Міністерство освіти і науки України

Львівський національний університет імені Івана Франка

Факультет електроніки та комп’ютерних технологій

Кафедра радіоелектронних і
комп’ютерних систем

КУРСОВА РОБОТА

на тему: **«Автоматизація тестування веб-ресурсу за допомогою мови Java»**

Виконав:

студент групи ФеІм-14с

_____Береза І.В.

Науковий керівник:

_____Флюнт О.Є.

Львів 2023

АНОТАЦІЯ

Ця курсова робота присвячена автоматизації тестування веб-ресурсу за допомогою мови програмування Java. Робота досліджує можливості Java для створення автоматизованих тестів та їх застосування у веб-розробці.

У роботі розглянуті основні принципи та інструменти автоматизованого тестування веб-ресурсів, а також досліджені фреймворки та бібліотеки, доступні для мови Java, для реалізації тестових сценаріїв.

ANNOTATION

This coursework is devoted to the automation of web resource testing using the Java programming language. The work explores the possibilities of Java for creating automated tests and their application in web development.

The work examines the main principles and tools of automated testing of web resources, as well as researches the frameworks and libraries available for the Java language for the implementation of test scenarios.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

UI - User Interface (інтерфейс користувача)

API - Application Programming Interface (інтерфейс програмування додатків)

QA - Quality Assurance (забезпечення якості)

Unit test - модульний тест, який перевіряє функціональність окремого модуля або компонента

Integration test - інтеграційний тест, який перевіряє взаємодію між різними модулями або компонентами

Functional test - функціональний тест, який перевіряє, чи відповідає програма вимогам та специфікаціям

Regression test - регресійний тест, який перевіряє, чи не виникли нові дефекти після внесення змін

Test case - тестовий випадок, опис окремого тесту з певними вхідними даними і очікуваними результатами

Mock - підробка, об'єкт, що імітує поведінку справжнього об'єкта для ізоляції тестуваного коду

Assertion - перевірка, яка стверджує, що певна умова є істинною

Test fixture - набір підготовлених даних або стану, необхідних для виконання тестів

ЗМІСТ

ВСТУП 5
1. АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ 6
1.1 Що таке автоматизоване тестування 6
1.2 Переваги та недоліки автоматизованого тестування в порівнянні з ручним 6
1.3 Види автоматизованого тестування 7
1.3.1 GUI tests 8
1.3.2 Integration tests 8
1.3.3 Unit tests 9
1.4 Аналіз методів веб-тестування 10
2. ОГЛЯД ІНСТРУМЕНТІВ ДЛЯ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ 13
2.1 Огляд мови програмування Java та її особливості 13
2.1.1 Виникнення JAVA 13
2.1.2 Основні принципи мови програмування Java 15
2.1.3 Переваги використання мови програмування Java 15
2.2 JUnit: Визначення та загальний огляд 16
2.2.1 Основні компоненти JUnit 17
2.3 Mockito 18
3. ПРАКТИЧНА РЕАЛІЗАЦІЯ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ З ВИКОРИСТАННЯМ JUNIT 20
3.1 Опис веб-додатка 20
3.2 Написання тестів 22
ВИСНОВОК 26
Перелік використаних джерел 27

ВСТУП

У сучасному світі розробка веб-додатків є необхідною складовою успішного бізнесу. Якість цих додатків визначається їхньою функціональністю, надійністю та коректністю роботи. Однак, зі зростанням розміру та складності веб-ресурсів стає все складніше та часозатратно проводити ручне тестування, яке потребує значних зусиль та ресурсів.

Автоматизація тестування веб-ресурсів є потужним інструментом, який дозволяє забезпечити швидке, ефективне та надійне тестування. Вона дозволяє автоматизовано виконувати тести, перевіряти функціональність, стабільність та відповідність веб-додатку вимогам та специфікаціям. Однією з потужних мов програмування, яка використовується для автоматизації тестування, є Java.

Цей курсовий проект присвячений автоматизації тестування веб-ресурсу за допомогою мови Java. Ми детально розглянемо основні принципи та методики автоматизованого тестування, а також покроково побудуємо автоматизовані тести для веб-ресурсу з використанням популярного фреймворку JUnit.

1. АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ

1.1 Що таке автоматизоване тестування

Автоматизоване тестування - це процес використання спеціалізованих програмних засобів для виконання тестових сценаріїв і перевірки функціональності, надійності та продуктивності програмного забезпечення. У цьому процесі тестування виконується автоматично без прямого втручання людини.

Автоматизоване тестування може бути використане для виконання рутинних, повторюваних тестових завдань, що можуть забрати багато часу, якщо вони виконуються вручну. Це також дозволяє виявляти помилки, що можуть бути пропущені під час ручного тестування. [1]

Для автоматизованого тестування використовуються спеціальні інструменти, такі як фреймворки для написання тестових сценаріїв, засоби запису та відтворення дій користувача, інструменти для генерації тестових даних, засоби аналізу результатів тестування тощо.

1.2 Переваги та недоліки автоматизованого тестування в порівнянні з ручним

Переваги:

- Збільшена ефективність: Автоматизоване тестування дозволяє виконувати багато тестових сценаріїв швидше і ефективніше, порівняно з ручним тестуванням, що допомагає зменшити час тестування та покращити продуктивність команди.
- Покриття тестування: Автоматизовані тестові сценарії можуть охоплювати широкий спектр тестових випадків і складних сценаріїв, що дозволяє забезпечити високу якість продукту та виявити потенційні

проблеми, які можуть бути пропущені при ручному тестуванні.

- Повторюваність: Автоматизовані тести можуть бути виконані повторно без втрати якості та точності результатів. Це дозволяє швидко перевіряти, чи впливають зміни у коді або веб-сторінці на функціональність продукту.
- Раннє виявлення помилок: Автоматизоване тестування дозволяє виявляти помилки на ранніх етапах розробки, що дозволяє економити час і зусилля, необхідні для виправлення помилок пізніше у процесі розробки. [1]

Недоліки:

- Початкові витрати: Розробка та налаштування автоматизованих тестів вимагає часу та зусиль, особливо на початкових етапах. Це може бути значним фактором, особливо для проектів з обмеженими ресурсами.
- Складність підтримки: Зі зростанням функціональності продукту та змінами у веб-сторінці автоматизовані тестові сценарії можуть потребувати регулярного оновлення та підтримки. Це може бути витратним і вимагати досвідчених автоматизаторів.
- Необхідність людського втручання: Деякі тести можуть вимагати людського втручання або експертної оцінки. Автоматизовані тестові сценарії не завжди можуть відтворити таку поведінку, що обмежує їх використання у деяких сценаріях тестування.

1.3 Види автоматизованого тестування

Існує кілька рівнів автоматичного тестування:

- GUI tests;
- Integration tests;
- Unit tests;

1.3.1 GUI tests

Автоматизація GUI (графічного інтерфейсу користувача) відноситься до процесу автоматизації тестування, який передбачає створення скриптів або програм, що відтворюють дії користувача на інтерфейсі програмного забезпечення. Цей підхід дозволяє автоматично перевіряти функціональність, надійність та коректність роботи програми через її графічний інтерфейс.

В процесі GUI-автоматизації, спеціальні тестові фреймворки або інструменти використовуються для запису та відтворення дій користувача на екрані, таких як натискання кнопок, введення тексту, переміщення миші та інші взаємодії з елементами інтерфейсу. Це дозволяє автоматично виконувати низку тестів для перевірки правильності роботи програми в різних сценаріях та умовах.

GUI-автоматизація дозволяє ефективно виявляти помилки та проблеми в інтерфейсі користувача, забезпечує швидкий та повторюваний процес тестування і дозволяє зосередитися на інших аспектах тестування, таких як функціональність та взаємодія з базою даних або серверами.

1.3.2 Integration tests

Integration tests (інтеграційні тести) відносяться до типу тестування, яке перевіряє взаємодію між різними компонентами або модулями програмного забезпечення. Цей вид тестування спрямований на перевірку правильності взаємодії між цими компонентами та виявлення можливих проблем або помилок, які можуть виникнути на стику їхніх функціональностей.

Під час проведення інтеграційних тестів, різні частини програмного забезпечення об'єднуються та тестуються як єдиний функціональний блок. Це

можуть бути, наприклад, взаємодіючі сервіси, модулі, компоненти, бази даних або зовнішні API. Тестування включає в себе передачу вхідних даних до системи та перевірку правильності виходу або взаємодії між компонентами.

Головна мета інтеграційних тестів полягає в перевірці того, що окремі компоненти працюють разом правильно, згідно з очікуваними специфікаціями та вимогами до системи. Це допомагає виявити проблеми, такі як несумісність між компонентами, проблеми з інтерфейсами, помилки передачі даних або недостовірні результати. Інтеграційні тести зазвичай проводяться після модульного тестування окремих компонентів, але перед системним тестуванням всієї системи.

1.3.3 Unit tests

Unit tests (модульні тести) відносяться до типу тестування, яке перевіряє правильність роботи окремих модулів або функцій програмного забезпечення. Вони фокусуються на найменших можливих одиницях коду, таких як функції, методи чи класи, і перевіряють їхню коректність та відповідність очікуваному результату.

Модульні тести зазвичай виконуються незалежно від інших частин програми, і для них можуть бути використані фейкові (mock) об'єкти або інші спеціальні механізми, що допомагають ізолювати модуль від залежностей. Це дозволяє перевірити, чи працює модуль правильно незалежно від інших компонентів системи.

Метою модульних тестів є переконатися, що кожен модуль програмного забезпечення працює ізольовано та правильно відповідає на вхідні дані, виконує очікувані обчислення та повертає очікувані результати. Вони допомагають виявляти програмні помилки, забезпечують стабільність та якість коду, а також сприяють легкості розробки та підтримки програмного забезпечення.

Модульні тести зазвичай розробляються розробниками програмного забезпечення на ранній стадії розробки, і вони часто автоматизовані, що дозволяє їх виконувати при кожній зміні коду для виявлення можливих проблем швидко та ефективно. [2]

1.4 Аналіз методів веб-тестування

До сучасних методів вебтестування за рівнями відносяться такі:

– модульне тестування (Unit Testing). Цей тип тестування проводиться розробниками до того, як збірка буде передана команді тестувальників для офіційного виконання тест кейсів. Модульне тестування проводиться відповідними розробникам. Розробники використовують тестові дані, окремі від тестових даних групи з забезпечення якості.

Мета модульного тестування – виділити кожну частину програми та показати, що окремі частини працюють правильно з точки зору вимог та функціональності.

Модульне тестування не може знайти кожну помилку програми. Неможливо оцінити кожен шлях виконання у кожному програмному застосуванні. Те саме стосується одиничного тестування. Існує обмеження на кількість сценаріїв і тестових даних, які розробник може використовувати для перевірки програмного коду. Тож після того, як він вичерпав усі варіанти, не залишається іншого вибору, як зупинити тестування модуля та об’єднати сегмент коду з іншими;

– інтеграційне тестування (Integration Testing). Тестування комбінованих частин програми, щоб визначити, чи правильно вони працюють разом, називається інтеграційним тестуванням. Також перевіряється взаємодії застосунку з зовнішніми системами. Існує два методи проведення тестування інтеграції: тестування знизу вгору та тестування інтеграції зверху вниз.

Тестування інтеграції знизу вгору починається з тестування окремих

одиниць, після чого поступово випробовуються поєднання вищих рівнів одиниць, що називаються модулями або збірками.

У тестуванні інтеграції зверху вниз, модулі вищого рівня перевіряються спочатку, а модулі нижчого рівня тестуються після цього. У всеосяжному середовищі розроблення програмного забезпечення зазвичай спочатку проводять тестування знизу вгору, після чого – тестування зверху вниз;

– системне тестування (System Testing). Це наступний рівень тестування – тестування системи в цілому. Після інтеграції всіх компонентів застосунок у цілому ретельно перевіряється на предмет того, що він відповідає стандартам якості. Цей тип тестування проводиться спеціалізованою групою тестування.

– регресійне тестування (Regression Testing). Щоразу, коли відбувається зміна в певній області коду програмного застосунка, це може вплинути й на інші області коду в програмі. Щоб перевірити, що виправлена помилка не призвела до порушення інших функціональних можливостей або правил, проводиться регресійне тестування. Його метою є забезпечити впевненість, що зміни, такі як виправлення помилок, не призвели до появи іншої помилки в застосунку.

– приймальне тестування (Acceptance Testing). Це найважливіший вид тестування, оскільки він проводиться командою із забезпечення якості, яка перевіряє, чи відповідає програма заявленим специфікаціям та чи відповідає вимогам клієнта. Команда QA матиме набір попередньо написаних сценаріїв 23 та тестових випадків, які будуть використані для тестування програми. Приймальне тестування призначене не лише для того, щоб вказати на прості орфографічні помилки, косметичні помилки або прогалини в інтерфейсі, але також вказати на будь-які помилки застосунку, які призведуть до збоїв у роботі системи або великих помилок у програмі. Виконуючи приймальні тести на системі, тестувальна група визначить, як буде працювати програма під час постійного використання. Існують також юридичні та договірні вимоги щодо прийняття системи в цілому.

– альфа-тестування. Цей тест є першим етапом тестування і проводитиметься серед команд (розробників та команд із забезпечення якості).

Тестування модулів, тестування інтеграції та тестування системи у поєднанні відомі як альфа-тестування. Під час цієї фази в застосунку буде перевірено наступне: орфографічні помилки, несправні посилання, застосунок буде протестовано на машинах з найнижчою специфікацією для перевірки часу завантаження та будь-яких проблем із затримкою; [1]

– бета-тестування. Цей тест виконується після успішного альфа тестування. При бета-тестуванні частина ймовірних користувачів тестує застосунок. Бета-тестування також відоме як тестування перед випуском. Бета-тестові версії програмного забезпечення інколи розподіляються серед широкої аудиторії в Інтернеті, частково для того, щоб виконати тест в реальних умовах. Якщо розроблюване програмне забезпечення призначене для внутрішнього користування в певній компанії, то для тестування залучають співробітників замовника;

– нефункціональне тестування. Цей вид заснований на тестуванні програми на основі її нефункціональних атрибутів. Нефункціональне тестування програмного забезпечення передбачає тестування програмного застосунку на основі вимог, які не є функціональними за своєю суттю, але також важливі, такі як продуктивність, безпека, користувальницький інтерфейс тощо;

– тестування продуктивності. Тестування продуктивності здебільшого використовується для виявлення будь-яких вузьких місць або проблем з продуктивністю, а не для пошуку помилок у програмному забезпеченні. Існують різні причини, які сприяють зниженню продуктивності програмного забезпечення.

2. ОГЛЯД ІНСТРУМЕНТІВ ДЛЯ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ

2.1 Огляд мови програмування Java та її особливості

JAVA — це обєктно–орієнтована мова програмування, розроблена Джеймсом Ослингом і запущена в 1995 році в якості одного із ключових компонентів платформи JAVA Sun Microsystems.

Синтаксис мови взятий в більшості з мови програмування C++ і C, про те JAVA має більш простішу обєктивну модель і більш низький рівень обєктів. JAVA – додатки як правило трансліровані в байт–код (довжина кожного коду операції — один байт). Який може бути оброблений любою віртуальною машиною JAVA (Java Virtual Mashine. JVM) незалежно від компютерної архітектури.

Першочергово реалізація посилань на компілятори JAVA, віртуальних машин і бібліотек класів, була представлена компанією Sun в 1995 році. В травні 1997 року разом з JAVA Communitz Process Sun розробила технологію JAVA згідно з GNU General Public License. Інші компанії також шукали альтернативи технології запропонованої Sun як GNU Compiler, для JAVA і GNU Classpath. [3]

2.1.1 Виникнення JAVA

Джеймс Гослінг почав розробляти проект мови програмування JAVA в липні 1991 року, для використання його в одному із своїх багаточисельних проектів set – top box. Мова спочатку називалась Oak “Дуб” на честь дуба, який ріс перед офісом Гослінга, але в кінці вибір був зупинений на JAVA, назва була вибрана із списку випадковим чином.

Гослінг вирішив запропонувати як додаток до віртуальної машини, яка буде мати стиль C C++. Sun випустила свій продукт під іменем JAVA 1.0. Девіз звучав (пишеш один раз запускаєш всюди). Слід відзначити і налаштовану безпеку, яка дозволяє використовувати межу на рівні файлів доступа.

Більшість веб–браузерів володіли можливістю запускати JAVA– аплети (програмні компонети в двоїчному коді які виконуються в вікні браузера) на веб–сторінках. Завдяки всьому цьому JAVA в дуже короткий час стала дуже популярною мовою.

В грудні 1998 року появилась JAVA 2. Нова версія пропонувала більшість конфігурацій, створених спеціально для різних типів платформ, наприклад JDEE додаток типу enterprise в той же час як stripped+down Sun ME був придуманий для мобільних платформ.

Sun перейменувала нові версії JD2 наступним чином: JAVA EE; JAVA ME; JAVA SE;

JAVA залишається стандартом, яким керується JAVA Community Process (процес який дозволяє заінтересованим лицям приймати участь у формуванні майбутніх специфікацій JAVA). Sun запропонувала більшість частин JAVA безкоштовно, незалежно на статус власника програмного забезпечення. Доходи від JAVA поступають за рахунок продажі спецалізованих продуктів, таких як Enterprise Java System.

13 листопада 2006 року Sun запустила декілька програм JAVA в виді програмного забезпечення, з відкритим вихідним кодом під ліцензією GNU General Public License (ліцензія на вільне програмне забезпечення).

8 травня 2008 року Sun завершила процес, який зробив придатним для використання всі коди ядра JAVA, з умовою розпоширення вільного програмного забезпечення з відкритим кодом, крім невеликої частини коду, на яку Sun розширює свої права.

2.1.2 Основні принципи мови програмування Java

Основні принципи мови програмування Java включають інкапсуляцію, наслідування та поліморфізм. Ці принципи дозволяють розробникам створювати складні програми з великою кількістю об’єктів, які взаємодіють між собою.

- Інкапсуляція – це принцип, який дозволяє захистити дані від зовнішнього доступу та зберегти конфіденційність. Дані у класі можуть бути приховані та доступні лише через методи цього класу.
- Наслідування – це принцип, який дозволяє створювати нові класи на основі вже існуючих класів. Новий клас, який наслідується від існуючого, може містити нові методи та властивості.
- Поліморфізм – це принцип, який дозволяє використовувати один і той же метод для різних об’єктів. Це означає, що метод може мати різну реалізацію в залежності від типу об’єкта, з яким він взаємодіє.

2.1.3 Переваги використання мови програмування Java

Java має багато переваг у порівнянні з іншими мовами програмування, такими як C++, Python, або JavaScript.

- Кросплатформенність – як уже зазначалось, програми, написані на мові програмування Java, можуть працювати на будь-якій операційній системі, яка має встановлену віртуальну машину Java (JVM). Це означає, що розробники можуть написати програми один раз, а потім запускати їх на будь-якому пристрої, що підтримує JVM, без необхідності в написанні окремої версії для кожної операційної системи.
- Безпека – Java має вбудовану систему безпеки, яка дозволяє запобігти вразливостям та атакам на програми. Це досягається за

допомогою віртуальної машини Java, яка запускає код в безпечному середовищі та перевіряє його на відповідність правилам безпеки.

- Велика спільнота розробників – Java має велику спільноту розробників та користувачів. Це означає, що у вас завжди є доступ до великої кількості інформації, підтримки та різноманітних бібліотек, які допоможуть вам в розробці програм.
- Велика кількість бібліотек та фреймворків – Java має велику кількість бібліотек та фреймворків, які значно спрощують розробку програм. Ці інструменти дозволяють вам використовувати готовий код та функціонал, що зменшує кількість написаного коду та скорочує час розробки програм.
- Об’єктно-орієнтований підхід – Java є мовою програмування з об’єктно-орієнтованим підходом. Це означає, що весь код написаний в Java складається з об’єктів, що значно спрощує розробку програм та підтримку коду в майбутньому.

2.2 JUnit: Визначення та загальний огляд

JUnit - це фреймворк з відкритим кодом, який використовується для написання та виконання модульних тестів мовою програмування Java. Це одна з найвідоміших платформ модульного тестування.[2]

JUnit, один з найпопулярніших фреймворків для тестування в мові Java, має цікаву історію та походження. Розробка JUnit почалася у 1997 році під керівництвом Кента Бека (Kent Beck) та Еріка Гамма (Erich Gamma). Вони створили JUnit як інструмент для автоматизованого тестування програмного забезпечення в рамках методології розробки програмного забезпечення Extreme Programming (XP).

У початкових версіях JUnit було використано підхід, відомий як "xUnit" - шаблон для фреймворків модульного тестування, який заснував Кент Бек.

Концепція "xUnit" полягає в створенні набору тестів, які перевіряють поведінку окремих модулів програми.

JUnit став одним з найпоширеніших фреймворків для тестування в світі Java завдяки своїй простоті використання та потужним можливостям. Він дозволяє розробникам швидко створювати тести та отримувати звіти про результати виконання. Крім того, JUnit постійно розвивається і оновлюється, додаючи нові функції та покращуючи його функціональні можливості.

З часом JUnit став стандартом у галузі тестування Java-програм та знайшов своє застосування у різних проектах та компаніях. Він інтегрується з іншими інструментами розробки та системами збирання проектів, такими як Apache Maven і Gradle, що сприяє зручному використанню та автоматизації тестування.[4]

2.2.1 Основні компоненти JUnit

- Аннотації: JUnit надає різні аннотації, які можна використовувати для позначення методів, які повинні бути виконані як тести. Наприклад, аннотація @Test вказує на те, що метод є тестовим.
- Assert-методи: JUnit надає різні методи для перевірки очікуваних результатів. Наприклад, метод assertEquals() порівнює два значення на рівність, а метод assertTrue() перевіряє, чи вираз є істинним.
- Тестові сюїти: JUnit дозволяє групувати тести в спеціальні об'єкти, відомі як тестові сюїти. Це дозволяє логічно організовувати тести та виконувати їх разом.
- Тестові запускачі: JUnit надає запускачі, які автоматично виконують тести в заданому порядку і збирають результати. Вони також надають можливість виконувати додаткові дії до і після тестів, такі як налаштування тестових умов або очищення ресурсів.

- Правила: JUnit дозволяє використовувати правила, які дозволяють налаштовувати середовище тестування або виконувати певні дії перед або після кожного тесту. Наприклад, правило TemporaryFolder створює тимчасовий каталог для використання в тестах.

JUnit має велику спільноту користувачів та широке використання у сфері розробки програмного забезпечення на Java. Він інтегрується з різними інструментами розробки та засобами для автоматизованого виконання тестів, такими як Maven, Gradle і IntelliJ IDEA. Крім того, JUnit підтримує розширення та налаштування, що дозволяє використовувати його для різних потреб тестування.[5]

Висновуючи, JUnit є потужним інструментом для модульного тестування Java-коду. Він дозволяє розробникам швидко писати тести, перевіряти правильність роботи коду та забезпечувати стабільність програмного забезпечення. Завдяки своїй простоті використання та широкому спектру функцій, JUnit є одним з найпопулярніших фреймворків для тестування Java-програм.

2.3 Mockito

Mockito - це бібліотека для тестування у програмуванні на Java, яка надає можливості для створення фейкових об'єктів (моків) для тестування коду, що залежить від інших об'єктів. Вона дозволяє створювати фейкові реалізації об'єктів залежностей (таких як бази даних, зовнішні сервіси або інші класи), щоб замінити їх під час виконання тестів.

За допомогою Mockito можна створювати мок-об'єкти, які поводяться як реальні об'єкти, але не мають реалізації реального функціоналу. Ви можете задавати поведінку мок-об'єктів, вказуючи, як вони повинні реагувати на виклики методів. Це дозволяє вам створювати контрольовані умови для

тестування різних сценаріїв без залучення реальних залежностей.

Mockito надає простий API для створення моків, встановлення очікувань, верифікації викликів методів та інших операцій, пов'язаних з тестуванням. Вона може бути використана в поєднанні з різними фреймворками для тестування, такими як JUnit або TestNG, для створення повноцінних тестів у вашому проекті.

Загалом, Mockito допомагає вам замінити реальні об'єкти на моки, що спрощує тестування, полегшує ізоляцію компонентів системи та допомагає виявляти та виправляти помилки.

3. ПРАКТИЧНА РЕАЛІЗАЦІЯ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ З ВИКОРИСТАННЯМ JUNIT

3.1 Опис веб-додатка

У цьому розділі ми дослідимо, як за допомогою фреймворка JUnit можна автоматизувати процес тестування. Ми напишемо тести для веб-ресурсу створеним з використанням фреймворку Spring . Програма представляє собою веб-додаток для керування книжковою бібліотекою, з можливістю отримання списку всіх книг або сортування за жанром через RESTful API. Вона використовує базу даних H2 для зберігання книжкових даних.

Основні компоненти цієї програми включають контролер ***(BookController)***, сервіс **(BookService)**, репозиторій **(BookRepository)** та сутність **(Book)**.

```
@RestController
@RequestMapping("/api/v1/books")
public class BookController {
```

Рис.3.1. Створення класу BookController з анотаціями ***@RestController*** і ***@RequestMapping*** для обробки HTTP-запитів на адресу ***/api/v1/books***

```
private final BookRepository bookRepository;

@Autowired
public BookServiceImpl(BookRepository bookRepository) {
  this.bookRepository = bookRepository;
}
```

Рис.3.2. Оголошення залежності від ***BookService*** через конструктор

```
@GetMapping
public List<Book> getList(@RequestParam(required = false) String genre)
  if (genre != null) {
    return bookService.getAllByGenre(genre);
  }

  return bookService.getAll();
}
}
```

Рис.3.2. Оголошення методу ***getList***, який обробляє GET-запити і повертає список книг залежно від заданого жанру або всі книги, якщо жанр не вказаний

```
spring:
  datasource:
    url: jdbc:h2:mem:library
    driver-class-name: org.h2.Driver
    username: sa
    password: sa
  jpa:
    database-platform: org.hibernate.dialect.H2Dialect
    defer-datasource-initialization: true
  h2:
    console:
      enabled: true
```

Рис.3.3. Фрагмент коду з файлу конфігурації ***application.yml*** для Spring

```
insert into book (title, author, genre, height, publisher)
values ('Fundamentals of Wavelets', 'Goswami, Jaideva', 'signal_processing', 228, 'Wiley'),
       ('Data Smart', 'Foreman, John', 'data_science', 235, 'Wiley'),
       ('God Created the Integers', 'Hawking, Stephen', 'mathematics', 197, 'Penguin'),
       ('Superfreakonomics', 'Dubner, Stephen', 'economics', 179, 'HarperCollins'),
       ('Orientalism', 'Said, Edward', 'history', 197, 'Penguin'),
       ('Nature of Statistical Learning Theory, The', 'Vapnik, Vladimir', 'data_science', 230, 'Springer'),
       ('Integration of the Indian States', 'Menon, V P', 'history', 217, 'Orient Blackswan'),
       ('Drunkard''s Walk, The', 'Mlodinow, Leonard', 'science', 197, 'Penguin');
```

Рис.3.4. Фрагмент коду з файлу ***data.sql***, який містить вставку даних до бази

3.2 Написання тестів

Ми напишемо тести для перевірки функціональності сортування книг за жанром у контролері ***BookController***. Точніше, ми перевіряємо, чи правильно повертається список книг з вказаним жанром при виклику методу ***getList()*** з параметром ***genre***.

```
@Test
public void shouldReturnBooksByGenre() {
```

Рис.3.2.1 Тут ми використовуємо аннотацію ***@Test***, яка позначає метод як тестовий, який потрібно виконати.

Далі використовуючи бібліотеку Mockito, ми встановлюємо очікування для методу ***findAll()*** об'єкта ***bookRepository***. Ми говоримо, що коли буде викликано метод **findAll()**, повернути список книг ***books***.

```
Mockito.when(bookRepository.findAll()).thenReturn(books);
```

Рис.3.2.2 Встановлення очікувань за допомогою Mockito

Тепер ми викликаємо метод ***getAllByGenre(genre***) на об'єкті ***bookService*** для отримання результату сортування книг за жанром.

```
List<Book> result = bookService.getAllByGenre(genre);
```

Рис 3.2.3 Виклик методу для тестування

За допомогою класу ***Assertions*** з JUnit, ми перевіряємо правильність результату:

Assertions.assertNotNull(result) перевіряє, що результат result не є ***null***.

Assertions.assertEquals(1, result.size()) перевіряє, що розмір результату result дорівнює 1, оскільки ми очікуємо, що буде знайдена одна книга з жанром "data_science".

Assertions.assertEquals(books.get(1), result.get(0)) перевіряє, що перший елемент у результаті ***result*** (єдиний елемент) співпадає з другою книгою у списку books. Це підтверджує, що сортування книг за жанром відбувається правильно.

```
Assertions.assertNotNull(result);
Assertions.assertEquals( expected: 1, result.size());
Assertions.assertEquals(books.get(1), result.get(0));
```

Рис. 3.2.4 Перевірка правильності результату

На рисунку 3.2.5 ми оголошуємо приватний метод ***createBooks(),*** який створює і повертає список книг. У цьому методі ми створюємо два об'єкти Book і заповнюємо їх даними. Перша книга має жанр ***"signal_processing",*** а друга

книга має жанр ***"data_science".*** Ми повертаємо ці книги у вигляді списку.

Цей метод допомагає нам створити вхідні дані для тесту, які використовуються для перевірки правильності сортування книг за жанром.

```
private List<Book> getBooks() {
  Book firstBook = new Book();
  Book secondBook = new Book();

  firstBook.setId(1L);
  firstBook.setTitle("Fundamentals of Wavelets");
  firstBook.setAuthor("Goswami, Jaideva");
  firstBook.setGenre("signal_processing");
  firstBook.setHeight(228);
  firstBook.setPublisher("Wiley");

  secondBook.setId(2L);
  secondBook.setTitle("Data Smart");
  secondBook.setAuthor("Foreman, John");
  secondBook.setGenre("data_science");
  secondBook.setHeight(235);
  secondBook.setPublisher("Wiley");

  return List.of(firstBook, secondBook);
}
```

Рис. 3.2.5 Оголошення приватного методу ***createBooks()***

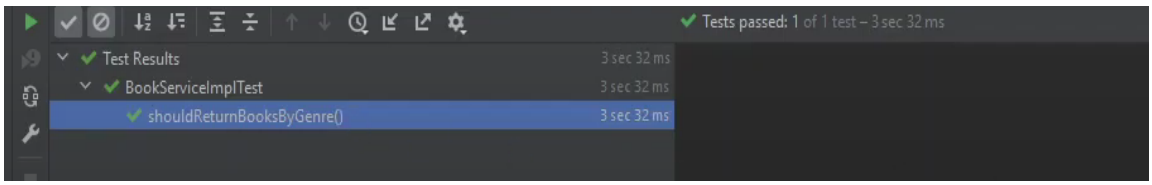

Рис. 3.2.1 Запуск тесту

У тестовому класі ***BookServiceImplTest*** було протестовано метод ***getAllByGenre(String genre)*** в сервісі ***BookServiceImpl***. Цей метод приймає рядок, який представляє жанр книги, і повертає список книг з вказаним жанром.

В тесті ми створили мок репозиторію ***bookRepository*** за допомогою фреймворку Mockito. Ми налаштували поведінку моку, щоб він повертав заданий список книг, коли викликається метод ***findAll()***.

Після цього ми викликали метод ***getAllByGenre(String genre)*** на екземплярі класу ***BookServiceImpl*** та передали жанр "data_science". Очікувалось, що метод правильно відсортовує книги за жанром і поверне список книг з жанром "data_science".

У тесті ми перевірили, чи отримали ми непорожній результат, чи кількість книг відповідає очікуваному значенню (1) та чи повернута книга відповідає першій книзі у списку книг.

Таким чином, тест перевіряє правильність функціонування методу ***getAllByGenre(String genre)***, забезпечуючи, що він вірно фільтрує книги за жанром та повертає очікуваний результат.

ВИСНОВОК

У рамках цієї курсової роботи було детально розглянуто процес автоматизації тестування веб-ресурсу за допомогою мови Java та фреймворку JUnit. Було розглянуто основні принципи тестування, переваги автоматизації та важливість використання тестових фреймворків для забезпечення ефективності і надійності тестів.

Ми розглянули реалізацію автоматизованих тестів на основі мови Java та фреймворку JUnit. Було показано, як створювати тестові класи, описувати окремі тести та використовувати асерти для перевірки очікуваних результатів. Також було розглянуто використання засобів мокування за допомогою Mockito для ізоляції залежностей та створення контрольованого середовища для тестування.

В цілому, автоматизація тестування за допомогою мови Java та фреймворку JUnit є потужним інструментом для забезпечення якості програмного забезпечення. Вона дозволяє розробникам швидко та ефективно виявляти та виправляти помилки.

Перелік використаних джерел

1. What is automation testing? [Electronic source]. – Access mode: https://www.globalapptesting.com/blog/what-is-automation-testing
2. Флагг Л. "Effective Unit Testing: A guide for Java developers". Shelter Island, 2013.Marquis, F. "JUnit recipes: Practical methods for programmer testing". Manning Publications, 2005.
3. Sierra, Kathy, and Bert Bates. "Head First Java." O'Reilly Media, 2019.
4. Bloch, Joshua. "Effective Java." Addison-Wesley Professional, 2018.Taha, T. "JUnit Interview Questions You'll Most Likely Be Asked". CreateSpace Independent Publishing Platform, 2011.
5. Rappin N., Fraser J.M. "JUnit in Action". Manning Publications, 2003.
6. Pashchenko, O., Pashchenko, Y. "JUnit Quick Guide: Learn Automated Testing with JUnit". Packt Publishing, 2018.

Додаток:

```
import io.devmentor.examples.library.entity.Book;
import io.devmentor.examples.library.repository.BookRepository;
import java.util.List;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
class BookServiceImplTest {

  @Mock
  private BookRepository bookRepository;

  @InjectMocks
  private BookServiceImpl bookService;

  @Test
  void shouldReturnBooksFilteredByGenre() {
```

```
  String genre = "data_science";
  List<Book> books = createBooks();

  Mockito.when(bookRepository.findAll()).thenReturn(books);

  List<Book> result = bookService.getAllByGenre(genre);

  Assertions.assertNotNull(result);
  Assertions.assertEquals(1, result.size());
  Assertions.assertEquals(books.get(1), result.get(0));
}

private List<Book> createBooks() {
  Book firstBook = new Book();
  Book secondBook = new Book();

  firstBook.setId(1L);
  firstBook.setTitle("Fundamentals of Wavelets");
  firstBook.setAuthor("Goswami, Jaideva");
  firstBook.setGenre("signal_processing");
  firstBook.setHeight(228);
  firstBook.setPublisher("Wiley");

  secondBook.setId(2L);
  secondBook.setTitle("Data Smart");
  secondBook.setAuthor("Foreman, John");
  secondBook.setGenre("data_science");
  secondBook.setHeight(235);
  secondBook.setPublisher("Wiley");
```

```
    return List.of(firstBook, secondBook);
  }
}
```