Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедр Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня
бакалавра
(назва освітньо-кваліфікаційного рівня)

студента *Фесенка Андрія Романовича*
(ПІБ)
академічної групи *122-21зск-1*
(шифр)
спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)
освітньої програмі *Комп'ютерні науки*
(назва освітньої програми)

на тему: *Розробка додатку автоматизації тестів REST API на основі фреймврку Cucumber*

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
кваліфікаційної роботи	*проф. Алексєєв М.О.*			
розділів:				
спеціальний	*проф. Алексєєв М.О*			
економічний	*доц. Касьяненко Л.В.*			
Рецензент				
Нормоконтролер	*доц. Гуліна І.Г.*			

Дніпро
2024

Міністерство освіти і науки України
НТУ «Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
завідувач кафедри
програмного забезпечення комп’ютерних систем
(повна назва)
М.О. Алексєєв
(підпис) (прізвище, ініціали)
« » 2024 року

ЗАВДАННЯ
на кваліфікаційну роботу
бакалавра
(назва освітньо-кваліфікаційного рівня)

студента *122-21зск-1* ***Фесенко. А.Р.***
(група) (прізвище та ініціали)
тема кваліфікаційної роботи *Розробка додатку автоматизації тестів REST API на основі фреймврку Cucumber*
затверджена наказом ректора НТУ «ДП» від *26.04.2024* № *366-с*

Розділ	Зміст виконання	Термін виконання
Спеціальний	***На основі матеріалів виробничої практики та інших науково-технічних джерел провести аналіз стану рішення проблеми та постановку задачі. Обґрунтувати вибір та здійснити реалізацію методів вирішення проблеми***	***13.05.2024 р.***
Економічний	***Провести розрахунок трудомісткості розробки програмного забезпечення, витрат на створення ПЗ й тривалості його розробки.***	***05.06.2024 р.***

Завдання видав *проф. Алексєєв М.О.*
(підпис) (посада, прізвище,ініціали)
Завдання прийняв до виконання *Фесенко А.Р.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *14.01.2024 р.*

Термін подання кваліфікаційної роботи до ЕК: *08.06.2024 р*

РЕФЕРАТ

Обсяг пояснювальної записки: 94 сторінок, включаючи 21 ілюстрацій, 0 таблиць, 2 додатки, 17 джерел згідно з переліком посилань.

Об'єкт дослідження: Система автоматизованого тестування REST API на основі фреймворку Cucumber.

Мета кваліфікаційної роботи: Розробка фреймворку для автоматизованого тестування REST API з використанням мови програмування JavaScript та інструментів Cucumber.js, Node.js, npm, що забезпечує підвищення продуктивності та якості програмного забезпечення.

Методи дослідження та апаратура: Використано методи дискретної математики, теорії ймовірностей і статистики, методи оптимізації та машинного навчання для аналізу, моделювання та оптимізації тестових сценаріїв. Розробка здійснювалась з використанням інструментів Visual Studio Code, Git, GitLab.

Основні конструктивні, технологічні й техніко-експлуатаційні характеристики та показники: Система забезпечує автоматичне виконання тестів, генерацію звітів, інтеграцію з CI/CD процесами, підтримку інформаційної безпеки та сумісності з різними середовищами розробки.

Практичне значення роботи та висновки: Запропонований фреймворк дозволяє значно зменшити час, необхідний для тестування, забезпечити високу якість програмного забезпечення та автоматизувати процеси тестування. Система може бути використана в різних галузях для тестування веб-додатків, мобільних додатків та інших програмних продуктів.

Ключові слова: АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ, REST API, CUCUMBER, CUCUMBER-HTML-REPORTER, JAVASCRIPT, NODE.JS, ТЕСТОВІ СЦЕНАРІЇ, CI/CD

ABSTRACT

Scope of the explanatory note: 94 pages, including 21 illustrations, 0 tables, 2 appendices, 17 sources according to the list of references.

Object of research: System of automated testing of REST API based on the Cucumber framework.

Purpose of the qualification work: Development of a framework for automated testing of REST API using JavaScript and tools such as Cucumber.js, Node.js, npm, aimed at increasing productivity and software quality.

Research methods and equipment: Methods of discrete mathematics, probability theory and statistics, optimization methods and machine learning were used for the analysis, modeling, and optimization of test scenarios. The development was carried out using Visual Studio Code, Git, GitLab tools.

Main constructive, technological, and technical-operational characteristics and indicators: The system ensures automatic execution of tests, report generation, integration with CI/CD processes, support for information security, and compatibility with different development environments.

Practical significance of the work and conclusions: The proposed framework significantly reduces the time required for testing, ensures high software quality, and automates testing processes. The system can be used in various fields for testing web applications, mobile applications, and other software products.

Forecast assumptions about the development of the research object: Further development of the framework involves integration with new tools and technologies, expanding functional capabilities, improving the user interface, and enhancing system performance.

Keywords: AUTOMATED TESTING, REST API, CUCUMBER, CUCUMBER-HTML-REPORTER, JAVASCRIPT, NODE.JS, TEST SCENARIOS, CI/CD

ЗМІСТ

РЕФЕРАТ .... - 3 -

ABSTRACT .... - 4 -

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ .... - 7 -

ВСТУП .... - 8 -

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧ .... - 10 -

1.1. Загальні відомості с предметної галузі .... - 10 -

1.1.1. Основні підходи до тестування .... - 10 -

1.1.2. Використання інтеграційних (сервісних) методів .... - 12 -

1.1.3. Оцінка взаємодії через тестування контрактів .... - 14 -

1.1.4. Розгляд архітектурних рішень у фреймворках для автоматизованого тестування .... - 15 -

1.1.5. Сучасні підходи до автоматизації тестування .... - 17 -

1.1.6. Важливість і переваги автоматизації тестування .... - 18 -

1.1.7. Висновки до розділу .... - 19 -

1.2. Призначення розробки та галузь застосування .... - 20 -

1.4. Постановка завдання .... - 22 -

1.5. Вимоги до програми або програмного виробу .... - 23 -

1.5.1. Вимоги до функціональних характеристик .... - 23 -

1.5.2. Вимоги до інформаційної безпеки .... - 24 -

1.5.3. Вимоги до складу та параметрів технічних засобів .... - 25 -

1.5.4. Вимоги до інформаційної та програмної сумісності .... - 25 -

РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ .... - 26 -

2.1. Функціональне призначення системи. .... - 26 -

2.2. Опис застосованих математичних методів. .... - 28 -

2.3. Опис використаних технологій та мов програмування .... - 29 -

2.4. Опис структури системи та алгоритмів її функціонування...... - 32 -
2.5. Обґрунтування та організація вхідних та вихідних даних програми.. - 34 -
2.6. Опис розробленої системи. .... - 36 -
2.6.1. Характеристика вхідної оперативної інформації...................... - 37 -
2.6.2. Моделювання фреймворку автоматизованого тестування .. - 39 -
2.6.3. Технологічне забезпечення задачі.. - 40 -
РОЗДІЛ 3. ТЕСТУВАННЯ СИСТЕМИ ТА ОЦІНКА ЇЇ ЕФЕКТИВНОСТІ .. - 52 -
3.1. Апі який був розроблений для цього проекту .... - 52 -
3.2. Схеми данних та її типи .... - 55 -
3.3. Тестування документації .... - 57 -
3.4. Тестування додатку за допомогою розробленого фреймворку - 57 -
РОЗДІЛ 4. ЕКОНОМІЧНИЙ РОЗДІЛ .... - 59 -
4.1. Розрахунок трудомісткості та вартості розробки програмного продукту .... - 59 -
4.2. Розрахунок витрат на створення програми................................ - 63 -
ВИСНОВКИ .... - 66 -
ПЕРЕЛІК ПОСИЛАНЬ.. - 67 -
ДОДАТОК А .... - 69 -
ДОДАТОК Б .... - 93 -

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ

API (Application Programming Interface) - Інтерфейс програмування додатків

BDD (Behavior-Driven Development) - Розробка, керована поведінкою

CI/CD (Continuous Integration/Continuous Deployment) - Безперервна інтеграція/Безперервне розгортання

IDE (Integrated Development Environment) - Інтегроване середовище розробки

npm (Node Package Manager) - Менеджер пакетів для Node.js

REST (Representational State Transfer) - Передача стану представлення

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

CSV (Comma-Separated Values) - Значення, розділені комами

HTML (HyperText Markup Language) - Мова розмітки гіпертексту

XML (eXtensible Markup Language) - Розширювана мова розмітки

JS (JavaScript) - Мова програмування JavaScript

Git - Система контролю версій

GitLab - Веб-сервіс для спільної розробки програмного забезпечення, що використовує Git для контролю версій

Log4js - Бібліотека для логування в середовищі виконання JavaScript

Allure - Фреймворк для створення звітів про тестування

VSCode (Visual Studio Code) - Інтегроване середовище розробки програмного забезпечення

SUT (System Under Test) - Система, що тестується

TDD (Test-Driven Development) - Розробка через тестування

UAT (User Acceptance Testing) - Приймальне тестування користувача

ВСТУП

Важливим етапом життєвого циклу розробки програмного забезпечення є перевірка його працездатності та відповідність встановленим вимогам. Автоматизоване тестування, на відміну від ручного, потребує менше часу і людських ресурсів, що робить створення систем для його автоматизації актуальним завданням.

Під час автоматизованого тестування виконуються тестові сценарії, результати яких порівнюються з очікуваними. Велика увага приділяється створенню таких сценаріїв, щоб охопити всі можливі варіанти використання програмного забезпечення.

Спеціалізовані інструменти та системи автоматизованого тестування дозволяють виконувати перевірки. Ці системи надають комплексне рішення для тестування, забезпечуючи платформу для реалізації процесу тестування, середовище для тестування та архітектурні рішення для опису сценаріїв, їх організації, виконання та перевірки результатів.

Автоматизація тестування є однією з основних сфер розвитку програмного забезпечення. Згідно зі звітом «Pulse of the Profession 2019», більшість команд використовують гнучкі або гібридні методології розробки (Scrum, Kanban тощо). У процесі гнучкої розробки велике значення надається контролю якості, що забезпечує швидку та ефективну роботу. Поряд із ручним тестуванням, команди все частіше використовують автоматизоване тестування, що спрощує виконання тестових сценаріїв і значно зменшує потребу в ручному тестуванні. Автоматизація тестування дозволяє швидко виявляти дефекти після внесення змін у код, тоді як ручне тестування зайняло б значно більше часу.

Міжнародна рада з контролю якості програмного забезпечення (International Software Testing Qualifications Board) визначає автоматизоване тестування як використання програм для підтримки тестування, таких як управління тестовими сценаріями, розробка тестових дизайнів та перевірка результатів. Автоматизація тестування дозволяє командам швидко і повторно виконувати тести, що особливо корисно для регресійного тестування – перевірки програмного забезпечення після модифікацій для виявлення дефектів або змін у немодифікованих частинах програми внаслідок внесених змін .

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧ

1.1. Загальні відомості с предметної галузі

1.1.1. Основні підходи до тестування

Автоматизація тестування програмного забезпечення є однією з ключових областей сучасної розробки. Існує багато методів і підходів до автоматизованого тестування, які спрямовані на підвищення ефективності та якості тестування. Серед найпоширеніших підходів можна виділити використання фреймворків на основі сценаріїв, керованих поведінкою (BDD), таких як Cucumber, які дозволяють створювати тестові сценарії, що легко читаються і підтримуються.

Відповідно до класифікації, яку вперше запропонував Майк Кон у своїй праці «Succeeding with Agile», стратегія автоматизованого тестування поділяється на три основні рівні:

1. **Модульне тестування**: на цьому рівні проводяться тести окремих модулів програмного забезпечення.
2. **Сервісне тестування**: цей рівень охоплює тести сервісів або API, які використовуються для взаємодії між різними частинами програми.
3. **Тестування користувацького інтерфейсу**: на цьому рівні проводяться тести, які перевіряють коректність роботи інтерфейсу програми з точки зору користувача.

Ці рівні формують концепцію «піраміди тестування», яка може бути доповнена варіантом піраміди, де кожен наступний рівень тестів, починаючи з модульних, розширює тестові сценарії попередніх рівнів і включає нові сценарії інтеграції компонентів системи.

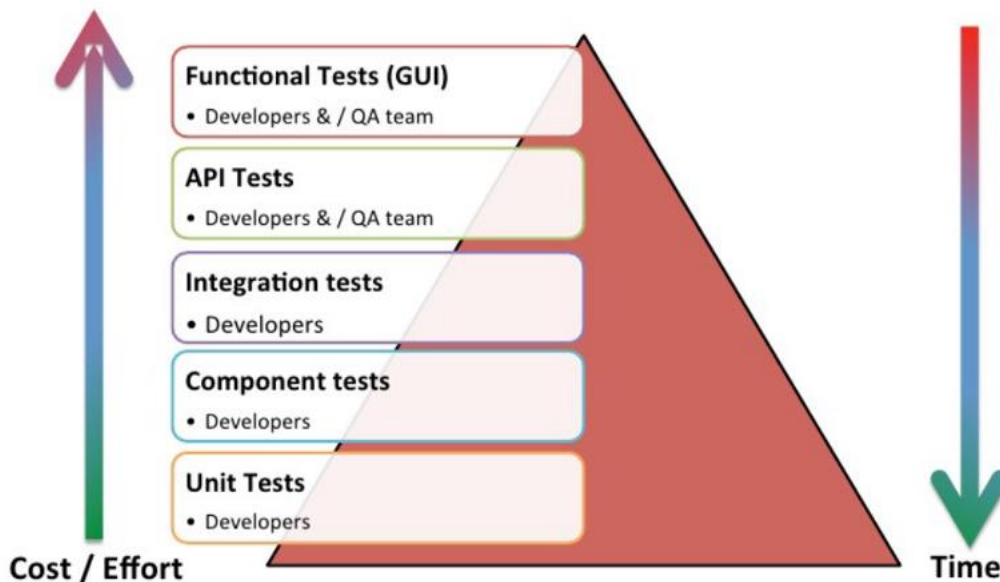


Рис. 1.1. – Розширена модель тестування

Кількість тестів на кожному рівні піраміди залежить від його місця в загальній структурі. Важливим аспектом піраміди є те, що із зростанням рівня тестування збільшуються зусилля і час, необхідні для розробки та підтримки тестових сценаріїв. Тести на вищих рівнях мають значну крихкість, оскільки навіть невеликі зміни в системі можуть призвести до невірної роботи великої кількості тестів. Хоча більшість функціональних тестів зазвичай орієнтована на інтерфейс користувача, такого як веб-сторінки або форми, особливість тестування API полягає в його безпосередньому спілкуванні за допомогою запитів без використання інтерфейсу користувача.

1.1.2. Використання інтеграційних (сервісних) методів

Інтеграційне тестування або сервісне тестування є процесом, під час якого перевіряються взаємодія та взаємозв'язок між різними компонентами системи, а також правильність їх інтеграції.

Зазвичай цей тип тестування використовується для перевірки функціональності API сервісу.

Інтеграційне тестування спрямоване на перевірку того, як різні модулі працюють як єдина система, а також на переконання, що окремо розроблені компоненти працюють разом правильно, а їх інтерфейси відповідають архітектурному дизайну системи.

У цьому виді тестування основна увага приділяється перевірці взаємодії різних систем згідно з принципом "клієнт-сервер".

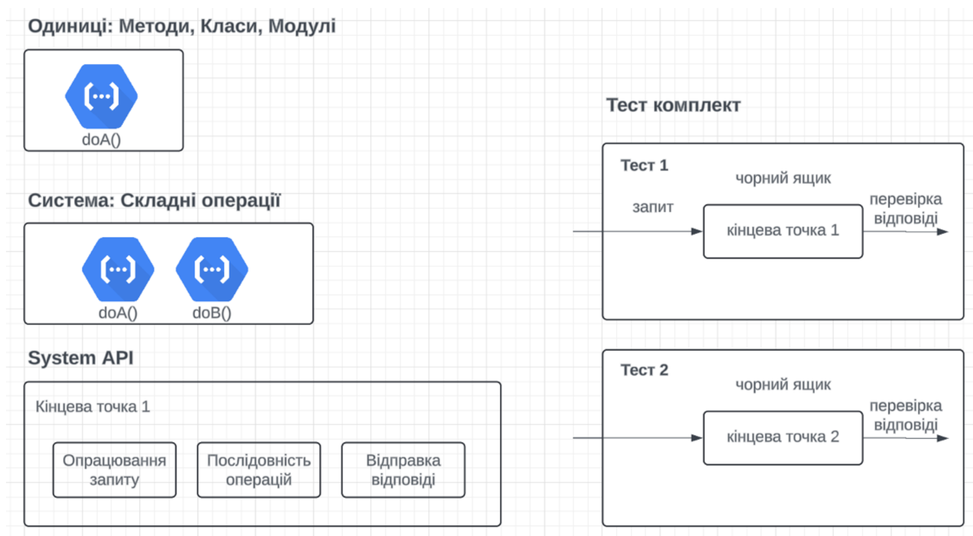


Рис. 1.2. Компоненти, включені у процес сервісного тестування.

Багато існуючих рішень не забезпечують достатньої гнучкості у створенні та підтримці тестових сценаріїв, що може призвести до зниження ефективності тестування. Крім того, необхідність ручного написання

великої кількості коду для тестів може стати перешкодою для їхнього широкого використання.

Програма призначена для використання в галузі розробки програмного забезпечення, зокрема для тестування веб-додатків, мобільних додатків і систем обміну даними. Вона дозволяє забезпечити високу якість та надійність програмного забезпечення шляхом автоматизації процесу тестування API.

Узагальнено кажучи, API забезпечує комунікацію між двома системами

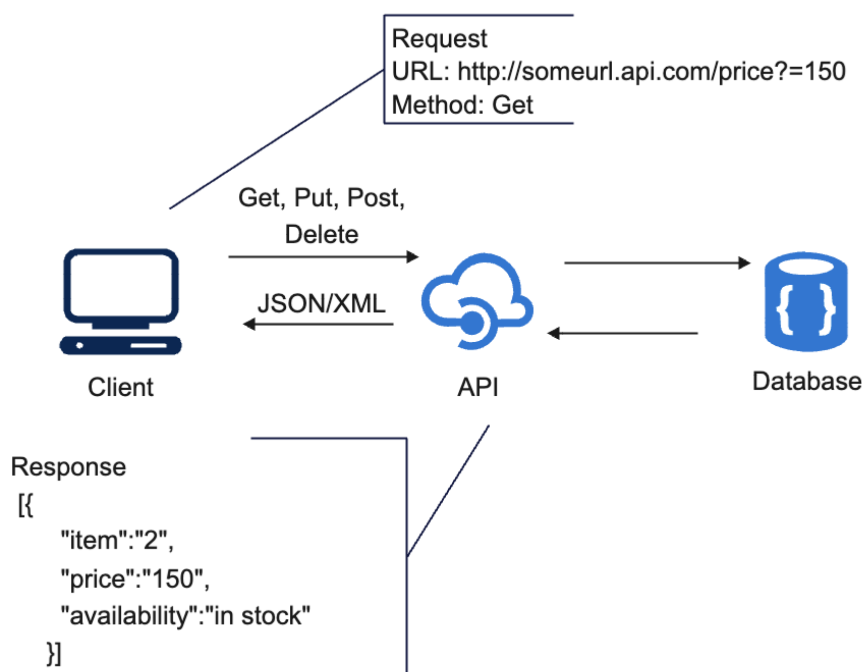


Рис. 1.3. Узагальнена структура REST API.

Як видно, між клієнтом і сервером відбувається обмін запитами та відповідями. Хоча клієнт і сервер можуть мати будь-яку мову програмування, проте HTTP є протоколом, який використовується для

передачі цих повідомлень. Цей зразок взаємодії запиту та відповіді є основним принципом роботи REST API.

1.1.3. Оцінка взаємодії через тестування контрактів

Сучасні організації розробляють та інтегрують окремі сервіси в єдину систему. Розбиття системи на невеликі сервіси передбачає взаємодію цих сервісів через чітко визначені інтерфейси.

Інтерфейси між різними додатками можуть мати різні формати та технології. Одним із найпоширеніших є використання REST та JSON через протокол HTTPS. Такий підхід передбачає наявність постачальника та споживача. Постачальник надає дані споживачам, в той час як споживач обробляє ці дані.

У світі REST, постачальник створює REST API з усіма необхідними кінцевими точками, а споживач використовує це API для отримання даних або здійснення змін в іншій службі. Контракт описується за допомогою DSL та включає в себе опис API у вигляді сценаріїв взаємодії.

Сценарії тестування контрактів взаємодії - це сценарії, які виконуються з певною періодичністю у системі неперервної інтеграції та перевіряють запити до реальних зовнішніх компонентів, щоб гарантувати, що їх контракт не був змінений.

Об'єктом застосування програми є різноманітні програмні продукти, які мають API для взаємодії з іншими системами або компонентами. Це можуть

бути як внутрішні системи компанії, так і публічні сервіси, що надають своїм користувачам можливості для інтеграції через API.

1.1.4. Розгляд архітектурних рішень у фреймворках для автоматизованого тестування

Багато існуючих рішень для автоматизації тестування REST API базуються на різних фреймворках, таких як Selenium, JUnit, TestNG, та інших. Вони мають свої переваги і недоліки, але не завжди можуть забезпечити достатню гнучкість і простоту використання для тестування API.

Фреймворки автоматизованого тестування, що базуються на архітектурі сценаріїв керованих поведінкою, є найбільш розповсюдженим підходом, спрямованим на аналіз причинно-наслідкових зв'язків у системі. Основною метою таких фреймворків є надання зручного інструментарію як для розробників, так і для інженерів з тестування, а також для управління в компанії. Це досягається за рахунок використання зовнішніх мов опису сценаріїв тестування, які нагадують природну мову, і які фреймворк подальше обробляє для генерації програмного коду тестових сценаріїв. Головним недоліком такого підходу є необхідність вміння програмування у тестувальників, а також необхідність створення додаткових сценаріїв тестування на мові програмування.

Гібридні фреймворки автоматизованого тестування поєднують у собі різноманітні архітектурні та підхідні рішення для опису тестових сценаріїв, що найкраще відображають процеси тестування конкретного бізнесу. Створення таких фреймворків може бути складним завданням, але вони найефективніше вирішують завдання тестування.

Характерною рисою будь-яких фреймворків автоматизованого тестування є процес роботи фреймворку, коли сценарії тестування завантажуються з певного джерела, кроки виконання сценаріїв відбуваються в певній послідовності з використанням додаткової інформації, а фреймворк управляє програмним застосунком та перевіряє результати виконання тестових сценаріїв.

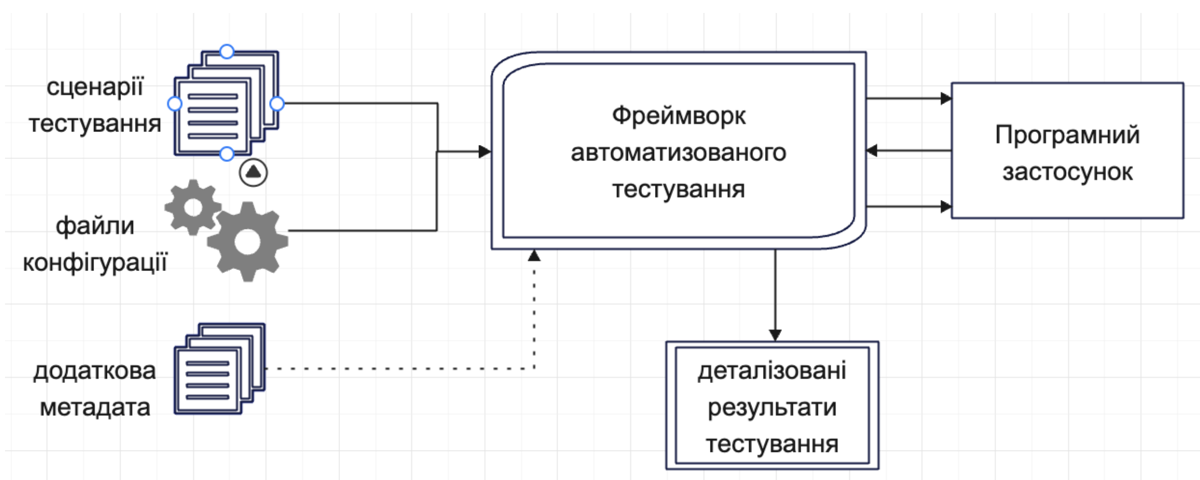


Рис. 1.5. Зразкова структура фреймворку для автоматизації тестування

На сьогоднішній день існує значна кількість інструментів, які дозволяють автоматизувати тестування API. Проте, багато з них вимагають глибоких знань у програмуванні або не забезпечують достатньої інтеграції з сучасними методологіями розробки програмного забезпечення, такими як Agile.

Однією з головних проблем є необхідність забезпечення високої якості тестування при мінімальних витратах часу і ресурсів. Важливою задачею є розробка системи, яка б дозволяла тестувальникам швидко і легко

створювати тестові сценарії, інтегрувати їх з процесом безперервної інтеграції та доставки (CI/CD) і отримувати зрозумілі звіти про результати тестування.

1.1.5. Сучасні підходи до автоматизації тестування

Наразі, у світі розробки програмного забезпечення спостерігається розширення інструментів та підходів до автоматизації тестування.

Інтеграція з CI/CD: Зараз автоматизоване тестування стає невід'ємною частиною процесу Continuous Integration (CI) та Continuous Deployment (CD). Інструменти автоматизованого тестування інтегруються з CI/CD платформами, щоб автоматично виконувати тести при кожній зміні в коді та автоматично розгортати нові версії програмного забезпечення.

Тестування з використанням візуальних елементів: Застосування інструментів, які дозволяють тестувати інтерфейс користувача на рівні візуальних елементів, набирає популярності. Такі інструменти дозволяють перевіряти вигляд і поведінку елементів інтерфейсу, що спрощує тестування складних UI-компонентів та забезпечує більшу точність.

Тестування API: Сучасні підходи до тестування включають широке використання тестування API. API тестування дозволяє перевірити роботу API без необхідності запуску повного інтерфейсу користувача. Це робить тестування більш швидким і ефективним.

Тестування мобільних додатків: З огляду на зростання популярності мобільних додатків, сучасні підходи до автоматизації тестування включають широке використання інструментів для тестування мобільних додатків. Ці інструменти дозволяють автоматизувати тестування функціональності, вигляду та продуктивності мобільних додатків на різних платформах та пристроях.

Використання штучного інтелекту та машинного навчання: Сучасні підходи до автоматизації тестування включають в себе використання штучного інтелекту та машинного навчання для автоматичного аналізу тестових результатів, виявлення дефектів та навіть автоматичної генерації тестових сценаріїв.

Ці підходи до автоматизації тестування спрямовані на підвищення ефективності та якості процесу розробки програмного забезпечення, зменшення часу на тестування та забезпечення швидкого виявлення і виправлення дефектів

1.1.6. Важливість і переваги автоматизації тестування

Автоматизація тестування в сучасній розробці програмного забезпечення є важливим етапом, який відіграє ключову роль у забезпеченні якості продукту та прискоренні процесу розробки

Підвищення швидкості тестування: Автоматизація тестування дозволяє значно прискорити процес тестування порівняно з ручним тестуванням. Автоматизовані тести можуть бути запущені швидше та виконані в декілька разів швидше, що дозволяє розробникам отримати швидку зворотню інформацію про якість коду.

Збільшення покриття тестування: Завдяки автоматизації тестування можна значно збільшити покриття функціональності продукту тестами. Автоматизовані тести можуть бути легко масштабовані і виконуватися великою кількістю разів без значного затримки.

Виявлення дефектів на ранніх етапах: Автоматизація тестування дозволяє виявляти дефекти та помилки в коді на ранніх етапах розробки. Це

дозволяє розробникам виправляти проблеми швидше та зменшує витрати на виправлення дефектів на пізніших етапах.

Зменшення людського фактору: Автоматизовані тести виконуються без участі людини, що зменшує вплив людського фактору на результати тестування. Це дозволяє уникнути помилок, пов'язаних з втомою, недбалістю чи неправильним виконанням тестових сценаріїв.

Економія часу і коштів: Хоча впередження та підтримка автоматизованих тестів вимагає певних затрат на початкову розробку, вони дозволяють економити час і кошти на тривалому проміжку часу. Час, витрачений на виконання автоматизованих тестів, значно менший, ніж на ручні тести.

Підвищення надійності програмного забезпечення: Завдяки систематичному та повторюваному виконанню автоматизованих тестів можна підвищити надійність програмного забезпечення, оскільки вони дозволяють виявляти та виправляти дефекти та помилки до випуску продукту в експлуатацію.

1.1.7. Висновки до розділу

Мета використання автоматизованих тестових фреймворків включає:

- Покращення продуктивності та спрощення процесу розробки тестових сценаріїв шляхом зменшення їх складності.
- Можливість повторного використання тестових сценаріїв для проведення регресійного тестування.
- Забезпечення структурованої методології розробки для підтримки цілісності дизайну системи та зменшення залежності від випадкових дефектів.

- Надання надійних інструментів для виявлення дефектів та аналізу їхніх причин з мінімальною участю людини.
- Зменшення залежності від експертів бізнес-домену шляхом автоматизованої реакції на зміни умов тестування та середовища.
- Ефективне використання ресурсів для тестування складних систем та неперервний контроль за процесом тестування.
- Створення тестових сценаріїв шляхом простого складання послідовностей кроків.
- Можливість розширення функціональності і незалежність від сторонніх продуктів.
- Легкість підключення нових бібліотек і запуск тестів через командний рядок.
- Можливість паралельного запуску тестів, використання Asserts і групування за тест-пакетами.
- Використання паралельного запуску тестів.
- Можливість опису взаємодії з усіма елементами додатку один раз.
- Ефективне використання часу для тестування, а не підготовки тестових даних.
- Краще зрозуміння причин викидання тестів.
- Додано логування в проект за допомогою Log4js, що дозволяє поділитися замовнику фінальним рапортом.
- Використання анотацій з cucumber-html-reporter для зрозумілого відображення ходу дій тесту без технічного бекграунду

1.2. Призначення розробки та галузь застосування

Повна назва розробленої системи: «Розробка додатку для автоматизації тестів REST API на основі фреймворку Cucumber»

Автоматизоване тестування: процес використання спеціалізованих інструментів для автоматичного виконання тестових сценаріїв з метою перевірки функціональності програмного забезпечення.
REST API (Representational State Transfer Application Programming Interface): архітектурний стиль для розробки веб-сервісів, який використовує HTTP запити для доступу і маніпуляції веб-ресурсами.

Cucumber: фреймворк для автоматизованого тестування, який підтримує поведінково-орієнтовану розробку (BDD) та дозволяє писати тести у вигляді читабельних сценаріїв на природній мові.

Причини виникнення необхідності розробки програмного забезпечення:

- Підвищення ефективності тестування: Автоматизація тестування дозволяє значно скоротити час, необхідний для виконання тестів, порівняно з ручним тестуванням.
- Зменшення витрат на тестування: Автоматизовані тести можуть виконуватись багаторазово без додаткових витрат, що дозволяє зменшити загальні витрати на забезпечення якості програмного забезпечення.
- Покращення якості програмного забезпечення: Регулярне та систематичне виконання автоматизованих тестів дозволяє виявляти та усувати дефекти на ранніх етапах розробки.
- Забезпечення безперервної інтеграції: Автоматизоване тестування інтегрується в процес безперервної інтеграції та доставки (CI/CD), що дозволяє швидко перевіряти зміни в коді та забезпечувати стабільність програмного забезпечення.

Розробка веб-додатків: Тестування REST API, яке використовується для взаємодії між фронтендом і бекендом.

Мобільні додатки: Перевірка API, що забезпечує функціональність мобільних додатків.

1.3. Підстава для розробки

Відповідно до ОКХ та ОПП, згідно навчального плану та графіків навчального процесу, в кінці навчання студент виконує кваліфікаційну роботу (проєкт). Тема роботи узгоджується з керівником проекту, випускаючою кафедрою, та затверджується наказом ректора. Отже, підставами для розробки (виконання кваліфікаційної роботи) є: − ОКХ та ОПП за напрямом підготовки 6.050101 «Комп'ютерні науки»; − Графік навчального процесу та навчальний план; − наказ ректора Національного технічного університету «Дніпровська політехніка» № 350-с від 16.05.2024 р; − завдання на дипломний проект на тему «».

1.4. Постановка завдання

Метою створення додатку автоматизованого тестування на основі фреймворку Cucumber є підвищення ефективності процесу тестування програмного забезпечення, зокрема REST API, зменшення витрат часу та ресурсів, а також забезпечення високої якості кінцевого продукту. Автоматизоване тестування дозволяє швидко і повторювано перевіряти функціональність системи, знаходити дефекти на ранніх етапах розробки та забезпечувати стабільну роботу програмних продуктів.

Об'єктами системи автоматизованого тестування є:

- API кінцеві точки, що підлягають тестуванню.
- Тестові сценарії, розроблені на основі фреймворку Cucumber.
- Тестові дані, необхідні для проведення тестів.

- Інструменти інтеграції з системами безперервної інтеграції (CI/CD).

Опис структури об'єктів інформаційної системи і перелік показників

- API кінцеві точки: визначаються URL, методи (GET, POST, PUT, DELETE) і параметри запитів.
- Тестові сценарії: складаються з опису тестових випадків у вигляді функціональних вимог (Given, When, Then), що легко читаються і підтримуються.
- Тестові дані: включають вхідні та очікувані результати, що використовуються в тестах для перевірки коректності роботи API.
- Інструменти інтеграції: забезпечують автоматичний запуск тестів при зміні коду та створення звітів про результати тестування.

Вихідна інформація системи автоматизованого тестування включає звіти про результати виконання тестів, що містять інформацію про пройдені та провалені тести, знайдені дефекти, час виконання тестів та інші важливі метрики. Ці звіти дозволяють розробникам і тестувальникам швидко реагувати на виявлені проблеми і підтримувати високу якість програмного забезпечення.

1.5. Вимоги до програми або програмного виробу

1.5.1. Вимоги до функціональних характеристик

- Система повинна підтримувати автоматизацію тестування REST API з використанням фреймворку Cucumber.

- Повинна забезпечувати можливість створення та виконання тестових сценаріїв для HTTP методів GET, POST, PUT, DELETE.
- Система має генерувати звіти про результати тестування з інформацією про пройдені та провалені тести, а також про знайдені дефекти.
- Повинна інтегруватися з інструментами безперервної інтеграції (CI/CD), такими як Jenkins, для автоматичного запуску тестів при зміні коду.
- Система повинна забезпечувати легкість налаштування та підтримки тестових сценаріїв і даних.
- Інтерфейс користувача повинен бути зручним та інтуїтивно зрозумілим для забезпечення швидкого навчання та ефективного використання системи.

1.5.2. Вимоги до інформаційної безпеки

Система повинна забезпечувати захист даних, що використовуються та генеруються під час тестування, від несанкціонованого доступу.

Тестові дані та результати тестування повинні зберігатися у захищених сховищах з обмеженим доступом.

Система повинна підтримувати аутентифікацію та авторизацію користувачів для контролю доступу до тестових сценаріїв та результатів тестування.

Всі комунікації між компонентами системи мають бути зашифровані для запобігання перехопленню даних.

Система повинна забезпечувати можливість аудиту дій користувачів та ведення журналу змін для відстеження діяльності.

1.5.3. Вимоги до складу та параметрів технічних засобів

1. Система повинна працювати на серверах з операційною системою Linux або Windows.
2. Мінімальні вимоги до апаратного забезпечення включають:
   - Процесор: Intel Core i5 або еквівалентний
   - Оперативна пам'ять: не менше 8 ГБ
   - Жорсткий диск: не менше 500 ГБ
3. Для забезпечення масштабованості та надійності система повинна підтримувати розгортання в хмарних середовищах, таких як AWS, Azure, або Google Cloud.
4. Система повинна забезпечувати можливість горизонтального масштабування для обробки великої кількості тестів одночасно.

1.5.4. Вимоги до інформаційної та програмної сумісності

- Система повинна бути сумісною з різними версіями REST API, що використовуються в тестованих програмах.
- Система повинна підтримувати інтеграцію з різними інструментами управління версіями, такими як Git, для забезпечення контролю версій тестових сценаріїв.
- Система повинна бути сумісною з різними середовищами розробки та тестування, включаючи IDE (наприклад, Visual Studio Code, IntelliJ IDEA) та системи управління проектами (наприклад, JIRA, Trello).
- Повинна забезпечувати можливість імпорту та експорту тестових сценаріїв та даних у стандартизованих форматах (наприклад, JSON, XML) для забезпечення сумісності з іншими системами.

– Система повинна підтримувати роботу з базами даних, що використовуються для зберігання тестових даних та результатів, включаючи MySQL, PostgreSQL, MongoDB.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1. Функціональне призначення системи.

Система автоматизованого тестування REST API на основі фреймворку Cucumber призначена для виконання наступних основних функцій:

1. Створення та управління тестовими сценаріями:

   - Система забезпечує інтерфейс для створення, редагування та видалення тестових сценаріїв.

   - Тестові сценарії створюються на основі підходу BDD (Behavior-Driven Development) з використанням синтаксису Gherkin.

2. Автоматичне виконання тестів:

   - Система дозволяє автоматично виконувати тестові сценарії для REST API кінцевих точок.

   - Підтримка виконання тестів для різних HTTP методів, таких як GET, POST, PUT, DELETE.

3. Генерація звітів про результати тестування:

   - Система генерує детальні звіти про результати виконання тестів, включаючи інформацію про пройдені та провалені тести, знайдені дефекти, час виконання тестів та інші метрики.

4. Інтеграція з CI/CD процесами:

    - Підтримка інтеграції з інструментами безперервної інтеграції та доставки, такими як Jenkins, що дозволяє автоматично запускати тести при зміні коду.

5. Забезпечення інформаційної безпеки:

    - Система забезпечує захист тестових даних та результатів тестування від несанкціонованого доступу.
    - Підтримка аутентифікації та авторизації користувачів для контролю доступу до тестових сценаріїв та результатів тестування.

6. Підтримка сумісності та масштабованості:

    - Система сумісна з різними середовищами розробки та інструментами, що використовуються в процесі розробки програмного забезпечення.
    - Підтримка масштабованості, що дозволяє обробляти велику кількість тестів одночасно і забезпечувати стабільну роботу навіть при великих навантаженнях.

Запропоновано розробити рішення шляхом створення та впровадження наступної функціональності:

1. Метод "Keyword Driven":

    У випадку, коли тестувальники не мають експертних знань у програмуванні, потрібно мати можливість замінити скрипти на текстові файли, що базуються на ключових словах. Цей підхід

дозволяє тестувальникам (автоматизаторам) створювати автоматизовані тести, просто описуючи кожен крок тесту.

2. Ітерації з різними наборами даних:

   Ця практика дозволяє тестувальникам використовувати одні й ті самі автоматичні тести, але запускати їх з різними наборами тестових даних. Наприклад, можна використовувати один тест для входу в систему, але виконувати його з різними комбінаціями імені користувача та пароля, щоб протестувати різні сценарії. Якщо фреймворк дозволяє визначати кілька ітерацій для запуску з різними даними, це значно зменшить час, потрібний для автоматизації.

2.2. Опис застосованих математичних методів.

У розробці системи автоматизованого тестування REST API на основі фреймворку Cucumber застосовуються різні математичні методи для забезпечення точності, надійності та ефективності тестування. Нижче наведено основні математичні методи, які використовуються в системі:

Методи дискретної математики:

- Графи та дерева: Для моделювання та аналізу структури тестових сценаріїв, де вершини графів представляють окремі тести, а ребра — послідовності виконання або залежності між тестами. Це дозволяє оптимізувати послідовність виконання тестів і виявляти цикли або некоректні залежності.

- Булева алгебра: Використовується для логічного моделювання умов тестування і перевірки результатів. Булева логіка допомагає

визначити правильність виконання тестів на основі умов та їх комбінацій.

Теорія ймовірностей і статистика:

- Аналіз ймовірності дефектів: Застосовується для оцінки ймовірності виникнення дефектів у різних частинах програмного забезпечення. Це допомагає визначити пріоритети для тестування та зосередитися на найбільш ризикованих областях коду.
- Статистичний аналіз результатів тестування: Використовується для аналізу результатів тестів, виявлення тенденцій і визначення показників якості програмного забезпечення. Це дозволяє виявити стабільність і надійність програмного забезпечення на основі великої кількості тестових запусків.

Методи оптимізації:

- Комбінаторна оптимізація: Застосовується для визначення оптимальної послідовності виконання тестів з метою мінімізації часу тестування та ресурсів. Це включає використання алгоритмів пошуку шляху, таких як алгоритм Дейкстри або A*, для знаходження найкоротших або найбільш ефективних шляхів виконання тестів.

2.3. Опис використаних технологій та мов програмування.

Система автоматизованого тестування REST API буде розроблена з використанням сучасних технологій і мов програмування, що забезпечують високу продуктивність, гнучкість і надійність.

Задача буде вирішена за допомогою створення фреймворка для автоматизованого тестування на мові програмування JavaScript. Для цього будуть використані такі інструменти розробки: Node.js, Cucumber.js, Visual

Studio Code, npm (Node Package Manager), а також системи контролю версій Git та GitLab.

Основні технології та мови програмування, що використовуються в розробці системи:

1. JavaScript:

   Основна мова програмування для розробки тестових сценаріїв і логіки автоматизації. JavaScript забезпечує гнучкість і широку підтримку різних інструментів і бібліотек.
   Можливість використання фреймворків та бібліотек для спрощення розробки, таких як Express.js для створення веб-додатків.

2. Node.js:

   Серверне середовище виконання для JavaScript, що дозволяє створювати швидкі та масштабовані додатки. Використовується для розробки серверної частини системи автоматизованого тестування.
   Підтримує асинхронну обробку запитів, що підвищує продуктивність і знижує затримки при виконанні великої кількості тестів.

3. Cucumber.js:

   Інструмент для автоматизованого тестування на основі підходу BDD (Behavior-Driven Development). Використовується для написання тестових сценаріїв у вигляді зрозумілих текстових описів на мові

Gherkin. Підтримує інтеграцію з іншими інструментами тестування, такими як Selenium для тестування веб-інтерфейсів.

4. npm (Node Package Manager):

   Менеджер пакетів для Node.js, що дозволяє легко встановлювати та керувати залежностями проекту. Великий репозиторій готових модулів і пакетів, які спрощують розробку і тестування.

5. Visual Studio Code:

   Інтегроване середовище розробки (VSCode), яке забезпечує зручні інструменти для написання, відлагодження та тестування коду. Підтримує роботу з різними мовами програмування і технологіями, що робить його універсальним інструментом для розробників.

6. Git та GitLab:

   Системи контролю версій та платформи для спільної розробки, що забезпечують збереження історії змін коду, зручну співпрацю в команді та автоматизацію процесів розгортання і тестування. Інтеграція з інструментами CI/CD для автоматичного запуску тестів і розгортання додатків.

Джерелами інформації для розробки нових тестів будуть використані сценарії, які були складені під час ручного тестування, а також наявні специфікації та вимоги до тестованого API.

Процес розробки буде складатися з наступних етапів:

1. Аналіз потреб у першорядному та додатковому функціоналі фреймворка для його користувачів.
2. Проектування архітектури розробки.
3. Розробка та програмування.
4. Проведення модульних тестів для фреймворка.
5. Впровадження фреймворка.
6. Надання підтримки та подальший розвиток функціоналу фреймворка.

2.4. Опис структури системи та алгоритмів її функціонування.

Система автоматизованого тестування складається з наступних основних компонентів:

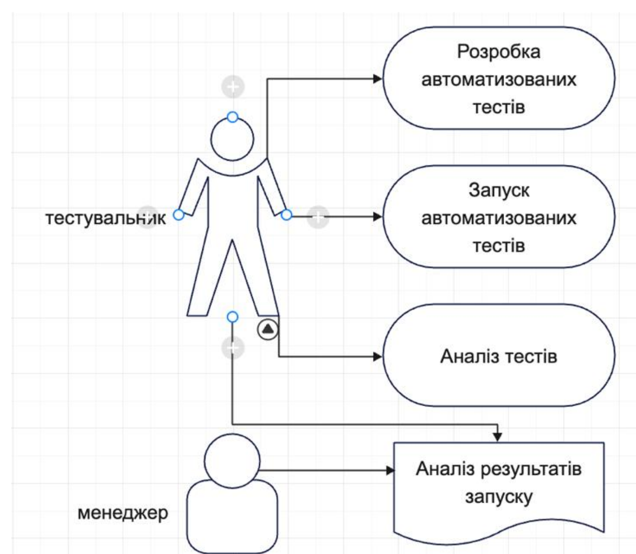


Рис. 2.1. Схема використання фреймворку автоматизації тестів

Тестувальник:

- Забезпечує інтерфейс для створення, редагування та видалення тестових сценаріїв.
- Використовує Cucumber.js для написання сценаріїв на мові Gherkin.

- Інтеграція з системою контролю версій для відстеження змін і спільної роботи над сценаріями.

Менеджер:

- Автоматично виконує тестові сценарії, взаємодіючи з REST API.
- Зберігає результати виконання тестів та генерує звіти.
- Підтримує паралельне виконання тестів для підвищення ефективності.

Модуль інтеграції з CI/CD:

- Забезпечує інтеграцію з інструментами CI/CD, такими як Jenkins, для автоматичного запуску тестів при зміні коду.
- Автоматичне розгортання нових версій додатків і запуск тестів після кожного коміту.

Модуль логування та звітності:

- Використовує Log4js для запису логів виконання тестів. Це потужна бібліотека для логування в середовищі виконання JavaScript, яка надає зручний інтерфейс для запису логів з різними рівнями важливості та налаштуванням транспортів для виведення логів в різні місця.
- Генерує звіти за допомогою фреймворку cucumber-html-reporter для зручного перегляду результатів тестування. Цей фреймворк є потужним інструментом для створення звітів про тестування, який підтримується для JavaScript. Він дозволяє створювати красиві та інформативні звіти, які включають в себе деталі про виконані тести, зображення скріншотів, інформацію про помилки та баги

- Звіти включають деталі про виконані тести, знайдені дефекти, час виконання тестів та інші важливі метрики.

Алгоритм функціонування:

1. Користувач створює тестовий сценарій за допомогою менеджера тестових сценаріїв.
2. Виконавець тестів автоматично запускає сценарії, взаємодіючи з API, та зберігає результати.
3. Модуль логування записує детальну інформацію про виконання тестів.
4. Модуль інтеграції з CI/CD автоматично запускає тести при кожній зміні в коді.
5. Модуль звітності генерує звіти, які надають детальну інформацію про результати тестування.

2.5. Обґрунтування та організація вхідних та вихідних даних програми.

Вхідні дані:

- Тестові сценарії: Опис тестових випадків на мові Gherkin, які включають вхідні дані, умови та очікувані результати.

- Тестові дані: Набори даних, що використовуються для виконання тестів, зберігаються у форматах JSON або CSV.

- Конфігураційні файли: Параметри налаштування середовища тестування, такі як URL API, облікові дані доступу тощо.

Вихідні дані:

– Звіти про результати тестування: Генеруються у форматах HTML, XML або JSON за допомогою фреймворку cucumber-html-reporter, містять інформацію про пройдені та провалені тести, знайдені дефекти, час виконання тестів тощо.

– Логи виконання тестів: Записуються за допомогою Log4js, містять детальну інформацію про виконання кожного тестового сценарію, що допомагає в діагностиці помилок.

Організація збору та обробки даних:

1. Тестові сценарії створюються та зберігаються в системі контролю версій (Git/GitLab).
2. Тестові дані зберігаються у структурованому форматі та завантажуються перед виконанням тестів.
3. Після виконання тестів результати зберігаються в базі даних або файловій системі для подальшого аналізу.

Звіти та логи автоматично генеруються та розміщуються в доступному для перегляду місці, такому як веб-сервер або репозиторій звітів.

2.6. Опис розробленої системи.

```
351  // POST endpoint to add a new student
352  app.post('/api/student', (req, res) => {
353      logger.debug('[API] Starting new student addition');
354      const { name, age, sex, fearFactor } = req.body;
355      // Check if age and fearFactor are provided and valid
356      logger.info(`[student] checking if age: ${age} and fearFactor: ${fearFactor} are provided and valid`);
357      if (!age || isNaN(age) || !fearFactor || isNaN(fearFactor)) {
358          logger.debug('[student] Invalid age or fearFactor provided');
359          return res.status(400).json({ error: 'Invalid age or fearFactor provided' });
360      }
361
362      // Determine if the student is female based on the sex field
363      logger.info(`[student] Determining if the student is female based on the sex field: ${sex}`);
364      const isFemale = sex.toLowerCase() === 'female';
365
366      // Create a new student object
367      const newStudent = {
368          id: generateId(),
369          name: name,
370          age: age,
371          sex: isFemale,
372          fearFactor: fearFactor
373      };
374      logger.info(`[student] Creating a new student object: ${newStudent}`);
375
376      // Add the new student to the students array
377      logger.info(`[student] adding the student to the students array`);
378      students.push(newStudent);
379
380      // Return the newly created student object in the response
381      res.status(201).json(newStudent);
382      logger.debug('[API] Finishing new student addition');
383  });
384
```

Рис. 2.2 Код реалізація класу, який взаємодіє з конкретним endpoint.

```
10  When('I create a debt for the student with the amount {int}', async function (amount) {
11      const debtDetails = {
12          studentId: this.studentID,
13          amount: amount
14      };
15      console.log(this.api)
16      response = await axios.post(`${this.api}/api/debt`, debtDetails);
17      this.response = response;
18  });
19
20  // Step definition ends...
21
22
23  // Step validation starts...
24
25  Then('The debt should be created with the amount {int}', function(expectedAmount) {
26      const debtData = this.response.data;
27      assert.strictEqual(debtData.amount, expectedAmount, `Expected debt amount ${expectedAmount}, but got ${debtData.amount}`);
28  })
29
30  Then('Id is assigned to the debt', function () {
31      const debtData = this.response.data;
32      assert.ok(debtData.id, 'Expected debt ID to be assigned, but it was not');
33  })
34
35  Then('The monthly percent of the newly created debt must be {int}', function (expectedMonthlyPercent) {
36      const debtData = this.response.data;
37      assert.strictEqual(debtData.monthlyPercent, expectedMonthlyPercent, `Expected monthly percent ${expectedMonthlyPercent}, but got ${
38  })
39
```

Рис. 2.3 Код реалізації тестових кроків

2.6.1. Характеристика вхідної оперативної інформації

Для фреймворку автоматизованого тестування, вхідною інформацією є тестові сценарії та дані, необхідні для їх виконання. Тестові сценарії складаються з наступних основних компонентів:

- Назва: Ідентифікатор тестового сценарію, що вказує на його мету.
- Опис: Може бути деталізованим та включати коментарі щодо виконання тестового сценарію.
- Передумови: Умови, які повинні бути виконані перед початком виконання сценарію.
- Кроки: Детальний опис кроків виконання сценарію, включаючи очікуваний результат.
- Післяумови: Дії, які повинні бути виконані після завершення тестового сценарію.
- Тестові сценарії у розробленому фреймворку будуть представлені у форматі, що відповідає методології BDD (Behaviour Driven Development). Сценарій тестування описується наступним чином:

Однією з найпоширеніших мов створення сценаріїв автоматизованого тестування є мова Gherkin. Gherkin не задає жорстких формальних правил для опису тестових сценаріїв, але використовує спеціальний набір ключових слів для структурування сценарію та опису необхідних артефактів тестування.

Ключові слова мови Gherkin:

- Feature: Ключове слово, що передує назві користувацької історії.

- As a: Назва або тип користувача, що стосується даної історії.
- In order to: Цілі або мета користувацької історії.
- I want to: Опис очікуваного результату чи цінності для користувача.
- Scenario: Початок кожного сценарію тестування з зазначенням його назви та мети. Якщо історія містить кілька сценаріїв, вказується порядковий номер.
- Given: Передумови. Якщо передумов декілька, кожна нова умова починається з нового рядка за допомогою слова «And».
- When: Подія, що запускає сценарій. Для складних подій, деталі описуються з нового рядка за допомогою «And» та «But».
- Then: Результат, який спостерігає користувач після виконання всіх кроків сценарію. Якщо результат складний, деталі описуються з нового рядка за допомогою «And» та «But».
- And: Допоміжне слово для логічної кон'юнкції.
- But: Допоміжне слово для логічного заперечення.

```gherkin
Feature: Debts
    as a user
    I want to be able to create, update and delete debts
    So that I can track the debts

    @active
    Scenario: Create debt for a specific student
    Given The API is running
    #precondition
      #Creating a new student
      When I create a new student with the following details:
      | name     | age | sex    | fearFactor     |
      | Andrii   | 28  | Male   | 1              |
      Then the response status code should be 201

    #executing the test
    When I create a debt for the student with the amount 1000
    Then the response status code should be 201
    And The debt should be created with the amount 1000
    And Id is assigned to the debt
    And The monthly percent of the newly created debt must be 0
```

Рис 2.4. Код реалізації умов на ситаксисі мови Gherkin

Мова Gherkin, призначена для опису тестових сценаріїв, підтримує можливість додавання коментарів за допомогою символу хеш (#) на початку будь-якого рядка у файлі. Щоб підвищити читабельність тексту, дозволено довільно розташовувати відступи та переносити рядки.

2.6.2. Моделювання фреймворку автоматизованого тестування

Основою для створення фреймворку стала методологія BDD (Behavior Driven Development). Система, що розробляється, використовує BDD фреймворк Cucumber.js.

Застосування BDD дозволяє створювати сценарії автоматизованого тестування звичайною мовою, що значно знижує бар'єр для входу нових розробників автотестів.

Для розробки обрана мова програмування JavaScript, яка є однією з найпоширеніших мов на сьогоднішній день.

Як апаратну платформу для запуску тестів з використанням фреймворку можна використовувати будь-який комп'ютер/сервер/віртуальну машину з мінімальною конфігурацією.

Мінімальні вимоги до конфігурації:

- Процесор: Intel Core i3 або аналогічний;
- Оперативна пам'ять: від 4 ГБ (рекомендовано 8 ГБ);
- Місце на диску: мінімум 10 ГБ (рекомендовано не менше 50 ГБ для довгострокового зберігання звітів та логів);
- Операційна система: Windows 10, Linux (сучасні дистрибутиви, такі як Ubuntu 20.04+), macOS 10.15 Catalina і вище.

Таке обладнання дозволить забезпечити стабільну роботу фреймворку та виконання тестів у різних середовищах розробки та тестування.

2.6.3. Технологічне забезпечення задачі

Технологічне забезпечення задачі автоматизованого тестування включає декілька ключових аспектів, які гарантують ефективне виконання тестових сценаріїв та надійність результатів тестування.

Збір інформації

Збір інформації в рамках системи полягає в отриманні готових тестових сценаріїв, які необхідно виконати. Ці сценарії можуть бути створені як вручну, так і автоматично, залежно від вимог та особливостей тестованої системи.

Подання інформації

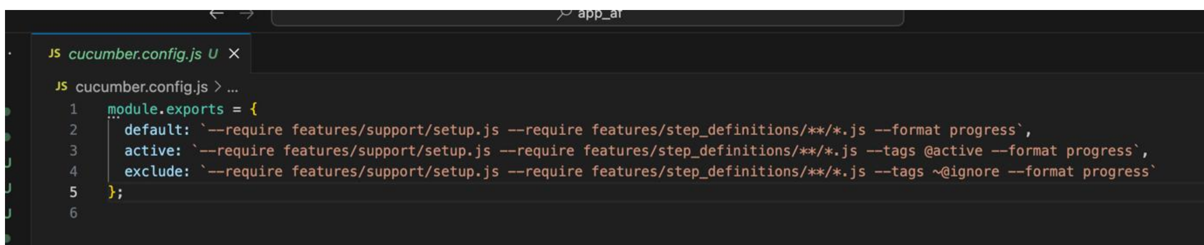
Подання тестових сценаріїв реалізується за допомогою сучасних бібліотек для unit-тестування. Серед найпоширеніших бібліотек можна виділити Mocha та Jest, які дозволяють структурувати тести, забезпечувати їх гнучке налаштування та виконання.

Інтеграція з фреймворками

Інтеграція тестових сценаріїв з фреймворками автоматизації, такими як Cucumber.js, дозволяє використовувати методологію BDD (Behavior Driven Development), що спрощує написання тестів зрозумілою мовою та підвищує зручність роботи з тестами для команд розробки та тестування.

Приклад зв'язку класу для запуску з тестовими сценаріями та набором тестових кроків

Для забезпечення виконання тестів необхідно реалізувати зв'язок між тестовими сценаріями і тестовими кроками. У JavaScript це може бути клас або файл, який використовує конфігурацію для визначення тестових методів та зв'язку з конкретними сценаріями і тестовими кроками:



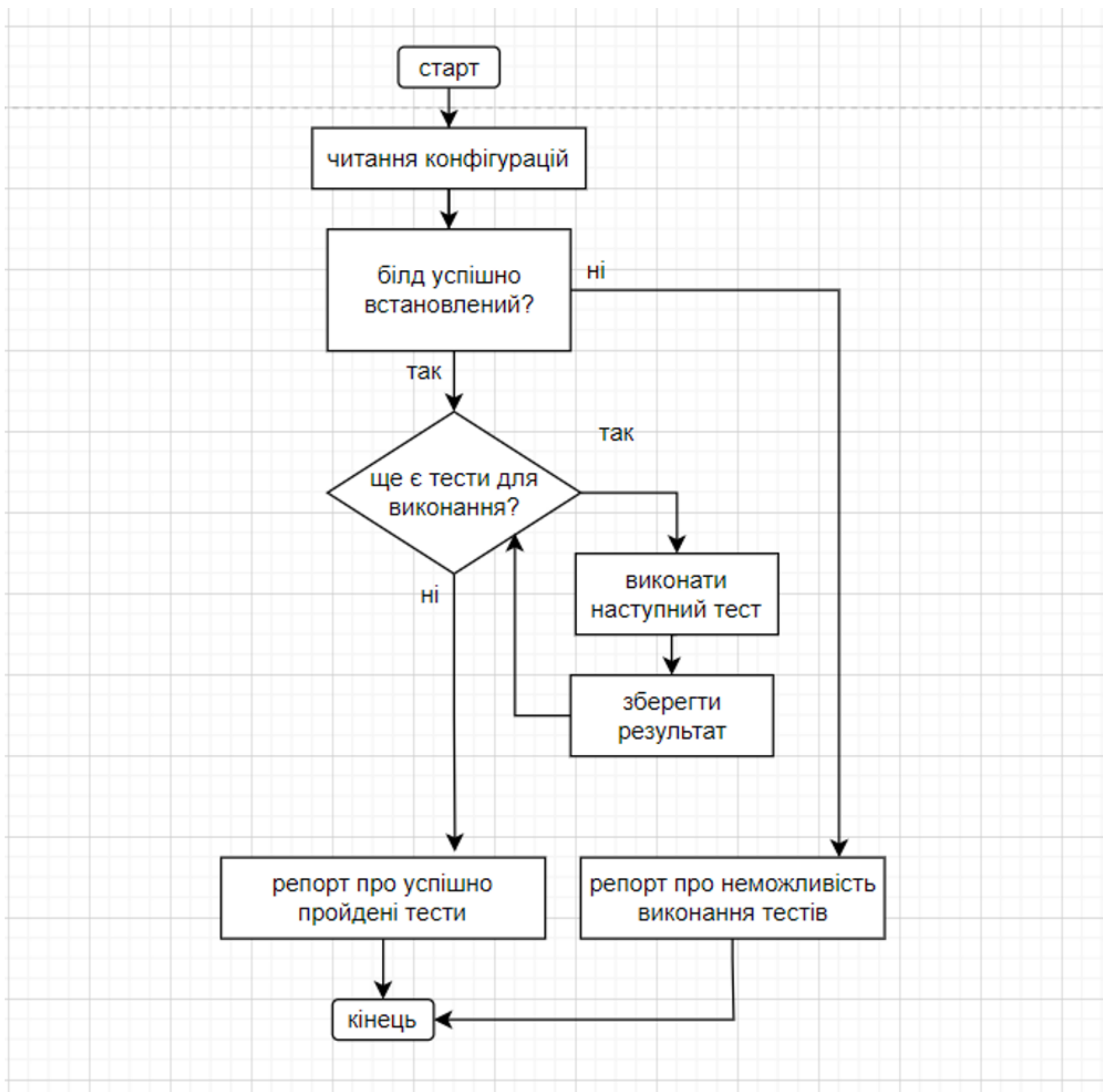
Сценарії повинні знаходитися в папці "features", що вказана в параметрі features конфігурації запуску.

У цьому прикладі створюється тестовий клас CustomWorld, який налаштовується для запуску тестів із використанням Cucumber.js. Тести налаштовуються на взаємодію з API, запускаючи сервер, відправляючи запити та перевіряючи відповіді.

Для створення звітів рекомендовано використовувати фреймворк cucumber-html-reporter, який надає розширені можливості для перегляду результатів виконання тестів та додавання логів.

Структура і виконання автоматизованого тесту

Послідовність дій тесту може бути описана наступним чином:



Важливі етапи в процесі виконання окремого тесту

1. Створення запиту

   Підтримувані методи запитів: GET, HEAD, POST, PUT, PATCH, OPTIONS, DELETE.

2. Установка перевірок (Assertions)
   Assertions пишуться безпосередньо у тестовому коді після виконання HTTP-запиту. Вони використовуються для перевірки відповідей сервера. Ви можете встановити перевірку для будь-якої частини відповіді на ваш вибір.

Assertions підтримують безліч порівнянь. Ось деякі з них:

```
assert.strictEqual(1, 1);
assert.notStrictEqual(1, 2);
assert.equal(1, '1');
assert.notEqual(1, 2);
assert.deepStrictEqual({ a: 1 }, { a: 1 });
assert.notDeepStrictEqual({ a: 1 }, { a: 2 });
assert.ok(true);
assert.ok(1);
assert.strictEqual(typeof 123, 'number');
assert.strictEqual(typeof 'hello', 'string');
const value = 5;
assert(value >= 1 && value <= 10);
const obj = { name: 'John' };
assert.property(obj, 'name');
assert.propertyVal(obj, 'name', 'John');
const str = 'hello world';
assert.include(str, 'world');
assert.notInclude(str, 'foo');
```

Якщо все було налаштовано та сконфігуровано правильно, наступним кроком є запуск тестів за допомогою Cucumber. Cucumber знайде

прив'язані до нього тестові сценарії у форматі Gherkin та виконає їх послідовно.

Після виконання тестів корисно отримати та зберегти наступну інформацію для подальшого аналізу:

- Час початку виконання тесту: Запис моменту, коли тест почав виконуватися.
- Час завершення виконання тесту: Запис моменту, коли тест завершився.
- Результат тесту: Відмітка про успішне проходження (passed), провал (failed) або блокування (blocked) тесту.
- Повний лог виконання тесту: Детальний журнал з інформацією про кожен етап виконання тесту, включаючи часові мітки для кожного запису.

За результатами виконання буде сформований набір файлів із результатами для фреймворку cucumber-html-reporter. Ці файли можна знайти у папці reports. Для перегляду звіту можна скористатися засобами фреймворку cucumber-html-reporter або відкрити файл cucumber-report.html.

JS generateReport.js > options

```javascript
const reporter = require('cucumber-html-reporter');
const fs = require('fs');
const path = require('path');

const reportDir = './reports';
const jsonFile = path.join(reportDir, 'cucumber_report.json');
const htmlFile = path.join(reportDir, 'cucumber_report.html');

if (!fs.existsSync(reportDir)) {
  fs.mkdirSync(reportDir);
}

const options = {
  theme: 'bootstrap',
  jsonFile: jsonFile,
  output: htmlFile,
  reportSuiteAsScenarios: true,
  scenarioTimestamp: true,
  launchReport: true,
  metadata: {
    "App Version": "1.0.0",
    "Test Environment": "STAGING",
    "Browser": "Chrome  89.0.4389.82",
    "Platform": "Windows 10",
    "Parallel": "Scenarios",
    "Executed": "Remote"
  }
};

try {
  if (fs.existsSync(jsonFile)) {
    reporter.generate(options);
  } else {
    console.error(`JSON report file not found: ${jsonFile}`);
  }
} catch (error) {
  console.error('Error generating report:', error);
}
```

reports > {} cucumber_report.json > ...


```
2    {
3      "description": "    as a user\n    I want to be able to create, update and delete debts\n    So that I can track
4      "elements": [
5        {
6          "description": "",
7          "id": "debts;create-debt-for-a-specific-student",
8          "keyword": "Scenario",
9          "line": 6,
10         "name": "Create debt for a specific student",
11         "steps": [
12           {
13             "arguments": [],
14             "keyword": "Given ",
15             "line": 7,
16             "name": "The API is running",
17             "match": {
18               "location": "features/step_definitions/student_steps.js:12"
19             },
20             "result": {
21               "status": "passed",
22               "duration": 263666
23             }
24           },
25           {
26             "arguments": [
27               {
28                 "rows": [
29                   {
30                     "cells": [
31                       "name",
32                       "age",
33                       "sex",
34                       "fearFactor"
35                     ]
36                   },
37                   {
38                     "cells": [
39                       "Andrii",
40                       "28",
```


Ln 1, Col 1   Spaces: 2   UTF-8   LF   {}

Вікно з результатами тесту відображається таким чином:

Feature: Students 2 1

As a user
I want to be able to create students
So that I can track their data afterwards

@active
Scenario: 2 1 1
Create a student

Given The API is running < 1ms
When I create a new student with the following details: 2ms

name	age	sex	fearFactor
Greg	25	Male	1

Then the response status code should be 200 Show Error - < 1ms

```
AssertionError [ERR_ASSERTION]: Expected status code 200,
   + expected - actual

   -201
   +200

   at CustomWorld.(anonymous) (/Users/andrii.fesenko/Desk
```

And The parameter 'name' should be equal to 'Greg' < 1ms

Cucumberjs Report api-automation Report generated a few seconds ago

All Scenarios 4
Passed Scenarios 3
Failed Scenarios 1

2 Features: 50% 50% Passed Failed
4 Scenarios: Failed 1 (25%) 75% Passed Failed

Expand All | Collapse All

Metadata
App Version: 1.0.0
Browser: Chrome 89.0.4389.82
Parallel: Scenarios
Test Environment: STAGING
Platform: Windows 10
Executed: Remote

Feature: Debts 1
Feature: Students 2 1

Рис. 2.5. Результати тесту в системі cucumber-html-report

Згідно з цим звітом, можна легко зрозуміти, що статус виконання тесту – «Passed», тобто тест пройдено успішно. У звіті також видно, що всі кроки позначені зеленими маркерами, що означає успішне виконання кожного кроку. У разі неуспішного проходження тесту, крок, на якому тест завершився помилкою, і всі наступні кроки будуть позначені іншими кольорами. Статус тесту в такому випадку зміниться на «Failed» (помилка). Таким чином, створивши кілька модулів та сервісів, і написавши тестовий сценарій зрозумілою мовою, за допомогою Cucumber можна запустити тестування API.

Feature: Debts 1

as a user
I want to be able to create, update and delete debts
So that I can track the debts

Scenario:
Create debt for a specific student 8

Given The API is running < 1ms
When I create a new student with the following details: 48ms

name	age	sex	fearFactor
Andrii	28	Male	1

Then the response status code should be 201 < 1ms
When I create a debt for the student with the amount 1000 3ms
Then the response status code should be 201 1ms
And The debt should be created with the amount 1000 < 1ms
And Id is assigned to the debt < 1ms
And The monthly percent of the newly created debt must be 0 < 1ms

У даному розділі було розглянуто питання створення програмного засобу для автоматизації тестування у вигляді єдиного фреймворка. Також обговорено проблеми та недоліки, які можна вирішити за допомогою цієї системи.

Обговорено важливість самого процесу тестування в рамках діяльності компанії-розробника програмного забезпечення, а також значення автоматизації тестування.

Була описана постановка задачі зі створення фреймворка, включаючи основні цілі, моделі вирішення та порядок роботи користувачів із системою.

Як створити фреймворк для автоматизації тестування:
Мова програмування: Обираючи мову програмування, ми забезпечуємо можливість розширювати фреймворк та уникати обмежень готових рішень.
Засоби збирання: Дозволяють легко додавати нові бібліотеки і запускати тести з консолі.
xUnit-бібліотеки: Роблять фреймворк тестовим і дозволяють запускати тести паралельно.
Моделі: Структурують фреймворк, дозволяючи використовувати різні його частини як модульні блоки.
Серіалізація моделей: Використання анотацій gson для швидкої серіалізації REST API-об’єктів у адаптерах під час підготовки тестових даних.
Репортинг: Полегшує процес дебагу.
Utils/ресурси: Допоміжні класи або файли для будь-яких дій, не пов'язаних із переліченими пунктами.

Статус коди

Документація API часто виглядає однаково, де видно, який метод використовувати, який URL, body, params headers і які відповіді мають приходити.

CRUD – абревіатура, яка позначає наступні операції:

- Create – створення
- Read – читання
- Update – оновлення
- Delete – видалення

GET – для отримання інформації від сервера.
POST – для передачі інформації від клієнта на сервер і підтвердження її отримання.
PUT – для оновлення інформації на сервері
DELETE - для видалення інформації з серверу

Cucumber

Фреймворк Cucumber використовується як частина BDD-підходу до розробки та як окремий інструмент для автоматизованого тестування.

Багато тестувальників вважають Cucumber лише обгорткою над кодом для краси, ігноруючи основну ідею концепції BDD. Це призводить до того, що функції інструмента використовуються неправильно, що суттєво ускладнює досягнення бажаного результату – зробити процес розробки зрозумілим і прозорим для всіх учасників.

BDD підхід об'єднує найкращі практики TDD, де розробка коду відбувається у прямій залежності від тестів. Це дозволяє зменшити розрив між технічними та нетехнічними учасниками команди.

Cucumber є популярним кросплатформенним BDD-фреймворком, який використовується для автоматизованого тестування, навіть якщо команда не використовує BDD-підхід до розробки.

Причини використання BDD-фреймворку Cucumber:

- Ідеальний варіант – розробка вже ведеться за BDD.
- Засіб для спілкування/зворотнього зв’язку із замовником – спілкування зрозумілою мовою.
- Жива документація – замість статичних тестів використовуються сценарії, які відображають актуальний стан проекту.

Антипатерни Cucumber

1. Послідовність дій замість бізнес-кроків

Автоматизація acceptance сценаріїв має орієнтуватися на бізнес-вимоги, а не лише на конкретну реалізацію.

1. Сценарії без чітких формулювань і промовних назв

Сценарії мають бути зрозумілими всім членам команди, включаючи розробників, менеджерів і аналітиків.

2. Занадто прості сценарії без використання засобів Cucumber

Використання списків, таблиць або багаторядкового тексту для передачі даних і полегшення сприйняття сценарію.

3. Плутанина у ключових словах Given, When, Then

Ключові слова мають відрізнятися не лише синтаксично, але і за змістом, що в них вкладено.

4. Занадто великі таблиці в Scenario Outline

Використовувати Scenario Outline з обережністю, щоб не ускладнити сценарії і не знизити їх читабельність.

Висновки по Cucumber:

- Робити сценарії виразними і зрозумілими.
- Писати бізнес-кроки і уникати зайвої деталізації.
- Використовувати Scenario Outline та Background обережно.
- Якісні та продумані сценарії полегшують підтримку і зменшують непорозуміння в команді.

РОЗДІЛ 3. ТЕСТУВАННЯ СИСТЕМИ ТА ОЦІНКА ЇЇ ЕФЕКТИВНОСТІ

3.1. Апі який був розроблений для цього проекту

API для Керування Боргами Студентів

Цей API дозволяє ефективно керувати процесом стягнення студентських боргів. Після впровадження, клієнти зможуть:

- Вносити, змінювати та переглядати інформацію про студентів, їх борги та колекторів у базі даних.
- Автоматично призначати колекторів та встановлювати дати зустрічей для врегулювання боргів.
- Переглядати та видаляти вже заплановані зустрічі.

Примітка: Кожна REST-кінцева точка (наприклад, /student) діє як окремий сервіс, схожий на мікросервісну архітектуру. Якщо одному сервісу необхідні дані від іншого, він здійснює HTTP-запит до потрібного сервісу. Цей продукт перебуває на етапі розробки і активований для тестування. Перезапуск програми відновлює базу даних до початкового стану.

Зустрічі (Appointments)

- GET - /api/appointment

  Отримати інформацію про всі заплановані зустрічі.
- POST - /api/appointment

  Додати нову зустріч. Дати починаються з наступного дня. Один колектор може мати одну зустріч на день. Ризик студента визначає необхідний рівень колектора. Оцінки ризику: 1 (низький), 2 (середній), 5 (високий). Сеньйорність колектора має бути вдвічі більшою за оцінку ризику.
- GET - /api/appointment/{id}

  Отримати інформацію про конкретну зустріч за її ID.
- DELETE - /api/appointment/{id}

  Скасувати зустріч за її ID.

Колектори (Collectors)

- GET - /api/collector

  Отримати інформацію про всіх колекторів.
- POST - /api/collector

  Додати нового колектора до системи.
- PUT - /api/collector

  Оновити інформацію про колектора.
- GET - /api/collector/{id}

  Отримати інформацію про конкретного колектора за його ID.
- DELETE - /api/collector/{id}

  Видалити колектора за його ID.

Заборгованості (Debts)

- GET - /api/debt
  Переглянути інформацію про всі борги студентів. Кожен борг перераховується з урахуванням відсотків. Пом'якшення застосовуються в залежності від віку та статі.
- POST - /api/debt
  Додати новий борг до системи. Останній день оновлення встановлюється як поточна дата.
- GET - /api/debt/{id}
  Переглянути інформацію про конкретний борг за ID.
- DELETE - /api/debt/{id}
  Видалити борг за його ID.

Студенти (Students)

- GET - /api/student
  Переглянути інформацію про всіх студентів у системі.
- POST - /api/student
  Додати нового студента до системи.
- PUT - /api/student
  Оновити інформацію про студента.
- GET - /api/student/{id}
  Переглянути інформацію про конкретного студента за його ID.
- DELETE - /api/student/{id}
  Видалити студента за його ID.

Цей API надає повний набір інструментів для управління студентськими боргами, автоматизуючи процеси та покращуючи ефективність роботи з боржниками.

3.2. Схеми данних та її типи

Appointment включає в себе такий набір данних

{

id: “”,

date: : “”,

studentId: “”,

collectorId: “”,

debtId: “”

}

```
const newAppointment = {
    id: generateId('appointment', appointments),
    date: formattedDate,
    studentId: studentId,
    collectorId: collector.id,
    debtId: debtId // Assigning the student's debt ID to the appointment
};
```

Collector включає в себе такий набір данних

{

id: “”,

name: “”,

seniority: “”

}

```
app.post('/api/collector',[
    body('name').notEmpty(),
    body('seniority').notEmpty().isInt({ min: 2, max: 10 })
```

Debt включає в себе такий набір данних

```
{
        id: “”,
        studentId: “”,
        amount: “”,
        totalAmount: “”,
        monthlyPercent: “”,
        creationDate: “”,
        lastUpdateDate: “”
}
```

```
const newDebt = {
    id: generateId(),
    studentId: studentId,
    amount: amount,
    totalAmount: totalAmount,
    monthlyPercent: interestRate * 100, // Convert interest rate to percentage
    creationDate: currentDate,
    lastUpdateDate: currentDate
};
```

Student включає в себе такий набір данних

```
{
        id: “”,
        name: “”,
        age: “”,
        sex: “”,
        fearFactor: “”,
}
```

```
const newStudent = {
    id: generateId(),
    name: name,
    age: age,
    sex: isFemale,
    fearFactor: fearFactor
};
```

3.3. Тестування документації

Тестування потрібно і можна розпочинати вже на етапі аналізу вимог та супровідної документації до продукту. Таким чином, вже на цьому етапі виявляються перші невідповідності або можливі баги:

1. Стать студента зберігається як пара ключ-значення sex та boolean, що прийнятно лише за умови більш інформативного декларування самого ключа, наприклад, male (чоловік) або female (жінка).
2. Незрозумілість у визначенні статі студента (sex – boolean).
3. Формат часу зберігається у дуже чіткому типі, що може створювати велику кількість помилок.
4. Так само про формат відображення заборгованості. Надмірно чіткі формати призводять до копчування помилок.
5. Немає початкової заборгованості студента. Відсутня інформація студента на початок програми (Наприклад кожен студент має 0 заборгованості на початку).
6. Неможливість отримати заборгованість на конкретну дату.
7. Неможливість відсортувати заборгованість по студенту.
8. Немає можливості надсилати запити PUT для ресурсу Appointment.
9. Немає можливості надсилати запити PUT для ресурсу Debt.

3.4. Тестування додатку за допомогою розробленого фреймворку

Зустріч/Appointment

- відправка запиту з неіснуючим debtId
  - Нові зустрічі отримують ідентифікаційний номер id = -1, якщо кількість профілів у базі досягає 15;

Колектор/Collector

POST - /api/collector/{id}

- Нові профілі колекторів створюються з id = -1, якщо кількість профілів у базі перевищує 15;

– Новий профіль колектора створюється навіть у випадках, коли вік студента вказано як 0 або від'ємне значення.

Заборгованість/Debt

POST - /api/debt/{id}

– Інформація про новий борг додається до системи, навіть якщо відсоткова ставка (monthlyPercent) вказана як 0 або від'ємне значення;
– Інформація про новий борг додається до системи, навіть якщо вказано неіснуючий ідентифікаційний номер студента.

Студент/Student

– Нові профілі студентів отримують id = -1, якщо кількість профілів у базі перевищує 15;
– Новий профіль студента створюється, навіть якщо вік студента вказано як 0 або від'ємне значення;
– Новий профіль студента створюється, навіть якщо показник ризику відрізняється від дозволених значень у документації (1, 2, 5);
– Новий профіль студента створюється, навіть якщо показник ризику має від'ємне значення.

Висновки до розділу

У цьому розділі було розглянуто структуру розроблюваної системи для автоматизованого тестування, а також описано деталі реалізованих функцій. В окремому підрозділі детально висвітлена архітектура проєкту, включаючи використані патерни, засоби, інструменти та мови програмування, які були застосовані.

Таким чином, було представлено загальну архітектуру системи та оцінено можливості застосування розробленого інструменту для конкретних

проєктів. Крім того, був наведений приклад реалізації системи тестування, що на практиці демонструє зручність розробки нових тестів.

На завершення, можна зробити висновок, що система є готовим до використання програмним засобом, який завдяки відкритому коду готовий до подальшого розширення функціоналу та можливостей.

Проте, тестована система стягнення заборгованості на даному етапі розробки має багато недоліків та потребує значних коригувань. Після фіксації всіх невідповідностей інформація про них повинна бути передана розробникам для внесення відповідних змін. Після отримання оновленої версії API від розробників, рекомендується провести регресійне тестування з детальним аналізом результатів

РОЗДІЛ 4. ЕКОНОМІЧНИЙ РОЗДІЛ

4.1. Розрахунок трудомісткості та вартості розробки програмного продукту

Під час розробки додатку для автоматизованого тестування API було проведено розрахунок трудомісткості та вартості, враховуючи наступні вихідні дані:

- Передбачуване число операторів програми: 2500;
- Коефіцієнт складності програми: 1,5;
- Коефіцієнт корекції програмного продукту в ході його розробки: 0,1;
- Годинна заробітна плата розробника: 200 грн/год (згідно зі статистикою з сайту WorkUA для України);
- Коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі: 1,3;
- Коефіцієнт кваліфікації програміста, залежний від стажу роботи: 0,8;

- Вартість машино-години ЕОМ: 25 грн/год.

Цінова політика використання інструменту для командної роботи, такого як GitHub, залежить від обраного тарифного плану. Існують безкоштовні тарифи для базового використання. Для командної роботи потрібно придбати платний тарифний план, який коштує 500 грн на користувача на місяць. Припустимо, що ви використовуєте інструмент 8 годин на день протягом 22 робочих днів на місяць, то погодинна вартість користування інструментом становить приблизно 2,84 грн/год.

Заробітну плату розробника за годину було розраховано за даними з сайту для пошуку роботи «WORK.UA». Середня заробітна плата розробника в Україні складає приблизно 35000 грн на місяць станом на травень 2024 рік. Припустимо, стандартний робочий тиждень становить 40 годин, тоді погодинна заробітна плата складає 200 грн/год.

Трудомісткість розробки програмного продукту можна розрахувати за допомогою формули:

$$t = t_o + t_u + t_a + t_n + t_{отл} + t_д \text{, людино-годин,} \tag{3.1}$$

де t_o – витрати праці на підготовку й опис поставленої задачі (приймається 50)

t_u – витрати праці на дослідження алгоритму рішення задачі;

t_a – витрати праці на розробку блок-схеми алгоритму;

t_n – витрати праці на програмування по готовій блок-схемі;

$t_{отл}$ – витрати праці на налагодження програми на ЕОМ;

$t_д$ – витрати праці на підготовку документації.

Складові витрати праці визначаються через умовне число операторів у програмному забезпеченні, яке розробляється.

Умовне число операторів (підпрограм):

$$Q = q \cdot C \cdot (1 + p), \tag{3.2}$$

де q – передбачуване число операторів (2500);

C – коефіцієнт складності програми (1,5);

p – коефіцієнт корекції програмного продукту в ході його розробки (0,1).

Таким чином, умовне число операторів становить:

Q = 2500 · 1,5 · (1 + 0,1) = 4125;

Витрати праці на вивчення опису задачі t_u визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_u = \frac{Q \cdot B}{(75..85) \cdot k} \text{ , людино-годин,} \tag{3.3}$$

де B – коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі (1,3);

k – коефіцієнт кваліфікації програміста, обумовлений стажем роботи з даної спеціальності (0,8);

З урахуванням коефіцієнта кваліфікації k = 0,8, отримуємо витрати праці на вивчення опису завдання:

t_u = (4125· 1,3) / (85 · 0,8) = 78,07 людино-годин.

Витрати праці на розробку алгоритму рішення задачі визначаються за формулою:

$$t_a = \frac{Q}{(20 \ldots 25) \cdot k} \text{ , людино-годин.} \tag{3.4}$$

де Q – умовне число операторів програми;

k – коефіцієнт кваліфікації програміста.

Підставивши відповідні значення в формулу (3.4), отримаємо:

t_a = 4125/ (20 · 0,8) = 257,81 людино-годин.

Витрати на складання програми по готовій блок-схемі:

(3.5)

$$t_n = \frac{Q}{(20\dots25)\cdot k}, \text{ людино-годин.}$$

t_n = 4125/ (25 · 0,8) = 206,25 людино-годин.

Витрати праці на налагодження програми на ЕОМ:

– за умови автономного налагодження одного завдання:

(3.6)

$$t_{отл} = \frac{Q}{(4..5)\cdot k}, \text{ людино-годин.}$$

$t_{отл}$ 125/ (5 · 0,8) = 1031,25 людино-годин.

– за умови комплексного налагодження завдання:

(3.7)

$$t_{отл}^{k} = 1{,}5 \cdot t_{отл}, \text{ людино-годин.}$$

$t_{отл}^{к}$ = 1,5 · 1031,25 = 1546,8 людино-годин.

Витрати праці на підготовку документації:

(3.8)

$$t_{\partial} = t_{\partial p} + t_{\partial o};$$

де $t_{\partial p}$ – трудомісткість підготовки матеріалів і рукопису;

(3.9)

$$t_{\partial p} = \frac{Q}{(15..20) \cdot k}$$, людино-годин.

(3.10)

$$t_{\partial o} = 0{,}75 \cdot t_{\partial p};$$

де $t_{\partial o}$- трудомісткість редагування, печатки й оформлення документації:

Підставляючи відповідні значення, отримаємо:

$t_{\partial p}$ = 4125/ (18 · 0,8) = 286,45 людино-годин.

$t_{\partial o}$= 0,75 · 286,45 = 214,83 людино-годин.

t_{∂} = 286,45 + 214,83 = 501,28 людино-годин.

Повертаючись до формули (3.1), отримаємо повну оцінку трудомісткості розробки програмного забезпечення:

t = 50 + 78,07 + 257,81 + 206,25 + 1031,25 + 286,46 = 1909,84 людино-годин.

За результатами розрахунків, загальна трудомісткість розробки даного програмного продукту складає 1909,84 людино-годин.

4.2. Розрахунок витрат на створення програми

Витрати на створення ПЗ $K_{ПО}$ включають витрати на заробітну плату виконавця програми $З_{ЗП}$ і витрат машинного часу, необхідного на налагодження програми на ЕОМ:

$$K_{ПО} = З_{ЗП} + З_{МВ}, \text{ грн},$$

(3.11)

$З_{ЗП}$– заробітна плата виконавців, яка визначається за формулою:

$$З_{ЗП} = t \cdot С_{ПР}\text{, грн,} \quad (3.12)$$

де t - загальна трудомісткість, людино-годин;

$С_{ПР}$– середня годинна заробітна плата програміста, грн/година.

З урахуванням того, що середня годинна зарплата розробника становить 200 грн / год, отримуємо:

$$З_{ЗП} = 1909{,}84 \cdot 200 = 381968 \text{ грн.}$$

Вартість машинного часу $З_{МВ}$, необхідного для налагодження програми на

ЕОМ, визначається за формулою: (3.13)

$$З_{мв} = t_{отл} \cdot С_{мч}\text{, грн,}$$

де $t_{отл}$ – трудомісткість налагодження програми на ЕОМ, год;

$С_{МЧ}$ – вартість машино-години ЕОМ, грн/год.

$$З_{МВ} = 1031{,}25 \cdot 25 = 25781{,}25\text{грн.}$$

Звідси витрати на створення програмного продукту:

$$Кпо = 381968 + 25781{,}25 = 407749{,}25\text{грн.}$$

Очікуваний період створення ПЗ:

$$\mathbf{T} = \frac{\mathbf{t}}{\mathbf{B_k} \cdot \mathbf{F_p}}\text{, міс,} \quad (3.14)$$

де B_k – число виконавців;

F_p – місячний фонд робочого часу (при 40 годинному робочому тижні F_p = 176 годин).

Звідси, за формулою (3.14) витрати на створення програмного продукту:

$$T = 1919{,}84 / (1 \cdot 176) = 10{,}90 \text{ міс.}$$

Висновок: Додаток для автоматизованого тестування API створено з метою підвищення ефективності тестування та скорочення витрат часу і ресурсів. Вартість розробки складає приблизно 407749,25 грн, а орієнтовний час розробки становить 10,90 місяці при стандартному 40-годинному робочому тижні і 176-годинному робочому місяці.

ВИСНОВКИ

Метою цієї випускної кваліфікаційної роботи було створення системи для автоматизації тестування API. В результаті виконаної роботи була розроблена система, яка забезпечує роботу з API, генерацію тестових даних, обробку логів, а також створення звітів про тестування за допомогою зовнішнього інструменту Allure.

При розробці враховувалися особливості конкретного API. Запропоноване рішення дозволить зменшити час, необхідний для тестування, і скоротити кількість фахівців, необхідних для його проведення, що, безумовно, підвищить ефективність компанії, яка впроваджуватиме цю систему у роботу тестового відділу.

Апробація системи проводилася на реальному завданні тестування API, що дозволило оцінити її зручність і практичність у застосуванні. Подальший розвиток системи передбачає розширення існуючих функцій та додавання нових, таких як розробка універсальних тестових кроків, оптимізація та прискорення роботи, а також створення системи аналізу причин помилок, виявлених під час тестування.

При розробці клієнт-серверного програмного забезпечення важливою є документація до API, яка забезпечує доступ до даних. Дуже часто клієнтська та серверна частини ПЗ розробляються різними командами, які можуть працювати на великій відстані або навіть у різних часових поясах. У таких випадках документація стає основним засобом організації взаємодії між цими командами, тому її якість є надзвичайно важливою.

ПЕРЕЛІК ПОСИЛАНЬ

1. Cucumber.js: Офіційний сайт та документація Cucumber для JavaScript https://github.com/cucumber/cucumber-js
2. Mocha: Популярний JavaScript фреймворк для тестування. https://mochajs.org/
3. SuperTest: Бібліотека для тестування HTTP з JavaScript. https://github.com/visionmedia/supertest
4. Allure Framework: Інструмент для генерації звітів про тестування. https://docs.qameta.io/allure/
5. Node.js: Платформа для виконання JavaScript на сервері. https://nodejs.org/en/docs/
6. Axios: Бібліотека для виконання HTTP запитів. https://axios-http.com/docs/intro
7. Cucumber HTML Reporter: Бібліотека для створення HTML-звітів для Cucumber. https://github.com/gkushang/cucumber-html-reporter
8. Cucumber.js with Allure: Інструкції щодо інтеграції Cucumber.js з Allure для генерації звітів. https://docs.qameta.io/allure/#_javascript_2
9. Setting up Mocha with Cucumber.js: Інструкції щодо налаштування Mocha з Cucumber.js. https://github.com/cucumber/cucumber-js/blob/main/docs/support_files.md#using-mocha
10. Cucumber.js Examples: Приклади використання Cucumber.js для написання тестів. https://github.com/cucumber/cucumber-js/tree/main/examples
11. Creating Test Reports: Інструкції щодо створення звітів про тестування за допомогою Cucumber HTML Reporter. https://github.com/gkushang/cucumber-html-reporter#options

12. Log4js: Бібліотека для логування у JavaScript. https://log4js-node.github.io/log4js-node/
13. Generating HTML Reports: Створення HTML-звітів з JSON-файлів. https://github.com/gkushang/cucumber-html-reporter#usage
14. BDD (Behavior Driven Development): Методологія розробки на основі поведінки. https://cucumber.io/docs/bdd/
15. IntelliJ IDEA: Інтегроване середовище розробки (IDE) для JavaScript та інших мов програмування. https://www.jetbrains.com/idea/documentation/
16. Git and GitLab: Системи контролю версій та спільної роботи над проєктами. https://git-scm.com/doc
17. GitLab https://docs.gitlab.com/

ДОДАТОК А

КОД ПРОГРАМИ

Server.js

```
const express = require('express');
const bodyParser = require('body-parser');
const { body, validationResult } = require('express-validator');
const moment = require('moment');

const log4js = require('../log4js.config.js');
const logger = log4js.getLogger();

const app = express();
const PORT = process.env.PORT || 3000;

// Middleware
app.use(express.json());

// Sample data (replace with your database implementation)
let appointments = [
  { id: 'ap1', debptid: 'debt1', collectorId: 'c3', date: '2025-03-03' }
];
let collectors = [
  { id: 'c1', name: 'Sam Kushi', seniority: 2 },
  { id: 'c2', name: 'Emi Bykon', seniority: 2 },
  { id: 'c3', name: 'Andrew Macks', seniority: 4 },
  { id: 'c4', name: 'Alan Dok', seniority: 4 },
  { id: 'c5', name: 'Peter Sdoa', seniority: 10 },
  { id: 'c6', name: 'Michael Kon', seniority: 10 },
];
let debts = [
  { id: 'debt1', studentId: 's1', amount: 1000, totalAmount: 1040.67,
monthyPercent: 20, creationDate: '2024-04-20', lastUpdateDate: '' },
  { id: 'debt2', studentId: 's2', amount: 2000, totalAmount: 2081.21,
monthyPercent: 30, creationDate: '2024-04-22', lastUpdateDate: '' }
];
let students = [
  { id: 's1', name: 'John', age: 18, sex: false, fearFactor: 2 },
  { id: 's2', name: 'Alice', age: 22, sex: true, fearFactor: 1 },
];

function generateId(apiType, items) {
```

```
    logger.info('[generateId] starting to generate id');
    if ((apiType === 'appointment' || apiType === 'collector' || apiType
=== 'student') && items.length >= 15) {
        return -1;
    } else {
        return Math.random().toString(36).substring(2, 10);
    }
}

function isCollectorAvailable(collectorId, appointmentDate,
studentFearFactor) {
    logger.info('[isCollectorAvailable] starting collector search');
    // Check if the collector's fear factor is at least 2 times bigger than the
student's fear factor
    const collector = collectors.find(collector => collector.id ===
collectorId);
    logger.info(`[isCollectorAvailable] Checking if the collector's fear
factor: ${collector.seniority} is at least 2 times bigger than the student's
fear factor: ${studentFearFactor}`);
    logger.info(`[isCollectorAvailable] Checking if the collector:
${collector.id} has no appointments for the provided date:
${appointmentDate}`);
    return collector && collector.seniority >= 2 * studentFearFactor &&
        // Check if the collector has no appointments for the provided date
        !appointments.some(appointment => appointment.collectorId ===
collectorId && appointment.date === appointmentDate);
}

// GET endpoint to get all appointments
app.get('/api/appointment', (req, res) => {
    logger.debug('[API] get all appointments');
    res.json(appointments);
});

// POST endpoint to create appointment
app.post('/api/appointment', [
    body('date').custom((value, { req }) => {
        // Check if the date is in YYYY-MM-DD format
        if (!moment(value, 'YYYY-MM-DD HH:mm:ss', true).isValid()) {
            throw new Error('Date must be in YYYY-MM-DD HH:mm:ss
format');
        }
        // Check if the date is in the future
```

```
    if (moment(value).isBefore(moment().startOf('day'))) {
      throw new Error('Appointment date cannot be in the past');
    }
    return true;
  }),
  body('studentId').notEmpty()
], (req, res) => {
  logger.debug('[API] starting appointment creation');
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    logger.error('[appointment] ', errors.array())
    return res.status(400).json({ errors: errors.array() });
  }

  // Check if there are available collectors for the appointment
  const appointmentDate = req.body.date;
  let formattedDate = moment(appointmentDate, 'YY-MM-DD
HH:mm:ss').format('YY-MM-DD');
  const studentFearFactor = students.find(student => student.id ===
req.body.studentId)?.fearFactor || "Student not found";
  const availableCollectors = collectors.filter(collector =>
isCollectorAvailable(collector.id, formattedDate, studentFearFactor));

  if (availableCollectors.length === 0) {
    return res.status(400).json({ message: 'No available collectors for
the appointment' });
  }

  // Check if the student has any existing debt
  const studentId = req.body.studentId
  logger.info('[appointment] checking if the student has any existing
debts', studentId);
  const studentDebt = debts.find(debt => debt.studentId === studentId);
  const debtId = studentDebt ? studentDebt.id : null;

  // Assign the appointment to the first available collector
  const collector = availableCollectors[0];
  const newAppointment = {
    id: generateId('appointment', appointments),
    date: formattedDate,
    studentId: studentId,
    collectorId: collector.id,
    debtId: debtId // Assigning the student's debt ID to the appointment
```

```
  };
  logger.info('[appointment] creating new appointment ',
newAppointment);
  logger.info('[appointment] assigning the appointment to the first
available collector', collector);

  // Add the new appointment to the appointments array
  logger.info('[appointment] adding the new appointment to the
appointments array ', newAppointment);
  appointments.push(newAppointment);
  res.status(201).json(newAppointment);
  logger.debug('[API] finishing appointment creation');
});

// GET endpoint to get appointment by id
app.get('/api/appointment/:id', (req, res) => {
  logger.debug('[API] starting "get appointment by id" function');
  const appointment = appointments.find(appt => appt.id ===
req.params.id);
  if (!appointment) {
    logger.debug('[appointment] get appointment error', appointment);
    res.status(204)
  } else {
    res.json(appointment);
  }
  logger.debug('[API] finishing "get appointment by id" function');
});

// DELETE endpoint to delete appointment by id
app.delete('/api/appointment/:id', (req, res) => {
  logger.debug('[API] starting appointment deleting');
  const index = appointments.findIndex(appt => appt.id ===
req.params.id);
  if (index === -1) {
    res.status(404).send('Appointment not found');
  } else {
    appointments.splice(index, 1);
    res.status(204).send('appointment has been deleted');
    logger.info('[appointment] appointment has been deleted ',
appointments[index]);
  }
});
```

```
// GET endpoint to get all collectors
app.get('/api/collector', (req, res) => {
   logger.debug('[API] starting "get all collectiors" function');
   res.json(collectors);
});

// POST endpoint to create a collector
app.post('/api/collector',[
   body('name').notEmpty(),
   body('seniority').notEmpty().isInt({ min: 2, max: 10
}).withMessage('seniority of collector must be an integer between 2 and
10')
], (req, res) => {
   logger.debug('[API] starting to create a collecter');
   const errors = validationResult(req);
   if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
   }
   logger.info('[collector] adding new collector ')
   const newCollector = req.body;
   newCollector.id = generateId('collector', collectors)
   collectors.push(newCollector);
   res.status(201).json(newCollector);
   logger.info('[collector] new collector has been added ', newCollector);
});

// PUT endpoint to update collector by id
app.put('/api/collector/:id', (req, res) => {
   logger.debug('[API] starting to uodate collector');
   const { id } = req.params;
   const updatedCollector = req.body;

   const index = collectors.findIndex(collector => collector.id === id);
   if (index !== -1) {
      logger.info('[collector] updating collector info ', collectors[index])
      updatedCollector.id = id; // Assign the existing ID
      collectors[index] = updatedCollector;
      res.json(updatedCollector);
   } else {
      logger.debug(`collector ${id} has not been found`);
      res.status(404).send('Collector not found');
   }
});
```

```
// DELETE endpoint to delete collector by id
app.delete('/api/collector/:id', (req, res) => {
   logger.debug('[API] starting collector deletion');
   const { id } = req.params;
   const index = collectors.findIndex(collector => collector.id === id);
   if (index !== -1) {
      logger.warn(`[collector] deleting the collector
${collectors[index]}`);
      collectors.splice(index, 1);
      res.status(204).send();
      logger.info(`[collector] collector ${id} has been deleted `);
   } else {
      logger.debug(`collector ${id} has not been found`);
      res.status(404).send('Collector not found');
   }
});

// GET endpoint to retrieve all debts
app.get('/api/debt', (req, res) => {
   logger.debug('[API] starting to retrieve all debts');
   res.json(debts);
});

// POST endpoint to add a new debt
app.post('/api/debt', (req, res) => {
   logger.debug('[API] starting new debt addition');
   const { studentId, amount, lastUpdateDate } = req.body;

   // Find the student by their ID
   logger.info(`[debt] Looking for the student ${studentId}`);
   const student = students.find(student => student.id === studentId);
   if (!student) {
      logger.debug(`[debts] student ${studentId} has not been found`);
      return res.status(404).json({ error: 'Student not found' });
   }

   const debtorAge = student.age;
   const isFemale = student.sex;

   // Calculate interest rate based on age and gender
   logger.info('[debt] calculating interest rate');
   let interestRate = 0; // Default interest rate
```

```
    if (debtorAge < 21) {
        interestRate = 0.1; // Apply -10% interest rate for debtors younger
than 21
    }
    if (debtorAge < 18) {
        interestRate = 0.2; // Apply an additional -20% interest rate for
debtors younger than 18
    }
    if (isFemale) {
        interestRate += 0.1; // Apply an additional -10% interest rate for
female debtors
    }

    // Calculate the total debt amount with interest
    let totalAmount = amount
    logger.info(`[debt] Calculating the total debt amount: ${totalAmount}
with interest: ${interestRate}`);

    // Check if the debt creation date is in the future
    const currentDate = new Date();
    logger.info(`[debt] checking if the debt creation date:
${lastUpdateDate || currentDate}  is in the future`);
    if (currentDate < new Date(lastUpdateDate)) {
        const daysSinceCreation = Math.ceil((new Date(lastUpdateDate) -
currentDate) / (1000 * 60 * 60 * 24));
        const dailyRate = interestRate / 30;
        for (let i = 0; i < daysSinceCreation; i++) {
            totalAmount *= (1 + dailyRate); // Apply daily interest rate
            logger.info('applying daily interest rate ', dailyRate)
            logger.info('applying it to the total amount', totalAmount)
        }
    }

    // Create a new debt object
    logger.info('[debt] creating a new debt object');
    const newDebt = {
        id: generateId(),
        studentId: studentId,
        amount: amount,
        totalAmount: totalAmount,
        monthlyPercent: interestRate * 100, // Convert interest rate to
percentage
        creationDate: currentDate,
```

```
        lastUpdateDate: currentDate
    };

    // Add the new debt to the debts array
    logger.info('[debt] adding the new debt to the debts array ', newDebt);
    debts.push(newDebt);

    // Return the newly created debt object in the response
    res.status(201).json(newDebt);
    logger.info('[debt] finishing debt addition');
});

// GET endpoint to retrieve a specific debt by id
app.get('/api/debt/:id', (req, res) => {
    logger.debug('[API] starting to retrieve a specific debt by id');
    const debt = debts.find(debt => debt.id === req.params.id);
    if (debt) {
        logger.info('return debt ', debt)
        res.json(debt);
    } else {
        logger.debug(`debt ${req.params.id} is not found`);
        res.status(204).send('Debt is not found!');
    }
});

// PUT endpoint to create a debt
app.put('/api/debt', (req, res) => {
    logger.debug('[API] Starting debt creation');
    const currentDate = new Date();

    debts.forEach(debt => {
        // Check if lastUpdateDate is today
        logger.info(`[debt] checking if lastUpdateDate:
${debt.lastUpdateDate} is today`);
        const lastUpdateDate = debt.lastUpdateDate ? new
Date(debt.lastUpdateDate) : new Date(debt.creationDate);
        if (lastUpdateDate.toDateString() === currentDate.toDateString()) {
            // If lastUpdateDate is today, do not update data
            logger.info(`[debt] debpt ${debt.id} has not been updated. The
last update date: ${debt.lastUpdateDate} is today`);
            return;
        }
```

```
        // Calculate the daily interest rate from the monthly percent
        const dailyRate = (debt.monthyPercent / 30) / 100;
        logger.info(`[debt] Calculating the daily interest rate: ${dailyRate}
from the monthly percent ${debt.monthyPercent}`);

        // Calculate the days elapsed since the last update date or creation
date
        const daysElapsed = Math.ceil((currentDate - lastUpdateDate) /
(1000 * 60 * 60 * 24));
        logger.info(`debt] Calculating the days elapsed: ${daysElapsed},
since the last update date: ${lastUpdateDate} or creation date`);

        // Apply the daily interest rate to update the total amount
        let totalAmount = debt.totalAmount || debt.amount; // Initialize
totalAmount with amount if not set
        logger.info(`Initializing totalAmount ${totalAmount}`)
        logger.info(`Applying the daily interest rate: ${dailyRate} to update
the total amount: ${totalAmount}`)
        totalAmount *= Math.pow((1 + dailyRate), daysElapsed); // Apply
interest rate

        // Update the totalAmount and lastUpdateDate for the debt
        logger.info(`Updating the totalAmount: ${debt.totalAmount} and
lastUpdateDate: ${debt.lastUpdateDate} for the debt`)
        debt.totalAmount = totalAmount;
        debt.lastUpdateDate = currentDate.toISOString();
    });

    // Return the updated debts array in the response
    res.json(debts);
    logger.debug('[API] Finishing debt creation');
});

// DELETE endpoint to delete a debt
app.delete('/api/debt/:id', (req, res) => {
    logger.debug('[API] Starting debt deletion');
    const { id } = req.params;
    const index = debts.findIndex(debt => debt.id === id);
    if (index !== -1) {
        logger.warn(`deleting debt ${debts[index]}`);
        debts.splice(index, 1);
        res.status(204).send();
        logger.info(`debt has been deleted`);
```

```
  } else {
    logger.debug(`[debt] debt ${id} has not been found`);
    res.status(404).send('Debt not found');
  }
});

// GET endpoint to retrieve all students
app.get('/api/student', (req, res) => {
  logger.debug('[API] Starting to retrieve all students');
  res.json(students);
});

// POST endpoint to add a new student
app.post('/api/student', (req, res) => {
  logger.debug('[API] Starting new student addition');
  const { name, age, sex, fearFactor } = req.body;
  // Check if age and fearFactor are provided and valid
  logger.info(`[student] checking if age: ${age} and fearFactor:
${fearFactor} are provided and valid`);
  if (!age || isNaN(age) || !fearFactor || isNaN(fearFactor)) {
    logger.debug('[student] Invalid age or fearFactor provided');
    return res.status(400).json({ error: 'Invalid age or fearFactor
provided' });
  }

  // Determine if the student is female based on the sex field
  logger.info(`[student] Determining if the student is female based on the
sex field: ${sex}`);
  const isFemale = sex.toLowerCase() === 'female';

  // Create a new student object
  const newStudent = {
    id: generateId('student', students),
    name: name,
    age: age,
    sex: isFemale,
    fearFactor: fearFactor
  };
  logger.info(`[student] Creating a new student object: ${newStudent}`);

  // Add the new student to the students array
  logger.info(`[student] adding the student to the students array`);
  students.push(newStudent);
```

```
    // Return the newly created student object in the response
    res.status(201).json(newStudent);
    logger.debug('[API] Finishing new student addition');
});

// PUT endpoint to update an existing student
app.put('/api/student/:id', (req, res) => {
    logger.debug('[API] Starting student updation');
    const { id } = re.params;
    const updatedStudent = req.body;

    const index = students.findIndex(student => student.id === id);
    if (index !== -1) {
        // Update the student record with the provided ID
        logger.info(`[student] Updating the student record with the provided
ID: ${id}`)
        students[index] = { ...updatedStudent, id: id };
        res.json(students[index]);
    } else {
        logger.debug(`[student] Student ${id} has not been found`);
        res.status(404).send('Student not found');
    }
});

// GET endpoint to retrieve a specific student by id
app.get('/api/student/:id', (req, res) => {
    logger.debug('[API] Starting to retrieve a student');
    const student = students.find(student => student.id === req.params.id);
    if (student) {
        res.json(student);
    } else {
        logger.debug(`[student] student ${req.params.id} has not been
found`);
        res.status(404).send('Student not found');
    }
});

// DELETE endpoint to delete a student
app.delete('/api/student/:id', (req, res) => {
    logger.debug('[API] Starting student deletion');
    const { id } = req.params;
    const index = students.findIndex(student => student.id === id);
```

```
    if (index !== -1) {
        logger.warn(`[student] deleting a student by ID: ${id}`);
        students.splice(index, 1);
        res.status(204).send();
        logger.info(`[student] student ${id} has been deleted`);
    } else {
        logger.info(`[student] student ${id} has not been found`);
        res.status(404).send('Student not found');
    }
});

module.exports = app;
//Start server
//app.listen(PORT, () => {
//    logger.info(`starting server on the port ${PORT}`);
//    console.log(`Server is running on http://localhost:${PORT}`);
//});
```

Appointment_steps.js

```
const { Given, When, Then, After } = require('@cucumber/cucumber');
const request = require('supertest');
const app = require('../../server/server.js');
const assert = require('assert');
const axios = require('axios');

let appointmentId;

// Step definition starts...
When('I request all appointments', async function () {
  response = await axios.get(`${this.api}/api/appointment`);
  this.response = response;
});

When('I create an appointment with valid data {string}', async function
(date) {
  const appointmentDetail = {
    studentId: this.studentID,
    date: date
    };
  try {
    const response = await axios.post(`${this.api}/api/appointment`,
appointmentDetail);
```

```
    this.response = response;
    appointmentId = this.response.data.id;
  } catch (error) {
    this.response = error.response;
  }
});

When('I create an appointment with past date {string}', async function
(date) {
  const appointmentDetail = {
    studentId: this.studentID,
    date: date
    };
  try {
    const response = await axios.post(`${this.api}/api/appointment`,
appointmentDetail);
    this.response = response;
  } catch (error) {
    this.response = error.response;
  }
});

When('I create an appointment with invalid data', async function
(dataTable) {
  const appointmentDetailsArray = dataTable.hashes();
  try {
    const response = await axios.post(`${this.api}/api/appointment`,
appointmentDetailsArray[0]);
    this.response = response;
  } catch (error) {
    this.response = error.response;
  }
});

When('I create an appointment with invalid date format {string}', async
function (date) {
  const appointmentDetail = {
    studentId: this.studentID,
    date: date
};
  try {
    const response = await axios.post(`${this.api}/api/appointment`,
appointmentDetail);
```

```
    this.response = response;
  } catch (error) {
    this.response = error.response;
  }
});

When('I request an appointment by id {string}', async function (id) {
  const response = await axios.get(`${this.api}/api/appointment/${id}`);
  this.response = response;
});

When('I delete an appointment by id', async function () {
  const response = await
axios.delete(`${this.api}/api/appointment/${appointmentId}`);
  this.response = response;
});
// Step definition ends...

// Step validation starts...

Then('the response should contain appointments', function () {
  assert.ok(Array.isArray(this.response.data));
});

Then('the response should contain appointment data', function () {
  assert.ok(this.response.data.id);
  assert.ok(this.response.data.studentId);
  assert.ok(this.response.data.date);
});

Then('I receive object of appointments', function () {
  const appointments = this.response.data
  assert.strictEqual(typeof appointments, 'object', 'Expected response to be
an object');
})

Then('the response should contain error message {string}', function
(errorMessage) {
  let error;
  console.log(this.response.status);
  if (this.response.data.errors) {
    if (Array.isArray(this.response.data.errors)) {
      error = this.response.data.errors[0].msg; // handle if it is an array
```

```
    } else {
      error = this.response.data.errors.message; // handle if it is an object
    }
  } else {
    error = this.response.data.error;
  }
  assert.strictEqual(error, errorMessage);
});

Then('the response should contain appointment with id {string}', function
(id) {
  assert.strictEqual(this.response.data.id, id);
});

// Step validation ends..
```

Debt_steps.js

```
const { Given, When, Then, After } = require('@cucumber/cucumber');
const request = require('supertest');
const app = require('../../server/server.js');
const assert = require('assert');
const axios = require('axios');


// Step definition starts...

When('I create a debt for the student with the amount {int}', async function
(amount) {
    const debtDetails = {
        studentId: this.studentID,
        amount: amount
    };
    console.log(this.api)
    response = await axios.post(`${this.api}/api/debt`, debtDetails);
    this.response = response;
});

When('I create a debt for the student that does not exist {string} with the
amount {int}', async function (student, amount) {
    const debtDetails = {
        studentId: student,
        amount: amount
    };
```

```
    console.log(debtDetails);
    try {
        response = await axios.post(`${this.api}/api/debt`, debtDetails);
        this.response = response;
    } catch (error) {
        if (error.response) {
            console.log('error response',error.response)
            this.response = error.response;
        } else {
            console.log('error',error)
            throw error;
        }
    }
});
// Step definition ends...


// Step validation starts...

Then('The debt should be created with the amount {int}', function(expectedAmount) {
    const debtData = this.response.data;
    assert.strictEqual(debtData.amount, expectedAmount, `Expected debt amount ${expectedAmount}, but got ${debtData.amount}`);
})

Then('Id is assigned to the debt', function () {
    const debtData = this.response.data;
    assert.ok(debtData.id, 'Expected debt ID to be assigned, but it was not');
})

Then('The monthly percent of the newly created debt must be {int}', function (expectedMonthlyPercent) {
    const debtData = this.response.data;
    assert.strictEqual(debtData.monthlyPercent, expectedMonthlyPercent, `Expected monthly percent ${expectedMonthlyPercent}, but got ${debtData.monthlyPercent}`);
})

Then('', function () {

})
```

```
Then('', function () {

})
// Steps validation ends...
```

Student_steps.js

```
const { Given, When, Then, After } = require('@cucumber/cucumber');
const request = require('supertest');
const app = require('../../server/server.js');
const assert = require('assert');
const axios = require('axios');


let studentID;
let response;


Given('The API is running', function () {
  console.log('Step definition: The API is running');
});

// Step definition starts...

When('I send a POST request to {string} with body:', async function
(endpoint, dataTable) {
  const studentDetailsArray = dataTable.hashes();
  const studentDetails = studentDetailsArray[0];
  response = await axios.post(`${this.api}${endpoint}`, studentDetails);
  this.response = response;
});

When('I request all students', async function () {
  response = await axios.get(`${this.api}/api/student`);
  this.response = response;
});

When('I create a new student with the following details:', async function
(dataTable) {
  const studentDetailsArray = dataTable.hashes();
  const studentDetails = studentDetailsArray[0];
  response = await axios.post(`${this.api}/api/student`, studentDetails);
  this.response = response;
  this.studentID = response.data.id;
```

```
});

When('I request a specific student by its ID {string}', async function (studentId) {
  const cleanStudentId = studentId.replace(/['"]+/g, ''); // Remove quotes
  response = await axios.get(`${this.api}/api/student/${cleanStudentId}`);
  this.response = response;
});

When('I create a new student with invalid age and the following details:', async function (dataTable) {
  const studentDetailsArray = dataTable.hashes();
  const studentDetails = studentDetailsArray[0];
  console.log(studentDetails);
  try {
    this.response = await axios.post(`${this.api}/api/student`, studentDetails);
  } catch (error) {
    this.response = error.response;
  }
});

When('I create {int} students', async function (numb) {
  studentIds = []; // Reset the student IDs array
  for (let i = 1; i <= numb; i++) {
    const studentDetails = {
      name: `Student${i}`,
      age: 20 + i,
      sex: 'Male',
      fearFactor: 1
    };
    const response = await axios.post(`${this.api}/api/student`, studentDetails);
    studentIds.push(response.data.id);
  }
});

// Step definition ends...
// Step validation starts...

Then('the response status code should be {int}', function (expectedStatusCode) {
  const actualStatusCode = this.response.status;
```

```
 assert.strictEqual(actualStatusCode, expectedStatusCode, `Expected status code ${expectedStatusCode}, but got ${actualStatusCode}`);
});

Then('The parameter {string} should be equal to {string}', function (parameter, value) {
 const responseData = this.response.data
 switch (true) {
   case parameter == 'name' :
     assert.strictEqual(responseData.name, value, `Expected value ${parameter}, but got value ${responseData.name}`); break
   case parameter == 'age' :
     assert.strictEqual(responseData.age, value, `Expected value ${parameter}, but got value ${responseData.age}`); break
   case parameter == 'id' :
     assert.strictEqual(responseData.id, value, `Expected value ${parameter}, but got value ${responseData.id}`); break
  }
})

Then('I receive object of students', function () {
 const students = this.response.data
 assert.strictEqual(typeof students, 'object', 'Expected response to be an object');
})

Then('The parameter {string} is type of string', function(parameter) {
 const responseData = this.response.data
 switch (true) {
   case parameter == 'sex' :
     assert.strictEqual(typeof responseData.sex, 'object', 'Expected response to be an object');
  }
})

Then('I verify the ID of the last student {int}', function (numb) {
 const lastStudentId = studentIds[numb-1];
 console.log('Student ID:', lastStudentId, studentIds.length);
 assert.strictEqual(typeof lastStudentId, 'string', `Expected ${numb} student ID to be a string but received id: ${lastStudentId}`);
 assert.ok(lastStudentId.length > 0, `Expected ${numb} student ID to be a non-empty string`);
});
```

```
Then('', function() {

})
// Steps validation ends...
```

Appointment.feature

```
Feature: Appointments API
    as a user
    I want to be able to create, update and delete appointments
    So that I can track the appointments

  Scenario: Get all appointments
    Given The API is running
    When I request all appointments
    Then the response status code should be 200
    And the response should contain appointments

  Scenario: Create a new appointment
    Given The API is running
    #precondition
      #Creating a new student
      When I create a new student with the following details:
      | name    | age | sex   | fearFactor    |
      | Andrii  | 28  | Male  | 1             |
      Then the response status code should be 201
    #executing the test
    When I create an appointment with valid data "2025-10-20 10:00:00"
    Then the response status code should be 201
    And the response should contain appointment data

 Scenario: Create an appointment with past date
    Given The API is running
    #precondition
      #Creating a new student
      When I create a new student with the following details:
      | name    | age | sex   | fearFactor    |
      | Andrii  | 28  | Male  | 1             |
      Then the response status code should be 201
    #executing the test
    When I create an appointment with past date "2023-05-20 10:00:00"
    Then the response status code should be 400
```

```
    And the response should contain error message "Appointment date cannot be in the past"

Scenario: Create an appointment with invalid student ID
    Given The API is running
    When I create an appointment with invalid data
      | date                | studentId |
      | 2024-10-20 10:00:00 | unknown   |
    Then the response status code should be 400
    And the response should contain error message "Student not found"

Scenario: Create an appointment with invalid date format
    Given The API is running
    #precondition
      #Creating a new student
      When I create a new student with the following details:
      | name    | age | sex   | fearFactor    |
      | Andrii  | 28  | Male  | 1             |
      Then the response status code should be 201
    #executing the test
    When I create an appointment with invalid date format "2025/05/20 10:00:00"
    Then the response status code should be 400
    And the response should contain error message "Date must be in YYYY-MM-DD HH:mm:ss format"

  Scenario: Get appointment by id
    Given The API is running
    When I request an appointment by id "ap1"
    Then the response status code should be 200
    And the response should contain appointment with id "ap1"

  Scenario: Delete an appointment by id
    Given The API is running
    #precondition
      #Creating a new student
      When I create a new student with the following details:
      | name    | age | sex   | fearFactor    |
      | Andrii  | 28  | Male  | 1             |
      Then the response status code should be 201
      #Creating appointment
      When I create an appointment with valid data "2025-10-20 10:00:00"
      Then the response status code should be 201
```

```
    #executing test
    When I delete an appointment by id
    Then the response status code should be 204
```

Debt.feature

```
Feature: Debts
    as a user
    I want to be able to create, update and delete debts
    So that I can track the debts

    Scenario: Create debt for a specific student
    Given The API is running
    #precondition
      #Creating a new student
      When I create a new student with the following details:
      | name    | age | sex   | fearFactor    |
      | Andrii  | 28  | Male  | 1             |
      Then the response status code should be 201
    #executing the test
    When I create a debt for the student with the amount 1000
    Then the response status code should be 201
    And The debt should be created with the amount 1000
    And Id is assigned to the debt
    And The monthly percent of the newly created debt must be 0

    Scenario: Create debt for unexisting
    Given The API is running
    When I create a debt for the student that does not exist "unknown" with the amount 1000
    Then the response status code should be 404
    And the response should contain error message "Student not found"
```

Student.feature

```
Feature: Students
  As a user
  I want to be able to create students
  So that I can track their data afterwards

  Scenario: Create a student
    Given The API is running
    When I create a new student with the following details:
    | name  | age | sex   | fearFactor    |
    | Greg  | 25  | Male  | 1             |
    Then the response status code should be 201
    And The parameter 'name' should be equal to 'Greg'
    And The parameter 'sex' is type of string

  Scenario: Get students
    Given The API is running
    When I request all students
    Then the response status code should be 200
    Then I receive object of students

  Scenario: Get a specific student by id
    Given The API is running
    When I request a specific student by its ID 's2'
    Then the response status code should be 200
    Then The parameter 'id' should be equal to 's2'

  Scenario: Create a student with age 0
    Given The API is running
    When I create a new student with invalid age and the following details:
      | name  | age | sex   | fearFactor    |
      | Test  | 0   | Male  | 1             |
    Then the response status code should be 400
    And the response should contain error message "invalid student age"

  Scenario: Create a student with age -1
    Given The API is running
    When I create a new student with invalid age and the following details:
      | name  | age | sex   | fearFactor    |
      | Test  | -1   | Male  | 1             |
    Then the response status code should be 400
    And the response should contain error message "invalid student age"
```

```
Scenario: Create a student with invalid fearFactor
  Given The API is running
  When I create a new student with invalid age and the following details:
    | name  | age | sex   | fearFactor    |
    | Test  | 25  | Male  | 6             |
  Then the response status code should be 400
  And the response should contain error message "invalid fearFactor"

Scenario: Create a student with invalid fearFactor
Given The API is running
When I create a new student with invalid age and the following details:
  | name  | age | sex   | fearFactor     |
  | Test  | 25  | Male  | -1             |
Then the response status code should be 400
And the response should contain error message "invalid fearFactor"

Scenario: Create 16 students and verify the ID of the 16th
  Given The API is running
  When I create 16 students
  Then I verify the ID of the last student 16
```

ДОДАТОК Б

ВІДГУК

Керівника економічного розділу

на кваліфікаційну роботу бакалавра на тему:

«Розробка додатку автоматизації тестів REST API на основі фреймворку Cucumber»

струдента групи 122-21зск-1 Фесенка Андрія Романовича

Керівник економічного розділу
доц. каф. ПЕП та ПУ, к.е.н **Л.В. Касьяненко**