Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ПОВЕДІНКОВИЙ ПІДХІД (BDD) ЯК ЕФЕКТИВНИЙ МЕТОД ДЛЯ ОРГАНІЗАЦІЇ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ В БЕЗПЕРЕВНІЙ ПОСТАВЦІ ПРОДУКТУ

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення” 121

Керівник курсової роботи
д.т.н., доц. Глибовець А.М.

_____
(підпис)
“____” _____ 2020 р.

Виконала студентка
Бенюх Л.І.
“____” _____ 2020 р.

Київ 2020

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри інформатики
к.ф-м.н., доц. Гороховський С.С
_____
(підпис)
“_____” _____2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студентці Бенюх Л.І. факультету інформатики 1 курсу МП
ТЕМА: Поведінковий підхід (BDD) як ефективний метод для організації автоматизованого тестування в безперервній поставці продукту

Вихідні дані:

Зміст ТЧ до курсової роботи:
Індивідуальне завдання
Вступ
1 Загальні положення автоматизованого тестування
2 Поведінковий підхід як один з інструментів в організації автоматизованого тестування
3 Система автоматизованого тестування на основі поведінкового підходу
Висновки
Список використаної літератури

Дата видачі “____” _____ 20__ р. Керівник _____
(підпис)

Завдання отримала_____
(підпис)

Тема: Поведінковий підхід (BDD) як ефективний метод для організації автоматизованого тестування в безперервній поставці продукту

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	02.12.2019	
2.	Огляд технічної літератури за темою роботи.	19.12.2019	
3.	Огляд літератури по організації автоматизованого тестування	21.01.2020	
4.	Аналіз підходів до автоматизованого тестування	20.02.2020	
5.	Вибір та ознайомлення з поведінковим підходом (BDD)	04.03.2020	
6.	Написання практичної частини, а саме розробка системи автоматизованого тестування	10.03.2020	
7.	Написання теоретичної роботи	15.04.2020	
8.	Підготовка до захисту роботи	12.05.2020	
9.	Коригування роботи за результатами попереднього огляду роботи	14.05.2020	
10.	Захист курсової роботи	29.05.2020	

Студентка Бенюх Л.І.

Керівник Глибовець А.М. “____” _____ 2020 р

ЗМІСТ

Анотація

У даній курсовій роботі розглянуто загальні відомості про організацію автоматизованого тестування та проаналізована ефективність застосування автоматизованого тестування на проекті. Також досліджено різні підходи до організації автоматизованого тестування за допомогою таких методів як написання коду через тестування, поведінковий підхід, підхід тестування за ключовими словами та набором даних. Також були розглянуті переваги, інструменти для організації поведінкового підходу тестування і розроблена система автоматизованого тестування для покриття тестами графічного інтерфейсу користувача та веб-сервісів (API тестування).

Ключові слова: поведінковий підхід (BDD), автоматизоване тестування, тестування графічного інтерфейсу, тестування веб-сервісів (API тестування), система для побудови автоматизованого тестування.

ВСТУП

Актуальність теми. Цифрова революція диктує свої вимоги до бізнесу, який стосується багатьох сфер (онлайн платформи по навчанню, розвагам, продажу, туризму тощо). Тому все більше підприємств прибігають за допомогою до різних інформаційних систем. В наші дні стрімко виросла потреба в автоматизації рутинних задач, з метою мінімізувати та оптимізувати час і кошти. Тому стоїть глобальне завдання, як прискорити час на доставку продукту до кінцевого користувача та при цьому не втратити якість продукту.

Саме автоматизація функціонального тестування відіграє дуже важливу роль у процесі отримання швидкого та ефективного результату тестування, і як результат доставки робочої версії продукту до кінцевого споживача у обмежений час.

Мета дослідження. Метою даної роботи є дослідження та використання підходів до автоматизованого тестування у середовищі швидких змін, гнучкої методології розробки та побудови безперервної доставки продукту кінцевому користувачу. Щоб досягти цієї мети необхідно виконати наступні завдання:

1. Дати визначення ролі автоматизованого тестування
2. Детально розглянути завдання, які має виконувати автоматизоване тестування.
3. Розглянути основні підходи до організації автоматизованого тестування на проекті.
4. Проаналізувати інструменти, які використовуються для організації поведінкового підходу.
5. Виділити основні переваги для побудови системи автоматизованого тестування за допомогою поведінкового підходу.
6. Використати поведінковий підхід для розробки системи автоматизованого тестування.

Об’єкт дослідження - концепція системи автоматизованого тестування на основі поведінкового підходу (BDD).

Предметом дослідження є можливість використання поведінкового підходу для отримання ефективного тестування у безперервній доставці коду користувачу.

Логіка дослідження зумовила таку структуру курсової роботи: вступ, 3 розділи, висновки та список використаної літератури.

У першому розділі буде проведене дослідження ролі та рівнів організації автоматизованого тестування, включаючи різні підходи.

У другому розділі буде досліджено ефективність реалізації поведінкового підходу автоматизованого тестування, інструменти та переваги.

Третій розділ буде присвячено безпосередньо опису процесу побудови системи автоматизованого тестування та її випробовування з подальшим аналізом.

РОЗДІЛ 1: Загальні положення автоматизованого тестування

1.1 Роль автоматизованого тестування у життєвому циклі продукту

Функціональне тестування допомагає забезпечити перевірку продукту згідно до вимог шляхом тестування функціоналу програмного забезпечення. Саме за допомогою тестування можливо перевірити, чи програма "працює" як передбачалось чи ні. У наш час відбувається оцифровування бізнесу і саме тому розробка інформаційних продуктів стає дуже актуальною. Як результат постає дуже важливе завдання перед командами з розробки забезпечувати постійне тестування та частішу доставку продукту до кінцевого споживача.

Автоматизація функціонального тестування відіграє дуже важливу роль у процесі отримання швидкого та ефективного результату тестування, і як результат доставку робочої версії продукту до кінцевого споживача у обмежений час [4]. Наприклад, працюючи з дуже комплексним і складним продуктом, автоматизація тестування відіграє ключову роль, коли потрібно знову і знову періодично перевіряти той самий функціонал з метою перевірити чи готовий продукт до поставки до кінцевого користувача чи ні.

Процес функціонального автоматизованого тестування полегшує виконання комплексних функціональних тестів, які допомагають розробити надійний продукт, гарантуючи, що програмне забезпечення не містить помилок.

Ручне тестування вимагає фізичного часу, і повторюваний характер тестування може стати одноманітним. Таким чином, для роботи у швидкому, змінному середовищі організації повинні зосередитись на впровадженні практики автоматизації тестів.

Отже, підсумовуючи вище зазначене, мені б хотілось виділити наступні причини чому потрібно запроваджувати автоматизоване тестування програмного забезпечення на проекті[4]:

1. Безперервна доставка продукту до користувача.

2. Перевірка системи в цілому та отримання результатів тестування при поставці кінцевому користувачу.
3. Можливість проходити тест сценарії, які неможливо виконати вручну одночасно (наприклад паралельні тести, тестування на декількох браузерах та операційних системах).
4. Швидкість виконання тесту та отримання результатів тестування.
5. Більш ефективне використання ресурсів тестування.

Саме тому стратегія по побудові автоматизованого тестування повинна бути включена у загальну стратегію тестування. Необхідно ретельно планувати впровадження автоматизованого тестування, щоб цей процес відбувся успішно для організації.

1.2 Процес побудови автоматизованого тестування

Як було зазначено вище автоматизоване тестування допомагає багатьом організаціям підвищити швидкість та ефективність процесів тестування програмного забезпечення.

Покриття тестуванням пов'язане з багатьма діями в рамках програмного забезпечення і життєвого циклу розробки продукту. Тому виділяють такі рівні тестування, як на рисунку 1.1, що наведений нижче.

Рисунок 1.1 - Рівні Тестування

Більш детальний опис різних рівнів тестування наведений у Таблиці 1.1.

Таблиця 1.1 - Опис тестування на різних рівнях

Рівень тестування	**Опис**	**Відповідальна сторона**	**Коментар**
Компонентне тестування	Тестування окремих модулів програми	Розробники	Тестування повинно відбуватися автоматично
Інтеграційне тестування	Тестування взаємодій обмеженого набору модулів програми	Розробники: взаємодія між усіма компонентами системи та сторонніми системами	Тестування проводиться автоматично
		Тест інженери: З боку системи - тестування інтеграції між усіма підсистемами та сторонніми системами	Тестування проводиться автоматично
Веб-сервісів (API) тестування	Тестування веб-сервісів (API) безпосередньо та в рамках тестування інтеграції з ціллю визначити чи вони відповідають вони вимогам	Тест інженери з автоматизованого тестування	Тестування проводиться автоматично
Тестування інтерфейсу користувача	Тестування графічного інтерфейсу програми для виявлення дефектів	Тест інженери з ручного та автоматизованого тестування	Тестування проводиться спочатку вручну, а потім автоматично

Продовження таблиці 1.1

Системне тестування	Основним напрямком тестування системи є перевірка визначених вимог, тестування інтегрованої системи в цілому	Тест інженери з ручного та автоматизованого тестування	Тестування проводиться спочатку вручну, а потім автоматично
Приймальне тестування	Тестування бізнес сценаріїв за участю клієнтів	Окрема група користувачів за допомогою команди з контролю якості (якщо потрібно)	Тестування може проводитися вручну та/або автоматично

В залежності від основної мети, яку покликана досягти автоматизація тестування виокремлюються різні області функціоналу продукту для покриття автоматичними тестами.

Основною метою автоматизації тестування для організації безперервної доставки продукту до користувача потрібно виділити такі області як[4]:

- покриття тестами самих критичних з точки зору бізнесу функцій, сценаріїв, бізнес-потоків та бізнес-процесів;
- покриття функціоналу, що розробляється у версії продукту, яка буде доставлена кінцеву користувачу;
- покриття тестами веб-сервісів (API);
- перевірка найважливіших з точки зору бізнесу сценаріїв, бізнес потоків та бізнес-процесів після змін у коді.

Також, наступні фактори грають визначальну роль при побудові успішного процесу автоматизованого тестування [4]:

1. **Побудова архітектури для автоматизації тестування.**

Архітектура автоматизації тестування дуже тісно пов'язана з архітектурою програмного забезпечення продукту. Тому необхідно виявити усі функціональні та нефункціональні вимоги з самого початку.

2. **Аналіз можливості покриття автоматизованими тестами.**

Продукт повинен бути розроблений таким чином, щоб можливо було його підтримувати автоматизованими тестами. Наприклад, у випадку тестування графічного інтерфейсу клієнта, побудова інтерфейсу не має бути сильно зав’язана на даних від сервера, а більше на їх відображення. У випадку тестування веб-сервісів (API) це може означати, що інтерфейси потрібно залишати відкритими, щоб можливо було покривати тестами.

3. **Створення стратегії по побудові автоматизації тестування на проекті.**

Практична та послідовна стратегія автоматизації тестів, яка стосується підтримки та розширенню продукту автоматичними тестами, допоможе спланувати процес по тестуванню оптимальним чином в рамках існуючого продукту. Створюючи стратегію автоматизації, необхідно враховувати витрати, переваги та ризики її імплементації для різних рівнів інформаційного продукту.

4. **Розробка системи з автоматизованого тестування.**

Для того, щоб створити простий у використанні та технічному обслуговуванні фреймворк, необхідно зробити наступне:

- побудувати тестовий стенд;
- встановити пріоритети для автоматизації тестових сценаріїв;
- постійно оновлювати автоматизовані тести;
- реалізувати систему безперервної поставки автоматичних тестів та їх запуск для перевірки збірок продукту протягом релізу, ітерації та нічни збірок, тощо;
- планувати та аналізувати регулярно тести для автоматизації.

1.3 Підходи для побудови автоматизованого тестування

Для реалізації ефективного автоматизованого тестування окрім побудови процесу автоматизованого тестування, також важливу роль відіграє сам підхід побудови тестів. Так підхід до автоматизації тестування буде

залежати від самого процесу розробки, тому що ці речі повинні бути взаємопов'язані. Виокремлюють такі основні підходи як [3]:

- розробка на основі тестів (англійською мовою Test Driven Development);
- поведінковий підхід тестування (англійською мовою Behaviour Driven Development);
- тестування на основі ключових слів (англійською мовою Keyword Driven Testing);
- тестування на основі даних (англійською мовою Data Driven Testing).

Давайте розглянемо, що означає кожен з підходів:

Розробка на основі тестів (англійською мовою Test Driven Development)

Передбачає спочатку організацію автоматизованого тестування за допомогою написання модульних, функціональних та інтеграційних тестів, і лише після написання самого робочого коду продукту. Тобто в першу чергу пишеться тест, який перевіряє коректність роботи ще не написаного коду. Далі програміст пише код, який виконує дії необхідні для проходження тесту. Коли тест буде успішно пройдено, можлива оптимізація чи доповнення написаного коду.

Цей підхід більше, ніж просто перевірка коректності, так як він впливає і на дизайн програми [8]. Якщо ви спочатку сфокусовані на тестах, вам простіше уявити, яка саме функціональність потрібна користувачеві. В результаті розробник продумає деталі інтерфейсу до реалізації продукту. Це, в свою чергу, скоротить час на розробку. Крім того, розробка на основі тестів зосереджує інженера на тестуванні окремих модулів, при цьому на місці ще нереалізованого функціоналу використовуються його заглушки.

Поведінковий підхід (англійською мовою Behaviour Driven Development)

По суті, поведінковий підхід є різновидом (доповнення) розробки на основі тестів з тією лише різницею, що цей підхід орієнтований на поведінку, яка покривається тестами (в тестовому підході основний фокус

йде безпосередньо на сам код). Суть поведінкового підходу полягає в розробці тестів, які описують систему термінами, зрозумілими не технічному спеціалісту. Це дає можливість прискорити процес отримання зворотного зв'язку від бізнесу. Тобто опис сценаріїв користувача відбувається природньою мовою – тобто мовою бізнесу. Детальніше ми розглянемо цей підхід у Розділі 2.

Тестування на основі ключових слів (англійською мовою Keyword Driven Testing)

Даний підхід передбачає використання ключових слів, що описують набір дій, потрібних для виконання конкретного кроку тестового сценарію. При такому підході в першу чергу визначається набір ключових слів, а тільки після цього асоціюється з функцією яка дія, пов'язана з даним ключовим словом. Наприклад, кожні кроки тесту, такі як натискання на кнопку миші чи клавіатури, відкриття або закриття браузера описуються певними ключовими словами («відкрити» - openbrowser, «натиснути» - click тощо). Використовуючи цей підхід, можна створювати прості функціональні тести на самих ранніх етапах розробки і тестувати додаток частинами.

Тестування на основі даних (англійською мовою Data Driven Testing)

При даному підході тестові дані зберігаються окремо від тестових сценаріїв, наприклад, у файлах або в базі даних. Такий розподіл значно спрощує самі тести та їх об’єм. Цей підхід використовується в проектах, де потрібно виконати тестування окремо запропонованих в кількох середовищах з великими наборами даних і стабільними тестовими прикладами.

РОЗДІЛ 2: Поведінковий підхід (BDD) як один з інструментів для організації автоматизації тестування

2.1 Загальне визначення поведінкового підходу

Як було зазначено, розробка за допомогою поведінкового підходу (BDD) була створена як доповнення підходу розробки на основі тестів (TDD) шляхом формалізації найкращих практик. Такий підхід допомагає всій команді зрозуміти дії та поведінку кінцевого користувача, що приводить до більш чітких вимог, тестів та, в кінцевому рахунку, до більш якісного продукту.

Часто затрачається багато часу на комунікацію між клієнтом та командою розробки для того, щоб зрозуміти, що потрібно розробити, що розроблено і що хочуть бачити кінцеві користувачі [3]. Ця комунікація часто є вузьким місцем у прогресі проекту, тому що технічні спеціалісти можуть невірно зрозуміти бізнесу, а бізнес-професіонали неправильно розуміють можливості своєї технічної команди. Як результат, кінцевий продукт технічно функціональний, але не відповідає точним вимогам бізнесу.

Поведінковий підхід є сукупністю практик для процесу розробки, спрямований на зменшення деяких ризиків, пов'язаних з комунікацією та зворотнім зв'язком між командою по розробці та клієнтом. Саме за допомогою цього підходу є можливість імітувати, як програма повинна вести себе з точки зору кінцевого користувача. Основною метою впровадження тестування за допомогою поведінкового підходу є поліпшення співпраці між зацікавленими сторонами, такими як розробники, тест інженери, менеджери продуктів та бізнес-аналітики. Існує широкий набір інструментів які підтримують впровадження поведінкового підходу, такі як Cucumber, JBehave, Gauge тощо.

З мого власного досвіду поведінковий підхід позитивно впливає на наступні аспекти розробки:

- Комунікація: розробка програмного забезпечення не стосується лише розробників. У команді присутні також тест інженери, проектні

менеджери, продуктові менеджери та інші зацікавлені сторони. Цей підхід дає можливість усім залишатися у контексті функціоналу протягом усього життєвого циклу продукту, що в цілому сприяє якісному спілкуванню.

- Прозорість процесу розробки: поведінковий підхід (BDD) описує функціонал простою мовою. Використовуючи мову, зрозумілу всім, кожен отримує чітку видимість прогресу проекту.
- Рівень задоволення кінцевих користувачів: орієнтуючись на потреби бізнесу, ви отримуєте задоволених користувачів, а це означає успішний бізнес.

Також тестові сценарії можна писати у реальному часі, опираючись на поведінку системи. Якщо команди по розробці працюють у змінному середовищі, яке базується на ітеративних доставках програмного забезпечення кінцевому користувачеві, поведінковий підхід це один з інструментів, який дає змогу проекту бути більш гнучким [2]. Він дозволяє переосмислити підхід до тестування в цілому, де сам процес має сенс.

Тест сценарії мають наступні властивості:

- Кожен тест повинен бути актуальним з точки зору продукту та створений, базуючись на конкретні приклади використання функціоналу.
- Тести мають завжди давати однакові результати, якщо код не змінювався.
- Кожен тестовий сценарій буде реальним сценарієм для користувача, а не простим тестовим випадком завдяки впровадження матриці "Роль-Поведінка-Причина" та формули "Дано-Коли-Тоді" базуючись на синтаксисі Gherkin.
- Документація з тестовими сценаріями зберігається в одному місці та постійно оновлюється.

Перш ніж вибрати інструмент для впровадження поведінкового підходу, потрібно виконати попередній аналіз. Для того, щоб зрозуміти мотивацію

для чого ви плануєте застосовувати цей підхід, а також хто буде використовувати цей інструмент.

Кожна організація має різні ролі, і поведінковий підхід не повинен належати виключно розробникам, ані інженерам з автоматизації тестування. Якщо ви не співпрацюєте близько з бізнесом, ви ніколи не побачите повну користь від цієї методології.

На ринку існує декілька інструментів з відкритим доступом для реалізації даного підходу. Розглянемо їх порівняння у Таблиці 2.1.

Таблиця 2.1 Порівняльна таблиця інструментів BDD

Назва	«Жива» документація	Зручність у написанні тестів і документації	Налаштування та робота	Звітування	Інтеграція з CI/CD
Cucumber [16]	Присутня	Зручно, з стандартами BDD	Зручний у використані, і нескладний у налаштуванні	Так, свій формат. В реальному часі	Так
Jbehave [17]	Частково	Зручно, з стандартами BDD	Складний у налаштуванні	Так, але потребує налаштувань	Так
Gauge [18]	Частково	Не дуже зручно(без більш стандартного формату)	Складний у налаштуванні	Так, але потребує налаштувань	Так

У моєму випадку я використовувала Cucumber тому що саме цей інструмент допоміг реалізувати такі потреби як:

- Взаємодія з бізнес представниками та ефективний спосіб при отриманні зворотного зв'язку.
- Зручність у написанні тестів для автоматизації.
- “Жива” документація.
- Візуалізація результатів тестування.
- Реалізувати систему безперервної поставки автоматичних.
- Бути впевненим, що вірний функціонал, затверджений з бізнес представниками на ранньому етапі розробки.

Давайте подивимось як працює Cucumber. Хочу зазначити, що в Cucumber можна використовувати не тільки для тестування та написання коду, але і для ведення документації продукту. Вона завжди буде в оновленому стані. На практиці це означає, що ваша документація, замість того, щоб писатися один раз, а потім поступово застарівати, стає “живою” та відображає актуальний стан проекту.

Отже давайте розглянемо як запускається тест в Cucumber. Спочатку Cucumber розпізнає специфіку функціоналу, яка описана простою мовою, потім перевіряє її на наявність сценаріїв для тестування, і потім запускає сценарії в системі. Кожен сценарій складається з набору кроків, які Cucumber обробляє. Саме завдяки цьому Cucumber може зрозуміти файли, де зберігається інформація про функціонал, базуючись на синтаксичних правилах. Синтаксичні правила, як уже було згадано, мають назву Gherkin. Давайте розглянемо приклади написання сценаріїв за допомогою синтаксису Gherkin

1. **Опис функціоналу у формі функціональних вимог**. Приклад можна знайти на Рисунку 2.2.

```
Feature: Sign up

  Sign up should be quick and friendly.

  Scenario: Successful sign up

    New users should get a confirmation email and be greeted
    personally by the site once signed in.

    Given I have chosen to sign up
    When I sign up with valid details
    Then I should receive a confirmation email
    And I should see a personalized greeting message

  Scenario: Duplicate email

    Where someone tries to create an account for an email address
    that already exists.

    Given I have chosen to sign up
    But I enter an email address that has already registered
    Then I should be told that the email is already registered
    And I should be offered the option to recover my password
```

Рисунок 2.2 - Приклад тестового сценарію [1]

На зображеному прикладі можна виокремити таку структуру:

- **Функціонал (англійською Feature):** Короткий, але повний опис необхідної функціональності, який запускає функцію і дає їй ім'я. Наступні три рядки описують переваги, які дає цей функціонал.
    - **Тестовий Сценарій (англійською Scenario):** Конкретна бізнес-ситуація для перевірки, що включає в себе детальний опис.
    - **“Дано” визначає етап**. Наведений крок описує передумови вашого тесту. Він встановлює будь-які тестові дані, які потрібні вашому тесту. Як правило, це включає такі речі, як створення будь-яких необхідних тестових даних або, для веб-програми, вхід у систему та перехід на потрібну сторінку.
    - **“Коли” включає дію для тестування**. На кроці коли описано головну дію або подію, яку ви потрібно перевірити. Це може бути користувач, який виконує якусь дію на веб-сайті, або інша подія, що не використовується інтерфейсом користувача, наприклад обробка транзакції або обробка повідомлення про подію. Ця дія призведе до певного спостережуваного результату, який ви підтвердите на кроці далі.
    - **“Тоді” описує очікуваний результат**. Крок далі порівнює спостережуваний результат або стан системи з тим, що ви очікуєте.

2. **Тестові сценарії у формі таблиці**. Таблиці містять індивідуальні кроки. Їх використовують тоді, коли багато повторів та потрібно протестувати великий набір даних (див. Рисунок 2.3).

```
Scenario: Transfer points between existing members
    Given the following accounts:
      | owner     | points | status-points |
      | Danielle  | 100000 | 800           |
      | Martin    | 50000  | 50            |
    When Martin transfers 40000 points to Danielle
    Then the accounts should be the following:
      | owner    | points | status-points  |
      | Danielle | 140000 | 800            |
      | Martin   | 10000  | 50             |
```

Рисунок 2.3 - Приклад використання таблиці у сценаріях [1]

3. **Тестові сценарії у формі таблиці з прикладами.** Це фактично чотири сценарії зведені в один - сценарій буде запускатися чотири рази, кожен раз зі значеннями з одного рядка в прикладі таблиці. Дані з таблиці

передаються на кожен крок через назви полів у кутових дужках: <status>, <base> тощо і можливо скоротити чотири сценарії до одного стислого сценарію та таблиці прикладів, як на Рисунку 2.4

```
Scenario: Earning extra points on flights by Frequent Flyer status
Given I am a <status> Frequent Flyer member
When I fly on a flight that is worth <base> base points
Then I should earn a status bonus of <bonus>
And I should have guaranteed minimum earned points per trip of <minimum>
And I should earn <total> points in all


Examples:
| status   | base | bonus | minimum | total |
| Standard | 439  | 0     | 0       | 439   |
| Silver   | 439  | 220   | 0       | 659   |
| Gold     | 439  | 329   | 1000    | 1000  |
| Gold     | 2040 | 1530  | 1000    | 3570  |
```

Рисунок 2.4 - Таблиця прикладів [1]

Давайте розглянемо, як додавати ці тест сценарії у реальний проект. Такі сценарії вносяться у файл .feature, який можна додати в проект разом із кодом. Організовувати файли .feature потрібно таким чином, щоб їх було легко знаходити та переглядати. Коли ви починаєте отримувати велику кількість файлів функцій у своєму проекті, важливо організувати у окрему папку як на Рисунку 2.5.

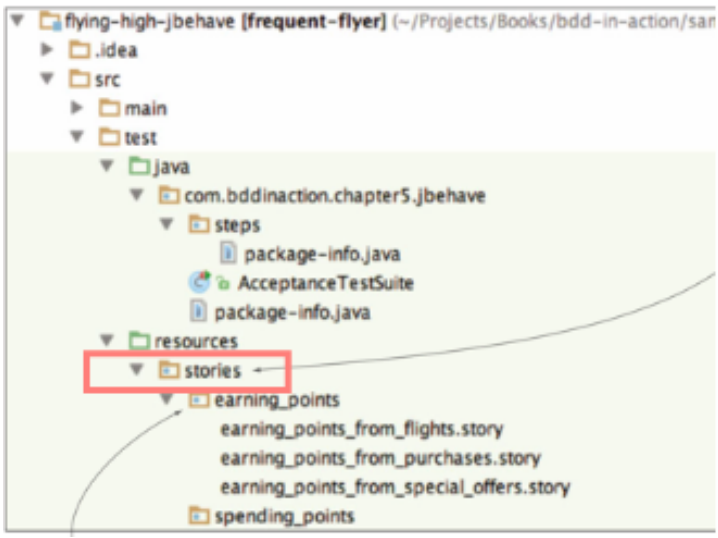


Рисунок 2.5 - Приклад організації структури проекту

Отже для того, щоб виконати ці дії, які наведені у тестових сценаріях, потрібно їх правильно оформлювати, а також додавати код (який буде виконувати дії та кроки [2]. Кожен крок у файлах описується окремим методом з використанням однієї з мов програмування, наприклад Java. Опис

кроків - це по суті код, який інтерпретують текст у .feature файлі (див. Рисунок 2.6).

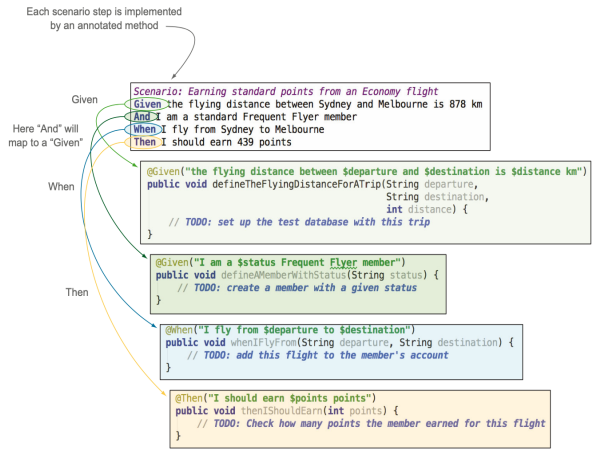


Рисунок 2.6 - Опис кроків через код

Для того, щоб тестові сценарії відображались у звіті кожен окремо, їх необхідно позначати тегами. Наприклад, візуалізація даних у звіті за тегами зображена на Рисунку 2.7.

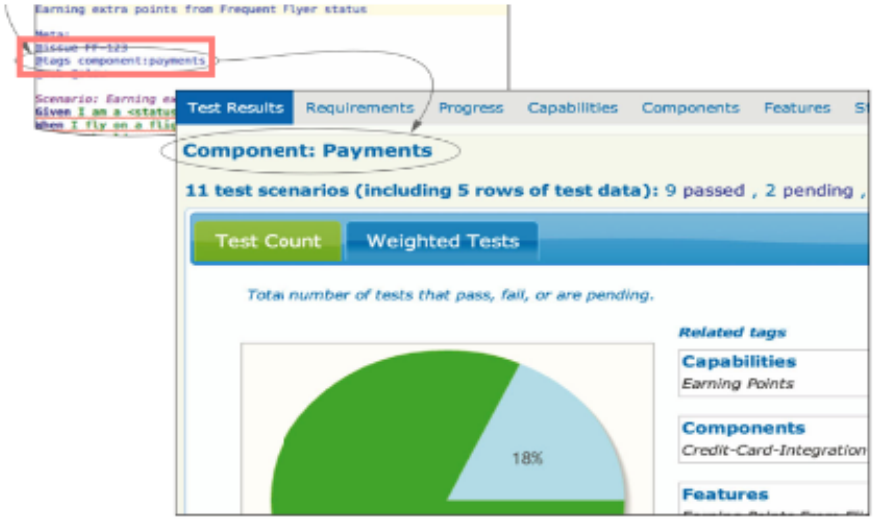


Рисунок 2.7 - Приклад додавання тегів для зручних звітів [2]

У тестовому наборі перевірок визначення кроку можуть бути одна чи дві строчки коду, запуск якого йде за допомогою використання бібліотек для автоматизації тестування, наприклад як бібліотека для тестування графічного інтрефейсу користувача Selenium чи для веб-сервісів (API тестування) RestAssured.

Підсумовуючи, для реалізації автоматичних функціональних тестів, проходять кроки зображені на Рисунку 2.8.

Рисунок 2.8 – Процес роботи з Cucumber

Отже поведінковий підхід дозволяє зрозуміти бізнес, та швидко адаптуватися до змін його потреб. Водночас команда з розробки може впевнитися, що розробляється актуальний для кінцевого користувача функціонал.

2.2 Безперервна поставка тестування на основі BDD

Якщо ви хочете швидше доставляти бізнес-цінність користувачам, вам потрібно мати можливість швидко та ефективно доставляти функціонал. Доставка функціоналу на живе середовище, як правило, є досить складним процесом. Потрібно створити програму з вихідного коду та запустити компонентні, інтеграційні, приймальні та автоматизовані тести. Потрібно додати його в пакет, що буде доставлятися до кінцевого споживача. Отже автоматизація цього процесу, а також впровадження автоматизованого тестування, лежать в основі стратегії з безперервної інтеграції та доставки продукту.

Існує багато інструментів для збірки вашого проекту. Так у світі Java можна використовувати Maven, Gradle або Ant. Для проекту, який описаний у практичній частині використовується саме Gradle для управління процесом збірки проекту.

Автоматизація тестування є запорукою ефективного процесу розгортання. Будь-які автоматизовані кроки в цьому процесі будуть швидшими та надійними, ніж ручні тести. Автоматизовані приймальні тести та жива документація, створені на основі поведінкового підходу (BDD), також допомагають підвищити впевненість у якості програми, дозволяючи тест інженерам витрачати менше часу на повторне тестування.

Автоматизовані тести повинні запускатися автоматично, як частина автоматизованого процесу збирання продукту. Незалежно від того, чи вони реалізовані як одиничні тести низького рівня або функціональні тести, вони повинні дотримуватися певної кількості обмежень. Зокрема,

- Кожен виконуваний тест повинен бути самодостатнім.
- Виконані тести повинні відповідати версії збірки.

Одним з найважливіших принципів, що лежать в основі гнучкої практики розробки програмного забезпечення, є зворотній зв'язок. Інформування про проблему - це перший крок до її вирішення. Затримка представляє ризик: чим швидше ви дізнаєтесь про потенційну проблему, тим швидше вона буде вирішена. Навіть позитивні відгуки корисні; коли ви можете підтвердити, що певна стратегія чи вирішення проблеми є успішним, ви можете використати це для подальшої роботи.

Основною метою безперервної інтеграції (CI) є забезпечення швидкого зворотного зв’язку про стан процесу збирання. Сервер CI є програмою, наприклад Jenkins, яка постійно відстежує сховище вихідного коду проекту на предмет змін. Кожного разу, коли відбувається зміна контролю над версією, сервер CI починає збірку (автоматично чи у ручному режимі) для компіляції та тестування цієї версії програми. Це означає, що всі автоматизовані тести виконуються для кожної нової версії бази коду. Якщо щось піде не так, команда негайно дізнається про це. У командах, які широко практикують CI, стан збірки сприймається дуже серйозно, і якщо збірка зламається, відповідальний розробник негайно припинить роботу та виправить проблему. Як виглядають тести можна побачити на Рисунку 2.9.

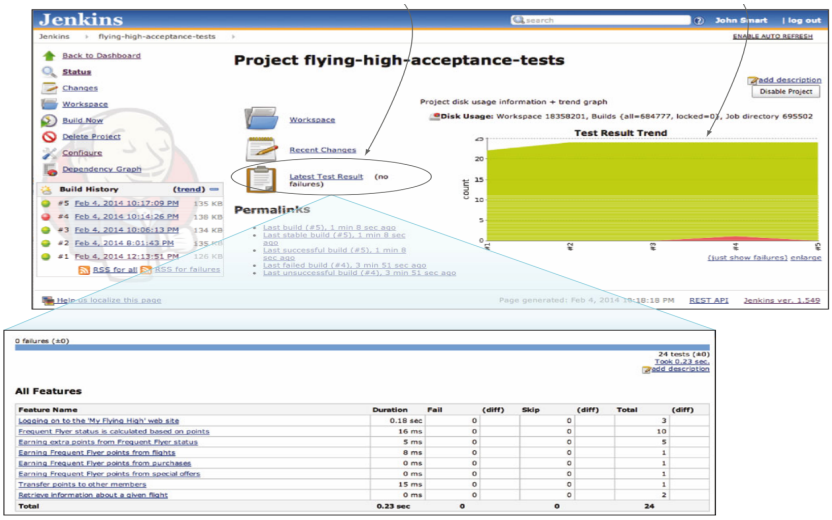


Рисунок 2.9 - Автоматизовані приймальні тести з використанням Jenkins

Висновки

Команди, які практикують поведінковий підхід (BDD), знаходяться в вигідному становищі для забезпечення постійної доставку тому що:

- безперервна доставка продукту користувачеві відбувається за допомогою перевірених та узгоджених тестів заздалегідь, що забезпечує високий рівень довіри до автоматизованих тестових сценаріїв;
- на роботу з документацією та ручне тестування не витрачається багато часу завдяки «живій» документації, яка також інтегрована в CI і повинна генеруватися автоматично для кожної нової збірки в CI;
- дуже часто виникає ситуація, коли ручні тест інженери не мають довіри до результатів автоматизованого тестування і витрачають час на повторне тестування уже автоматизованих сценаріїв, тому що не розуміють, що покрито, а що ні. З поведінковим підходом всі члени команди розуміють, що написано і тому, якщо і доводиться повторювати тести вручну, то тільки якщо вони не були покриті автоматизованими тестами
- можливо пришвидшити виконання приймальних тестів, запустивши їх паралельно, використовуючи Selenium Grid або на одній машині, або на декількох машинах.

РОЗДІЛ 3: Система автоматизованого тестування на основі поведінкового підходу

3.1 Розробка системи для автоматизованого тестування

3.1.1 Огляд системи та інструменти

Система автоматизованого тестування побудована на базі фреймворку, який включає у себе такі рівні автоматизованого тестування як:

- Тестування веб-сервісів - API тестування;
- Функціональні автоматизовані тести графічного інтерфейсу користувача.

Система для автоматизації тестування має структуру програмного продукту. Як продукт, фреймворк для автоматизації тестування визначає функції програми, такі як обробка зовнішніх файлів, взаємодія з графічним інтерфейсом, надає шаблони для структури тестів. Загалом, розробка фреймворка для автоматизованого тестування є нічим іншим як розробкою стандартного програмного забезпечення.

Опис функціоналу та самі тести були написані на основі поведінкового підходу, використовуючи Cucumber та Gherkin синтаксис.

Система була інтегрована з сервісом безперервної поставки коду Jenkins, а результатом кожного тестування є автоматично-згенерований звіт на основі Cucumber.

Для автоматизації та випробовування самої системи були написані таки автоматичні тести:

- Тестовий онлайн магазин від Selenium для тестування графічного інтерфейсу користувача: http://automationpractice.com/index.php;
- Ресурс прогнозу погоди з відкритим API: https://restapi.demoqa.com/utilities/weather/city/lviv.

Детальний опис технологій та інструментів, які використовувались для написання системи з автоматизованого тестування можна знайти у Таблиці 3.1.

Таблиця 3.1 – Технології та Інструменти

Назва інструменту	**Тестування веб-сервісів (API тестування)**	**Тестування графічного інтерфейсу**
Мова програмування	Java 8	
Середовище програмування	IntelliJ IDEA	
Інструмент для написання тестів у стилі BDD [1]	Cucumber 5.7.0	
Бібліотека для використання Cucumber для Java [13]	Cucumber JVM 1.0.6	
Інструмент для звітування [1]	Cucumber Reporting 5.2.1	
Синтаксис для написання тестів [1]	Gherkin 13.0.0	
Бібліотека для запуску тестів у веб-браузері [10]	-	Selenium Version 3.141.59
Менеджмент бібліотек для роботи з веб-драйвером [10]	-	WebDriver Manager 3.8.1
Бібліотека для роботи з веб-сервісами (API) [11]	Rest Assured 4.3.0	-
Бібліотека для роботи з Redis [12]	Jedis 3.2.0	
Бібліотека для паралельного запуску тестів [14]	Courgette 4.6.2	
Бібліотека для збору логів для Java	Logback 1.2.3	
Інструмент для збирання проекту [8]	Gradle	
Інструмент для віртуалізації операційної системи [15]	Docker	
Інструмент для управління інтеграцією [9]	Jenkins	

3.1.2 Огляд розробки системи для автоматизованого тестування

Давайте розглянемо детальніше з яких функціональних частин для повноцінного тестування складається фреймворк. Почнемо з загального опису компонентів, які в нього входять, а потім перейдемо до більш детального опису покриття тестами кожного рівня тестування. Як виглядає проект у середовищі розробки IntelliJ IDEA та з яких частин він складається можна знайти на Рисунку 3.1.

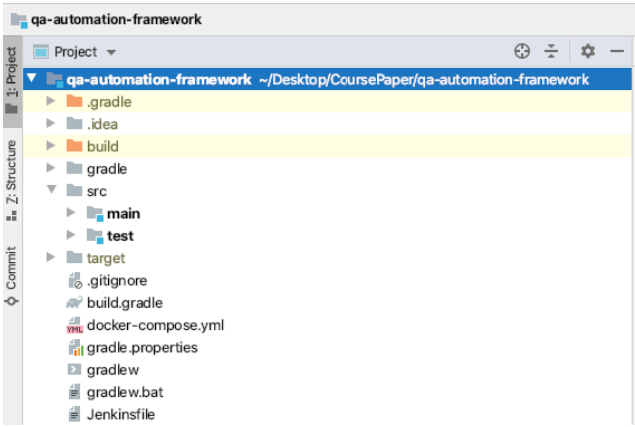
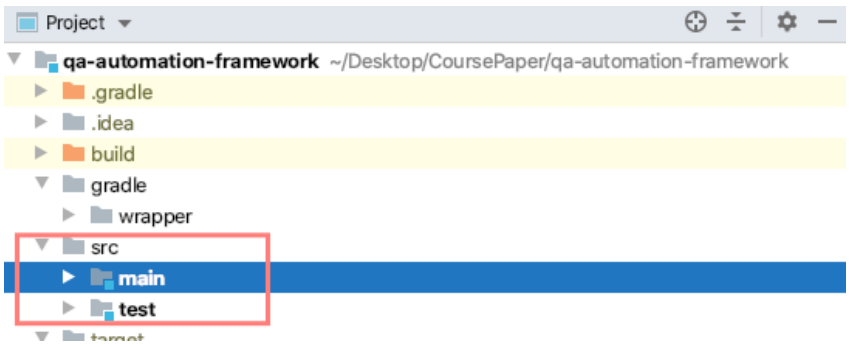


Рисунок 3.1 – Структура проекту

У ньому є основні та додаткові компоненти, які забезпечують роботу системи. Деякі компоненти використовуються для тестування веб-сервісів (API тестування) і тестування графічного інтерфейсу одночасно. Тоді як інші лише одного чи іншого вказаного тестування.

Основні ресурси проекту знаходяться у папці src, яка включає main та test.



Рисунку 3.2 – Основні компоненти фреймворку

Директорія resources включає саб-директорію feature, який у свою чергу містить детальні сценарії тестів – для API та UI - описані за допомогою Gherkin (див. Рисунок 3.3).

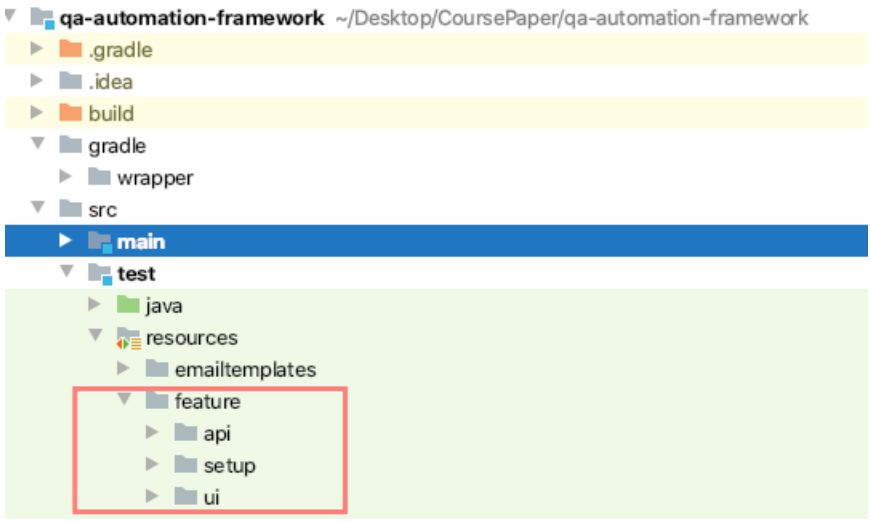


Рисунок 3.3 - Feature файли

Приклад опису перевірки зображений на Рисунку 3.4.

```
Feature: Test Weather API.

Background:
    Given API baseUrl "https://restapi.demoqa.com/utilities/weather/city/"

@api @api-001 @smoke
Scenario Outline:
    Given City of "<city>"
    And invoke weather API
    Then verify the response code is "<code>" and body contains city name "<verifyCity>"
    Examples:
        | city        | code | verifyCity |
        | kyiv        | 200  | Kyiv       |
        | lviv        | 200  | Lviv       |
        | incorrect | 400  |            |
```

Рисунку 3.4 – Приклад опису функціоналу за допомогою Gherkin

Код для розпізнавання кроків функціоналу знаходиться у папці test (див. Рисунок 3.5). Всі тести поділяються за напрямком тестування: API тестування чи тестування графічного інтерфейсу (UI).

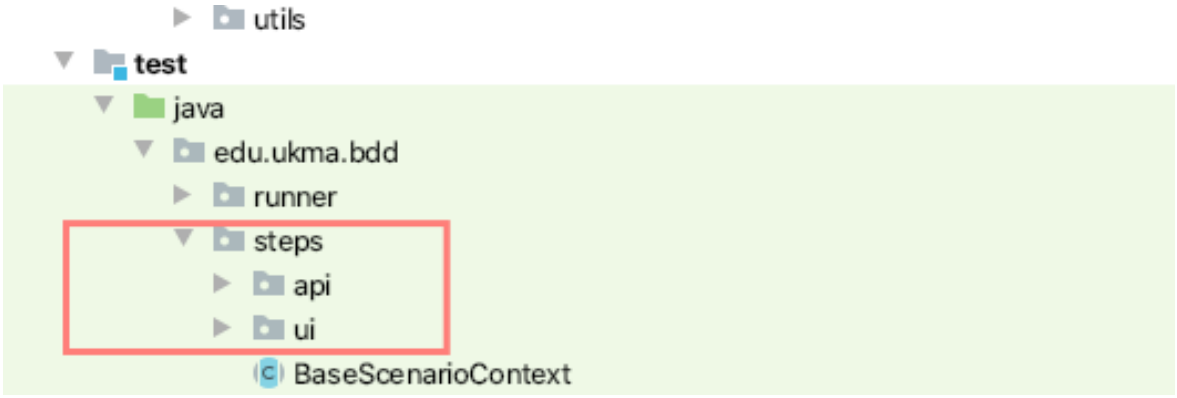


Рисунок 3.5 – Розташування кроків автоматизованих тестів

Реалізовані функції для опису кроків сценаріїв саме API тестів подані Рисунку 3.6. Дані функції описують логіку кожного кроку описаного за допомогою мови Gherkin.

В тестах імпортується бібліотека Cucumber, яка потрібна для роботи з відповідним функціоналом та бібліотека RestAssured, тому що виконується перевірка веб-сервісів (API тестування).

```
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;

import io.restassured.response.Response;
import io.restassured.specification.RequestSpecification;

public class APITest {

  private String givenCity;
  private RequestSpecification request;
  private Response response;
  private String baseUrl;

  public APITest() {

  }

  @Given("API baseUrl \"([^\"]*)\"$")
  public void api_url(String baseUrl) { this.baseUrl = baseUrl; }

  @Given("^City of \"([^\"]*)\"$")
  public void city_of(String city) { this.givenCity = city; }

  @Given("^invoke weather API$")
  public void invoke_weather_api() {
    this.request = given().pathParam( parameterName: "city",givenCity);
    this.response = this.request.when().
                    get( path: this.baseUrl+ "{city}");
  }

  @Then("^verify the response code is \"([^\"]*)\" and body contains city name \"([^\"]*)\"$")
  public void verify_response(int code, String city) {
    this.response.then().
        assertThat().
        statusCode(code).
        body( path: "City", city.trim().isEmpty() ? Matchers.nullValue() : Matchers.equalTo(city));
  }

`
```

Рисунок 3.6 – Детальний опис тестового сценарію для API

Логіка для запуску знаходиться у папці runner (див. Рисунок 3.7).

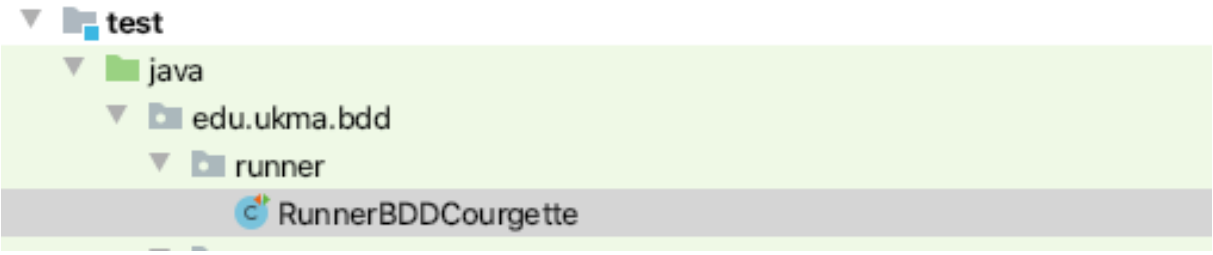


Рисунок 3.7 – Розташування файлу логіки запуску тестів

RunnerBDDCourgette клас (див. Рисунок 3.8) містить конфігурацію логіки за якою потрібно запускати автоматизовані тести і включає такі параметри як:

- Кількість потоків для виконання тестів;
- Потрібно чи не потрібно показувати результати тестів;
- Потрібно чи не потрібно пройти заново тест, який не працює тощо.

```
package edu.ukma.bdd.runner;

import ...

@RunWith(Courgette.class)
@CourgetteOptions(
        runLevel = CourgetteRunLevel.FEATURE,
        showTestOutput = true,
        rerunFailedScenarios = false,
        threads = 2,
        cucumberOptions = @CucumberOptions(
                features = {
                        "src/test/resources/feature/setup",
                                "src/test/resources/feature/ui",
                                "src/test/resources/feature/api"
                        },
                tags = "@smoke",
                dryRun = false,
                strict = true,
                monochrome=true,
                glue = {"edu.ukma.bdd.steps.ui",
                         "edu.ukma.bdd.steps.api"},
                plugin = {
                        "pretty",
                                "html:build/reports/cucumberreport/index",
                                "json:build/reports/cucumberreport/cucumber.json"}
                ))

public class RunnerBDDCourgette {

}
```

Рисунок 3.8 – RunnerBDDCourgette клас для запуску тестів

Перед тестами, необхідно розвернути базу Redis для того, щоб працювати з тестовими даними у паралельних потоках. Опис інструкції, як працювати з базою Redis є у файлі docker-compose.yml.

Існує також папка utils, яка містить додатковий функціонал, наприклад, такий як робота з базою Redis – клас RedisUtility. Завдяки даному класу і є можливим підключення та робота з самою базою (див. Рисунок 3.9).

```java
package edu.ukma.bdd.utils;

import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.Transaction;

public final class RedisUtility {

    private final static Logger LOGGER = LoggerFactory.getLogger(RedisUtility.class);

    private static JedisPool pool;
    private static Jedis jedis;

    private static boolean isRedisAlive;

    private RedisUtility() {
    }

    static {
        configureJedisPool();
    }

    private static void configureJedisPool() {
        try {
            LOGGER.info("Configuring jedis pool");
            pool = new JedisPool(System.getProperty("env.redisHost"), Constants.REDIS_PORT);
            jedis = pool.getResource();
            isRedisAlive = true;
        } catch (Exception e) {
            LOGGER.error("Failed to configure redis", e);
            isRedisAlive = false;
        }
    }

    public static String acquireUser(String userKey) {
        if (!isRedisAlive) {
            String userName = Constants.FALLBACK_USER;
            LOGGER.warn("Returning fallback user {} as jedis is not configured", userName);
            return userName;
        }
```

Рисунок 3.9 – Опис класу RedisUtility

Конфігурація логування системи відбувається за допомогою LogBack бібліотеки з можливість додаткових налаштувань через resources -> logback.xml (див. Рисунок 3.10).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <!-- Stop output INFO at start -->
    <statusListener class="ch.qos.logback.core.status.NopStatusListener" />

    <!--  Console Appender -->
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>
                %d{yyyy-MM-dd HH:mm:ss} %-5p [%t] --- %c{1}:%L - %m %n
            </pattern>
        </encoder>
    </appender>

    <root level="info">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

Рисунок 3.10 - LogBack логування

3.1.3 Огляд компонентів для побудови автоматизованого тестування з точки зору інтерфейсу користувача

Деякі основні компоненти такі, як feature файл, тести, логування були описані у підрозділі 3.1.2. Розглянемо детально реалізацію компонентів для автоматизованого тестування графічного інтерфейсу (див. Рисунок 3.11).

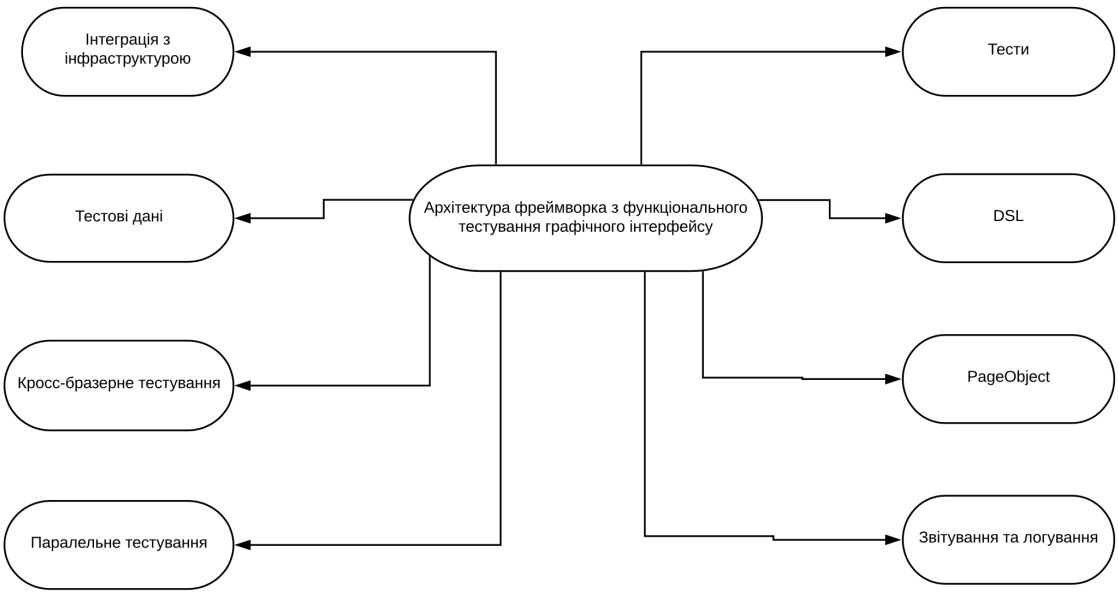


Рисунок 3.11 – Компоненти фреймворку для автоматизованого тестування інтерфейсу користувача

Фреймворк для автоматизованого тестування створений за допомогою PageObject підходу. Основна суть цього підходу полягає в тому, що окремо пишуть логіку для роботи з кожною сторінкою веб-сайту. Так, на кожен елемент сторінки (наприклад, кнопка, текст введення, тощо) визначають дії на цій сторінці. Тест сценаріїв (які представляють фактичні тестові випадки) можуть використовувати один чи декілька об'єктів (англійською PageObject). Як це працює можна знайти на Рисунку 3.12.

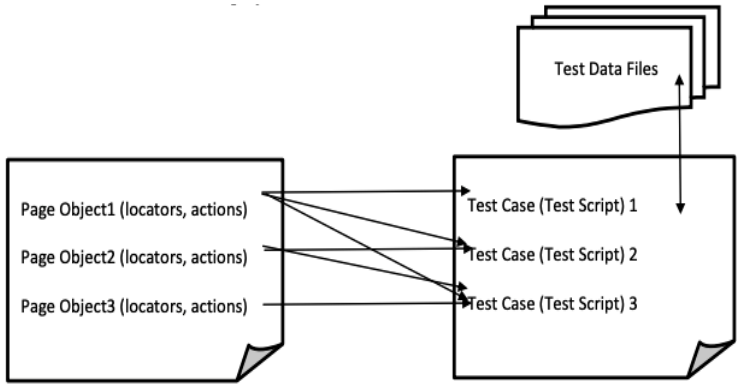


Рисунок 3.12 – Логіка взаємодії тестових сценаріїв та об’єктів сторінок

Цей підхід дуже зручний для написання автоматизованих тестів. Якщо, наприклад, дизайн однієї зі сторінок змінюється, то потрібно буде переписати лише відповідний головний клас, що описує цю сторінку.

Основні переваги:

- Поділ коду тестів і опису сторінок;
- Об'єднання всіх дій для роботи з веб-сторінкою в одному місці.

Детальніше розглянемо структуру папки page, у якій знаходяться описи класів. Її розташування та зміст можна знайти на Рисунку 3.13

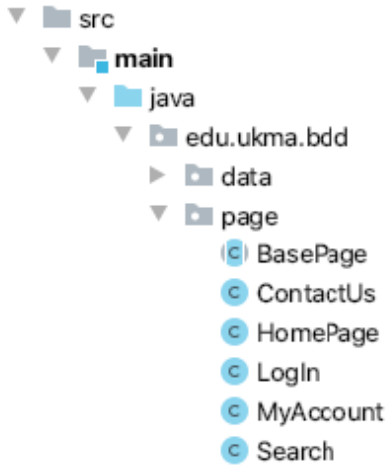


Рисунок 3.13 - Page Objects розміщення

Ця папка складається з класів, які місятять опис сторінки з точки зору функціоналу:

- BasePage – головний клас, в якому прописується базовий набір типових методів, що можуть бути використані іншими класами;
- ContactUs – клас, що унаслідується від BasePage та описує логіку роботи з сторінкою Contact Us;
- HomePage - клас, що унаслідується від BasePage та описує логіку роботи з сторінкою Home Page;
- LogIn - клас, що унаслідується від BasePage та описує логіку роботи з сторінкою Log In;
- MyAccount - клас, що унаслідується від BasePage та описує логіку роботи з сторінкою My Account.
- Search - клас, що унаслідується від BasePage та описує логіку роботи з сторінкою Search.

Розглянемо для прикладу клас Search (див. Рисунок 3.14).

```
package edu.ukma.bdd.page;

import java.util.List;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Search extends BasePage {

    private final static Logger LOGGER = LoggerFactory.getLogger(Search.class);

    public Search(WebDriver webDriver) { super(webDriver); }

    public boolean verifyProduct(String expectedProduct) {
        boolean isExpectedProduct = false;
        List<WebElement> productList = getArrayOfElementByCSS( webElementKey: "searchResultListCSS");
        LOGGER.info("Expecetd product is {}", expectedProduct);
        for (WebElement product : productList) {
            LOGGER.info("Product title is {}", product.getText());
            if (product.getText().contains(expectedProduct)) {
                isExpectedProduct = true;
            }
        }
        return isExpectedProduct;
    }
}
```

Рисунок 3.14 – Опис класу Пошук

Спочатку ми імпортуємо бібліотеку Selenium для запуску та роботи з різними веб-браузерами. Також клас унаслідує від базового класу Base Page універсальні методи та містить додаткові методи для роботи на сторінці Search. Методи у свою чергу описують дії та локатори за допомогою яких і здійснюється активність на сторінці пошуку та сам пошук.

Опис тестів знаходиться у окремій папці test, яка в свою чергу містить автоматизовані тести (див. Рисунок 3.15), які виконуються на підставі описаних кроків сценаріїв у файлах типу feature.

```
package edu.ukma.bdd.steps.ui;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import org.junit.Assert;

import edu.ukma.bdd.page.HomePage;

public class SearchTest {

    private UIScenarioContext context;

    public SearchTest(UIScenarioContext context) { this.context = context; }

    @Given("^User is browsing 'Automation Practice' website$")
    public void user_is_browsing_Automation_Practice_website() { context.homePage = new HomePage(context.getWebDriver()); }

    @When("^User clicks \"([^\"]*)\" on \"([^\"]*)\"$")
    public void user_on_on_on_ecom(String clickableField, String pageName) {
        if (pageName.equals("Homepage")) {
            switch (clickableField) {
            case "Contact us":
                context.contactUs = context.homePage.clickOnContactUsTab();
                break;
            case "Sign In":
                context.logIn = context.homePage.clickOnSignInTab();
                break;
            default:
                throw new RuntimeException("Invalid option " + clickableField);
            }
        }
    }

    @Then("^User lands on \"([^\"]*)\" Page$")
    public void user_lands_on(String arg1) throws Throwable {
        boolean isOnPage = false;
        switch (arg1) {
        case "automationpractice.com/index.php?controller=contact":
            isOnPage = context.contactUs.isOnPage(arg1);
            break;
        case "automationpractice.com/index.php?controller=authentication&back=my-account":
            isOnPage = context.logIn.isOnPage(arg1);
            break;
        case "automationpractice.com/index.php?controller=my-account":
            isOnPage = context.myAccount.isOnPage(arg1);
            break;
```

Рисунок 3.15 – Опис тестових сценаріїв на сторінці пошуку

У тестах, зображених на рисунку вище, імпортується сама бібліотека Cucumber, яка потрібна для роботи з відповідним функціоналом, а також написання тестових сценаріїв.

Приклад як виглядає функціонал Search у feature файлі можна знайти на Рисунку 3.16.

```
Feature: Feature to test searching scenarios

@ui @search @ui-001 @smoke
Scenario:
    Given User is browsing 'Automation Practice' website
    When User searches for "Blouse"
    Then User lands on "automationpractice.com/index.php?controller=search" Page
    And Verifies that the results contain the word "Blouse"
```

Рисунок 3.16 – Сценарій для тестування функціоналу (feature)

Селекторів, за допомогою яких відбувається пошук елементів на сторонках, система берез із заздалегідь підготовленого файлу selectors.properties (див. Рисунок 3.17 та Рисунок 3.18).

Рисунок 3.17 – Розташування файлу selectors.properties

```
#Automation Practice Home Page
contactUsTabCSS=#contact-link > a
signInTabCSS=#header > div.nav > div > div > nav > div.header_user_info > a
searchBarCSS=#search_query_top
searchButtonCSS=#searchbox > button

#Automation Practice Sign In Page
emailAddressCSS=#email
passwordCSS=#passwd
signInButtonCSS=#SubmitLogin > span
authenticationFailedAlertMessageCSS=#center_column > div.alert.alert-danger > p
registerEmailAddressCSS=#email_create
createAccountButtonCSS=#SubmitCreate
titleChoiceCSS=#account-creation_form > div:nth-child(1) > div.clearfix > div:nth-child(3) > label
firstnameCSS=#customer_firstname
lastnameCSS=#customer_lastname
addressFirstnameCSS=#firstname
addressLastnameCSS=#lastname
addressCSS=#address1
cityCSS=#city
stateSelectCSS=#id_state
postalCodeCSS=#postcode
mobileNumberCSS=#phone_mobile
registerButtonCSS=#submitAccount

#Automation Practice Search result page
searchResultListCSS=#center_column > ul > li > div > div.right-block > h5 > a
```

Рисунку 3.18 – Збережені селектори у файлі selectors.properties

Тестові дані, що використовуються для створення тестових користувачів (у випадку коли їх не існує) та збереження їх до бази Redis збережені у файлі testdata.properties (див. Рисунок 3.19).

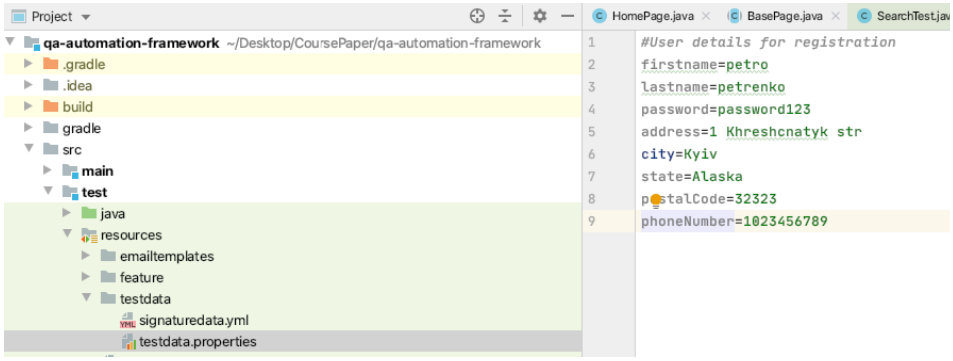


Рисунок 3.19 - testdata.properties файл з тестовими даними

Для роботи безпосередньо з різними веб-браузерами, у папці utils написаний клас DriverUtility (див. Рисунок 3.20). Даний клас дозволяє створювати браузери з різними налаштуваннями (приватний режим, headless).

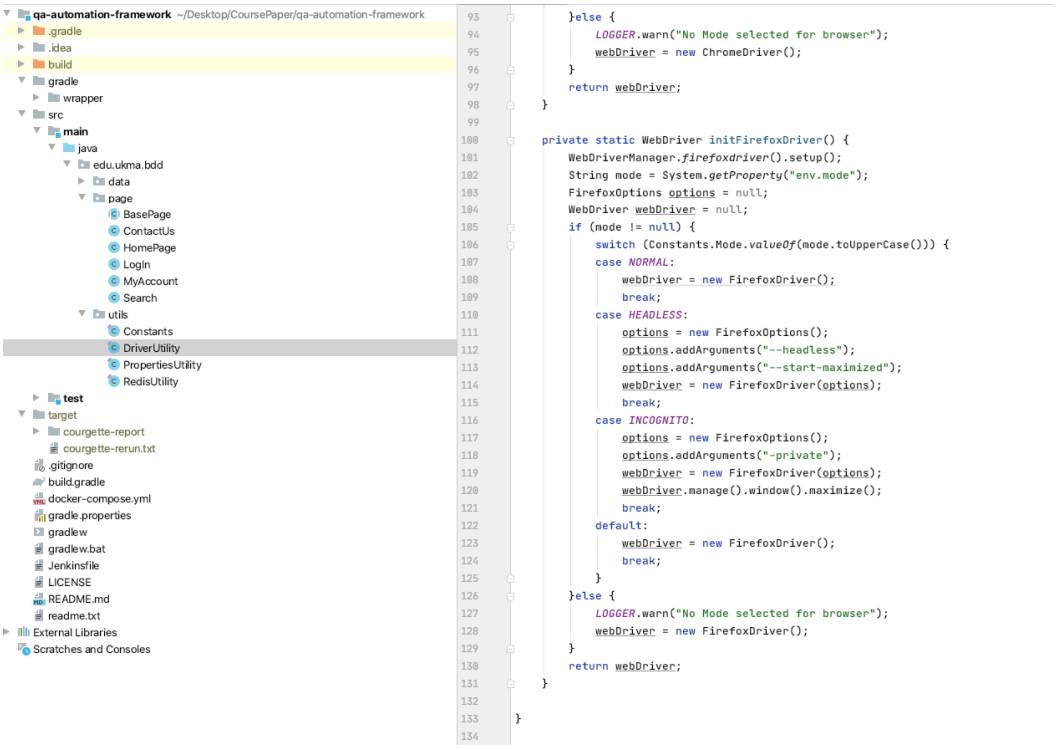


Рисунок 3.20 – Опис класу DriverUtility

3.1.4 Огляд компонентів для побудови автоматизованого тестування з точки зору веб-сервісів (API)

Система включає наступні основні компоненти для автоматизації веб-сервісу API (див. Рисунок 3.21).

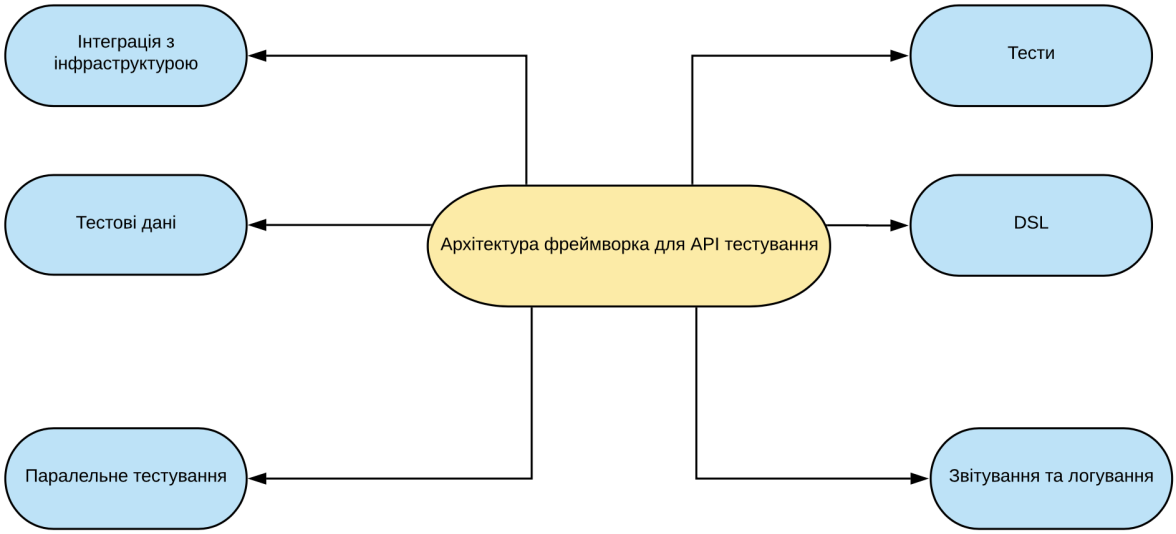


Рисунок 3.21 - Компоненти фреймворку для автоматизованого тестування веб-сервісів (API)

Як можна побачити з Рисунку 3.21, компоненти для роботи з API тестуванням багато в чому схожі з аналогами для автоматизації UI тестування. Водночас PageObject підхід для API тестів тут не був реалізований.

Тести API також написанні на основі поведінкового тестування з використанням синтаксису Gherkin. Розглянемо приклад таких тестів для перевірки сервісу прогнози погоди, а саме такі тестові сценарії:

- Перевірка функціоналу API для видачі прогнозу погоди по двом містам України – Kyiv та Lviv;
- Перевірка на некоректну назву (див. Рисунок 3.23).

```
Feature: Test Weather API.

Background:
    Given API baseUrl "https://restapi.demoqa.com/utilities/weather/city/"

@api @api-001 @smoke
Scenario Outline:
    Given City of "<city>"
    And invoke weather API
    Then verify the response code is "<code>" and body contains city name "<verifyCity>"
    Examples:
        | city        | code | verifyCity |
        | kyiv      | 200  | Kyiv       |
        | lviv      | 200  | Lviv       |
        | incorrect | 400  |            |
```

Рисунок 3.23 – Сценарії BDD для тестування API сервісу

Функціонал самих BDD кроків описаний в файлі, що знаходиться у src -> test -> api (див. Рисунок 3.24). Детально даний функціонал ми розглянули у попередніх розділах.

```
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;

import io.restassured.response.Response;
import io.restassured.specification.RequestSpecification;

public class APITest {

  private String givenCity;
  private RequestSpecification request;
  private Response response;
  private String baseUrl;

  public APITest() {

  }

  @Given("API baseUrl \"([^\"]*)\"$")
  public void api_url(String baseUrl) { this.baseUrl = baseUrl; }

  @Given("^City of \"([^\"]*)\"$")
  public void city_of(String city) { this.givenCity = city; }

  @Given("^invoke weather API$")
  public void invoke_weather_api() {
    this.request = given().pathParam( parameterName: "city",givenCity);
    this.response = this.request.when().
                    get( path: this.baseUrl+ "{city}");
  }

  @Then("^verify the response code is \"([^\"]*)\" and body contains city name \"([^\"]*)\"$")
  public void verify_response(int code, String city) {
    this.response.then().
       assertThat().
       statusCode(code).
       body( path: "City", city.trim().isEmpty() ? Matchers.nullValue() : Matchers.equalTo(city));
  }

}
```

Рисунок 3.24 – Функції для реалізації кроків BDD сценаріїв автоматизації API

3.2 Впровадження системи автоматизованого тестування у Jenkins

Запустити тести та отримати результати можна двома способами:

- Вручну;
- Автоматично за допомогою Jenkins.

Автоматичний запуск відбувається за допомогою сервісу Jenkins. Конфігурація самого запуску тесту виконана за допомогою Jenkinsfile таким чином, що дозволяє перед стартом вказати:

- гілку репозиторія, де знаходиться тест;
- вид браузера;
- режим браузера;
- який саме тест за тегом (@smoke, @regression).

Приклад самого Jenkinsfile показаний на Рисунку 3.25

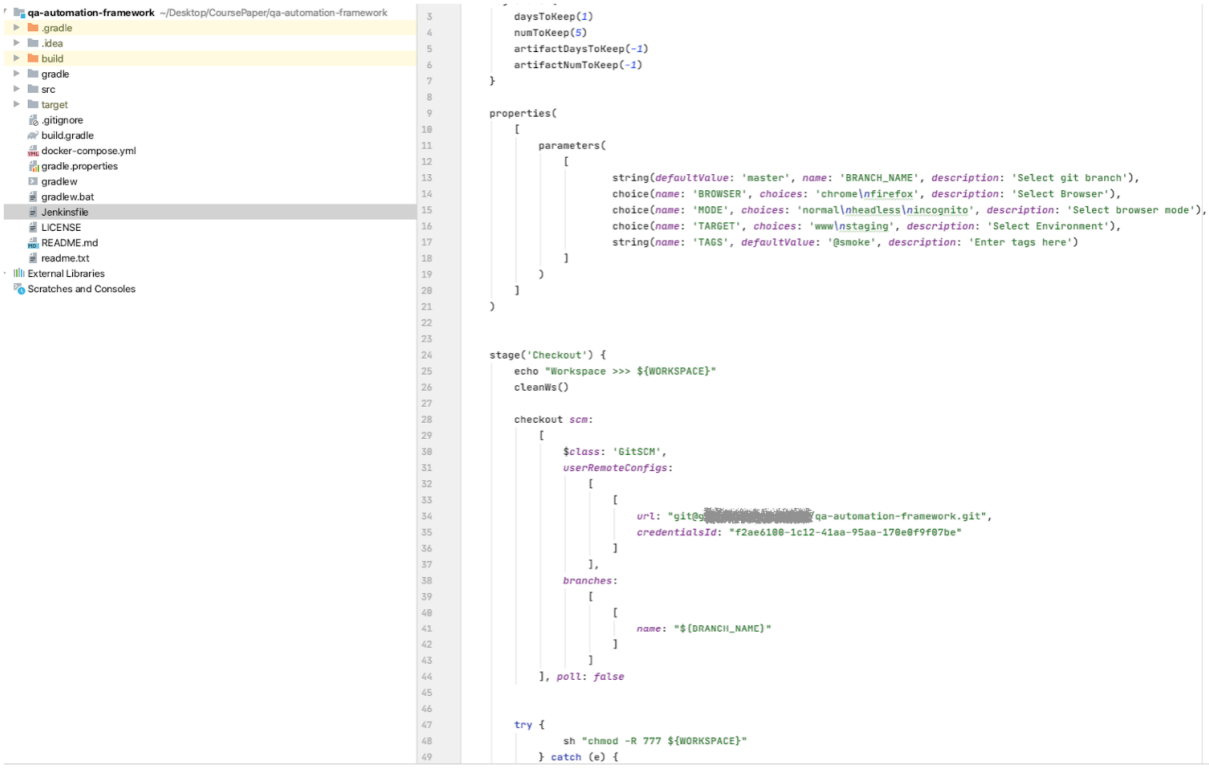


Рисунок 3.25 - Jenkinsfile конфігурація

Давайте подивимось, як відбувається взаємодія між компонентами системи та як організований потік (pipeline) запуску тестів і отримання результатів. Jenkins отримує тести з GitHub сховища. Потім запускає процес збірки проекту за допомогою gradle, який в свою чергу запускає Cucumber та Courgette. Останній дозволяє запустити тести у декілька потоків. По

закінченню виконання тестування, зберігається автоматичний звіт на базі інструментарію бібліотеки Cucumber. Весь описаний процес зображений на Рисунку 3.26.

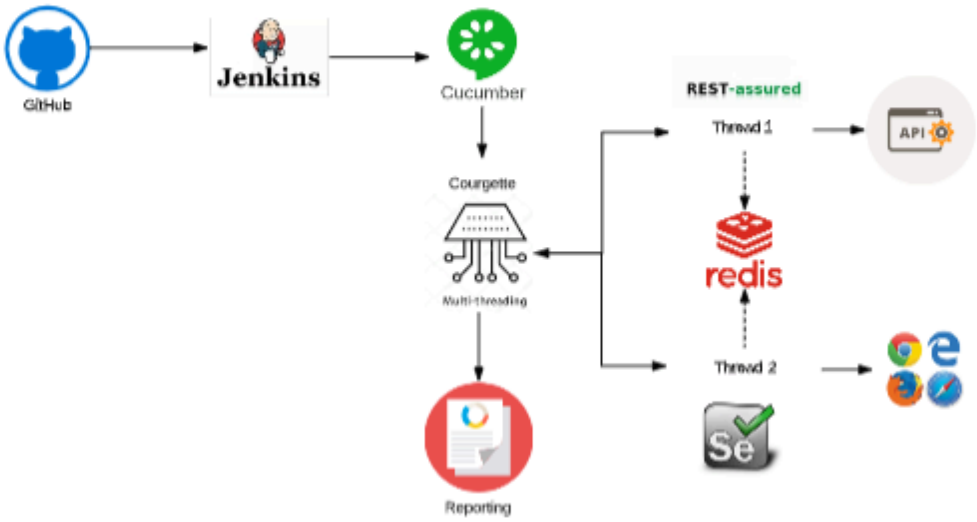


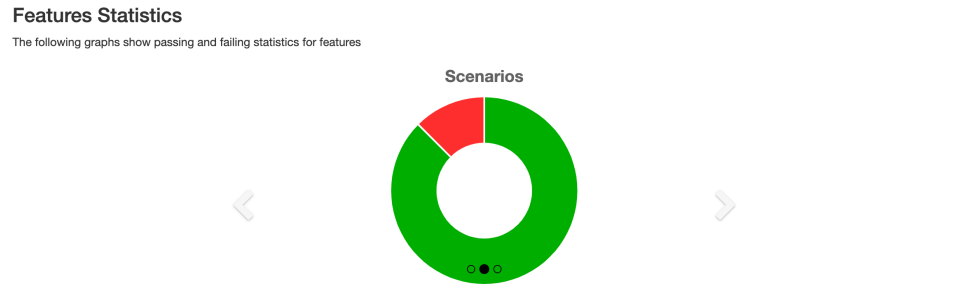
Рисунок 3.26 – Jenkins pipeline Архітектура взаємодії компонентів

3.3 Візуалізація результатів тестування за допомогою Jenkins та Cucumber

Результати успішних та ні тестів можна отримати за допомогою бібліотеки Cucumber reports безпосередньо у системі Jenkins. Дана бібліотека генерує репорт по закінченню тестування на основі згаданих раніше тегів, наприклад @ui, @api.

Давайте розглянемо даний звіт детальніше.

- За функціоналом тестування



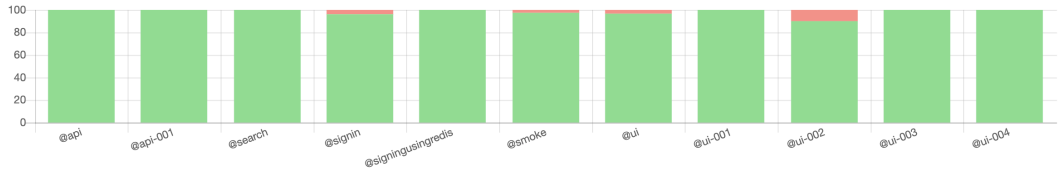
	Steps						Scenarios			Features	
Feature	Passed	Failed	Skipped	Pending	Undefined	Total	Passed	Failed	Total	Duration	Status
Feature to test searching scenarios	4	0	0	0	0	4	1	0	1	4.684	Passed
Feature to test sign in scenarios	25	1	0	0	0	26	3	1	4	36.592	Failed
Test Weather API.	12	0	0	0	0	12	3	0	3	5.903	Passed
	41	1	0	0	0	42	7	1	8	47.179	3
	97.62%	2.38%	0.00%	0.00%	0.00%		87.50%	12.50%			66.67%

Рисунок 3.27 – Результат тестування за функціоналам

- За окремим тегом

Tags Statistics

The following graph shows passing and failing statistics for tags



	Steps						Scenarios			Features	
Tag	Passed	Failed	Skipped	Pending	Undefined	Total	Passed	Failed	Total	Duration	Status
@api	9	0	0	0	0	9	3	0	3	5.900	Passed
@api-001	9	0	0	0	0	9	3	0	3	5.900	Passed
@search	4	0	0	0	0	4	1	0	1	4.684	Passed
@signin	25	1	0	0	0	26	3	1	4	36.592	Failed
@signingusingredis	16	0	0	0	0	16	2	0	2	22.689	Passed
@smoke	38	1	0	0	0	39	7	1	8	47.176	Failed
@ui	29	1	0	0	0	30	4	1	5	41.276	Failed
@ui-001	4	0	0	0	0	4	1	0	1	4.684	Passed
@ui-002	9	1	0	0	0	10	1	1	2	13.903	Failed

Рисунок 3.28 – Результат тестування по тегам

- По крокам тестування

Steps Statistics

The following graph shows step statistics for this build. Below list is based on results. step does not provide information about result then is not listed below. Additionally @Before and @After are not counted because they are part of the scenarios, not steps.

Implementation	Occurrences	Average duration	Max duration	Total durations	Ratio
edu.ukma.bdd.steps.api.APITest.api_url(java.lang.String)	3	0.001	0.001	0.003	100.00%
edu.ukma.bdd.steps.api.APITest.city_of(java.lang.String)	3	0.003	0.004	0.010	100.00%
edu.ukma.bdd.steps.api.APITest.invoke_weather_api()	3	1.706	2.886	5.119	100.00%
edu.ukma.bdd.steps.api.APITest.verify_response(int,java.lang.String)	3	0.257	0.716	0.771	100.00%
edu.ukma.bdd.steps.ui.SearchTest.user_is_browsing_Automation_Practice_website()	5	0.024	0.064	0.123	100.00%
edu.ukma.bdd.steps.ui.SearchTest.user_lands_on(java.lang.String)	11	0.006	0.009	0.071	90.91%
edu.ukma.bdd.steps.ui.SearchTest.user_on_on_on_ecom(java.lang.String,java.lang.String)	4	2.600	3.190	10.400	100.00%
edu.ukma.bdd.steps.ui.SearchTest.user_search_for(java.lang.String)	3	3.873	4.550	11.621	100.00%
edu.ukma.bdd.steps.ui.SearchTest.verifies_all_result_product_contain_the_word(java.lang.String)	3	0.070	0.094	0.210	100.00%
edu.ukma.bdd.steps.ui.SignInTest.inputs_email_address_and_password()	2	4.990	5.377	9.981	100.00%
edu.ukma.bdd.steps.ui.SignInTest.inputs_email_address_and_password(java.lang.String,java.lang.String)	2	4.435	4.614	8.870	100.00%
edu.ukma.bdd.steps.ui.UIScenarioContext.endMethod(io.cucumber.java.Scenario)	5	0.201	0.471	1.009	100.00%
edu.ukma.bdd.steps.ui.UIScenarioContext.startMethod(io.cucumber.java.Scenario)	5	5.517	6.405	27.585	100.00%
13	**52**	**1.457**	**27.585**	**1:15.773**	**Totals**

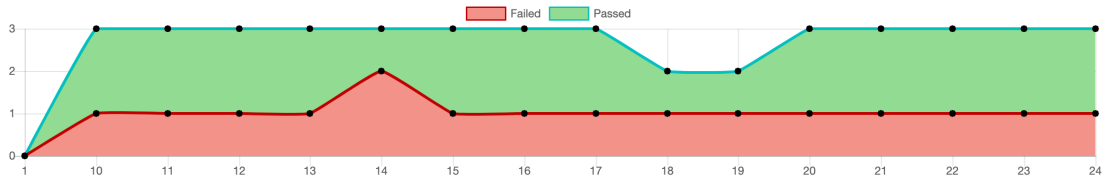
Рисунок 3.29 – Результат тестування по крокам

- Тренди успішності від запуску до запуску

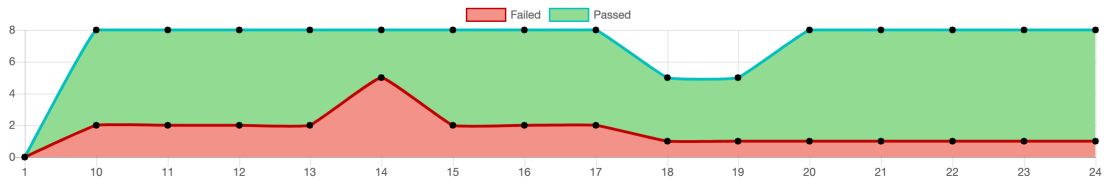
Trends Statistics

The following graph shows features, scenarios and steps for a period of time.

Features:



Scenarios:



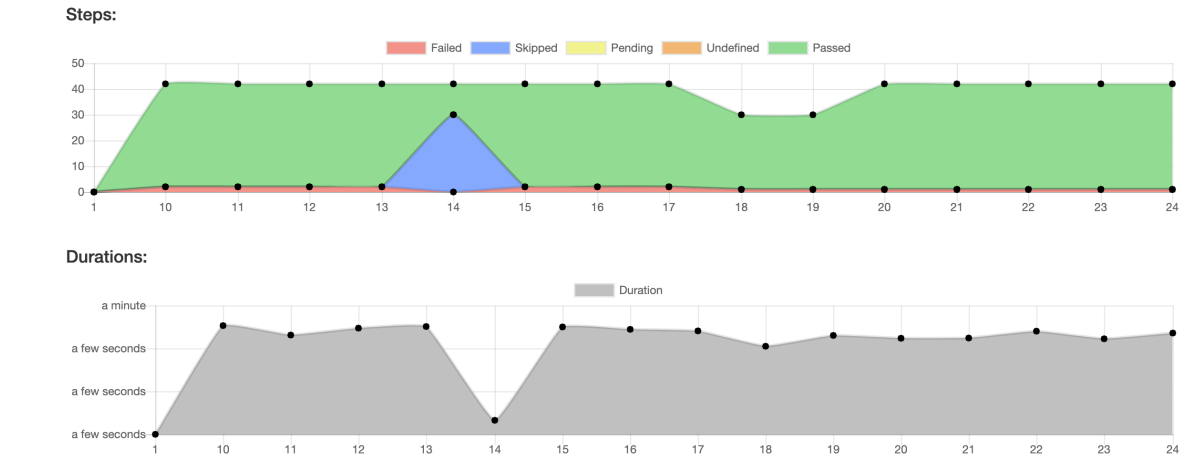


Рисунок 3.30 – Результат тестування у тренді

- Неуспішні тести

Рисунок 3.31 – Відображення деталей по неуспішним тестам

ВИСНОВКИ

Дослідивши концептуальні підходи до організації автоматизованого тестування та поєднавши розробку системи для її реалізації, можна з впевненістю стверджувати, що автоматизація тестування відіграє важливу роль в оптимізації безперервної поставки інформаційного продукту до кінцевого користувача. Швидкий зворотний зв'язок допомагає виявити функціональні помилки на ранніх етапах та підвищити загальний рівень якості системи. Таким чином, забезпечивши і високий рівень задоволення користувача.

Для опису та реалізації самого тестового функціоналу можна підійти різними шляхами. Але як показало дослідження здійснене під час написання даної курсової роботи, оптимальним рішенням є використання поведінкового підходу (BDD). Такий підхід дозволяє писати тести зрозумілою мовою для усіх сторін – бізнес та інженерна команда – швидко, покращуючи розуміння та комунікацію між членами команди. Наприклад, тестові сценарії можуть бути написані навіть представниками бізнесу у реальному часі, базуючись на їх вимоги.

Наступним важливим кроком є інтеграція системи автоматичного тестування з сервісом безперервної поставки коду. Саме ця інтеграція дозволяє швидко виявити та виправити потенційні помилки функціоналу.

Поєднання поведінкового підходу до написання тестів разом з інтеграцією CI системи дає змогу досягти синергічного ефекту в оптимізації затрат часу та швидкості зворотного зв’язку протягом тестування інформаційного продукту. Саме такий підхід був досліджений, описаний та реалізований у даній курсовій роботі.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Matt Wynne, Aslak Hellesoy. The Cucumber Book: Behaviour-Driven Development for Testers and Developers - Pragmatic Programmers, LLC, 2012 - p. 30 - 450
2. John Ferguson Smart. BDD in Action - Manning Publications Co, 2015 - p.18 - p.12 - 100, p. 161-166, p. 260-342
3. Enrique Amodeo.Learning Behaviour-driven Development with JavaScript - Packt Publishing, 2015 - p. 7 - 24
4. Graham Bath, Rex Black. STQB Advanced Level Test Automation Engineer - International Software Testing Qualifications Board, 2019
5. Lisa Grispin, Janet Gregory. A practical guide for testers and Agile teams - Pearson Education, Inc., 2009, p. 100- 380
6. Ben Rady, Rod Coffin. Continuous testing with Ruby. Rails and JavaScript - Pragmatic Programmers, LLC., 2011 - p.23 - 44
7. Paul M.Duval. Continuous Integration - Pearson Education, Inc, 2007 - p 40 - 86
8. Gradle build tool - 2020 - [Електронний ресурс] - Режим доступу https://gradle.org/
9. Jenkins. Build great things at any scale - 2020 - [Електронний ресурс] - Режим доступу https://www.jenkins.io/
10. The Selenium Browser Automation - 2020 - [Електронний ресурс] - Режим доступу https://www.selenium.dev/documentation/en/webdriver/
11. Rest Assured - 2020 - [Електронний ресурс] - Режим доступу https://github.com/rest-assured/rest-assured/wiki/GettingStarted
12. Redis - 2020 - [Електронний ресурс] - Режим доступу https://redis.io/
13. Cucumber JVM - 2020 - [Електронний ресурс] - Режим доступу https://github.com/cucumber/cucumber-jvm
14. Courgette JVM - 2020 - [Електронний ресурс] - Режим доступу https://github.com/rgov/courgette-build

15. Docker - 2020 - [Електронний ресурс] - Режим доступу https://www.docker.com/
16. JBehave - 2020 - [Електронний ресурс] - Режим доступу https://jbehave.org/
17. Gauge - 2020 - [Електронний ресурс] - Режим доступу https://gauge.org/