клас `BaseApiController` містить спільний конструктор із `APIRequestContext` і метод `expectStatus` для узгоджених перевірок коду відповіді.

Приклад повного CRUD-циклу для ферми наведено у `tests/api/farm/farm.spec.ts` (`@api @regression`): створення за допомогою `FarmFactory.valid()`, реєстрація ідентифікатора в `dataRegistry`, читання через `userApiContext.get` із розбором JSON (`utils/api/parseJsonWithFallback.ts`), оновлення назви та видалення з подальшим `getFarmById` з очікуванням 404.

Аналогічні CRUD-набори реалізовано у `tests/api/field/field.spec.ts`, `tests/api/crop/crop.spec.ts`, `tests/api/planting/planting.spec.ts` та `tests/api/harvest/harvest.spec.ts` з тим самим маркером `@api @regression` і використанням відповідних фабрик із `data/factories/`.

Файл `tests/api/contracts/contracts.sanity.spec.ts` перевіряє наявність ключових полів у відповідях `GET` за ідентифікатором і в масивах списків для ферм, полів, культур, посівів і зборів; для сценарію зборів використовується допоміжна функція `setupPlanting` з `domain.fixture.ts` для підготовки ланцюга залежностей.

Валідація вхідних даних на боці API зосереджена у `tests/api/validation/validation.spec.ts`: через `userApiContext` надсилаються некоректні payload-и на `POST /api/farms`, `/api/fields`, `/api/crops` та пов'язані маршрути з очікуванням 400; для частини кейсів із недійсними зовнішніми ключами допускається 400 або 404 залежно від реакції сервера.

3.4 Сквозні E2E-сценарії та перевірки безпеки

Далі розглянуто два класи інтеграційних перевірок, що виходять за межі окремих UI- або API-специфікацій: сквозні бізнес-сценарії у каталозі

`tests/e2e/business/` та тести безпеки й моделі власності у `tests/security/`. Обидва типи файлів запускаються в межах проєкту `ui` Playwright, оскільки маркер `@api` у них не використовується.

Каталог `tests/e2e/business/` містить два файли `\*.spec.ts`. У `farm-field.spec.ts` (`@ui @critical`) поєднано `loginViaUi`, програмне створення ферми через `FarmController`, заповнення форми поля (`FieldFormPage`) та перевірки у `FieldListPage` і `FarmListPage` із реєстрацією ідентифікаторів у `dataRegistry`. У `farm-to-harvest.spec.ts` тим самим маркером будується повний ланцюг сутностей через контролери та перевіряється відображення ключових записів у відповідних UI-списах.

У `tests/security/` згруповано 12 файлів `\*.spec.ts` за предметними підкаталогами (`farm`, `field`, `crop`, `planting`, `harvest`, `ownership`, `ui` для прямих посилань). Типові маркери — `@security`, `@ownership`, а також `@api` або `@ui` залежно від каналу перевірки. Утиліта `expectForbiddenOrNotFound` (`utils/api/securityAssertions.ts`) фіксує очікування кодів 403 або 404 для несанкціонованих операцій.

Послідовність склейки UI-автентифікації, API-підготовки даних і кроків форм для сценарію «ферма → поле» наведена на рисунку 3.4.

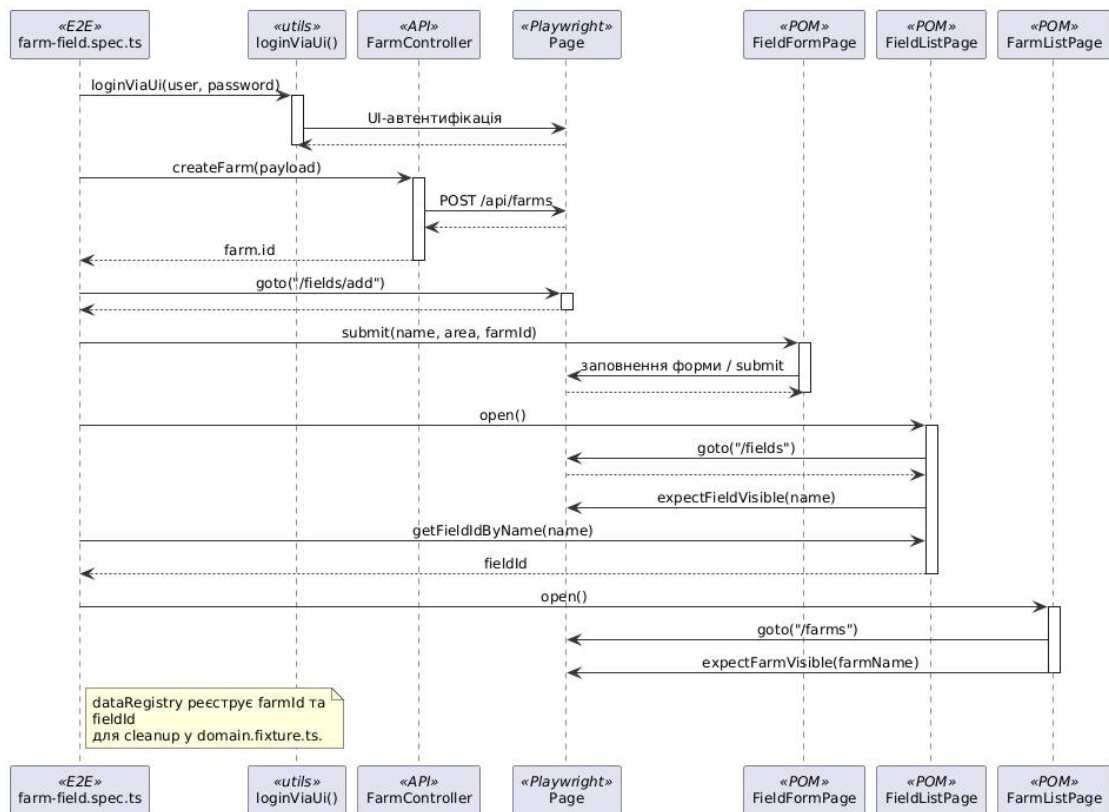


Рис. 3.4 Діаграма послідовностей сквозного сценарію «ферма → поле»
(`tests/e2e/business/farm-field.spec.ts`)

Перевірка блокування прямих UI-маршрутів редагування/видалення чужої ферми та узгодженості API-читання ілюструється на рисунку 3.5 (файл `tests/security/ui/farm-direct-link.security.spec.ts`).

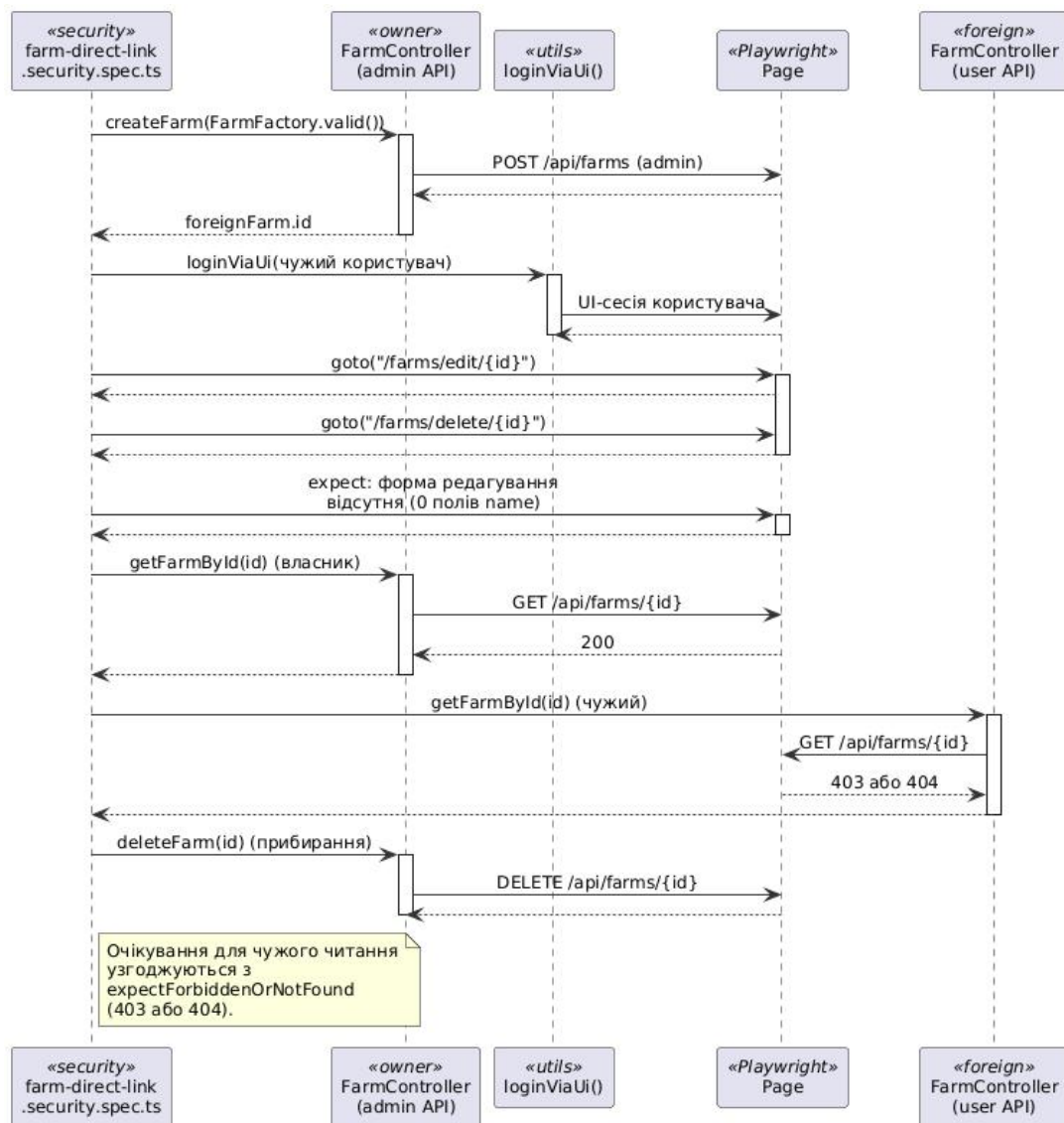


Рис. 3.5 Діаграма послідовностей перевірки заборони чужих UI-маршрутів і відмови при читанні чужого ресурсу API (`expectForbiddenOrNotFound`)

3.5 Звітність Playwright, артефакти діагностики та інтерпретація результатів прогонів

Нижче викладено можливості стандартного HTML-звіту Playwright та супутніх артефактів діагностики, які формуються згідно з `playwright.config.ts` у каталозі `test-automation-framework/` і зберігаються після прогону у `playwright-report/` та `test-results/`. Інтерфейс звіту

орієнтований на інженера тестування й розробника: він дає зведений зріз стану набору (успішні, невдалі, нестабільні та пропущені тести), дозволяє швидко знайти проблемний сценарій у межах конкретного файлу `*.spec.ts` і перейти до деталей помилки без необхідності повторно відтворювати весь прогон локально лише заради одного рядка логу.

Після локального виконання тестів звіт відкривають командою ``npm run report:open`` (у ``package.json`` це виклик ``playwright show-report``). У конфігурації вказано репортер ``html`` з параметром ``open: "never"`` (HTML-звіт генерується, але браузер не відкривається автоматично) та репортер ``list`` для стислого консольного підсумку під час прогону. Глобальні опції ``use`` містять ``trace: "on-first-retry"`, `screenshot: "only-on-failure"`, `video: "retain-on-failure"```: для невдалих або повторних спроб зберігаються матеріали, які доповнюють текст помилки у веб-інтерфейсі звіту. У workflow ``.github/workflows/test-automation-pr-gate.yml`` передбачено завантаження каталогів ``playwright-report`` та ``test-results`` як артефактів job, що полегшує перегляд результатів у команді. Нижче наведено розгорнутий опис елементів HTML-звіту та місця для ілюстрацій додаткових артефактів (рис. 3.6–3.10).

Головний екран HTML-звіту Playwright поєднує функції дашборда й навігатора. У верхній частині зазвичай відображаються підсумкові лічильники за статусами виконання (успіх, збій, нестабільність, пропуск), сумарний час прогону та службові відомості (наприклад, мітка часу запуску). Поруч розташовані фільтри за статусом і рядок пошуку: за назвою тесту, фрагментом шляху до файлу або за текстовим маркером на кшталт ``@api``, ``@security``, ``@smoke``. Такий поділ знижує час на локалізацію регресії: замість перегляду всієї консолі достатньо відфільтрувати «невдалі» або ввести ключове слово предметної області (ferm, field, ownership тощо).

Нижче за зведенням тести згруповано за файлами специфікацій; кожен блок можна розгортати й згортати. Для кожного окремого тесту видно тривалість виконання, що допомагає виявляти «повільні» кейси та нестабільність через тайм-аути. Для невдалих або повторених спроб поруч із

записом часто доступне посилання на trace (наприклад, «View Trace»): воно веде до інтерактивного переглядача з покроковим журналом дій браузера й мережесих звернень. Для проєктів `ui` та `api`, заданих у `playwright.config.ts`, у звіті відображаються окремі контексти прогону, що полегшує відокремлення помилок інтерфейсу від помилок контракту REST.

Перевага HTML-звіту для кваліфікаційної роботи полягає в наочності: читач бачить не абстрактний «pass/fail», а структуру реального набору тестів фреймворку AgroManagement, відповідність імен тестів бізнес-вимогам і наявність тегів як засобу керованої вибірки. Для автора автотестів звіт є робочим інструментом причинного аналізу: від зведення до конкретного assertion, далі — до trace, знімка екрана або відео, якщо конфігурація зберігання артефактів це дозволяє.

Зведений вигляд HTML-звіту з панеллю підсумку, фільтрами, пошуком і згрупованими специфікаціями наведено на рисунку 3.6.

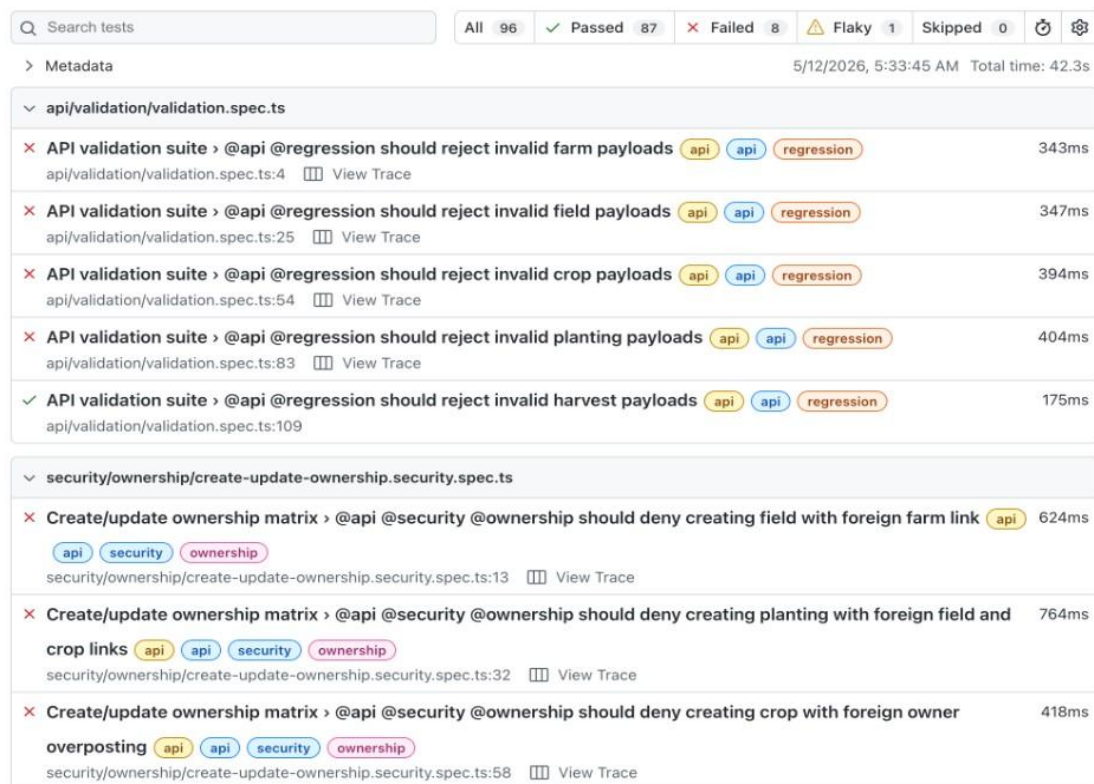


Рис. 3.6 Зведений HTML-звіт Playwright: підсумок прогону, фільтри за статусом, пошук, групування тестів за файлами специфікацій і посилання на trace

Після вибору невдалого тесту звіт переходить до картки деталізації. Тут відображаються повідомлення про помилку (зокрема результат `expect(...)` для assertion), фрагмент стеку з посиланням на рядок специфікації, а також службові вкладки або блоки з вкладеннями (attachments), якщо під час прогону збережено додаткові дані. Для API-тестів корисно бачити контекст запиту й відповіді в trace або в логах мережі; для UI-тестів — прив'язку до кроку взаємодії зі сторінкою. Такий рівень деталізації скорочує цикл «збій → гіпотеза → виправлення», бо дозволяє одразу зорієнтуватися, чи це регресія даних, зміна селектора, валідація на сервері чи проблема авторизації.

Приклад картки невдалого тесту з повідомленням про помилку та контекстом наведено на рисунку 3.7.



Рис. 3.7 Картка невдалого тесту в HTML-звіті Playwright

Оскільки у `playwright.config.ts` для trace встановлено режим `on-first-retry`, повний trace зазвичай доступний для спроб після першого повтору. Переглядач trace відтворює хронологію дій (переходи, кліки, заповнення полів), показує мережеві запити й відповіді, скріншоти ключових кроків усередині сценарію. Це корисно для демонстрації в дипломній роботі поведінки тесту «як на лінійці часу» та для пояснення, на якому етапі виявлено розбіжність між очікуваним і фактичним результатом.

Фрагмент інтерфейсу trace viewer з покроковим виконанням наведено на рисунку 3.8.

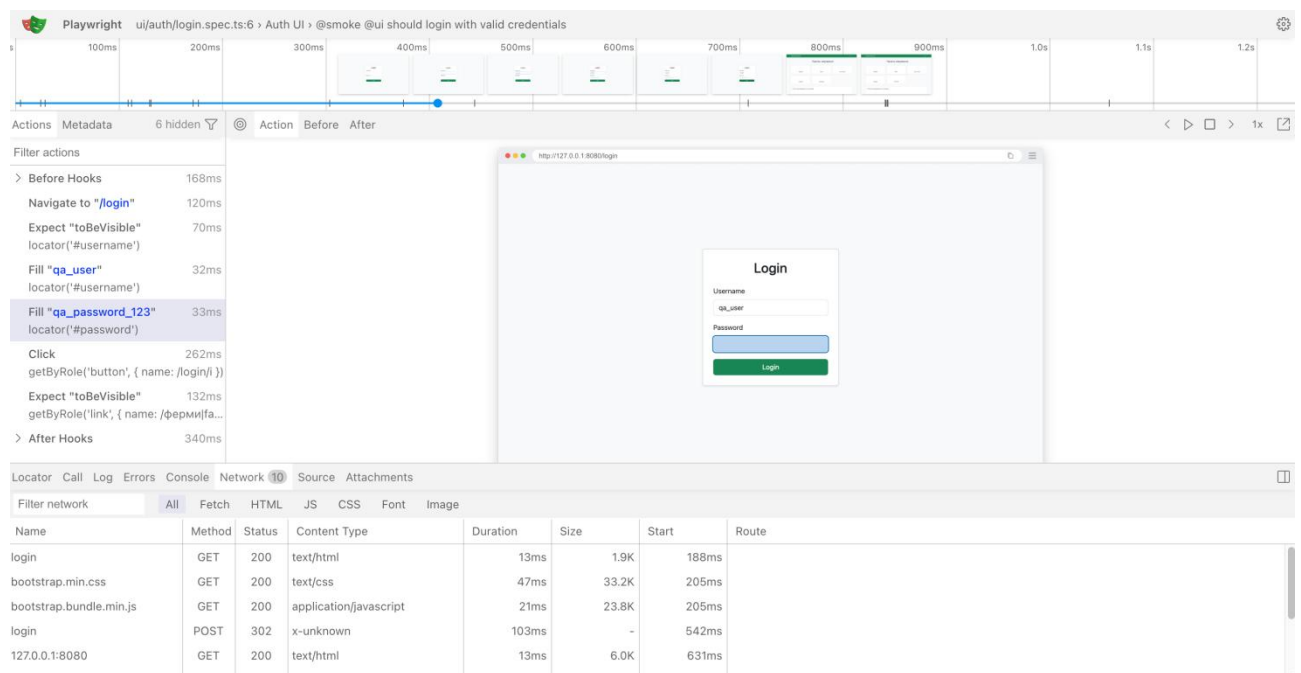


Рис. 3.8 Фрагмент Playwright Trace Viewer з покроковим виконанням тесту

Параметр `screenshot: "only-on-failure"` зменшує обсяг зайвих файлів: знімок екрана створюється лише для невдалих тестів і зберігається поруч із метаданими у `test-results/`, а в HTML-звіті може відображатися як вкладення. Візуальний знімок стану сторінки доповнює текст помилки, коли assertion стосується DOM-структури або видимості елемента: він фіксує фактичний UI

на момент збою. Приклад знімка екрана для невдалого тесту наведено на рисунку 3.9.

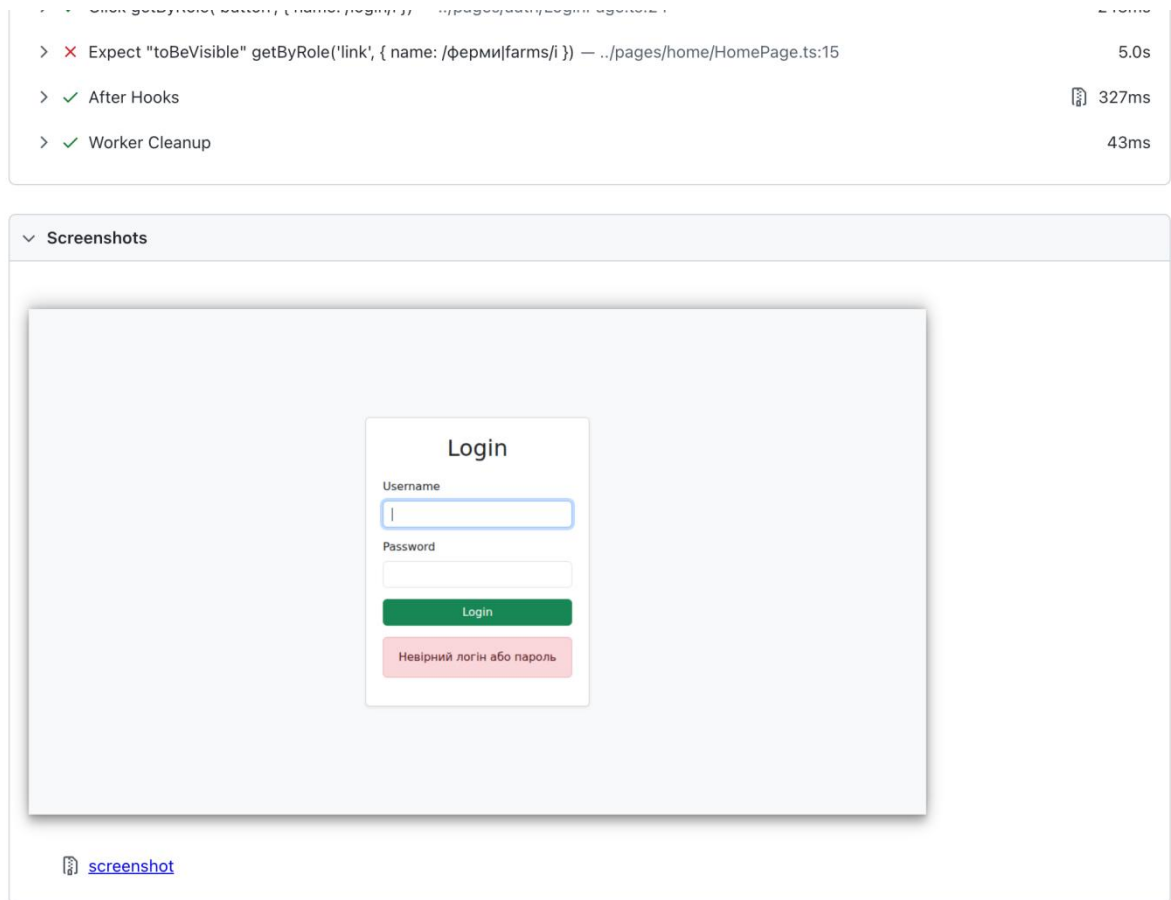


Рис. 3.9 Скріншот стану сторінки при падінні UI-тесту

Значення ``video: "retain-on-failure"``` забезпечує збереження відеозапису спроби для невдалих тестів. Відео корисне, коли потрібно показати динаміку інтерфейсу: послідовність переходів, анімації, спливаючі вікна, стан завантаження. У сукупності з trace і знімком екрана це дає три рівні діагностики — часова шкала подій, статичний кадр і безперервний рух — і підвищує переконливість розділу про тестування в роботі.

Наявність відео для невдалого тесту в інтерфейсі звіту наведено на рисунку 3.10.

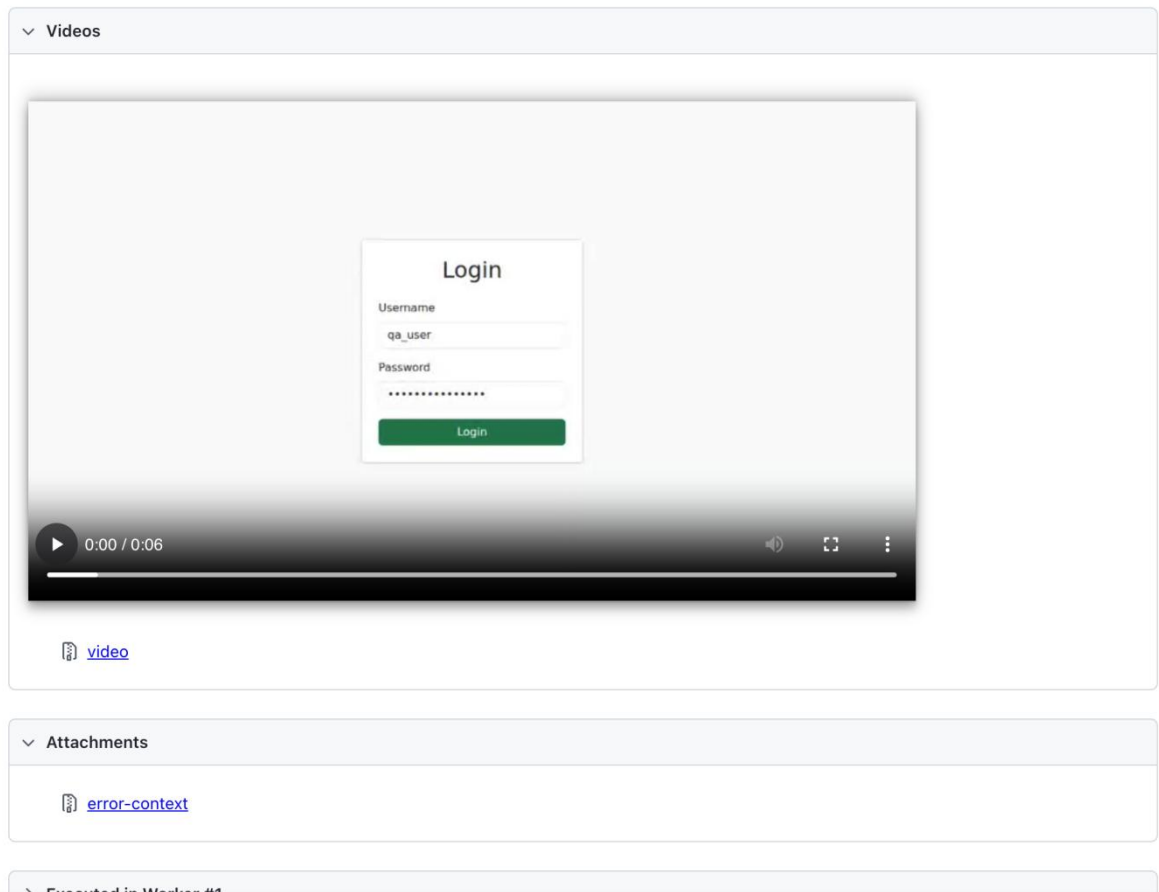


Рис. 3.10 Відеозапис спроби невдалого тесту у звіті Playwright

Отже HTML-звіт Playwright забезпечує зручну навігацію від зведеної статистики до деталей конкретного тесту, а налаштовані `trace`, знімки екрана та відео підсилюють діагностичну цінність автоматизованого прогону для вебзастосунку AgroManagement. Рисунки 3.6–3.10 ілюструють ланцюжок «зведення → картка збою → `trace` → знімок → відео»; консольний репортер `'list'` залишається допоміжним засобом під час виконання в терміналі й у логах CI, проте окремі скріншоти консольного виводу для цього розділу не передбачено.

4 ОЦІНКА ЕФЕКТИВНОСТІ

4.1 Аналіз швидкості виконання тестів

У цьому підрозділі зосереджуємося на емпіричній тривалості: що саме показує підсумок GitHub Actions порівняно з локальним часом Playwright у консолі, і як інтерпретувати різницю без повторення параметрів паралелізму та переліку npm-скриптів, уже наведених у розділі 3 [7; 12].

Конфігурація `playwright.config.ts` (паралелізм, повтори в CI, репортери) описана в розділі 3; для оцінки швидкості важливо розрізняти тривалість усього job (оточення, залежності, тести, публікація звіту) і час лише рушія тестів або локального запуску, що наочно видно на рис. 4.1–4.2 [9].

Вибір набору (`'smoke'`, `'full'`, `'security'` тощо) керується `'SELECTED_SUITE'` у workflow і скриптами `'package.json'`; кількісні відмінності між «коротким gate» і повним зрізом знімаються з міток часу в Actions та з HTML-звіту [12].

Окремий нюанс інтерпретації часу пов'язаний із повторними спробами в CI (`'retries'` у `playwright.config.ts`): у звіті може з'явитися кілька спроб одного тесту, тому сумарна тривалість не завжди дорівнює «одному проходженню» сценарію. Для оцінки ефективності варто зазначати, чи аналізується перша спроба, остання чи сума спроб [2; 3; 9].

У сумарний час CI входять очікування сервісів, `'npm ci'`, встановлення Chromium і публікація звіту; ці хвилини не тотожні внутрішній сумі тривалостей окремих `'test()'` у звіті Playwright [12].

Повторюваність вимірів є умовою коректного порівняння: зміна версії Playwright, кешу залежностей або паралелізму на ранері може зсунути хвилини навіть без змін у тестах. Тому у тексті роботи доцільно вказувати мітку коміту, тип події CI та обраний набір, коли наводяться абсолютні значення тривалості [12].

Для зіставлення наборів доцільна компактна таблиця: набір / подія CI, кількість тестів за звітом, час тестів, час job; значення підставляються з конкретних прогонів без дублювання пояснень з п. 3.

HTML-звіт Playwright дозволяє уточнити не лише сумарний час, а й «розподіл» між файлами специфікацій і окремими тестами; це корисно, коли потрібно пояснити, який саме шар набору дає основний внесок у хвилини прогону [7; 8].

Порівнюючи локальний запуск і CI, варто також пам'ятати про відмінність «холодного» старту оточення в хмарі та теплий локальний кеш: однакові цифри секунд рідко означають однакові накладні зусилля інженера на підготовку оточення [12].

Політика збоїв (`STRICT_MODE``, інформаційний `full``) впливає на інтерпретацію часу job; без повторення коду workflow зазначимо, що деталі режимів наведені в `.github/workflows/test-automation-pr-gate.yml``, а організаційний сенс — у п. 4.3 [12].

З погляду характеристики якості за ISO/IEC 25010 час відгуку тестового конвеєра пов'язаний із властивостями продуктивності та ефективності використання ресурсів у широкому сенсі: надмірна тривалість знижує практичну придатність автоматизації, тоді як збалансований поділ наборів дозволяє поєднувати швидкий контроль і глибоку перевірку [1]. Для кваліфікаційної роботи бакалавра достатньо продемонструвати розуміння цих залежностей і вміння читати наявні метрики, а не будувати складні моделі прогнозування часу. Для команди це означає, що «довгий» job не завжди є ознакою низької якості коду: частина часу відображає інфраструктурну варіативність і політику повторів, тому порівняння прогонів доцільно робити за однакових умов оточення та однакового набору [1].

Як ілюстрацію зняття показників наведено два зрізи з реальних матеріалів прогону. На рисунку 4.1 показано сторінку підсумку workflow GitHub Actions для пайплайну AgroManagement Autotests Pipeline (файл `.github/workflows/test-automation-pr-gate.yml``): сумарна тривалість 3 хв 11 с

охоплює повний цикл job у середовищі CI, включно з підготовкою залежностей, контейнерного стеку, виконанням обраного набору тестів та публікацією HTML-звіту. Зокрема, крок Run autotests тривав 2 хв 49 с, а крок Publish Playwright report — 9 с. У підсумку job зазначено, що набір `full` завершився з помилками тестів, проте для цього сценарію workflow налаштовано як інформаційний (неблокуючий), що узгоджується з поясненням політики `STRICT\_MODE` у тексті підрозділу [12].

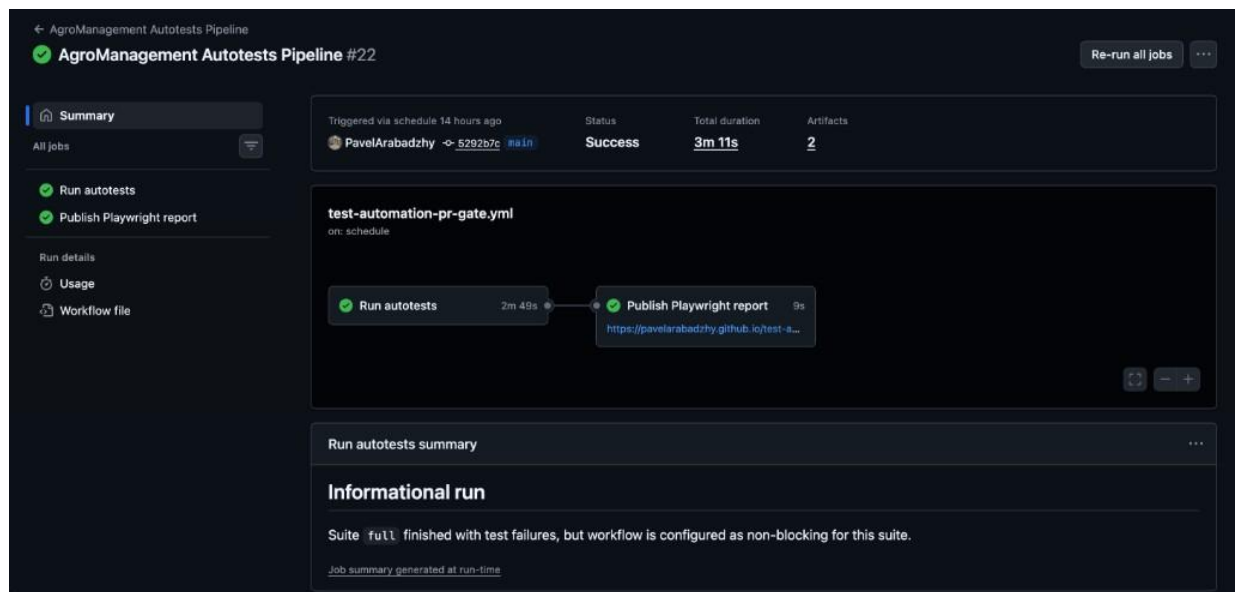


Рис. 4.1 Приклад підсумку GitHub Actions: сумарний час пайплайну, тривалість job Run autotests та кроку публікації звіту Playwright (набір `full`)

Для порівняння на рисунку 4.2 наведено локальний прогін у терміналі IDE: 86 тестів успішно виконано за 34,4 с. Цей час відображає лише роботу тестового рушія Playwright після готовності оточення на машині розробника й не включає кроки CI (Docker, очікування сервісів, `npm ci`, `npx playwright install`). Тому різниця порядків між хвилинами повного пайплайну та десятками секунд локального прогону є очікуваною й пояснюється різницею контекстів, а не повільністю самих тестів [7; 8; 12].

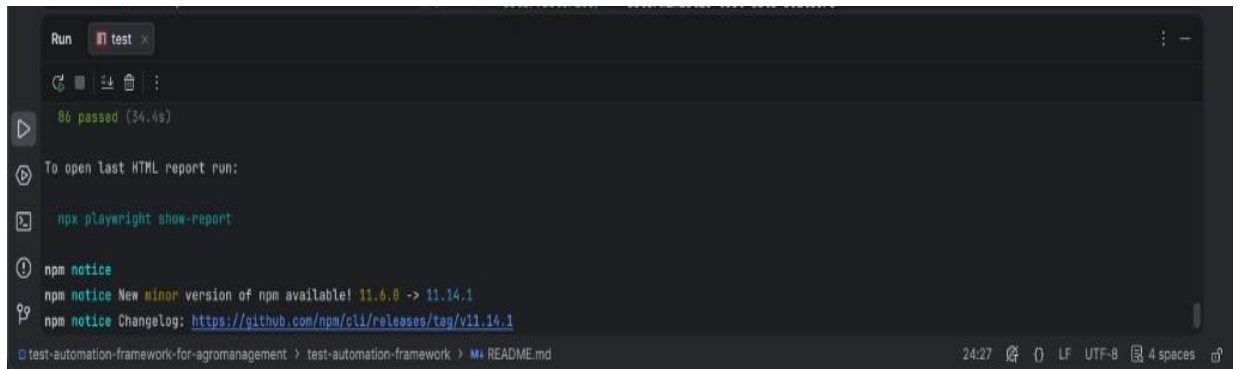


Рис. 4.2 Локальне виконання Playwright: кількість пройдених тестів і час прогону (приклад з консолі)

Узагальнюючи підрозділ 4.1, рисунки 4.1–4.2 підкреслюють головне: тривалість job не дорівнює «лише тестам». Деталі пайплайну залишаються в workflow з розділу 3; тут зафіксовано лише зняті інтервали та підхід до табличного порівняння наборів.

Що підтверджено файлами проєкту (підрозділ 4.1): фактичні знімки та числа з обраного прогону; для методики — ``.github/workflows/test-automation-pr-gate.yml`` і ``.package.json`` (деталі конфігурації — у розділі 3).

Що потребує уточнення або доповнення вручну (підрозділ 4.1): оновити скріншоти за зміною UI GitHub; заповнити таблицю тривалостей для інших наборів після нових прогонів.

4.2 Охоплення системи автоматизованими тестами

Під «покриттям» тут розумітимемо охоплення функціональних сценаріїв і ризиків автотестами, а не code coverage Spring-додатку; інструментальний збір покриття коду у проєкті не налаштовувався [7; 8].

Структуру каталогу ``.tests/``, проєкти ``.ui/`` і ``.api/`` і зміст ключових специфікацій розкрито в розділі 3; у підрозділі 4.2 лишаємо лише зведення для оцінки охоплення — таблицю 4.1 [9].

Автоматизація не замінює повністю дослідницьке тестування та оцінку

зручності: існують «сліпі зони» візуальних дрібниць, нестандартних розкладок даних і сценаріїв, які важко стабілізувати у вигляді assertion. Тому охоплення в таблиці 4.1 слід читати як навмисно обраний набір ризиків, а не як абсолютну повноту всіх можливих перевірок [7].

Маркери '@smoke', '@regression', '@security', '@critical', '@ownership' задають шари швидкого gate, регресії, безпеки та сквозних ланцюжів; приклади файлів наведено в розділі 3, тут — лише узагальнення в табл. 4.1 [4; 8; 12].

Маркери варто трактувати як мінімальний механізм прослідковуваності: вони пов'язують тест із рівнем критичності і типом ризику (швидкий gate, регресія, безпека, сквозний ланцюг). У дипломній роботі корисно пояснити, як саме ці мітки підтримують керований вибір набору в СІ без ручного відбору тестів «на око» [8; 12].

Стислу відповідність між функціональними областями, каталогами тестів, маркерами та типовими наборами СІ наведено в таблиці 4.1.

Таблиця 4.1

Узагальнене охоплення функціональних областей AgroManagement
автотестами (каталоги, маркери, набори запуску)

Функціональна область	Каталог специфікацій	Проект маркери /	Приклад набору CI / npm
Швидкий контур доступності	`tests/api/health`, `tests/ui/auth`	`api` / `@api` `@smoke`; `ui` / `@smoke @ui`	`npm run test:smoke`, `SELECTED_SUITE=smoke`
Регресійні CRUD та форми	`tests/api/*`, `tests/ui/*` (окрім вузьких smoke)	`@api` `@regression`; `@ui @regression`	`npx playwright test --grep @regression`, `regression`
Безпека та володіння даними	`tests/security/**`	`@security`, `@ownership`, змішані `@ui` / `@api`	`npm run test:security`, `SELECTED_SUITE=security`
Сквозні бізнес-ланцюжки	`tests/e2e/business`	`@ui @critical`	`npm run test:ui` / повний `npm test`

4.3 Доцільність впровадження автоматизації тестування

Доцільність у вузькому організаційному сенсі оцінюємо через вплив на процес якості: зменшення рутинної регресії, передбачуваність результату перевірки після зміни, спільну картину стану для розробника й тестувальника, а також змогу швидко локалізувати збій за артефактами. Ці ефекти не зводяться до однієї метрики, але їх можна підкріпити спостереженнями з CI та звітів Playwright [12].

Практична доцільність проявляється у зменшенні координаційних втрат: замість узгодження «хто і що перевірить перед злиттям» команда спирається на автоматичний gate та опубліковані артефакти. Це не усуває людську оцінку ризику, але переносить дискусію на рівень зміни наборів і

пріоритетів, а не на рівень ручного відтворення тих самих кліків [12].

Специфікації Playwright виконують роль «живої» документації процесу: новий учасник команди може швидше зрозуміти очікувану поведінку системи, прочитавши назви тестів і теги, ніж лише текстовий чек-лист. Для кваліфікаційної роботи це важливий аргумент на користь підтримуваності фреймворку, навіть якщо обсяг коду тестів зростає [8].

Логіка ``smoke`` + ``STRICT_MODE`` проти інформаційного ``full`` уже проілюстрована часом `job` у п. 4.1 (рис. 4.1); тут підкреслимо лише наслідок для процесу: швидкий блокуючий контур і періодичний глибокий зріз без дублювання опису кроків `workflow` [12].

Окремий аспект доцільності — узгодженість правил репозиторію з реальною політикою ризику: якщо нічний повний прогін інформаційний, команда свідомо приймає, що частина відомих проблем продукту не блокує `pipeline`. Таке рішення має бути пояснене в тексті як управлінський компроміс між швидкістю зворотного зв'язку і повнотою блокування [12].

Додаткова доцільність — у поєднанні статичного ``typecheck``, збереження артефактів (``if: always``) і ``cleanup:data``: менше «шуму» від типових помилок, більше матеріалів для аналізу збою й чистіші повторні прогони [7; 9; 12; 15].

Підтримка автотестів також потребує інженерного супроводу: оновлення селекторів, адаптація до змін UI, ревізія даних у фабриках. Доцільність тут полягає в тому, що ці зусилля концентруються в репозиторії фреймворку й піддаються рев'ю, на відміну від розпорошених ручних перевірок без єдиного джерела істини [8].

Нарешті, наявність стандартизованого конвеєра підсилює культуру прозорості: результати менш залежать від індивідуальної дисципліни «запустив локально», оскільки для гілки існує спільний артефактний слід. Це знижує інформаційну асиметрію між авторами змін і рев'юерами [12].

Зв'язок із надійністю та повторюваністю перевірок за ISO/IEC 25010 і з проблемою нестабільних тестів у літературі підкріплює організаційний

аргумент без повторення технічних деталей пайплайна [1; 2; 3].

На основі підрозділу 4.3, доцільність автоматизації для AgroManagement полягає в інтегрованому контролі змін і зменшенні ручної регресії, у підвищеній відтворюваності перевірок і в прозорості діагностики. Технічна реалізація пайплайна деталізована в розділі 3, а п. 4.1–4.2 додають вимірювані та структурні аргументи; разом це формує цілісну картину практичної придатності рішення для командного процесу [1; 7; 12].

ВИСНОВКИ

У результаті виконаної кваліфікаційної роботи розроблено фреймворк автоматизованого тестування для вебзастосунку AgroManagement на базі Playwright і TypeScript. Реалізація забезпечує систематичну перевірку авторизації, доменних сутностей і пов'язаних бізнес-процесів, зменшує ризик регресій при змінах коду та скорочує час виявлення дефектів за рахунок повторюваних прогонів у локальному середовищі та CI.

У процесі роботи виконано такі завдання:

1. Досліджено предметну область вебзастосунку AgroManagement та визначено п'ять критичних доменних сутностей для автоматизованої перевірки: Farm, Field, Crop, Planting, Harvest.

2. Проведено порівняльний аналіз інструментів Selenium, Cypress, Playwright за 8 критеріями (швидкість, кросбраузерність, підтримка API, паралелізм, складність налаштування, стабільність, CI/CD, асинхронність) - на основі чого обрано Playwright як найбільш доцільний.

3. На основі аналізу AgroManagement визначено 6 функціональних і 4 нефункціональних вимог до фреймворку задля забезпечення повноти та відтворюваності автоматизованих перевірок.

4. Спроектовано багат шарову архітектуру фреймворку та модель організації тестового коду з використанням патернів Page Object Model, API Controllers, Factory та AAA-pattern для підвищення підтримуваності й масштабованості набору тестів.

5. Реалізовано фреймворк автоматизованого тестування на стеку TypeScript + Playwright + Node.js, який інтегрований із системою GitHub Actions (CI/CD) для перевірки основних функціональних сценаріїв вебзастосунку AgroManagement.

6. Проведено запуски автоматизованих тестів та оцінено час виконання набору з 96 тестів: повний прогон у CI/CD триває близько 42 секунд, тоді як

ручне тестування тих самих сценаріїв займає близько 5 годин, що забезпечує прискорення у понад 400 разів.

Отриманий результат підтверджує практичну доцільність запропонованого рішення: сформований фреймворк забезпечує відтворювані перевірки критичних сценаріїв AgroManagement, інтеграцію з конвеєром злиття змін і зручну діагностику за HTML-звітами Playwright, що сприяє підтримці стабільної якості вебзастосунку в умовах ітеративної розробки.

Робота пройшла апробацію. За її результатами опубліковано такі тези доповідей:

1. Арабаджи П.О., Бажан Ю.П. Аналіз застосування автоматизованого тестування для інформаційної системи AgroManagement. Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, ДУІКТ. Збірник тез. - К.: ДУІКТ, 2025, С.200-202.

2. Арабаджи П.О., Бажан Ю.П. Аналіз критеріїв вибору інструменту автоматизованого тестування для інформаційної системи AgroManagement. Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, ДУІКТ. Збірник тез. - К.: ДУІКТ, 2026, С.334-335.

ПЕРЕЛІК ПОСИЛАНЬ

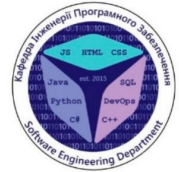
1. ISO/IEC 25010:2011. Information technology — Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. ISO. 2011. URL: <https://www.iso.org/standard/35733.html> (дата звернення: 06.04.2025).
2. Tahir A., Rasheed S., Dietrich J., Hashemi N., Zhang L. Test flakiness' causes, detection, impact and responses: A multivocal review // Journal of Systems and Software. 2023. Vol. 206. P. 111789. URL: <https://doi.org/10.1016/j.jss.2023.111789> (дата звернення: 13.04.2025).
3. Parry O., Kapfhammer G. M., Hilton M., McMinn P. Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models // Empirical Software Engineering. 2023. Vol. 28. Article 72. URL: <https://doi.org/10.1007/s10664-023-10307-w> (дата звернення: 21.04.2025).
4. OWASP Application Security Verification Standard (ASVS). OWASP Foundation. URL: <https://owasp.org/www-project-application-security-verification-standard/> (дата звернення: 02.04.2025).
5. Selenium Documentation. Waits. Selenium Project. URL: <https://www.selenium.dev/documentation/webdriver/waits/> (дата звернення: 08.04.2025).
6. Cypress. Official site. Cypress, Inc. URL: <https://www.cypress.io/> (дата звернення: 17.04.2025).
7. Playwright. Fast and reliable end-to-end testing. Microsoft. URL: <https://playwright.dev/> (дата звернення: 24.04.2025).
8. Playwright Documentation. Writing tests. Microsoft. URL: <https://playwright.dev/docs/writing-tests> (дата звернення: 10.04.2025).
9. Playwright Documentation. Test configuration. Microsoft. URL: <https://playwright.dev/docs/test-configuration> (дата звернення: 28.04.2025).

10. Selenium. Official site. Selenium Project. URL: <https://www.selenium.dev/> (дата звернення: 04.04.2025).
11. Cypress Documentation. Why Cypress. Cypress, Inc. URL: <https://docs.cypress.io/app/get-started/why-cypress> (дата звернення: 15.04.2025).
12. GitHub Docs. Quickstart for GitHub Actions. GitHub, Inc. URL: <https://docs.github.com/en/actions/quickstart> (дата звернення: 19.04.2025).
13. Playwright Documentation. Page object models. Microsoft. URL: <https://playwright.dev/docs/pom> (дата звернення: 26.04.2025).
14. Playwright Documentation. Test fixtures. Microsoft. URL: <https://playwright.dev/docs/test-fixtures> (дата звернення: 01.04.2025).
15. Documentation — TypeScript. Handbook. TypeScript contributors. URL: <https://www.typescriptlang.org/docs/> (дата звернення: 23.04.2025).
16. dotenv. npm package page. npm, Inc. URL: <https://www.npmjs.com/package/dotenv> (дата звернення: 29.04.2025).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ



Розробка фреймворку автоматизованого тестування для web-застосунку AgroManagement

Виконав студент 4 курсу
групи ПД-41
Арабаджи Павло Олександрович
Керівник роботи
викладач кафедри ІПЗ Бажан Юрій Павлович

Київ – 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: автоматизація тестування основних функціональних сценаріїв вебзастосунку AgroManagement для зменшення кількості помилок та часу їх виявлення.

Об'єкт дослідження: процес автоматизації тестування вебзастосунку AgroManagement.

Предмет дослідження: методи та засоби автоматизації тестування.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Дослідити предметну область та функціональні можливості вебзастосунку AgroManagement з метою визначення критичних зон перевірок і обґрунтування вимог до автоматизації тестування.
2. Провести порівняльний аналіз існуючих інструментів автоматизованого тестування вебзастосунків для обрання найбільш доцільного інструмента реалізації фреймворку.
3. Визначити функціональні та нефункціональні вимоги до тестового фреймворку задля забезпечення повноти та відтворюваності автоматизованих перевірок.
4. Визначити архітектуру фреймворку автоматизованого тестування та модель організації тестового коду для підвищення підтримуваності й масштабованості набору тестів.
5. Реалізувати фреймворк автоматизованого тестування для перевірки основних функціональних сценаріїв вебзастосунку AgroManagement.
6. Провести запуски автоматизованих тестів та оцінити результати його застосування з метою підтвердження практичної доцільності автоматизації та ефективності запропонованого рішення.

3

АНАЛІЗ АНАЛОГІВ

Критерій	Selenium	Cypress	Playwright
Швидкість виконання тестів	середня	середня	висока
Підтримка браузерів	Chrome, Firefox, Edge	Chrome-базовані браузери	Chrome, Firefox, Edge, Safari
Підтримка API тестування	обмежена	часткова	повна
Паралельний запуск тестів	складний	обмежений	вбудований
Складність налаштування	висока	середня	низька
Стабільність тестів	середня	висока	висока
Підтримка CI/CD	підтримується	підтримується	підтримується
Робота з асинхронністю	складна	автоматична	автоматична

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1. Автоматизована перевірка авторизації користувачів із використанням валідних та невалідних даних.
2. Автоматизована перевірка CRUD-операцій для сутностей Farm, Field, Crop, Planting та Harvest.
3. Виконання UI тестів для перевірки коректності роботи інтерфейсу користувача.
4. Виконання API тестів для перевірки бізнес-логіки системи.
5. Формування звітів про результати виконання тестів із відображенням статусу кожного тесту.
6. Логування результатів виконання тестів із фіксацією помилок.

Нефункціональні вимоги:

1. Час виконання повного набору автоматизованих тестів не перевищує 5 хвилин.
2. Модульна архітектура фреймворку для спрощення підтримки та розширення покриття тестів.
3. Стабільність виконання тестів забезпечується отриманням однакових результатів при не менше ніж 3 повторних запусках.
4. Підтримка інтеграції фреймворку з CI/CD системою (GitHub Actions) для автоматизації запуску тестів.

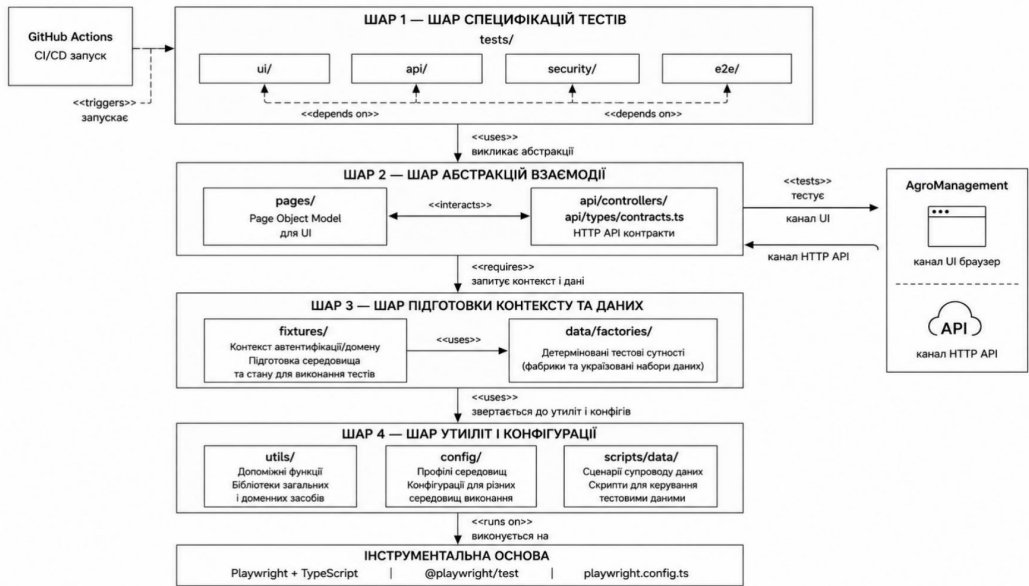
5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

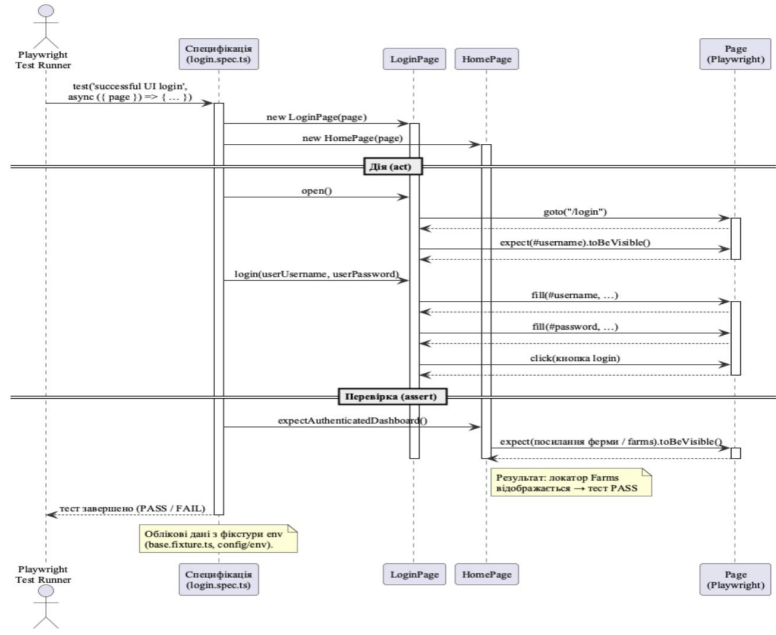


6

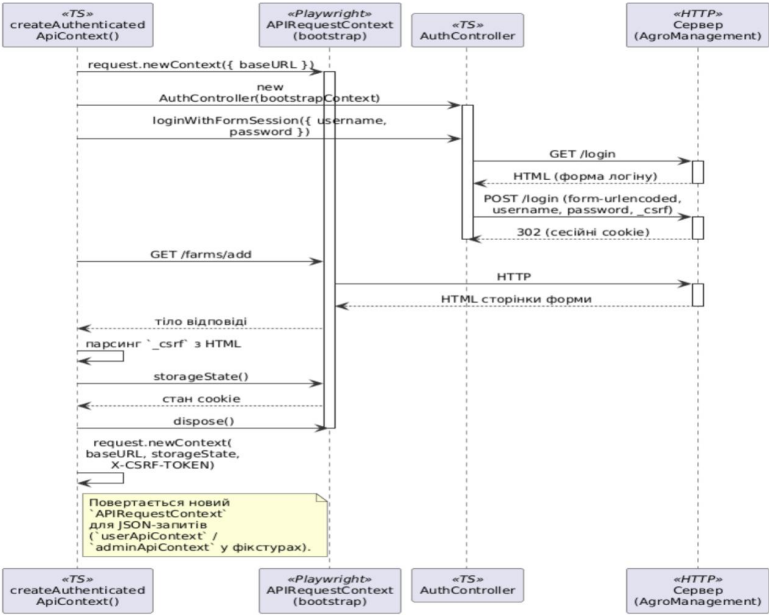
БАГАТОШАРОВА АРХІТЕКТУРА ФРЕЙМВОРКУ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ



ДІАГРАМА ПОСЛІДОВНОСТІ UI-ТЕСТУ АВТОРИЗАЦІЇ



ДІАГРАМА ПОСЛІДОВНОСТЕЙ ПІДГОТОВКИ АРІ-КОНТЕКСТУ



ЕКРАННІ ФОРМИ

Search tests

All 96

Passed 87

Failed 8

Flaky 1

Skipped 0

Metadata

5/12/2026, 5:33:45 AM Total time: 42.3s

api/validation/validation.spec.ts

API validation suite > @api @regression should reject invalid farm payloads

api

api

regression

343ms

api/validation/validation.spec.ts:4

View Trace

API validation suite > @api @regression should reject invalid field payloads

api

api

regression

347ms

api/validation/validation.spec.ts:25

View Trace

API validation suite > @api @regression should reject invalid crop payloads

api

api

regression

394ms

api/validation/validation.spec.ts:54

View Trace

API validation suite > @api @regression should reject invalid planting payloads

api

api

regression

404ms

api/validation/validation.spec.ts:83

View Trace

API validation suite > @api @regression should reject invalid harvest payloads

api

api

regression

175ms

api/validation/validation.spec.ts:109

security/ownership/create-update-ownership.security.spec.ts

Create/update ownership matrix > @api @security @ownership should deny creating field with foreign farm link

api

security

ownership

624ms

security/ownership/create-update-ownership.security.spec.ts:13

View Trace

Create/update ownership matrix > @api @security @ownership should deny creating planting with foreign field and crop links

api

api

security

ownership

764ms

security/ownership/create-update-ownership.security.spec.ts:32

View Trace

Create/update ownership matrix > @api @security @ownership should deny creating crop with foreign owner overposting

api

api

security

ownership

418ms

security/ownership/create-update-ownership.security.spec.ts:58

View Trace

HTML-звіт Playwright: підсумок прогону та навігація

ЕКРАННІ ФОРМИ

API validation suite

@api @regression should reject invalid farm payloads

api/validation/validation.spec.ts:4

View Trace | 343ms

Run 115ms

Retry #1 228ms

Errors

Error: expect(received).toBe(expected) // Object.is equality
Expected: 400
Received: 201
12 |
13 | // assert
> 14 | expect(emptyName.status()).toBe(400);
15 |
16 | // act
17 | const emptyLocation = await userApiContext.post("/api/farms", {
at /home/runner/work/test-automation-framework-for-agromanagement/test-automation-framework-for-agromanage

Test Steps

Filter steps

Before Hooks

POST "/api/farms" — api/validation/validation.spec.ts:9

Expect "toBe" — api/validation/validation.spec.ts:14

HTML-звіт Playwright: деталізація невдалого тесту

ЕКРАННІ ФОРМИ

Playwright

ui/auth/login.spec.ts:6 • Auth UI • @smoke @ui should login with valid credentials

Timeline

Actions Metadata

Filter actions

Before Hooks

Navigate to "/login"

Expect "toBeVisible"

Fill "qa_user"

Fill "qa_password_123"

Click

Expect "toBeVisible"

After Hooks

Locator

Call

Log

Errors

Console

Network

Source

Attachments

Filter network

All

Fetch

HTML

JS

CSS

Font

Image

Name	Method	Status	Content Type	Duration	Size	Start	Route
login	GET	200	text/html	13ms	1.9K	188ms	
bootstrap.min.css	GET	200	text/css	47ms	33.2K	205ms	
bootstrap.bundle.min.js	GET	200	application/javascript	21ms	23.8K	205ms	
login	POST	302	x-unknown	103ms	-	542ms	
127.0.0.1:8080	GET	200	text/html	13ms	6.0K	631ms	

Playwright Trace Viewer: покроковий timeline, дії, мережа, скрінні кроків

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Арабаджи П.О., Бажан Ю.П. Аналіз застосування автоматизованого тестування для інформаційної системи AgroManagement. Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, ДУІКТ. Збірник тез. - К.: ДУІКТ, 2025, С.200-202.

2. Арабаджи П.О., Бажан Ю.П. Аналіз критеріїв вибору інструменту автоматизованого тестування для інформаційної системи AgroManagement. Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, ДУІКТ. Збірник тез. - К.: ДУІКТ, 2026, С.334-335.

13

ВИСНОВКИ

1. Досліджено предметну область вебзастосунку AgroManagement та визначено п'ять критичних доменних сутностей для автоматизованої перевірки: Farm, Field, Crop, Planting, Harvest.
2. Проведено порівняльний аналіз інструментів Selenium, Cypress, Playwright за 8 критеріями (швидкість, кросбраузерність, підтримка API, паралелізм, складність налаштування, стабільність, CI/CD, асинхронність) - на основі чого обрано Playwright як найбільш доцільний.
3. На основі аналізу AgroManagement визначено 6 функціональних і 4 нефункціональних вимог до фреймворку задля забезпечення повноти та відтворюваності автоматизованих перевірок.
4. Спроектовано багатoshарову архітектуру фреймворку та модель організації тестового коду з використанням патернів Page Object Model, API Controllers, Factory та AAA-pattern для підвищення підтримуваності й масштабованості набору тестів.
5. Реалізовано фреймворк автоматизованого тестування на стеку TypeScript + Playwright + Node.js, який інтегрований із системою GitHub Actions (CI/CD) для перевірки основних функціональних сценаріїв вебзастосунку AgroManagement.
6. Проведено запуски автоматизованих тестів та оцінено час виконання набору з 96 тестів: повний прогон у CI/CD триває близько 42 секунд, тоді як ручне тестування тих самих сценаріїв займає близько 5 годин, що забезпечує прискорення у понад 400 разів.

14

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

```
// ===== playwright.config.ts =====
import { defineConfig, devices } from "@playwright/test";
import { ENV } from "../config/env";

export default defineConfig({
  testDir: "./tests",
  fullyParallel: true,
  forbidOnly: !!process.env.CI,
  retries: process.env.CI ? 1 : 0,
  workers: process.env.CI ? 2 : undefined,
  reporter: [
    ["html", { open: "never" }],
    ["list"],
  ],
  use: {
    baseURL: ENV.baseUrl,
    trace: "on-first-retry",
    screenshot: "only-on-failure",
    video: "retain-on-failure",
  },
  projects: [
    {
      name: "ui",
      use: {
        ...devices["Desktop Chrome"],
      },
      grepInvert: /@api/,
    },
    {
      name: "api",
      use: {
        baseURL: ENV.apiUrl,
      },
      grep: /@api/,
    },
  ],
});

// ===== package.json =====
{
  "name": "test-automation-framework",
  "version": "1.0.0",
  "private": true,
  "description": "Playwright + TypeScript test automation framework for AgroManagement",
  "scripts": {
    "test": "playwright test",
    "test:clean": "playwright test; EXIT_CODE=$?; npm run cleanup:data; exit $EXIT_CODE",
    "test:ui": "playwright test --project=ui",
    "test:api": "playwright test --project=api",
    "test:smoke": "playwright test --grep @smoke",
    "test:security": "playwright test --grep @security",
    "test:headed": "playwright test --headed",
    "test:debug": "playwright test --debug",
    "cleanup:data": "ts-node scripts/data/cleanupTestData.ts",
    "report:open": "playwright show-report",
    "typecheck": "tsc --noEmit"
  },
  "dependencies": {
    "dotenv": "^16.4.5"
  },
  "devDependencies": {
    "@playwright/test": "^1.54.2",
    "@types/node": "^22.8.7",
    "ts-node": "^10.9.2",

```

```
    "typescript": "^5.6.3"
  }
}

// ===== config/env.ts =====
import dotenv from "dotenv";
import { devEnvDefaults } from "../env/dev.env";
import { stageEnvDefaults } from "../env/stage.env";

dotenv.config();

export type EnvironmentName = "dev" | "stage";

export interface FrameworkEnv {
  testEnv: EnvironmentName;
  baseUrl: string;
  apiBaseUrl: string;
  testDataPrefix: string;
  userUsername: string;
  userPassword: string;
  adminUsername: string;
  adminPassword: string;
}

const TEST_ENV = (process.env.TEST_ENV ?? "dev") as EnvironmentName;
const envDefaults = TEST_ENV === "stage" ? stageEnvDefaults : devEnvDefaults;

export const ENV: FrameworkEnv = {
  testEnv: TEST_ENV,
  baseUrl: process.env.BASE_URL ?? envDefaults.baseUrl,
  apiBaseUrl: process.env.API_BASE_URL ?? process.env.BASE_URL ?? envDefaults.apiBaseUrl,
  testDataPrefix: process.env.TEST_DATA_PREFIX ?? "qaauto",
  userUsername: process.env.USER_USERNAME ?? "",
  userPassword: process.env.USER_PASSWORD ?? "",
  adminUsername: process.env.ADMIN_USERNAME ?? "",
  adminPassword: process.env.ADMIN_PASSWORD ?? ""
};

export function assertRequiredEnv(): void {
  const required = [
    { key: "USER_USERNAME", value: ENV.userUsername },
    { key: "USER_PASSWORD", value: ENV.userPassword },
    { key: "ADMIN_USERNAME", value: ENV.adminUsername },
    { key: "ADMIN_PASSWORD", value: ENV.adminPassword }
  ];

  const missing = required.filter((item) => !item.value).map((item) => item.key);
  if (missing.length > 0) {
    throw new Error(`Missing required env vars: ${missing.join(", ")}`);
  }
}
```

```
// ===== config/env/dev.env.ts =====
import { FrameworkEnv } from "../env";

export const devEnvDefaults: Pick<FrameworkEnv,
"baseUrl" | "apiBaseUrl"> = {
  baseUrl: "http://localhost:8080",
  apiBaseUrl: "http://localhost:8080"
};

// ===== api/types/contracts.ts =====
export interface ApiResponseMessage {
  success: boolean;
  message: string;
}

export interface LoginRequest {
  username: string;
  password: string;
}

export interface Farm {
  id?: number;
  name: string;
  location: string;
}

export interface Field {
  id?: number;
  name: string;
  area: number;
  farm: { id: number };
}

export interface Crop {
  id?: number;
  name: string;
  expectedYield: number;
}

export interface Planting {
  id?: number;
  plantingDate: string;
  expectedHarvestDate: string;
  field: { id: number };
  crop: { id: number };
}

export interface Harvest {
  id?: number;
  planting: { id: number };
  harvestDate: string;
  yieldAmount: number;
}

// ===== api/controllers/BaseApiController.ts =====
import { APIRequestContext, expect } from
"@playwright/test";

export class BaseApiController {
  constructor(protected readonly request: APIRequestContext) {}

  protected async expectStatus(response: { status(): number },
expectedStatus: number): Promise<void> {
    expect(response.status(), "Unexpected status
code").toBe(expectedStatus);
  }
}
```

```
// ===== api/controllers/AuthController.ts =====
import { APIRequestContext } from "@playwright/test";
import { BaseApiController } from "../BaseApiController";
import { ApiResponseMessage, LoginRequest } from
"../types/contracts";

export class AuthController extends BaseApiController {
  constructor(request: APIRequestContext) {
    super(request);
  }

  async loginViaApi(payload: LoginRequest):
Promise<ApiResponseMessage> {
    const response = await this.request.post("/api/auth/login",
{
      data: payload
    });
    await this.expectStatus(response, 200);
    return response.json();
  }

  async loginWithFormSession(payload: LoginRequest):
Promise<void> {
    const loginPage = await this.request.get("/login");
    await this.expectStatus(loginPage, 200);
    const html = await loginPage.text();
    const tokenMatch =
html.match(/name="_csrf" s+value="([^\"]+)"/);
    if (!tokenMatch) {
      throw new Error("CSRF token was not found on /login");
    }
    const csrfToken = tokenMatch[1];

    const response = await this.request.post("/login", {
      headers: {
        "Content-Type": "application/x-www-form-urlencoded"
      },
      maxRedirects: 0,
      form: {
        username: payload.username,
        password: payload.password,
        _csrf: csrfToken
      }
    });

    if (response.status() !== 302) {
      throw new Error("Expected 302 after form login, got
${response.status()}");
    }
    const location = response.headers()["location"] ?? "";
    if (location.includes("/login")) {
      throw new Error("Form login failed for user
'${payload.username}', redirected to ${location}");
    }
  }
}

// ===== api/controllers/FarmController.ts =====
import { APIRequestContext } from "@playwright/test";
import { BaseApiController } from "../BaseApiController";
import { Farm } from "../types/contracts";

export class FarmController extends BaseApiController {
  constructor(request: APIRequestContext) {
    super(request);
  }

  async createFarm(payload: Farm): Promise<Farm> {
    const response = await this.request.post("/api/farms",
```

```

{ data: payload });
  await this.expectStatus(response, 201);
  return this.toFarm(await response.json(), payload);
}

async getFarmById(id: number): Promise<{ status: number;
body?: Farm }> {
  const response = await this.request.get(`/api/farms/${id}`);
  if (response.status() === 200) {
    return { status: 200, body: this.toFarm(await
response.json(), { id, name: "", location: "" }) };
  }
  return { status: response.status() };
}

async listFarms(): Promise<Farm[]> {
  const response = await this.request.get("/api/farms");
  await this.expectStatus(response, 200);
  const body = await response.json();
  if (!Array.isArray(body)) return [];
  return body.map((item) => this.toFarm(item, { name: "",
location: "" }));
}

async updateFarm(id: number, payload: Farm):
Promise<Farm> {
  const response = await this.request.put(`/api/farms/${id}`,
{ data: payload });
  await this.expectStatus(response, 200);
  return this.toFarm(await response.json(), { ...payload, id });
}

async deleteFarm(id: number): Promise<void> {
  const response = await
this.request.delete(`/api/farms/${id}`);
  await this.expectStatus(response, 204);
}

private toFarm(raw: unknown, fallback: Farm): Farm {
  const data = (raw ?? {}) as Record<string, unknown>;
  return {
    id: typeof data.id === "number" ? data.id : fallback.id,
    name: typeof data.name === "string" ? data.name :
fallback.name,
    location: typeof data.location === "string" ?
data.location : fallback.location
  };
}
}

```

```

// ===== api/controllers/FieldController.ts =====
import { APIRequestContext } from "@playwright/test";
import { BaseApiController } from "../BaseApiController";
import { Field } from "../types/contracts";

```

```

export class FieldController extends BaseApiController {
  constructor(request: APIRequestContext) {
    super(request);
  }

```

```

  async createField(payload: Field): Promise<Field> {
    const response = await this.request.post("/api/fields",
{ data: payload });
    await this.expectStatus(response, 201);
    return this.toField(await response.json(), payload);
  }

```

```

  async listFields(): Promise<Field[]> {
    const response = await this.request.get("/api/fields");
  }

```

```

    await this.expectStatus(response, 200);
    const body = await response.json();
    if (!Array.isArray(body)) return [];
    return body.map((item) => this.toField(item, { name: "",
area: 0, farm: { id: 0 } }));
  }

```

```

  async getFieldById(id: number): Promise<{ status: number;
body?: Field }> {
    const response = await this.request.get(`/api/fields/${id}`);
    if (response.status() !== 200) {
      return { status: response.status() };
    }
    return { status: 200, body: this.toField(await
response.json(), { id, name: "", area: 0, farm: { id: 0 } }) };
  }

```

```

  async updateField(id: number, payload: Field):
Promise<Field> {
    const response = await this.request.put(`/api/fields/${id}`,
{ data: payload });
    await this.expectStatus(response, 200);
    return this.toField(await response.json(), { ...payload, id });
  }

```

```

  async deleteField(id: number): Promise<void> {
    const response = await
this.request.delete(`/api/fields/${id}`);
    await this.expectStatus(response, 204);
  }

```

```

  private toField(raw: unknown, fallback: Field): Field {
    const data = (raw ?? {}) as Record<string, unknown>;
    const farm = (data.farm ?? {}) as Record<string,
unknown>;
    return {
      id: typeof data.id === "number" ? data.id : fallback.id,
      name: typeof data.name === "string" ? data.name :
fallback.name,
      area: typeof data.area === "number" ? data.area :
fallback.area,
      farm: {
        id: typeof farm.id === "number" ? farm.id :
fallback.farm.id
      }
    };
  }
}

```

```

// ===== api/controllers/CropController.ts =====
import { APIRequestContext } from "@playwright/test";
import { BaseApiController } from "../BaseApiController";
import { Crop } from "../types/contracts";

```

```

export class CropController extends BaseApiController {
  constructor(request: APIRequestContext) {
    super(request);
  }

```

```

  async createCrop(payload: Crop): Promise<Crop> {
    const response = await this.request.post("/api/crops",
{ data: payload });
    await this.expectStatus(response, 201);
    return this.toCrop(await response.json(), payload);
  }

```

```

  async getCropById(id: number): Promise<{ status: number;
body?: Crop }> {
    const response = await this.request.get(`/api/crops/${id}`);
  }

```

```

    if (response.status() !== 200) return { status:
response.status() };
    return { status: 200, body: this.toCrop(await
response.json(), { id, name: "", expectedYield: 0 }) };
  }

  async updateCrop(id: number, payload: Crop):
Promise<Crop> {
    const response = await this.request.put(`/api/crops/${id}`,
{ data: payload });
    await this.expectStatus(response, 200);
    return this.toCrop(await response.json(), { ...payload, id });
  }

  async deleteCrop(id: number): Promise<void> {
    const response = await
this.request.delete(`/api/crops/${id}`);
    await this.expectStatus(response, 204);
  }

  async listCrops(): Promise<Crop[]> {
    const response = await this.request.get("/api/crops");
    await this.expectStatus(response, 200);
    const body = await response.json();
    if (!Array.isArray(body)) return [];
    return body.map((item) => this.toCrop(item, { name: "",
expectedYield: 0 }));
  }

  private toCrop(raw: unknown, fallback: Crop): Crop {
    const data = (raw ?? {}) as Record<string, unknown>;
    return {
      id: typeof data.id === "number" ? data.id : fallback.id,
      name: typeof data.name === "string" ? data.name :
fallback.name,
      expectedYield: typeof data.expectedYield === "number" ?
data.expectedYield : fallback.expectedYield
    };
  }
}

```

// ===== fixtures/base.fixture.ts =====

```

import { test as base } from "@playwright/test";
import { assertRequiredEnv, ENV } from "../config/env";

```

```

type BaseFixture = {
  env: typeof ENV;
};

```

```

export const test = base.extend<BaseFixture>({
  env: async ({}, use) => {
    assertRequiredEnv();
    await use(ENV);
  }
});

```

```

export { expect } from "@playwright/test";

```

// ===== fixtures/auth.fixture.ts =====

```

import { APIRequestContext, request } from
"@playwright/test";
import { AuthController } from
"./api/controllers/AuthController";
import { test as base } from "../base.fixture";

```

```

type AuthFixtures = {
  userApiContext: APIRequestContext;
  adminApiContext: APIRequestContext;
};

```

```

async function createAuthenticatedApiContext(
  baseUrl: string,
  username: string,
  password: string
): Promise<APIRequestContext> {
  const bootstrapContext = await
request.newContext({ baseUrl });
  const authController = new
AuthController(bootstrapContext);
  await authController.loginWithFormSession({ username,
password });

```

// Extract CSRF token from a Thymeleaf form page after authentication.

```

const formPageResponse = await
bootstrapContext.get("/farms/add");
if (formPageResponse.status() !== 200) {
  throw new Error("Unable to open CSRF source page,
status=${formPageResponse.status()}");
}
const html = await formPageResponse.text();
const tokenMatch =
html.match(/name="_csrf" s+value="([^\"]+)"/);
if (!tokenMatch) {
  throw new Error("CSRF token was not found in /farms/add
page");
}
const csrfToken = tokenMatch[1];

```

```

const state = await bootstrapContext.storageState();
await bootstrapContext.dispose();

```

```

return request.newContext({
  baseUrl,
  storageState: state,
  extraHTTPHeaders: {
    "X-CSRF-TOKEN": csrfToken
  }
});
}

```

```

export const test = base.extend<AuthFixtures>({
  userApiContext: async ({ env }, use) => {
    const context = await
createAuthenticatedApiContext(env.apiUrl,
env.userUsername, env.userPassword);
    await use(context);
    await context.dispose();
  },
  adminApiContext: async ({ env }, use) => {
    const context = await
createAuthenticatedApiContext(env.apiUrl,
env.adminUsername, env.adminPassword);
    await use(context);
    await context.dispose();
  }
});

```

```

export { expect } from "../base.fixture";

```

// ===== fixtures/domain.fixture.ts =====

```

import { CropController } from
"./api/controllers/CropController";
import { FieldController } from
"./api/controllers/FieldController";
import { FarmController } from
"./api/controllers/FarmController";
import { HarvestController } from

```

```

"/api/controllers/HarvestController";
import { PlantingController } from
"/api/controllers/PlantingController";
import { CropFactory } from "../data/factories/CropFactory";
import { FieldFactory } from "../data/factories/FieldFactory";
import { FarmFactory } from "../data/factories/FarmFactory";
import { HarvestFactory } from
"/data/factories/HarvestFactory";
import { PlantingFactory } from
"/data/factories/PlantingFactory";
import { randomSuffix } from "../utils/data/random";
import { TestDataRegistry } from
"/utils/data/TestDataRegistry";
import { test as base } from "../auth.fixture";

type DomainFixtures = {
  farmController: FarmController;
  fieldController: FieldController;
  cropController: CropController;
  plantingController: PlantingController;
  harvestController: HarvestController;
  dataRegistry: TestDataRegistry;
};

export const test = base.extend<DomainFixtures>({
  farmController: async ({ userApiContext }, use) => {
    await use(new FarmController(userApiContext));
  },
  fieldController: async ({ userApiContext }, use) => {
    await use(new FieldController(userApiContext));
  },
  cropController: async ({ userApiContext }, use) => {
    await use(new CropController(userApiContext));
  },
  plantingController: async ({ userApiContext }, use) => {
    await use(new PlantingController(userApiContext));
  },
  harvestController: async ({ userApiContext }, use) => {
    await use(new HarvestController(userApiContext));
  },
  dataRegistry: async ({ farmController, fieldController,
    cropController, plantingController, harvestController }, use)
=> {
    const registry = new TestDataRegistry();
    await use(registry);

    for (const harvestId of registry.consumeHarvestIds()) {
      try {
        await harvestController.deleteHarvest(harvestId);
      } catch {
        // best-effort cleanup
      }
    }

    for (const plantingId of registry.consumePlantingIds()) {
      try {
        await plantingController.deletePlanting(plantingId);
      } catch {
        // best-effort cleanup
      }
    }

    for (const fieldId of registry.consumeFieldIds()) {
      try {
        await fieldController.deleteField(fieldId);
      } catch {
        // best-effort cleanup
      }
    }
  }
});

```

```

for (const cropId of registry.consumeCropIds()) {
  try {
    await cropController.deleteCrop(cropId);
  } catch {
    // best-effort cleanup
  }
}

for (const farmId of registry.consumeFarmIds()) {
  try {
    await farmController.deleteFarm(farmId);
  } catch {
    // best-effort cleanup
  }
}
});

export async function setupFarmAndField(
  farmController: FarmController,
  fieldController: FieldController,
  dataRegistry: TestDataRegistry
): Promise<{ farmId: number; fieldId: number; farmName:
string; fieldName: string }> {
  const farmPayload = FarmFactory.valid();
  const farm = await farmController.createFarm(farmPayload);
  if (!farm.id) throw new Error("Farm ID was not returned");
  dataRegistry.registerFarmId(farm.id);

  const fieldName = randomSuffix("field");
  const field = await fieldController.createField({
    name: fieldName,
    area: 15.25,
    farm: { id: farm.id }
  });
  if (!field.id) throw new Error("Field ID was not returned");
  dataRegistry.registerFieldId(field.id);

  return {
    farmId: farm.id,
    fieldId: field.id,
    farmName: farm.name,
    fieldName
  };
}

export async function setupFieldAndCrop(
  farmController: FarmController,
  fieldController: FieldController,
  cropController: CropController,
  dataRegistry: TestDataRegistry
): Promise<{ fieldId: number; cropId: number; fieldName:
string; cropName: string }> {
  const farm = await
  farmController.createFarm(FarmFactory.valid());
  if (!farm.id) throw new Error("Farm ID was not returned");
  dataRegistry.registerFarmId(farm.id);

  const field = await
  fieldController.createField(FieldFactory.valid(farm.id));
  if (!field.id) throw new Error("Field ID was not returned");
  dataRegistry.registerFieldId(field.id);

  const crop = await
  cropController.createCrop(CropFactory.valid());
  if (!crop.id) throw new Error("Crop ID was not returned");
  dataRegistry.registerCropId(crop.id);

  return { fieldId: field.id, cropId: crop.id, fieldName:
    field.name, cropName: crop.name };
}

```

```

    }

    export async function setupPlanting(
      farmController: FarmController,
      fieldController: FieldController,
      cropController: CropController,
      plantingController: PlantingController,
      dataRegistry: TestDataRegistry
    ): Promise<{ plantingId: number }> {
      const { fieldId, cropId } = await
        setupFieldAndCrop(farmController, fieldController,
          cropController, dataRegistry);
      const planting = await
        plantingController.createPlanting(PlantingFactory.valid(field
          Id, cropId));
      if (!planting.id) throw new Error("Planting ID was not
        returned");
      dataRegistry.registerPlantingId(planting.id);

      return { plantingId: planting.id };
    }

    export async function setupHarvest(
      farmController: FarmController,
      fieldController: FieldController,
      cropController: CropController,
      plantingController: PlantingController,
      harvestController: HarvestController,
      dataRegistry: TestDataRegistry
    ): Promise<{ plantingId: number; harvestId: number }> {
      const { plantingId } = await setupPlanting(farmController,
        fieldController, cropController, plantingController,
        dataRegistry);
      const harvest = await
        harvestController.createHarvest(HarvestFactory.valid(plantin
          gId));
      if (!harvest.id) throw new Error("Harvest ID was not
        returned");
      dataRegistry.registerHarvestId(harvest.id);

      return { plantingId, harvestId: harvest.id };
    }

    export { expect } from "../auth.fixture";

    // ===== pages/auth/LoginPage.ts =====
    import { expect, Locator, Page } from "@playwright/test";

    export class LoginPage {
      private readonly usernameInput: Locator;
      private readonly passwordInput: Locator;
      private readonly submitButton: Locator;
      private readonly errorAlert: Locator;

      constructor(private readonly page: Page) {
        this.usernameInput = page.locator("#username");
        this.passwordInput = page.locator("#password");
        this.submitButton = page.getByRole("button", { name:
          /login/i });
        this.errorAlert = page.locator(".alert-danger");
      }

      async open(): Promise<void> {
        await this.page.goto("/login");
        await expect(this.usernameInput).toBeVisible();
      }

      async login(username: string, password: string):
        Promise<void> {
          await this.usernameInput.fill(username);
          await this.passwordInput.fill(password);
          await this.submitButton.click();
        }

      async expectInvalidLoginMessage(): Promise<void> {
        await expect(this.errorAlert).toBeVisible();
      }
    }

    // ===== pages/home/HomePage.ts =====
    import { expect, Locator, Page } from "@playwright/test";

    export class HomePage {
      private readonly farmsCard: Locator;

      constructor(private readonly page: Page) {
        this.farmsCard = page.getByRole("link", { name:
          /ферми|farms/i });
      }

      async open(): Promise<void> {
        await this.page.goto("/");
      }

      async expectAuthenticatedDashboard(): Promise<void> {
        await expect(this.farmsCard).toBeVisible();
      }
    }

    // ===== pages/farm/FarmListPage.ts =====
    import { expect, Locator, Page } from "@playwright/test";

    export class FarmListPage {
      private readonly addFarmButton: Locator;

      constructor(private readonly page: Page) {
        this.addFarmButton = page.getByRole("link", { name:
          /додати ферму|add farm/i });
      }

      async open(): Promise<void> {
        await this.page.goto("/farms");
        await expect(this.addFarmButton).toBeVisible();
      }

      async clickAddFarm(): Promise<void> {
        await this.addFarmButton.click();
      }

      async openEditByFarmName(name: string): Promise<void> {
        const row = this.page.locator("tr", { hasText: name });
        await expect(row).toBeVisible();
        const byHref = row.locator('a[href*="/edit/"]');
        if (await byHref.count()) {
          await byHref.click();
          return;
        }
        await row.getByRole("link", { name:
          /редагувати|edit/i }).click();
      }

      async deleteByFarmName(name: string): Promise<void> {
        const row = this.page.locator("tr", { hasText: name });
        await expect(row).toBeVisible();
        this.page.once("dialog", (dialog) => dialog.accept());
        const byHref = row.locator('a[href*="/delete/"]');
        if (await byHref.count()) {

```

```

        await byHref.click();
        return;
    }
    await row.getByRole("link", { name:
/видалити|delete/i }).click();
}

async expectFarmVisible(name: string): Promise<void> {
    await expect(this.page.locator("tr", { hasText:
name })).toBeVisible();
}

async expectFarmHidden(name: string): Promise<void> {
    await expect(this.page.locator("tr", { hasText:
name })).toHaveCount(0);
}

async getFarmIdByName(name: string): Promise<number |
undefined> {
    const row = this.page.locator("tr", { hasText: name });
    await expect(row).toBeVisible();
    let href = await
row.locator('a[href*="/edit/"]').getAttribute("href");
    if (!href) {
        href = await row.getByRole("link", { name:
/редагувати|edit/i }).getAttribute("href");
    }
    if (!href) return undefined;
    const idMatch = href.match(/\/farms\/edit\/(\d+)/);
    return idMatch ? Number(idMatch[1]) : undefined;
}

// ===== pages/farm/FarmFormPage.ts =====
import { expect, Locator, Page } from "@playwright/test";

export class FarmFormPage {
    private readonly nameInput: Locator;
    private readonly locationInput: Locator;
    private readonly saveButton: Locator;

    constructor(private readonly page: Page) {
        this.nameInput = page.locator('input[name="name"]');
        this.locationInput = page.locator('input[name="location"]');
        this.saveButton = page.getByRole("button", { name:
/зберегти|save/i });
    }

    async expectFormVisible(): Promise<void> {
        await expect(this.nameInput).toBeVisible();
        await expect(this.locationInput).toBeVisible();
    }

    async submit(name: string, location: string): Promise<void>
    {
        await this.nameInput.fill(name);
        await this.locationInput.fill(location);
        await this.saveButton.click();
    }
}

// ===== pages/field/FieldListPage.ts =====
import { expect, Locator, Page } from "@playwright/test";

export class FieldListPage {
    private readonly addFieldButton: Locator;

    constructor(private readonly page: Page) {
        this.addFieldButton = page.getByRole("link", { name:

```

```

/додати поле|add field/i });
    }

    async open(): Promise<void> {
        await this.page.goto("/fields");
        await expect(this.addFieldButton).toBeVisible();
    }

    async clickAddField(): Promise<void> {
        await this.addFieldButton.click();
    }

    async deleteByFieldName(name: string): Promise<void> {
        const row = this.page.locator("tr", { hasText: name });
        await expect(row).toBeVisible();
        this.page.once("dialog", (dialog) => dialog.accept());
        const byHref = row.locator('a[href*="/delete/"]');
        if (await byHref.count()) {
            await byHref.click();
            return;
        }
        await row.getByRole("link", { name:
/видалити|delete/i }).click();
    }

    async expectFieldVisible(name: string): Promise<void> {
        await expect(this.page.locator("tr", { hasText:
name })).toBeVisible();
    }

    async expectFieldHidden(name: string): Promise<void> {
        await expect(this.page.locator("tr", { hasText:
name })).toHaveCount(0);
    }

    async getFieldIdByName(name: string): Promise<number |
undefined> {
        const row = this.page.locator("tr", { hasText: name });
        await expect(row).toBeVisible();
        let href = await
row.locator('a[href*="/edit/"]').getAttribute("href");
        if (!href) {
            href = await row.getByRole("link", { name:
/редагувати|edit/i }).getAttribute("href");
        }
        if (!href) return undefined;
        const idMatch = href.match(/\/fields\/edit\/(\d+)/);
        return idMatch ? Number(idMatch[1]) : undefined;
    }
}

// ===== pages/field/FieldFormPage.ts =====
import { Locator, Page } from "@playwright/test";

export class FieldFormPage {
    private readonly nameInput: Locator;
    private readonly areaInput: Locator;
    private readonly farmSelect: Locator;
    private readonly saveButton: Locator;

    constructor(private readonly page: Page) {
        this.nameInput = page.locator('input[name="name"]');
        this.areaInput = page.locator('input[name="area"]');
        this.farmSelect = page.locator('select[name="farm.id"]');
        this.saveButton = page.getByRole("button", { name:
/зберегти|save/i });
    }

    async open(): Promise<void> {

```

```

    await this.page.goto("/fields/add");
  }

  async expectFormVisible(): Promise<void> {
    await this.nameInput.waitFor({ state: "visible" });
  }

  async submit(name: string, area: number, farmId: number):
  Promise<void> {
    await this.nameInput.fill(name);
    await this.areaInput.fill(area.toString());
    await this.farmSelect.selectOption({ value:
  farmId.toString() });
    await this.saveButton.click();
  }
}

```

// ===== utils/auth/uiAuth.ts =====

```

import { Page } from "@playwright/test";
import { LoginPage } from "../../pages/auth/LoginPage";

```

```

export async function loginViaUi(page: Page, username:
string, password: string): Promise<void> {
  const loginPage = new LoginPage(page);
  await loginPage.open();
  await loginPage.login(username, password);
}

```

// ===== utils/data/random.ts =====

```

import { ENV } from "../../config/env";

export function randomSuffix(prefix: string): string {
  return `${ENV.testDataPrefix}-${prefix}-${Date.now()}-
${Math.floor(Math.random() * 10000)}`;
}

```

// ===== utils/data/TestDataRegistry.ts =====

```

export class TestDataRegistry {
  private farmIds: number[] = [];
  private fieldIds: number[] = [];
  private cropIds: number[] = [];
  private plantingIds: number[] = [];
  private harvestIds: number[] = [];

  registerFarmId(id?: number): void {
    if (id) this.farmIds.push(id);
  }

  registerFieldId(id?: number): void {
    if (id) this.fieldIds.push(id);
  }

  registerCropId(id?: number): void {
    if (id) this.cropIds.push(id);
  }

  registerPlantingId(id?: number): void {
    if (id) this.plantingIds.push(id);
  }

  registerHarvestId(id?: number): void {
    if (id) this.harvestIds.push(id);
  }

  consumeFarmIds(): number[] {
    return [...this.farmIds].reverse();
  }
}

```

```

  consumeFieldIds(): number[] {
    return [...this.fieldIds].reverse();
  }

  consumeCropIds(): number[] {
    return [...this.cropIds].reverse();
  }

  consumePlantingIds(): number[] {
    return [...this.plantingIds].reverse();
  }

  consumeHarvestIds(): number[] {
    return [...this.harvestIds].reverse();
  }
}

```

// ===== utils/api/parseApiResponse.ts =====

```

import { ApiResponse } from "@playwright/test";

```

```

export async function parseJsonWithFallback(
  response: ApiResponse
): Promise<Record<string, unknown>> {
  try {
    return (await response.json()) as Record<string, unknown>;
  } catch (error) {
    throw new Error(`Expected valid JSON response body, but
  parsing failed: ${String(error)}`);
  }
}

```

// ===== utils/api/securityAssertions.ts =====

```

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

```

```

export function expectForbiddenOrNotFound(status: number):
void {
  expect([403, 404]).toContain(status);
}

```

// ===== data/factories/FarmFactory.ts =====

```

import { Farm } from "../../api/types/contracts";
import { randomSuffix } from "../../utils/data/random";

```

```

export class FarmFactory {
  static valid(overrides?: Partial<Farm>): Farm {
    return {
      name: randomSuffix("farm"),
      location: randomSuffix("location"),
      ...overrides
    };
  }
}

```

// ===== tests/ui/auth/login.spec.ts =====

```

import { LoginPage } from "../../pages/auth/LoginPage";
import { HomePage } from "../../pages/home/HomePage";
import { test, expect } from "../../fixtures/base.fixture";

```

```

test.describe("Auth UI", () => {
  test("@smoke @ui should login with valid credentials",
  async ({ page, env }) => {
    // arrange
    const loginPage = new LoginPage(page);
    const homepage = new HomePage(page);

    // act
    await loginPage.open();
    await loginPage.login(env.userName,

```

```

env.userPassword);

    // assert
    await homePage.expectAuthenticatedDashboard();
  });

  test("@smoke @ui should show error on invalid login",
  async ({ page, env }) => {
    // arrange
    const loginPage = new LoginPage(page);

    // act
    await loginPage.open();
    await loginPage.login(env.userName,
    `${env.userPassword}-wrong`);

    // assert
    await loginPage.expectInvalidLoginMessage();
    await expect(page).toHaveURL(/login/);
  });
});

// ===== tests/ui/farm/farm.spec.ts =====
import { test, expect } from "../../fixtures/domain.fixture";
import { FarmFactory } from
  "../../data/factories/FarmFactory";
import { FarmFormPage } from
  "../../pages/farm/FarmFormPage";
import { FarmListPage } from
  "../../pages/farm/FarmListPage";
import { loginViaUi } from "../../utils/auth/uiAuth";

test.describe("Farm UI", () => {
  test("@ui @regression should display farms list", async
  ({ page, env }) => {
    // arrange
    await loginViaUi(page, env.userName,
    env.userPassword);
    const listPage = new FarmListPage(page);

    // act
    await listPage.open();

    // assert
    await expect(page).toHaveURL(/farms/);
  });

  test("@ui @regression should create farm", async ({ page,
  env, dataRegistry }) => {
    // arrange
    await loginViaUi(page, env.userName,
    env.userPassword);

    const listPage = new FarmListPage(page);
    const formPage = new FarmFormPage(page);
    const createPayload = FarmFactory.valid();

    // act
    await listPage.open();
    await listPage.clickAddFarm();
    await formPage.expectFormVisible();
    await formPage.submit(createPayload.name,
    createPayload.location);

    // assert
    await listPage.expectFarmVisible(createPayload.name);

    const createdFarmId = await
    listPage.getFarmIdByName(createPayload.name);

```

```

    dataRegistry.registerFarmId(createdFarmId);
    expect(createdFarmId).toBeTruthy();
  });

  test("@ui @regression should update farm", async ({ page,
  env, farmController, dataRegistry }) => {
    // arrange
    await loginViaUi(page, env.userName,
    env.userPassword);

    const createPayload = FarmFactory.valid();
    const createdFarm = await
    farmController.createFarm(createPayload);
    dataRegistry.registerFarmId(createdFarm.id);
    const updatedName = `${createPayload.name}-updated`;

    const listPage = new FarmListPage(page);
    const formPage = new FarmFormPage(page);
    // act
    await listPage.open();
    await
    listPage.openEditByFarmName(createPayload.name);
    await formPage.submit(updatedName,
    createPayload.location);

    // assert
    await listPage.expectFarmVisible(updatedName);
  });

  test("@ui @regression should delete farm", async ({ page,
  env, farmController }) => {
    // arrange
    await loginViaUi(page, env.userName,
    env.userPassword);

    const createPayload = FarmFactory.valid();
    const createdFarm = await
    farmController.createFarm(createPayload);

    const listPage = new FarmListPage(page);
    // act
    await listPage.open();
    await listPage.deleteByFarmName(createPayload.name);

    // assert
    await listPage.expectFarmHidden(createPayload.name);

    const afterDelete = await
    farmController.getFarmById(createdFarm.id!);
    expect(afterDelete.status).toBe(404);
  });
});

// ===== tests/api/farm/farm.spec.ts =====
import { FarmFactory } from
  "../../data/factories/FarmFactory";
import { test, expect } from "../../fixtures/domain.fixture";
import { parseJsonWithFallback } from
  "../../utils/api/parseApiResponse";

test.describe("Farm API", () => {
  test("@api @regression should create farm", async
  ({ farmController, dataRegistry }) => {
    // arrange
    const payload = FarmFactory.valid();

    // act
    const created = await farmController.createFarm(payload);
    dataRegistry.registerFarmId(created.id);

```

```

// assert
expect(created.id).toBeTruthy();
expect(created.name).toBe(payload.name);
expect(created.location).toBe(payload.location);
});

test("@api @regression should get farm by id", async
({ farmController, dataRegistry, userApiContext }) => {
  // arrange
  const payload = FarmFactory.valid();
  const created = await farmController.createFarm(payload);
  dataRegistry.registerFarmId(created.id);

  // act
  const byId = await
userApiContext.get(`/api/farms/${created.id}`);

  // assert
  expect(byId.status()).toBe(200);
  const body = await parseJsonWithFallback(byId);
  expect(body.id).toBe(created.id);
  expect(body.name).toBe(payload.name);
  expect(body.location).toBe(payload.location);
});

test("@api @regression should update farm", async
({ farmController, dataRegistry }) => {
  // arrange
  const payload = FarmFactory.valid();
  const created = await farmController.createFarm(payload);
  dataRegistry.registerFarmId(created.id);

  // act
  const updated = await
farmController.updateFarm(created.id!, {
    name: `${payload.name}-v2`,
    location: payload.location
  });

  // assert
  expect(updated.name).toContain("-v2");
  expect(updated.id).toBe(created.id);
});

test("@api @regression should delete farm", async
({ farmController }) => {
  // arrange
  const payload = FarmFactory.valid();
  const created = await farmController.createFarm(payload);

  // act
  await farmController.deleteFarm(created.id!);
  const afterDelete = await
farmController.getFarmById(created.id!);

  // assert
  expect(afterDelete.status).toBe(404);
});
});

// ===== tests/security/ui/farm-direct-link.security.spec.ts =====
import { FarmController } from
"../../api/controllers/FarmController";
import { FarmFactory } from
"../../data/factories/FarmFactory";
import { test, expect } from " ../../fixtures/auth.fixture";
import { loginViaUi } from " ../../utils/auth/uiAuth";

```

```

import { expectForbiddenOrNotFound } from
"../../utils/api/securityAssertions";

test.describe("Farm UI direct-link security", () => {
  test("@ui @security @ownership should block foreign
edit/delete direct links", async ({
    page,
    env,
    userApiContext,
    adminApiContext
  }) => {
    // arrange
    const ownerClient = new
FarmController(adminApiContext);
    const foreignClient = new
FarmController(userApiContext);
    const foreignFarm = await
ownerClient.createFarm(FarmFactory.valid());
    expect(foreignFarm.id).toBeTruthy();
    await loginViaUi(page, env.userUsername,
env.userPassword);

    // act
    await page.goto(`/farms/edit/${foreignFarm.id}`);
    await page.goto(`/farms/delete/${foreignFarm.id}`);
    const afterDeleteAttempt = await
ownerClient.getFarmById(foreignFarm.id!);
    const foreignRead = await
foreignClient.getFarmById(foreignFarm.id!);

    // assert
    await
expect(page.locator("input[name='name']").toHaveCount(0);
    expect(afterDeleteAttempt.status).toBe(200);
    expectForbiddenOrNotFound(foreignRead.status);

    await ownerClient.deleteFarm(foreignFarm.id!);
  });
});

// ===== tests/e2e/business/farm-field.spec.ts =====
import { test, expect } from " ../../fixtures/domain.fixture";
import { loginViaUi } from " ../../utils/auth/uiAuth";
import { FarmListPage } from
" ../../pages/farm/FarmListPage";
import { FieldFormPage } from
" ../../pages/field/FieldFormPage";
import { FieldListPage } from
" ../../pages/field/FieldListPage";
import { randomSuffix } from " ../../utils/data/random";

test.describe("Business flow Farm -> Field", () => {
  test("@ui @critical should create field for an owned farm",
async ({
    page,
    env,
    farmController,
    dataRegistry
  }) => {
    // arrange
    await loginViaUi(page, env.userUsername,
env.userPassword);

    const farm = await farmController.createFarm({
      name: randomSuffix("farm-business"),
      location: randomSuffix("location-business")
    });
    dataRegistry.registerFarmId(farm.id);
    expect(farm.id).toBeTruthy();
  });
});

```

```
// act
await page.goto("/fields/add");
const fieldForm = new FieldFormPage(page);
const fieldList = new FieldListPage(page);
const fieldName = randomSuffix("field-business");
await fieldForm.submit(fieldName, 12.5, farm.id!);

// assert
await fieldList.open();
await fieldList.expectFieldVisible(fieldName);
const createdFieldId = await
fieldList.getFieldIdByName(fieldName);
dataRegistry.registerFieldId(createdFieldId);
expect(createdFieldId).toBeTruthy();

const farmList = new FarmListPage(page);
await farmList.open();
await farmList.expectFarmVisible(farm.name);
});
});

// ===== .github/workflows/test-automation-pr-gate.yml
=====
# ===== .github/workflows/test-automation-pr-gate.yml
(фрагменти) =====
name: AgroManagement Autotests Pipeline

on:
  pull_request:
    paths:
      - "agro-management/**"
      - "test-automation-framework/**"
      - ".github/workflows/test-automation-pr-gate.yml"
  push:
    branches: ["main"]
    paths:
      - "agro-management/**"
      - "test-automation-framework/**"
      - ".github/workflows/test-automation-pr-gate.yml"
  schedule:
    # Nightly full suite: 00:00 UTC ≈ 02:00–03:00
    Europe/Kyiv (GitHub cron is UTC-only)
    - cron: "0 0 * * *"
  workflow_dispatch:
    inputs:
      suite:
        description: "Select test suite to run"
        required: true
        default: smoke
        type: choice
        options:
          - smoke
          - ui
          - api
          - regression
          - security
          - smoke+security
          - full

permissions:
  contents: read
  pages: write
  id-token: write

jobs:
  test-suite:
    name: Run autotests
    runs-on: ubuntu-latest
```

```
env:
  BASE_URL: http://127.0.0.1:8080
  API_BASE_URL: http://127.0.0.1:8080
  USER_USERNAME: ${ secrets.USER_USERNAME }
  USER_PASSWORD: ${ secrets.USER_PASSWORD }
  ADMIN_USERNAME:
  ${ secrets.ADMIN_USERNAME }
  ADMIN_PASSWORD:
  ${ secrets.ADMIN_PASSWORD }
  SELECTED_SUITE: ${ github.event_name ==
'schedule' && 'full' || github.event_name ==
'workflow_dispatch' && github.event.inputs.suite || 'smoke' }
  # Strict only for PR/push (smoke gate) and manual smoke;
  nightly full is informational (known product failures).
  STRICT_MODE: ${ github.event_name ==
'pull_request' || github.event_name == 'push' ||
(github.event_name == 'workflow_dispatch' &&
github.event.inputs.suite == 'smoke') }

steps:
  - name: Checkout
    uses: actions/checkout@v4

  - name: Validate required GitHub secrets
    run: |
      missing=0
      for key in USER_USERNAME USER_PASSWORD
      ADMIN_USERNAME ADMIN_PASSWORD; do
        if [ -z "${key}" ]; then
          echo "Missing required secret: $key"
          missing=1
        fi
      done
      if [ "$missing" -ne 0 ]; then
        echo "One or more required secrets are missing."
        exit 1
      fi

  - name: Setup Node
    uses: actions/setup-node@v4
    with:
      node-version: 20
      cache: npm
      cache-dependency-path: test-automation-
framework/package-lock.json

  - name: Install test framework dependencies
    working-directory: test-automation-framework
    run: npm ci

  - name: Quality gate (typecheck)
    working-directory: test-automation-framework
    run: npm run typecheck

  - name: Install Playwright browser
    working-directory: test-automation-framework
    run: npx playwright install --with-deps chromium

  - name: Build and start infrastructure
    working-directory: agro-management
    run: docker compose up -d --build mysql keycloak

  - name: Wait for MySQL
    working-directory: agro-management
    # ... (фрагмент workflow пропущено: очікування
    MySQL/Keycloak, запуск app, seed ролей і реєстрація
    користувачів — рядки 95–237) ...

  - name: Run selected suite
    id: run-suite
    working-directory: test-automation-framework
```

```

run: |
  set +e
  exit_code=0
  echo "Selected suite: $SELECTED_SUITE"
  echo "Strict mode: $STRICT_MODE"

case "$SELECTED_SUITE" in
  smoke)
    npm run test:smoke || exit_code=$?
    ;;
  ui)
    npm run test:ui || exit_code=$?
    ;;
  api)
    npm run test:api || exit_code=$?
    ;;
  regression)
    npx playwright test --grep @regression ||
exit_code=$?
    ;;
  security)
    npm run test:security || exit_code=$?
    ;;
  smoke+security)
    npm run test:smoke || exit_code=$?
    npm run test:security || exit_code=$?
    ;;
  full)
    npm test || exit_code=$?
    ;;
  *)
    echo "Unknown suite: $SELECTED_SUITE"
    exit_code=2
    ;;
esac

echo "exit_code=$exit_code" >>
"$GITHUB_OUTPUT"

```

```

if [ "$exit_code" -ne 0 ] && [ "$STRICT_MODE" =
"true" ]; then
  echo "Strict mode is enabled -> failing workflow."
  exit "$exit_code"
fi

if [ "$exit_code" -ne 0 ]; then
  echo "Non-blocking failures detected for suite
'$SELECTED_SUITE'."
fi

- name: Publish non-blocking test summary
  if: always()
  run: |
    if [ "${{ steps.run-suite.outputs.exit_code }}" != "0" ]
    && [ "$STRICT_MODE" != "true" ]; then
      echo "## Informational run" >>
"$GITHUB_STEP_SUMMARY"
      echo "Suite\`$SELECTED_SUITE\` finished with
test failures, but workflow is configured as non-blocking for
this suite." >> "$GITHUB_STEP_SUMMARY"

- name: Cleanup test data
  if: always()
  working-directory: test-automation-framework
  run: npm run cleanup:data

- name: Upload test artifacts
  if: always()
  uses: actions/upload-artifact@v4
  with:
    name: ${{ env.SELECTED_SUITE }}-artifacts
    path: |
      test-automation-framework/test-results
      test-automation-framework/playwright-report
    if-no-files-found: ignore

```