Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра програмного забезпечення

Бакалаврська кваліфікаційна робота

на тему: «Розробка програмного засобу для автоматизації тестування REST API у процесі CI/CD»

Виконав: студент 4 курсу
групи 2ПІ-21б
спеціальності
121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Давиденко М.О.
(прізвище та ініціали)
Керівник: к.т.н., доц. каф. ПЗ Хошаба О.М.
(прізвище та ініціали)
Рецензент: к.т.н., доц. каф. ЗІ Лукічов В.В.
(прізвище та ініціали)

Допущено до захисту
Зав. кафедр _____
«9» 06 2025 р.

ВНТУ – 2025

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти перший бакалаврський
Галузь знань 12 - Інформаційні технології
Спеціальність 121 – Інженерія програмного забезпечення
Освітньо-професійна програма - Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
Романюк О. Н.
_21_березня 2025 року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Давиденко Миколі Олеговичу

1. Тема роботи - «Розробка програмного засобу для автоматизації тестування REST API у процесі CI/CD»

Керівник роботи: Хошаба Олександр Мирославович, к.т.н., доцент кафедри ПЗ, затверджені наказом вищого навчального закладу від 20 березня 2025 р. № 97.

2. Строк подання студентом роботи 2 червня 2025 р.

3. Вихідні дані до роботи: кількість REST API ендпоїнтів для тестування — до 60; кількість параметризованих тестів — до 200; швидкість виконання REST API тестів — до 1000 запитів/хв; середній час CI/CD пайплайну — до 4 хвилин; обсяг даних для data-driven тестування — до 1000 записів; інтеграція з Jenkins, Docker, REST Assured, Allure Reports.

4. Зміст розрахунково-пояснювальної записки: вступ; аналіз предметної галузі автоматизації тестування REST API у CI/CD; порівняльна характеристика інструментів тестування REST API; методика інтеграції тестів у пайплайн CI/CD; проектування програмного засобу; реалізація модуля REST Assured тестування; генерація та візуалізація звітів; контрактне тестування та мокування; верифікація вручну у пайплайні Jenkins; висновки; список використаних джерел; додатки.

5. Перелік графічного матеріалу: мета та задачі роботи, порівняльна характеристика інструментів тестування REST API, проектування програмного засобу, методика інтеграції тестів у пайплайн CI/CD, блок-схема алгоритмів, тестування програмного засобу, верифікація у пайплайні Jenkins; висновки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Хошаба О.М., к.т.н., доцент кафедри ПЗ	24.03.25	01.06.25

7. Дата видачі завдання 24 березня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз проблеми впровадження сучасних підходів та практик автоматизації тестування	25.03.2025-03.04.2025	вик.
2	визначення процесу автоматизації тестування REST API, проектування основних та додаткових модулів програмного засобу	04.04.2025-14.04.2025	вик.
3	Розробка основних та додаткових модулів програмного засобу	15.04.2025-05.05.2025	вик.
4	опис модульного підходу в архітектурі тестування, тестування контрольного прикладу	06.05.2025-19.05.2025	вик.
5	Оформлення матеріалів до захисту БКР	20.05.2025-30.05.2025	вик.

Студент _____ Давиденко М.О.

(підпис) (прізвище та ініціали)

Керівник бакалаврської кваліфікаційної роботи _____ Хошаба О.М.

(підпис) (прізвище та ініціали)

АНОТАЦІЯ

УДК 004.415

Давиденко М.О. Розробка програмного засобу для автоматизації тестування REST API у процесі CI/CD : бакалаврська кваліфікаційна робота зі спеціальності 121 Інженерія програмного забезпечення, освітня програма – інженерія програмного забезпечення. Вінниця: ВНТУ, 2025. 124 С.

На укр. мові. Бібліогр. : 46 назв ; рис. : 16; табл. 12.

У бакалаврській кваліфікаційній роботі розглянуто процес створення програмного засобу для автоматизації тестування REST API у процесі CI/CD.

Також виконано аналіз сучасних підходів до автоматизації тестування REST API, проведено техніко-економічне обґрунтування вибору технологій, реалізовано архітектуру програмного засобу з використанням Java, Spring Boot, REST Assured та Jenkins. Запропонований засіб дозволяє автоматично запускати тести після кожного коміту, формувати звіти, проводити контрактне тестування, мокування та сервісну віртуалізацію.

ANNOTATION

Davydenko M.O. Development of software testing for REST API testing automation in the CI/CD process: bachelor's thesis on the specialty 121 Software engineering, educational program - software engineering. Vinnytsia: VNTU, 2025. 124 with.

In Ukrainian language. Bibliographer : 46 titles; Fig. : 16; table 12.

The bachelor's qualification work implies creating a software task for REST API testing automation in the CI/CD process.

The work analyses modern approaches to REST API testing automation, conducts a feasibility study of the choice of technologies, and implements a software architecture using Java, Spring Boot, REST Assured, and Jenkins. The proposed tool allows you to automatically run tests after each commit, generate reports, conduct contract testing, mocking, and service virtualisation.

ЗМІСТ

ВСТУП

Обґрунтування вибору теми дослідження. В теперішніх умовах інформаційних технологій програмне забезпечення розробляється із зростаючими вимогами до якості, надійності та швидкості оновлення. Особливої актуальності набувають технології безперервної інтеграції та доставки (CI/CD), які передбачають автоматизацію всіх етапів життєвого циклу ПЗ - від коміту змін до їхнього впровадження у продуктивне середовище. Одним із критичних етапів CI/CD є тестування, зокрема тестування REST API, що виступає посередником між мікросервісами, фронтендом та бекендом, забезпечуючи стабільність усієї системи. Тому, автоматизація тестування REST API є важливою складовою сучасного підходу до забезпечення якості, оскільки дозволяє виявляти помилки до потрапляння змін у продакшн, знижує технічний борг та сприяє надійності інтеграції компонентів.

Проблема дослідження полягає у недостатньому рівні автоматизації процесів тестування REST API в багатьох компаніях, де все ще використовуються ручні сценарії або неповністю інтегровані засоби перевірки, що унеможливлює реалізацію принципів DevOps. Для цього, постає необхідність у створенні програмного засобу, який забезпечить повну автоматизацію тестування REST API з інтеграцією у пайплайни CI/CD, із підтримкою сучасних практик: контрактного тестування, мокування, data-driven тестів, генерації звітності, автоматичного створення середовищ та моніторингу результатів.

Таким чином, одним із найефективніших рішень є використання зв’язки Java + Spring Boot + REST Assured + Jenkins + Docker. Такий стек дозволяє будувати модульні, реюзабельні тести, інтегровані у CI/CD пайплайни з високою гнучкістю та масштабованістю. Робота спрямована на створення програмного засобу, який автоматизує всі аспекти перевірки REST API в рамках життєвого циклу розробки програмного забезпечення.

Зв’язок роботи з науковими програмами, планами, темами. Робота виконувалась згідно плану виконання наукових досліджень на кафедрі програмного забезпечення

Мета та завдання. Метою роботи є підвищення ефективності забезпечення якості програмного забезпечення шляхом розробки програмного засобу для повної автоматизації тестування REST API та його інтеграції у процес CI/CD, що дозволяє скоротити час релізу, зменшити кількість помилок та покращити масштабованість тестових сценаріїв.

Відповідно до поставленої мети виконані наступні задачі:

- виконати аналіз предметної галузі автоматизації тестування REST API в процесі CI/CD;
- виконати аналіз впровадження сучасних підходів та практик автоматизації тестування REST API в CI/CD;
- визначити процес автоматизації тестування REST API;
- запропонувати метод паралельного виконання REST-assured тестів;
- виконати проектування основних та додаткових модулів програмного комплексу автоматизації тестування REST API;
- розробити основні та додаткові модулі програмного засобу;
- описати модульний підхід в архітектурі тестування REST API;
- виконати тестування контрольного прикладу для основного модуля AuthTestModule.

Об’єктом дослідження є процес забезпечення якості взаємодії компонентів програмного забезпечення у середовищі CI/CD.

Предметом дослідження є методи, інструменти та засоби автоматизації тестування REST API, інтегровані у пайплайни безперервної інтеграції.

Методи дослідження. У процесі дослідження використовувалися: методи об’єктно-орієнтованого проектування (UML-діаграми); методи тестування (TDD, BDD); інструменти для автоматизованого тестування REST API (REST Assured, Postman, Newman); засоби оркестрації CI/CD (Jenkins, GitLab CI, Docker); технології мокування та сервісної віртуалізації (WireMock, Beeceptor); сучасні підходи щодо контрактного тестування (OpenAPI, Swagger).

Новизна отриманих результатів:

1. Запропоновано метод автоматизованого безперервного тестування REST API, у якому, на відміну від існуючих рішень, поєднано архітектуру мікросервісів

на базі Spring Boot із Docker-контейнеризацією та Jenkins-пайплайном для data-driven сценаріїв, що дозволило скоротити тривалість CI/CD-процесу на 16%.

2. Запропоновано метод паралельного виконання REST-assured тестів у Jenkins Pipeline, який враховує ресурси Docker-контейнерів та залежності мікросервісів, що дозволило підвищити пропускну здатність CI/CD-процесу на 28%.

3. Подальшого розвитку отримала модель генерації тестових даних для data-driven REST-assured тестів у Jenkins Pipeline, що дозволяє знизити на 17% витрати часу на підготовку тестових сценаріїв за рахунок використання централізованого сховища шаблонів даних.

Практична цінність отриманих результатів. Практична цінність отриманих результатів полягає у створенні ефективного інструменту автоматизованого тестування REST API, який може бути інтегрований у пайплайни CI/CD, що забезпечує скорочення часу тестування, зменшення людського фактору, раннє виявлення дефектів, підвищення стабільності релізів, а також масштабованість рішень. Розроблений програмний засіб може бути впроваджений у будь-яке підприємство, яке використовує DevOps-підходи та мікросервісну архітектуру.

Особистий внесок здобувача. Усі наукові результати, викладені у бакалаврській кваліфікаційній роботі, отримані автором особисто. Автору належать такі результати: розробка архітектура програмної системи; створення алгоритмів у вигляді UML-діаграм; розробка модулів клієнтської та серверної частин.

Апробація результатів роботи. Результати роботи були представленні на конференції «X Міжнародної науково-практичної конференції з проблем вищої освіти і науки "Інформаційні технології в освіті, науці і виробництві (ІТОНВ-2025)"».

Публікації. Результати роботи були опубліковані в матеріалах конференції – «X Міжнародної науково-практичної конференції з проблем вищої освіти і науки "Інформаційні технології в освіті, науці і виробництві (ІТОНВ-2025)"» [46].

1 АНАЛІЗ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз предметної галузі автоматизації тестування REST API

У сучасних умовах цифрової трансформації бізнесу, розробка програмного забезпечення здійснюється в умовах високої конкуренції та стрімкого темпу змін. Компанії прагнуть якомога швидше постачати нові функціональні можливості кінцевим користувачам, що призводить до широкого впровадження підходів безперервної інтеграції та доставки (CI/CD). Ці підходи дають змогу автоматизувати значну частину процесів - від коміту коду до його розгортання у продуктивне середовище.

У цьому контексті тестування REST API займає ключову позицію, оскільки саме API забезпечує взаємодію між мікросервісами, мобільними застосунками, фронтенд і бекенд складовими систем. Помилки в API можуть спричинити збої у цілісному функціонуванні програмного забезпечення, незалежно від правильності роботи окремих його компонентів.

Враховуючи складність та обсяг сучасних систем, автоматизація тестування REST API стала не просто бажаною, а необхідною умовою досягнення якості та надійності на всіх етапах CI/CD [1,2].

Загальна концепція CI/CD та місце тестування REST API у цьому процесі полягає в наступному. Continuous Integration (CI) передбачає регулярне об'єднання коду, розміщення його у спільному репозиторії, після чого автоматично запускаються процеси компіляції, перевірки, побудови та тестування. Якщо будь-який з етапів CI завершується помилкою, зміни не потрапляють у подальший етап життєвого циклу розробки.

Безперервна інтеграція (англ. Continuous Integration, CI) - це практика програмної інженерії, яка полягає у регулярному злитті змін у кодовій базі в спільний централізований репозиторій, зазвичай кілька разів на день. Основна мета CI полягає у забезпеченні своєчасного виявлення помилок інтеграції, скороченні часу між написанням коду та його перевіркою в умовах повної системної збірки. У типовому CI-процесі після коміту коду до репозиторію (наприклад, Git) автоматично ініціюється серія попередньо визначених дій:

компіляція проєкту, виконання юніт-тестів, статичний аналіз коду, створення артефактів збірки та запуск інтеграційних тестів [1, 5].

Ці дії, як правило, реалізуються за допомогою спеціалізованих інструментів CI-серверів, таких як Jenkins, GitLab CI/CD, Travis CI, CircleCI або GitHub Actions, які виступають центральними елементами автоматизації процесу. Після кожного нового коміту зміни тестуються незалежно і в ізольованому середовищі, що дозволяє забезпечити високу надійність та контроль якості. Якщо хоча б один із етапів CI завершується невдачею - наприклад, тест не проходить або не вдається побудова - система фіксує це як build failure, і зміни не потрапляють у наступну фазу розробки або доставки [3, 6]. Таким чином, CI сприяє не лише прозорості й відповідальності всередині команди, але й формує умови для безперервного зворотного зв’язку, що в свою чергу зменшує ризик накопичення технічного боргу. Крім того, автоматизація таких завдань, як аналіз якості коду (наприклад, за допомогою SonarQube), перевірка безпеки (наприклад, через Snyk), чи динамічні тести API - дозволяє масштабувати CI до середовищ із сотнями розробників без втрати ефективності [2, 7].

Continuous Delivery/Deployment (CD) - це розширення CI, яке дозволяє автоматично або напівавтоматично доставляти оновлення в тестові та продуктивні середовища. Автоматичне тестування REST API є обов’язковим компонентом цього етапу, адже саме воно дозволяє впевнено гарантувати відсутність критичних дефектів у логіці взаємодії систем.

Безперервне постачання (Continuous Delivery, CD) - це етап життєвого циклу розробки програмного забезпечення, який розширює можливості Continuous Integration (CI), дозволяючи автоматично або напівавтоматично доставляти зміни коду у різні середовища: тестове, стадійне (staging), а в деяких випадках - і у продуктивне. Основна мета CD - забезпечити готовність коду до розгортання в будь-який момент часу, за умови проходження усіх необхідних автоматичних перевірок [4, 6].

Коли зміни успішно інтегруються в CI-пайплайн, вони передаються в CD, де відбуваються такі типові дії: розгортання у тестове середовище, автоматизоване приймальне тестування, інтеграційне тестування, а також тестування інтерфейсів

прикладного програмування (REST API). В умовах мікросервісної архітектури REST API є основним каналом зв'язку між сервісами, тому його надійність - критично важлива. Автоматизоване REST API тестування на цьому етапі дозволяє виявити критичні логічні помилки взаємодії компонентів, порушення у структурі запитів чи відповідей, а також неконсистентність даних [2, 7].

Continuous Deployment є наступним етапом після Continuous Delivery і передбачає повну автоматизацію - успішно протестовані зміни без участі людини одразу переходять до продуктивного середовища. Це практикується у високонавантажених, динамічних середовищах, де швидкість оновлень є конкурентною перевагою (наприклад, Amazon, Netflix, Spotify) [8].

У сучасних CI/CD системах розгортання часто реалізується за допомогою канарейкових релізів, blue-green деплойменту, rolling-оновлень, що дозволяє мінімізувати ризики, пов’язані з впровадженням нових версій. Проте незалежно від обраної стратегії, автоматичне тестування REST API залишається незамінним засобом валідації функціональності систем на всіх рівнях - від індивідуального сервісу до комплексної інтеграції [3, 5].

Таким чином, CD - це не лише технічний процес доставки, а й стратегія управління якістю та надійністю змін, де REST API тестування виступає як запобіжник проти деградації системної поведінки.

Таким чином, REST API тести мають бути (рис. 1.1):

- повністю автоматизованими;

- інтегрованими в пайплайни CI/CD;

- достатньо гнучкими, щоб адаптуватися до змін у структурах запитів, відповідей та бізнес-логіки [3,4].

Для забезпечення стабільності, масштабованості та високої швидкості розгортання в CI/CD-процесах, REST API тести повинні бути повністю автоматизованими (рис. 1.1), тобто виконуватись без участі людини, згідно з наперед визначеними тригерами та сценаріями. Повна автоматизація REST API тестування охоплює кілька обов’язкових складових.

1. Створення тестових сценаріїв у коді або у сценарійно-орієнтованій формі.

Тестові випадки мають бути або частиною коду (наприклад, написані за допомогою бібліотек REST Assured, SuperTest, HTTPretty), або збережені у форматах JSON/YAML (наприклад, для Newman або Karate). Це дозволяє:

- стандартизувати тест-кейси;
- версіонувати їх разом з кодом програми;
- робити їх повторно використуваними [4, 6].

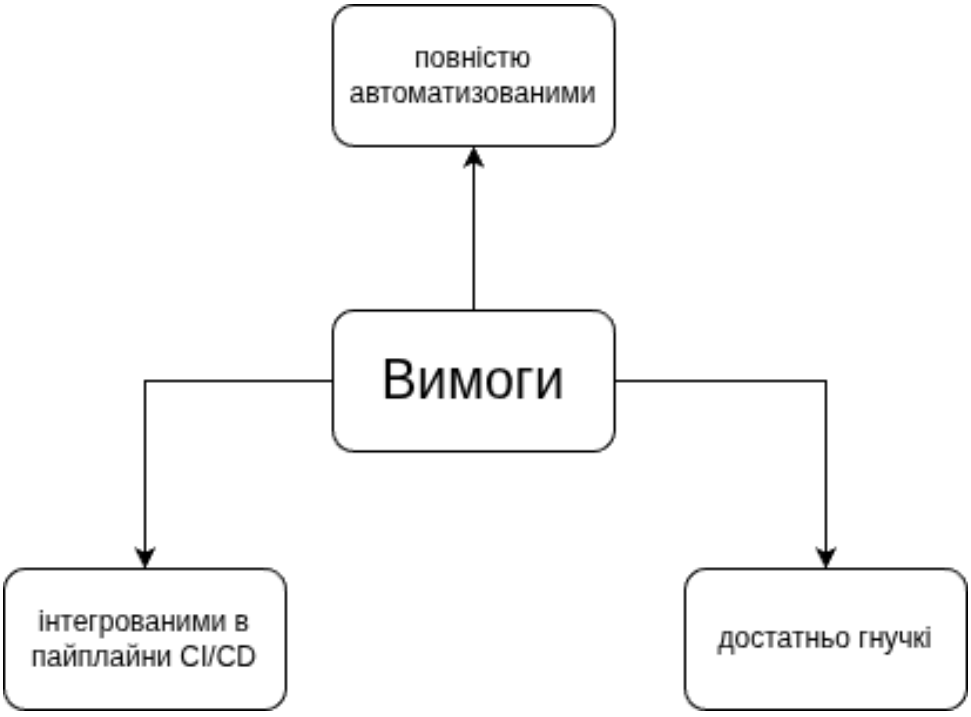

Рисунок 1.1 - Вимоги щодо програмного забезпечення на основі пайплайнів CI/CD з використанням REST API технологій

2. Інтеграція з CI-сервером.

Інструменти тестування REST API мають підтримувати CLI (Command Line Interface) або HTTP-тригерний запуск, що дозволяє легко вбудовувати їх у CI/CD пайплайни. Наприклад:

Postman → запуск через Newman в CLI: newman run collection.json.

Karate або REST Assured → запуск як частина JUnit-тестів у Jenkins, GitHub Actions чи GitLab CI [2, 5].

3. Автоматичне створення тестових середовищ (test environment provisioning).

Для запуску REST API тестів потрібне попереднє створення інфраструктури: підняття контейнерів, запуск мок-серверів, налаштування авторизації (JWT, OAuth, API Key). Це досягається через:

- Docker Compose;
- Helm-чарти в Kubernetes;
- Terraform або Ansible (у складніших системах) [7].

4. Валідація результатів на основі відповідей API.

REST API тести мають включати:

- перевірку HTTP-статусів (200, 401, 404 тощо),
- перевірку структур JSON-відповідей (schema validation),
- логічні перевірки (бізнес-правила),
- асерти (assertions) на ключові поля, структури, значення [3].

Приклад:

```
"assert": {
  "status": 200,
  "json.body.user.name": "expectedName"
}
```

5. Збір результатів і візуалізація.

Автоматизовані тести повинні повертати результат у форматах, зручних для подальшої аналітики (наприклад, JUnit XML, Allure, HTML-звіти). Це дозволяє CI-системі:

- автоматично маркувати білд як «failed»;
- створювати лог-файли;
- інтегруватися з дашбордами моніторингу (наприклад, Grafana + Prometheus) [6, 8].

6. Автоматичний запуск на кожному кроці пайплайну.

REST API тести мають запускатися:

- при кожному pull/merge request;
- при nightly build (щоденна перевірка стабільності);
- при релізі;
- або перед деплоєм (pre-deploy checks).

Це забезпечує повне охоплення життєвого циклу змін і дозволяє оперативно реагувати на будь-які дефекти [2, 6]. Тому, повна автоматизація REST API тестів забезпечує не лише контроль якості, а й інфраструктурну стійкість CI/CD-процесу. Такий підхід дозволяє мінімізувати ризики, підвищити швидкість релізів, покращити зворотній зв’язок та впровадити сучасні DevOps практики з найкращими показниками ефективності.

Інтеграція REST API тестів у процеси CI/CD - це необхідний крок для досягнення стійкої, повторюваної та контрольованої розробки програмного забезпечення, особливо в умовах мікросервісної архітектури. Така інтеграція дозволяє забезпечити безперервну перевірку працездатності API на всіх етапах життєвого циклу зміни коду - від першого коміту до розгортання в продуктивному середовищі. Для цього використовуються наступні механізми інтеграції REST API тестів у процеси CI/CD.

1. Використання YAML/DSL-опису пайплайнів.

У CI/CD платформах (Jenkins, GitHub Actions, GitLab CI, Azure DevOps) пайплайни зазвичай описуються мовами декларативного сценарію - YAML або Groovy, де REST API тести вбудовуються як окремі кроки (jobs/stages). Наприклад, у GitLab CI:

```
api_test:
  stage: test
  script:
    - newman run postman_collection.json --reporters cli,junit
  artifacts:
    paths:
      - newman-report.xml
```

Такий підхід забезпечує:

- повторюваність тестів;
- контроль версій сценаріїв тестування;
- легке масштабування (додавання нових середовищ, паралелізація тощо) [2, 5].

2. Вбудовування у логіку тригерів.

API тести мають запускатись за чітко визначеними подіями:

- на кожен pull/merge request (перед об’єднанням змін до основної гілки);

- на кожному коміті в основну гілку;

- на етапі підготовки до релізу (release candidate);

- на кожному автоматичному nightly build.

Це дозволяє оперативно виявляти порушення функціональності API, пов’язані з новими змінами [3].

3. Контейнеризація середовища для тестування.

Оскільки REST API тести мають залежність від працюючого бекенд-сервісу, перед їх запуском слід автоматично створити або емулятор (mock), або повноцінне середовище:

- запуск контейнерів із API (Docker Compose);

- використання Kubernetes для масштабованого CI/CD;

- паралельне створення мок-сервісів для нестабільних або залежних API [4, 6].

Цей крок включається як окремий job, наприклад:

services:

- *name: api:latest*

  alias: backend

4. Автоматичне прийняття рішень на основі результатів тестів.

CI/CD-системи повинні обробляти результати REST API тестів і приймати рішення автоматично:

- якщо всі тести пройдені, то пайплайн продовжується;

- якщо виявлено помилки, то пайплайн переривається;

- якщо помилка в «критичному» API, то створюється автоматичне повідомлення в Slack/Email/JIRA [7].

Це можливо завдяки генерації JUnit XML або JSON звітів, які CI-сервери можуть обробляти нативно.

5. Моніторинг і репортинг.

Інтегровані REST API тести мають автоматично наступним чином:

- генерувати HTML або Allure-звіти;

- завантажуватись як артефакти кінцевого бінарного коду;

- бути доступними для аналізу безпосередньо через CI-інтерфейс [6].

Це створює зворотній зв'язок для розробника, який бачить не лише результат, а й контекст помилки.

Таким чином, інтеграція REST API тестування в CI/CD пайплайни дозволяє перевіряти не лише функціональність коду, але й стабільність взаємодії компонентів у реальному або емуляторному середовищі. Така інтеграція значно знижує ризики релізу дефектного коду та сприяє побудові надійної DevOps-культури, де тестування не є ізольованим етапом, а інтегрованою частиною життєвого циклу ПЗ [2, 3, 5 - 7].

Однією з ключових вимог до ефективного автоматизованого тестування REST API у CI/CD-середовищах є гнучкість тестових сценаріїв - здатність швидко та безперешкодно адаптуватися до змін у специфікації API, бізнес-логіці чи форматі даних. У динамічному середовищі сучасного програмного забезпечення, де оновлення відбуваються кілька разів на тиждень, статичні, жорстко закодовані тести швидко втрачають актуальність, тому необхідно впроваджувати гнучкі та адаптивні підходи наступним чином [3, 4].

1. Використання параметризованих тестів і шаблонів.

Гнучкі API тести повинні базуватися на шаблонах, що дозволяють підставляти змінні значення у структуру запиту. Наприклад:

```
{
  "userId": "{{user_id}}",
  "email": "{{email}}"
}
```

Параметри ({{user_id}}, {{email}}) можуть заповнюватися на етапі виконання тесту з:

- глобального файлу конфігурації;

- відповіді попереднього запиту (chain testing);

- змінних середовища (environment variables) [3].

Це дозволяє швидко адаптувати тести при зміні логіки або структури API.

2. Використання JSON Schema Validation.

Для перевірки структури відповіді REST API доцільно застосовувати валидацію на основі JSON-схем, а не перевірку жорстко закодованих значень. Це дозволяє:

- перевіряти наявність необхідних ключів;

- перевіряти типи даних;

- дозволяти змінні поля/значення, зберігаючи стабільність тесту.

Наприклад, у Karate або REST Assured можна вказати файл JSON Schema, який виконує перевірку відповідності структури:

```
{
  "type": "object",
  "properties": {
    "id": { "type": "integer" },
    "name": { "type": "string" }
  },
  "required": ["id", "name"]
}
```

Цей підхід забезпечує стійкість тестів до мінімальних змін у відповідях, якщо вони не є критичними [4, 6].

3. Чітке розділення тестової логіки від даних (data-driven testing).

Гнучкі REST API тести мають підтримувати data-driven підхід - коли логіка тесту зберігається окремо від вхідних даних. Наприклад, набір даних може зберігатися у вигляді CSV, JSON або YAML, а тестовий сценарій зчитує ці дані динамічно. Це дозволяє:

- швидко додавати нові варіанти кейсів без зміни коду;

- централізовано змінювати дані при оновленні вимог;

- масштабувати покриття [3].

4. Побудова модульних і реюзабельних тестів.

Код або сценарії тестів мають бути компонентними: часто повторювані дії (автентифікація, створення користувача, очищення середовища) винесені в окремі функції або макроси. Це дозволяє:

- зменшити дублювання коду;

- централізовано змінювати бізнес-логіку;

- зберігати консистентність [5].

5. Мокування та стабільність при зовнішніх змінах.

При тестуванні API, які залежать від зовнішніх сервісів, слід застосовувати мок-сервери (mock servers) або сервіси-емулятори (наприклад, WireMock, Mockoon, Beeceptor), які дозволяють стабілізувати поведінку тестів та не залежати від недоступних або нестабільних джерел [4].

6. Інтеграція з OpenAPI/Swagger.

Якщо REST API розробляється за контрактом OpenAPI (Swagger), то тести можна генерувати або валідувати автоматично на основі цього контракту. Інструменти типу Dredd, Swagger Test Templates або RestAssured з плагінами дозволяють:

- автоматично оновлювати тести при зміні OpenAPI-документа;

- перевіряти відповідність API до контракту [6].

Таким чином, REST API тести мають бути гнучкими не лише у технічному, а й у концептуальному плані - адаптуватись до змін не через переписування, а через архітектуру тестування, яка підтримує масштабування, параметризацію, шаблонізацію та автоматичну валідацію. Саме це забезпечує стійкість до змін бізнес-вимог і високий темп розгортання оновлень без втрати якості [3, 4, 6].

Для реалізації вимог щодо програмного забезпечення на основі пайплайнів CI/CD з використанням REST API технологій (рис. 1.1) існує велика кількість інструментів, які дозволяють реалізувати автоматизоване тестування REST API з різними рівнями складності та гнучкості. Наведено найбільш вживані та поширені з них у середовищах CI/CD (табл. 1.1):

- Postman + Newman - надзвичайно популярний набір інструментів. Postman дозволяє швидко створювати інтерактивні запити та тести, а Newman - виконувати їх у командному рядку, що робить його зручним для інтеграції з CI/CD пайплайнами [5];

- REST Assured - Java-бібліотека для тестування REST API з підтримкою BDD-підходу. Надзвичайно гнучка, але потребує знань програмування. Часто використовується разом з JUnit/TestNG у CI [6];

- Karate - фреймворк, який поєднує тестування API з можливостями сценарного опису, де не потрібні складні знання програмування. Karate підтримує інтеграцію з Maven, Gradle, Jenkins [7];

- SoapUI - зручний для функціонального та навантажувального тестування як REST, так і SOAP API. Підтримує створення складних тестових наборів, сценаріїв, assertions, має як безкоштовну, так і комерційну версію [8];

- JMeter - хоч першочергово відомий як інструмент для навантажувального тестування, JMeter також підтримує REST API тести. Завдяки підтримці CLI-режиму, зручний для запуску у CI/CD пайплайнах [9].

Таблиця 1.1 - Порівняльна характеристика найбільш поширеного програмного забезпечення у середовищах CI/CD

Інструмент	Простота інтеграції	Гнучкість	Підтримка сценаріїв	Автоматизація запуску	Популярн. у спільноті
Postman + Newman	Висока	Середня	Так	Так	Висока
REST Assured	Середня	Висока	Так	Так	Висока
Karate	Висока	Висока	Так	Так	Середня
SoapUI	Середня	Висока	Так	Так	Середня
JMeter	Низька	Середня	Так	Так	Висока

Таким чином, на підставі проведеного аналізу можна зробити наступні висновки. Postman підходить для команд, яким потрібна швидка інтерактивна перевірка REST API з можливістю масштабування за допомогою Newman. Недоліком є складність створення дуже гнучких логічних перевірок [5]. REST Assured є найбільш придатним для середовищ, де використовується Java. Він дозволяє глибоку інтеграцію з існуючими фреймворками тестування та забезпечує високу масштабованість [6]. Karate чудово підходить для проєктів, де потрібне швидке написання зрозумілих тестів із мінімальним кодом, особливо у мікросервісній архітектурі [7]. SoapUI залишається релевантним для складних корпоративних систем, де є потреба в SOAP-сумісності або розширеній

функціональності перевірок [8]. JMeter може бути доцільним як частина навантажувального тестування, проте для функціонального тестування REST API він поступається іншим рішенням [9].

Отже, автоматизоване тестування REST API є критично важливим для забезпечення стабільності, безпеки та функціональності програмного забезпечення в умовах безперервної розробки та доставки. Основні переваги його інтеграції у CI/CD:

- миттєве зворотне повідомлення, де використання тестування REST API одразу після коміту дозволяє виявити помилки, що могли б порушити інтеграцію сервісів [3, 6];

- раннє виявлення збоїв в інтеграції, де REST API тести перевіряють відповідність запитів і відповідей між мікросервісами, запобігаючи появі неочевидних дефектів на продакшні [2, 7];

- масштабованість і гнучкість, де існують інструменти типу REST Assured або Karate, що дозволяють легко масштабувати тести у великих проєктах [6, 9];

- безперервність якості: інтегровані тести в CI/CD пайплайни гарантують, що якість не є одноразовим етапом, а постійним процесом [10].

Загалом, вибір інструментів і стратегій залежить від технологічного стеку, культури команди та вимог до масштабованості. Усі наведені інструменти мають свої сильні сторони, але оптимальний вибір формується за рахунок їх інтеграції в загальний пайплайн CI/CD [1, 5, 9].

1. 2 Аналіз впровадження сучасних підходів автоматизації тестування

Одним із ключових підходів до впровадження сучасних підходів та практик, підвищення ефективності та надійності автоматизованого тестування REST API є контрактне тестування (англ. Contract-based testing), яке передбачає використання специфікацій API у вигляді формалізованих контрактів (наприклад, OpenAPI/ Swagger) як єдиного джерела істини для всіх учасників процесу розробки: бекенд-, фронтенд-розробників, тестувальників і DevOps-фахівців [11].

Так, у класичних підходах, коли тестувальники створюють ручні або автоматизовані сценарії на основі документації чи UI, можливі розбіжності між очікуваною та фактичною поведінкою API (табл. 1.2). Контрактне тестування усуває цю проблему шляхом жорсткого узгодження формату запитів і відповідей ще на етапі проектування. Контракт, як правило, фіксує:

- HTTP-методи та шляхи;
- структуру запитів і відповідей (у форматі JSON/XML);
- типи даних;
- обов’язкові/необов’язкові поля;
- приклади викликів і відповідей [12].

Це дозволяє реалізувати автоматизовану валідацію відповідності API-контракту через інструменти на кшталт Dredd, Pact, Swagger Validator, Schemathesis. У CI/CD-пайплайні такий підхід інтегрується на ранніх етапах (зазвичай одразу після збірки) і дає змогу миттєво виявити відхилення від специфікації, перш ніж код потрапить до QA або продакшну [13].

Таблиця 1.2 - Порівняльна характеристика впровадження контрактного та класичного API тестування

Критерій впровадження	Контрактне тестування (OpenAPI/Swagger)	Класичне тестування (Postman, REST Assured)
Вихідна точка	Автоматизований контракт	Опис/API документація вручну
Валідація відповідності схеми	Так	Частково / опціонально
Виявлення конфліктів API	Автоматично на рівні специфікації	Після тестів / вручну
Інтеграція в CI/CD	Так, у вигляді окремого кроку	Так, але залежить від підходу
Підтримка змін	Висока, через оновлення одного контракту	Низька, потрібна правка багатьох тестів
Доступність для команди	Відкрита специфікація для всіх	Сценарії зберігаються в коді або тест-наборі

Таким чином, контрактне тестування виконує не лише технічну, але й комунікаційну функцію, забезпечуючи формалізовану домовленість між командами і мінімізуючи помилки, що виникають через неправильне трактування логіки API. Тому, контрактне тестування REST API є ефективним методом забезпечення консистентності між очікуваною та реальною поведінкою та впровадженням сервісів. Його використання особливо доцільне в CI/CD-процесах із високою частотою змін, багатьма залежностями між мікросервісами та великою кількістю учасників розробки. Валідація проти OpenAPI дозволяє запобігати дефектам ще до моменту розгортання, зменшуючи навантаження на QA і час релізу [11 - 13].

До впровадження сучасних підходів та практик автоматизації тестування REST API в CI/CD відносяться Shift-left testing, що являє собою підхід до забезпечення якості програмного забезпечення, яке полягає у перенесенні тестових активностей на максимально ранні етапи життєвого циклу розробки. Термін «shift-left» походить від лінійної діаграми SDLC (Software Development Life Cycle), де етапи розвитку йдуть зліва направо: планування → розробка → тестування → розгортання. Згідно з цим підходом, тестування має розпочинатись ще до написання коду, тобто «зсунутися вліво» [14, 15].

У контексті REST API shift-left testing означає:

- створення тестів ще до реалізації функціоналу API;

- участь тестувальників у плануванні вимог і контрактів API;

- використання моків і емуляторів для тестування очікуваної поведінки до появи реального сервісу;

- інтеграція тестів у перші етапи пайплайну CI.

Завдяки shift-left підходу команди досягають зменшення кількості дефектів, зниження вартості виправлень і прискорення циклу зворотного зв’язку (табл. 1.3). Особливо ефективно цей підхід працює в зв’язці з контрактним тестуванням (OpenAPI), де API специфікація є вихідною точкою не лише для розробників, а й для створення автоматизованих тестів [13, 16].

REST API тестування у shift-left може бути реалізоване через:

- контрактні валідації ще до написання коду;
- TDD-підхід для розробки API;
- запуск тестів на pre-commit хуках або sandbox середовищах;
- автоматизоване виконання API тестів на кожному pull request [14].

Таблиця 1.3 - Порівняльна характеристика основних критеріїв впровадження сучасних підходів на основі Shift-left та традиційного тестування API

Критерій	Shift-left API тестування	Традиційне API тестування
Момент створення тестів	До або під час розробки API	Після завершення реалізації
Технічний борг	Мінімальний	Зростає зі складністю проєкту
Виявлення дефектів	На ранньому етапі	На етапі QA
Залучення тестувальників	З етапу проєктування	З етапу тестування
Інтеграція з контрактами	Висока (через OpenAPI, Pact)	Часткова або відсутня
Стійкість до змін	Вища, завдяки ранньому узгодженню	Нижча, через післяфактум перевірки

Таким чином, такі сучасні підходи та практики впровадження автоматизації тестування як Shift-left testing у сфері REST API це не просто оптимізація порядку виконання тестів, а зміна підходу до розробки в цілому, що орієнтована на якість з перших хвилин. Цей підхід дозволяє виявляти логічні, структурні та функціональні помилки ще до того, як вони потраплять у кодову базу, що робить CI/CD-процеси безпечнішими, передбачуванішими й економнішими [14 - 16].

У складних інформаційних системах з розподіленою архітектурою, особливо у мікросервісному підході, REST API часто взаємодіє з великою кількістю залежних компонентів - базами даних, сторонніми API, сервісами автентифікації тощо. У процесі тестування ці залежності можуть бути:

- нестабільними або недоступними;
- комерційними (з оплатою за кожен виклик);
- ще не реалізованими на момент розробки або тестування.

У таких умовах застосовується мокування (mocking) та віртуалізація сервісів (service virtualization) - дві тісно пов’язані техніки (табл. 1.4), що дозволяють емулювати поведінку зовнішніх залежностей REST API у контрольованому середовищі [17, 18].

Мок (mock) - це штучний сервіс або скрипт, що імітує роботу реального API або іншого зовнішнього компоненту (табл. 1.4). Він повертає наперед визначені відповіді на відповідні запити. Моки можуть бути статичними (фіксовані JSON-відповіді) або динамічними (на основі логіки або сценаріїв). До поширених інструментів підтримки сервісу мок відносяться Postman Mock Servers, WireMock, Beeceptor, Mockoon, Prism [19]. Мокування корисне у випадках, коли:

- сервіс ще не готовий;
- потрібно відтестувати граничні/неочікувані сценарії;
- потрібно уникнути надмірного навантаження на реальний API.

Інший підход впровадження відноситься до віртуалізації сервісів (табл. 1.4). Service virtualization - це розширений підхід, що охоплює не лише імітацію API, а й імітацію поведінки сервісу як цілісної системи, включаючи час відповіді, помилки, затримки, навантаження. Це дозволяє створювати складні середовища тестування, наближені до продакшну. До поширених інструментів підтримки сервісу відносяться такі інструменти як CA Service Virtualization, Parasoft Virtualize, Hoverfly, Traffic Parrot. До переваги віртуалізації відноситься:

- можливість тестування негативних сценаріїв (наприклад, timeouts, 5xx помилки);
- підвищена надійність тестів у нестабільному середовищі;
- повна автоматизація поведінки зовнішніх залежностей [20].

Таким чином, мокування та віртуалізація сервісів є ключовими техніками для забезпечення стабільності, передбачуваності та гнучкості тестування REST API у складних або непередбачуваних умовах. Їх використання дозволяє уникати збоїв через зовнішні залежності, забезпечувати стабільні CI/CD пайплайни та пришвидшувати випуск якісного ПЗ [17, 18, 20].

Таблиця 1.4 - Порівняльна характеристика основних критеріїв впровадження сучасних підходів на основі Mocking та Service Virtualization

Критерій	Мокування (Mocking)	Віртуалізація сервісів (Service Virtualization)
Рівень імітації	Запити та відповіді API	Поведінка всього сервісу
Складність реалізації	Низька / середня	Висока
Підтримка негативних сценаріїв	Частково	Так
Підтримка затримок/ таймаутів	Рідко	Так
Використання у CI/CD	Так	Так
Орієнтованість	Локальне/тестове середовище	Ентерпрайз-інтеграція, DevOps

Сучасні методики розробки програмного забезпечення прагнуть інтегрувати тестування безпосередньо у процес створення функціоналу. Серед таких підходів особливої актуальності набувають TDD (Test-Driven Development) та BDD (Behavior-Driven Development), які все частіше застосовуються не лише для модульного тестування (табл. 1.6), а й для розробки та перевірки REST API [21, 22].

Використання Test-Driven Development (TDD) полягає в наступному. У межах підходу TDD, перед написанням будь-якої логіки API спочатку створюється тестовий сценарій, що перевіряє очікувану поведінку (наприклад, отримання коду 200 для певного запиту). Лише після цього реалізується код, необхідний для проходження тесту (табл. 1.6). Це гарантує, що вся функціональність має відповідне покриття тестами з самого початку.

Для REST API це означає:

- створення тестів на ендпоїнти ще до розробки;
- перевірку вхідних параметрів, валідацію, коди відповіді;
- швидке виявлення відхилень при рефакторингу [21].

TDD ефективно реалізується в середовищах з REST Assured (Java), SuperTest (Node.js), PyTest (Python), де можна легко моделювати запити й перевіряти відповіді у вигляді коду (табл. 1.5).

Таблиця 1.5 - Порівняльна характеристика основних критеріїв впровадження сучасних підходів на основі TDD та BDD

Критерій	TDD	BDD
Орієнтація	Тестування коду	Тестування поведінки
Мова опису	Програмна (Java, JS, Python)	Людино-зрозуміла (Gherkin)
Автоматизація	Через фреймворки тестування	Через фреймворки сценарного тестування
Покриття	Технічне	Бізнес-рівень
Приклад інструментів	REST Assured, PyTest	Cucumber, Karate, Behave
Використання в CI/CD	Висока інтеграція	Висока інтеграція

На відміну від TDD, Behavior-Driven Development (BDD) орієнтований не лише на поведінку коду, а й на бізнес-логіку, яку ця поведінка реалізує. Тести у BDD формуються у вигляді сценаріїв, які описують очікувану поведінку системи з точки зору кінцевого користувача або бізнес-аналітика. Формат таких сценаріїв часто базується на шаблоні:

Given [умова],

When [подія],

Then [очікуваний результат].

Для REST API BDD дозволяє:

- описувати сценарії використання API мовою, зрозумілою всім учасникам команди;

- генерувати автоматизовані тести із специфікацій;

- підтримувати зрозумілий зв'язок між вимогами та реалізацією.

До поширених REST API BDD інструментів відносяться Cucumber, Karate DSL, Behave (Python). Karate також особливо популярний для API, адже дозволяє поєднувати BDD-сценарії з REST-запитами без потреби у повноцінному коді [23].

Таким чином, застосування таких підходів як TDD і BDD для тестування REST API дає змогу організувати розробку з акцентом на якість, передбачуваність і відстежуваність вимог. Вони не лише сприяють автоматизації, а й забезпечують зрозумілий місток між технічними і бізнес-командами, що критично важливо в

умовах CI/CD, де швидкість релізів поєднується з вимогами до надійності [21 - 23].

1.3 Постановка задач дослідження

Проведення аналізу предметної галузі автоматизації тестування REST API в процесі CI/CD та виконання аналізу впровадження сучасних підходів та практик автоматизації тестування дозволило визначили наступне. Основними задачами роботи є:

- виконання аналізу предметної галузі автоматизації тестування REST API в процесі CI/CD;
- виконання аналізу впровадження сучасних підходів та практик автоматизації тестування REST API в CI/CD;
- визначення процесу автоматизації тестування REST API;
- запропонування методу паралельного виконання REST-assured тестів;
- виконання проектування основних та додаткових модулів програмного комплексу автоматизації тестування REST API;
- розробка основних та додаткових модулів програмного засобу;
- описання модульного підходу в архітектурі тестування REST API;
- виконання тестування контрольного прикладу для основного модуля AuthTestModule.

2 ДОСЛІДЖЕННЯ МЕТОДІВ ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАСОБУ

2.1 Визначення процесу автоматизації тестування REST API

У сучасних умовах стрімкого розвитку інформаційних технологій, що супроводжується активною імплементацією гнучких методологій розробки програмного забезпечення (Agile, DevOps), автоматизація тестування REST API стає ключовим фактором забезпечення надійності та стабільності розроблюваних програмних рішень. REST API є основним способом комунікації між мікросервісами у сучасній архітектурі, особливо в екосистемі, що базується на Java та Spring Boot. Отже, автоматизація тестування таких API в умовах безперервної інтеграції та постачання (CI/CD) стає не лише доцільною, а й критично необхідною.

Тому, у сучасному процесі розробки програмного забезпечення, що дедалі більше орієнтується на безперервну інтеграцію та постачання (CI/CD), забезпечення якості продукту має бути не відкладеним етапом, а органічною частиною життєвого циклу розробки (рис. 2.1). У цьому контексті автоматизація тестування REST API виступає не як додатковий інструмент, а як інфраструктурна необхідність.

По-перше, REST API є центральною точкою взаємодії між мікросервісами або зовнішніми клієнтами та сервером. Помилка в API викликає каскадне порушення роботи всієї системи. У класичному підході з ручним тестуванням така помилка може бути виявлена занадто пізно - після деплойменту в продуктивне середовище (рис. 2.1). Автоматизоване тестування, інтегроване у пайплайн CI/CD, дає змогу виявляти критичні помилки ще до етапу деплойменту, забезпечуючи безперервну перевірку стабільності API після кожної зміни коду.

По-друге, часті оновлення коду - які є типовими у DevOps або Agile-підходах - потребують постійної перевірки на регресію. Ручне тестування стає непрактичним через свою трудомісткість, особливо коли йдеться про десятки або сотні кінцевих точок API. Автоматизація дозволяє повторно використовувати вже створені сценарії, запускати їх паралельно у Docker-контейнерах, і отримувати зведені звіти без залучення людини.

По-третє, автоматизація тестування REST API в CI/CD підтримує високий рівень надійності і контроль якості в масштабованих мікросервісних архітектурах. Замість того, щоб покладатися на кінцеве ручне тестування продукту, перевірки здійснюються на кожному кроці: від мержу pull request до деплойменту в staging або production.

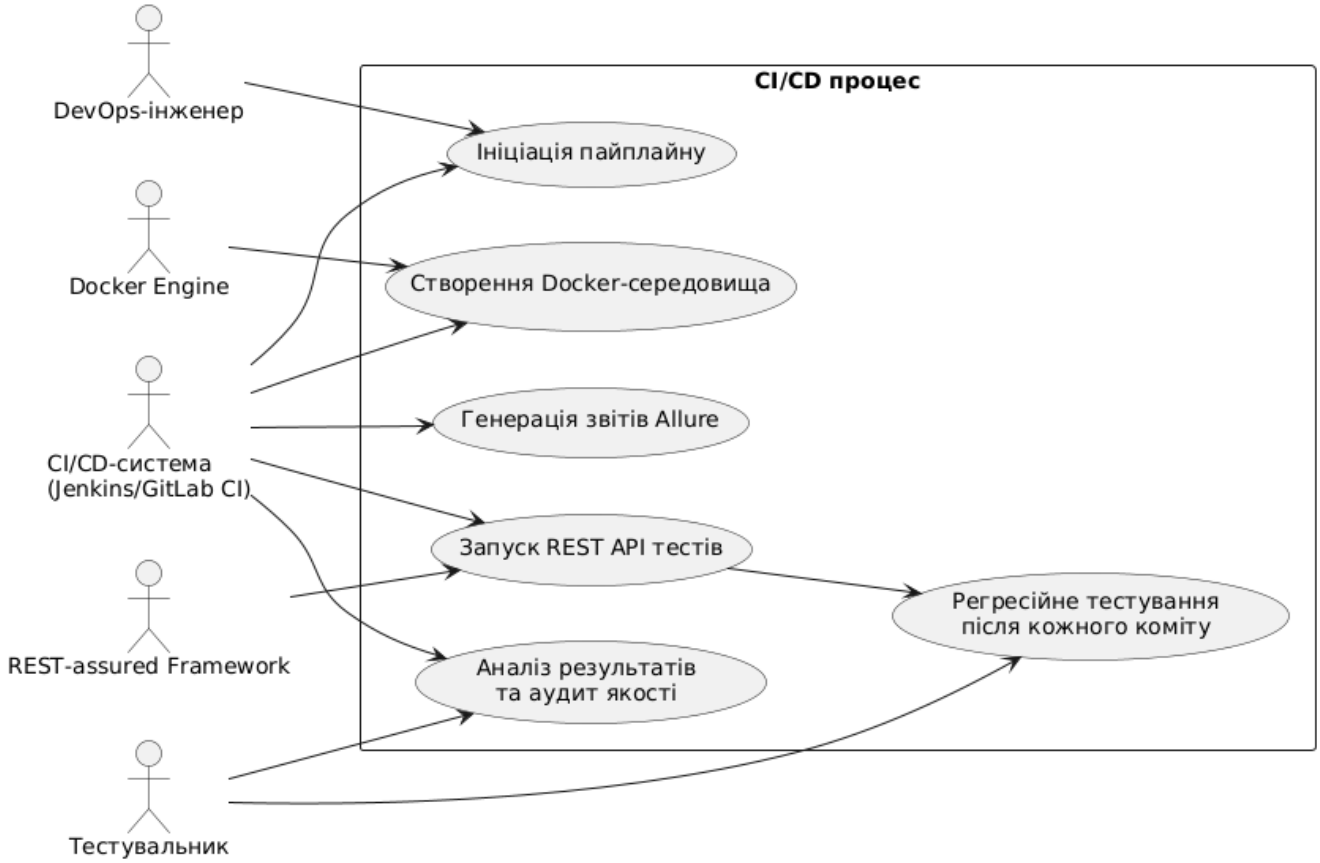

Рисунок 2.1 - UML діаграму використання процесу розробки програмного забезпечення на основі безперервної інтеграції та постачання (CI/CD)

Нарешті, інтеграція з такими системами як Jenkins, GitLab CI, Docker і фреймворками на зразок REST-assured дозволяє повністю автоматизувати перевірку бізнес-логіки API, підтримуючи відтворюваність середовища, прозорість та аудит. Таким чином, автоматизація проведення та проектування тестування REST API - це не просто покращення процесу тестування, а необхідна умова ефективного CI/CD, яка гарантує:

- стабільність релізів,

- швидкість виявлення помилок,

- зменшення людського втручання,

- підвищення продуктивності команди.

Процес автоматизації тестування REST API повинен інтегруватися безпосередньо в життєвий цикл розробки програмного забезпечення. Це означає, що кожне оновлення коду має супроводжуватись автоматичним запуском тестів, перевіркою відповідей API, їхньої коректності, стабільності та відповідності функціональним вимогам.

Необхідність інтеграції автоматизації тестування REST API необхідний в життєвому циклі розробки програмного забезпечення. Життєвий цикл розробки програмного забезпечення (SDLC) передбачає послідовність фаз - від збору вимог до обслуговування та супроводу системи. У контексті сучасних підходів (Agile, DevOps, Continuous Delivery), тестування більше не розглядається як окремий завершальний етап - воно має бути вбудоване у всі фази SDLC, починаючи з проєктування.

Автоматизація тестування REST API повинна інтегруватися безпосередньо в SDLC з кількох ключових причин.

Раннє виявлення дефектів (Shift-Left Testing). Інтеграція автоматизованих тестів на ранніх фазах дозволяє виявляти помилки ще до моменту деплойменту, що значно знижує вартість їх усунення. Це особливо актуально для REST API, де некоректна поведінка одного мікросервісу може спричинити збої в роботі всієї системи.

Підвищення швидкості розробки. Усі оновлення коду, коміти та pull-реквести повинні автоматично супроводжуватись виконанням REST API тестів. Це дозволяє паралелізувати розробку та тестування, зменшуючи час випуску нових функцій (time-to-market).

Постійна перевірка інтеграції компонентів. REST API часто є точками перетину між різними сервісами та командами. Автоматизовані інтеграційні тести дозволяють підтвердити коректність взаємодії компонентів у реальному середовищі, ще до стадії ручної перевірки.

Забезпечення відповідності вимогам. Автоматизовані REST API тести, які формуються на основі специфікацій (наприклад, OpenAPI/Swagger), дозволяють

гарантувати відповідність реалізації заявленим вимогам у динамічному середовищі.

Безперервна якість (Continuous Quality). Тестування REST API має бути не подією, а постійним процесом, який автоматично виконується після кожної зміни. Такий підхід підтримує високу якість ПЗ у будь-який момент часу, незалежно від частоти оновлень або кількості одночасних розробників.

Масштабованість та відтворюваність. Інтеграція з CI/CD системами та контейнеризація (через Docker) забезпечують високу масштабованість, дозволяють повторювати тести в ізольованому середовищі, що критично важливо для перевірки REST API з різними конфігураціями.

Таким чином, автоматизація REST API тестування як невід'ємна частина SDLC:

- мінімізує ризики;
- підвищує ефективність розробки;
- сприяє безперервному контролю якості на кожному етапі життєвого циклу.

Вибір зв'язки Java + Spring Boot для проектування та розробки програмного засобу зумовлений її широкою популярністю в корпоративному середовищі, багатим інструментарієм для реалізації REST API та високим ступенем інтеграції з інструментами CI/CD. Spring Boot спрощує розгортання мікросервісів, дозволяє легко налаштовувати REST-контролери, використовує вбудовану систему валідації, обробки винятків, документації (через Swagger/OpenAPI) тощо. У межах мікросервісної архітектури окремі сервіси виконують ізольовані бізнес-функції, взаємодіючи через RESTful API. Саме тому на кожен мікросервіс повинні бути написані незалежні модулі тестування REST API.

Вибір архітектурної основи під час проектування створенні програмного засобу для автоматизації тестування REST API має вирішальне значення для ефективності, масштабованості та підтримуваності кінцевого рішення. У межах цієї магістерської роботи обрано технологічний стек, що включає мову програмування Java, фреймворк Spring Boot і мікросервісну архітектуру (рис. 2.2). Такий вибір є результатом техніко-економічного обґрунтування, що базується на наступних критеріях.

1. Зрілість та стабільність Java як технології для розробки API. Java є однією з найбільш стабільних і надійних мов програмування, яка протягом десятиліть демонструє свою придатність для побудови високонавантажених корпоративних систем. Завдяки суворій типізації, розвиненій екосистемі та активній спільноті Java забезпечує:

- зручність для тестування (JUnit, TestNG, Mockito, REST-assured);
- підтримку інструментів CI/CD (Maven/Gradle, Jenkins, SonarQube);
- портативність через JVM та сумісність з Docker-контейнерами.

2. Потужність Spring Boot як фреймворку для побудови REST API. Spring Boot - це спрощене середовище побудови додатків на основі Spring Framework, яке автоматизує конфігурацію, управління залежностями та розгортання. Для реалізації REST API Spring Boot надає:

- простий механізм створення контролерів (@RestController);
- вбудовану підтримку валідації, обробки помилок, документації через OpenAPI (Swagger);
- гнучку інтеграцію з системами безпеки, логування та базами даних;
- сумісність із Spring Test, що дозволяє писати модульні й інтеграційні REST API тести.

Spring Boot також ідеально підходить для запуску у контейнеризованих середовищах, зокрема, завдяки підтримці "fat jar", що дозволяє легко об'єднувати застосунок із усіма залежностями.

3. Мікросервісна архітектура як основа масштабованості та гнучкості. Мікросервісна архітектура дозволяє розділити складний програмний продукт на незалежні компоненти (сервіси), які:

- легко розгортаються окремо;
- тестуються незалежно один від одного;
- масштабуються автономно;
- мають власний життєвий цикл.

У контексті тестування REST API це означає, що:

- можна ізолювати API кожного сервісу та тестувати його незалежно;

- у CI/CD пайплайні можна автоматизувати перевірки лише зміненого мікросервісу;

- застосовується підхід "test early, test often", який відповідає DevOps-принципам.

4. Широка підтримка та стандартизація (рис. 2.2). Java + Spring Boot + мікросервіси - це усталений стандарт в індустрії, який підтримується:

- великою кількістю документації;

- активною спільнотою;

- великою кількістю готових рішень (Spring Cloud, Netflix OSS, Kubernetes-орієнтовані бібліотеки).

Це забезпечує стабільну базу для реалізації не тільки REST API, а й процесу їх автоматизованого тестування в масштабованому, продуктивному середовищі. Отже, поєднання Java, Spring Boot і мікросервісної архітектури не лише відповідає технічним вимогам проекту, але й забезпечує найвищу сумісність із сучасними практиками автоматизації тестування, CI/CD та контейнеризації.

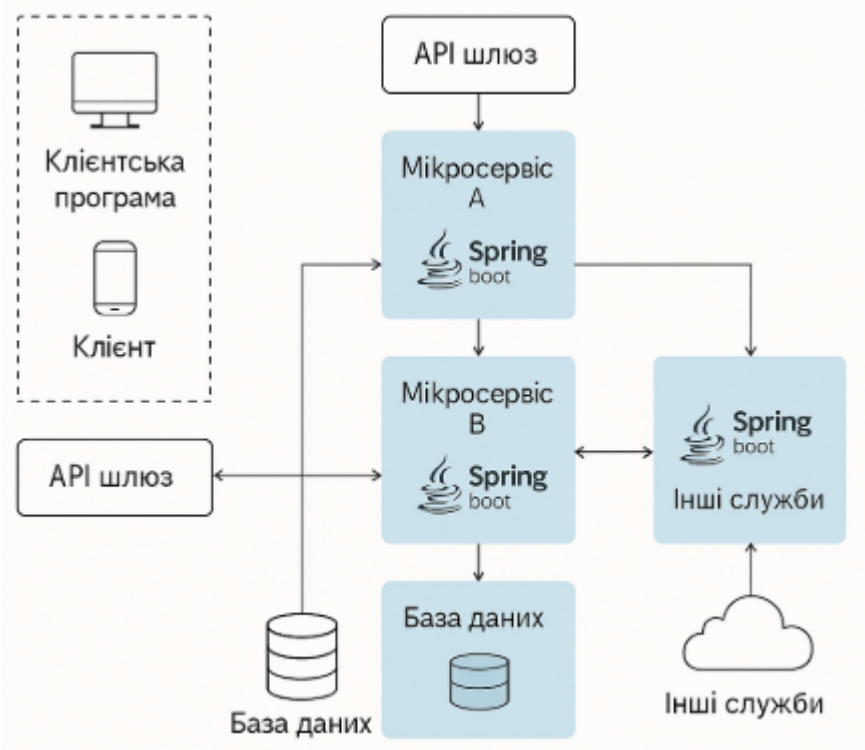

Рисунок 2.2 - Обгрунтування вибору архітектурної основи для проектування програмного засобу

Особливе значення набуває використання бібліотеки REST-assured, яка забезпечує декларативний та зручний синтаксис для перевірки REST API. Наприклад, з її допомогою легко реалізується аутентифікація, перевірка статус-кодів, валідація JSON-відповідей, тестування із параметрами та виклики на основі шаблонів.

Data-driven підхід у REST API тестуванні полягає в наступному. Одним із найбільш ефективних підходів до організації автоматизованих тестів є data-driven testing - тестування, що передбачає запуск одного і того самого тест-кейсу з різними вхідними даними. Такий підхід дозволяє досягнути масштабованості, мінімізувати дублювання коду та спростити супровід тестів.

У контексті REST API data-driven підхід реалізується шляхом передачі параметрів запитів (URL, headers, body) та очікуваних відповідей ззовні (наприклад, із CSV, JSON, Excel або баз даних). При цьому REST-assured може використовуватись у поєднанні з JUnit або TestNG, де параметризовані тести створюються динамічно під час виконання. Наприклад, можна автоматизувати перевірку 50 різних сценаріїв логінізації або створення ресурсів одним тестом, що зчитує дані з JSON-масиву.

Такий підхід особливо вигідний при CI/CD, оскільки дає змогу легко додавати нові сценарії без зміни структури тестів, а також інтегрувати масове тестування на ранніх етапах життєвого циклу програмного продукту.

Інтеграція автоматизованих тестів REST API у процес безперервної інтеграції (CI) та безперервного постачання (CD) забезпечує їх систематичне виконання після кожного коміту до репозиторію (табл. 2.2).

Інтеграція REST API тестів у процес CI/CD виконується після кожного коміту таким чином. У системах безперервної інтеграції (CI), таких як Jenkins, GitLab CI, GitHub Actions або TeamCity, автоматизовані REST API тести інтегруються як етап (stage) у пайплайні. Після кожного коміту (push) до репозиторію, тригер запускає Jenkins pipeline, який включає такі кроки:

- отримання змін з репозиторію (наприклад, з Git);
- збирання проекту (наприклад, за допомогою Maven/Gradle);
- створення Docker-образу програми (опціонально);

- запуск середовища (test/staging) у Docker-контейнерах;
- виконання REST API тестів (JUnit + REST-assured);
- збір результатів (Allure, Surefire);
- розгортання (deployment), якщо всі тести успішні.

Таблиця 2.2 - Критерії проведення автоматизованого тестування (REST-assured + CI/CD) REST API

Критерій	Автоматизоване тестування	Коментар
Швидкість виконання	Швидке, виконується паралельно в контейнерах	Автоматизація дає значну перевагу
Масштабованість	Висока	Data-driven тестування дозволяє ефективне масштабування
Людський фактор	Мінімальна ймовірність помилок	Знижується ризик некоректного вводу
Інтеграція в CI/CD	Повністю інтегрована	Jenkins/Docker забезпечують велику гнучкість
Покриття сценаріїв	Повне, включно з негативними сценаріями	Параметризовані тести ефективно покривають edge cases
Аналіз результатів	Автоматизована звітність	Наприклад, через Allure
Навчальна крива	Середня	Потребує значних знань Jenkins, Docker, Java

Таке інтегроване виконання дає змогу:

- автоматично запускати тести без участі розробника;
- гарантовано перевіряти критичні API після кожної зміни;
- блокувати подальші етапи деплойменту в разі невдачі тестів.

В наслідок цього, для забезпечення ефективності, автоматизовані тести REST API повинні відповідати набору об'єктивних критеріїв. Ці критерії діляться на структурні (що тестувати) та операційні (як і коли тестувати).

Інтеграція ручних тестів у процес CI/CD на перший погляд може здаватися суперечністю. Проте існують реальні сценарії, коли ручне тестування REST API

частково інтегрується в CI/CD, зокрема через напівавтоматизацію, тригери перевірки, або контроль якості перед деплойментом (табл. 2.3).

Таблиця 2.3 - Критерії проведення ручного тестування

Критерій	Ручне тестування	Коментар
Швидкість виконання	Повільне, залежить від швидкості людини	Ручне тестування не дає перевагу перед автоматизованим
Масштабованість	Обмежена	Data-driven тестування не дозволяє масштабування в повній мірі
Людський фактор	Висока ймовірність помилок	Підвищений ризик некоректного вводу даних
Інтеграція в CI/CD	Неможлива або складна	не забезпечується значна гнучкість
Покриття сценаріїв	Лише частина сценаріїв	Параметризовані тести дуже помірно покривають edge cases
Аналіз результатів	Потребує ручного збору	Власні програмні інструменти
Навчальна крива	Низька	Не потребує значних знань Jenkins, Docker, Java в наслідок обмеженого їх використання

Інтеграція ручних тестів REST API у процес CI/CD: особливості та реалізація виконується наступним чином. У традиційній практиці ручне тестування REST API зазвичай проводиться окремо від автоматизованих пайплайнів CI/CD. Проте в умовах, коли проєкт знаходиться на ранніх стадіях розробки, або при відсутності повної автоматизації, можливе включення ручного етапу перевірки в структуру CI/CD-процесу. Така інтеграція має специфічну форму та функціональність.

При цьому Pipeline зупиняється на ручному етапі (manual gate), де у пайплайні CI/CD створюється окремий етап ручного підтвердження, що призупиняє автоматичне просування до наступних фаз (наприклад, розгортання на staging або production), поки не буде вручну перевірено REST API.

Наведемо такий Jenkins приклад (Pipeline input step):

```
stage('Manual API Check') {
   steps {
        input message: 'Проведіть ручне тестування REST API у Postman та натисніть "Продовжити"'
   }
}
```

Нотифікація до тестувальника полягає в тому, що система CI/CD надсилає повідомлення (email, Slack, Teams) до відповідального QA-інженера, що нова версія сервісу потребує перевірки. У повідомленні міститься:

- версія сервісу;
- список кінцевих точок;
- чек-лист ручного тестування;
- посилання на середовище (test/staging).

Ручна перевірка у Postman, Swagger, curl також проводиться поширено, де тестувальник перевіряє REST API за перевіреним сценарієм (CRUD, авторизація, edge cases). Тестові результати часто фіксуються вручну або у вигляді Postman Collection Reports.

Верифікація вручну у пайплайні виконується після успішного ручного тестування QA інженер натискає кнопку "Продовжити" в CI/CD-середовищі, що активує наступний етап (наприклад, деплоймент).

Архівація або логування результатів являють собою чек-листи, звіти (PDF, Excel, JSON) або колекції Postman, що можуть зберігатися у системі контролю версій або архівуватись у CI/CD-середовищі як артефакти.

В загальному випадку, ручне тестування REST API - це процес перевірки функціональності, надійності, безпеки та відповідності API до специфікації, який виконується без використання автоматизованих фреймворків. В основі цього підходу лежать чітко визначені критерії, які визначають, що саме має бути перевірено у рамках кожного сценарію. При цьому, ручне тестування REST API базується на чітко визначених, багаторазово повторюваних критеріях, які охоплюють функціональність, безпеку, стабільність і відповідність специфікації.

Незважаючи на обмеження в масштабованості, цей підхід залишається необхідним на етапах дослідження, специфікації, smoke-тестування або коли автоматизація ще не впроваджена.

Таким чином, хоча ручне тестування REST API не може бути повністю інтегрованим у CI/CD, воно може виступати в якості контрольного бар'єру (manual gate) перед критичними діями. Це дозволяє підтримувати базову перевірку у проектах без повної автоматизації, або при критично важливих релізах, де необхідна експертна перевірка.

Найчастіше для реалізації інтеграції REST-assured тестів у Jenkins Pipeline з використанням Docker використовують Jenkins - відкриту систему автоматизації, яка дозволяє налаштовувати складні пайплайни (pipeline) для оркестрації всього життєвого циклу тестування. Щоб досягти масштабованості, модульності та ізоляції, усі компоненти CI/CD пайплайну зазвичай розгортаються у Docker-контейнерах. Це дозволяє:

- уникнути залежностей між середовищами;
- виконувати тести паралельно;
- масштабувати запуск тестів у декількох ізольованих екземплярах;
- зменшити час виконання тестів та загального CI/CD процесу.

Розглянемо типову схему CI/CD процесу з автоматизованим тестуванням REST API (рис. 2.3) під час якої виконуються наступні дії:

- Push коду до репозиторію (наприклад, GitLab/GitHub);
- Jenkins отримує тригер на оновлення коду;
- ініціалізується Jenkins pipeline;
- будується Docker-образ з мікросервісом;
- розгортається тестове середовище;
- запускається контейнер з REST-assured тестами (наприклад, за допомогою Maven).

Далі формуються результати тестів (наприклад, Allure reports) публікуються у Jenkins (рис. 2.3). Якщо всі тести пройдено успішно - Jenkins передає управління етапу деплойменту.

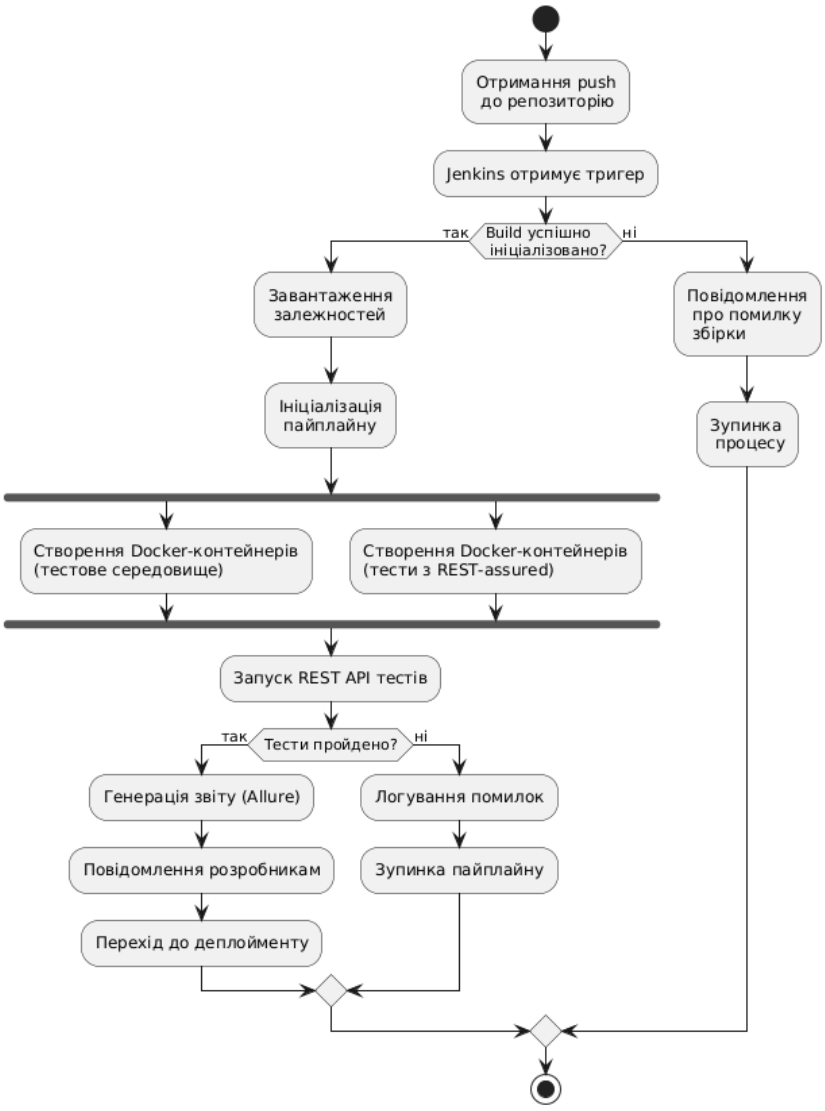

Рисунок 2.3 - UML-діаграма активностей процесу інтеграції REST-assured тестів у Jenkins Pipeline з використанням Docker

UML-діаграма (рис. 2.3) наочно демонструє повний життєвий цикл автоматизованого тестування REST API, починаючи з моменту оновлення коду до деплойменту або відхилення змін. Вона чітко ілюструє важливість автоматизації в рамках CI/CD, а також впровадження інструментів контейнеризації та звітності для масштабованої та керованої розробки. На ній подано детальний опис UML-діаграми активностей, що моделює процес інтеграції REST-assured тестів у Jenkins Pipeline із використанням Docker-контейнерів. Така діаграма відображає не лише основні етапи CI/CD, а й варіанти розгалужень, умовні переходи та паралельне виконання завдань - тобто вона відповідає вимогам до розгалуженої UML-діаграми активностей.

Наступна UML-діаграма послідовності (рис. 2.4) відображає взаємодію між основними компонентами процесу автоматизованого тестування REST API в CI/CD, та ілюструє типовий сценарій, починаючи з запуску пайплайну в системі безперервної інтеграції Jenkins. Першим етапом у цій послідовності є ініціація Jenkins, який надсилає команду до Docker Engine на створення контейнера з тестовим середовищем, що містить реалізацію REST-assured тестів. Docker, у свою чергу, інтерпретує цю команду та запускає відповідний тестовий контейнер, всередині якого відбувається виконання автоматизованих тестів. У межах виконання тестів, REST-assured надсилає HTTP-запити до API Endpoint, який, як правило, реалізовано за допомогою Spring Boot. Кожен запит відповідає певному сценарію тестування: перевірка статус-кодів, правильності відповіді, обробки некоректних даних, авторизації тощо. API обробляє ці запити та повертає відповіді (у форматі JSON, XML або іншому, залежно від специфікації), які REST-assured зчитує, аналізує та порівнює з очікуваними результатами.

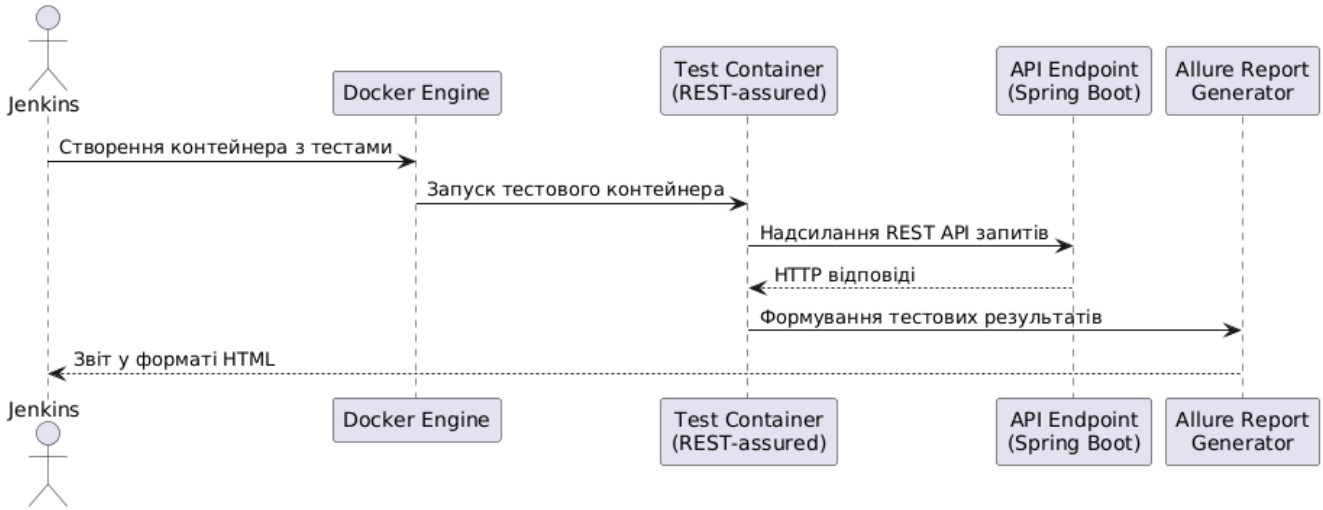

Рисунок 2.4 - UML-діаграма послідовності процесу взаємодії між основними об'єктами автоматизованого тестування REST API в CI/CD

Після завершення виконання всіх тестів, тестовий контейнер передає зібрані результати до модуля формування звітності - Allure Report Generator (рис. 2.4). Цей компонент обробляє метадані тестів (наприклад, які саме кейси пройшли/провалились, скільки часу зайняло виконання, які помилки було виявлено) та формує зручний у використанні HTML-звіт. Після генерації звіту Allure повертає

його Jenkins, що дозволяє відповідальним членам команди (розробникам, тестувальникам, DevOps-інженерам) переглядати результати в інтегрованому вигляді безпосередньо в CI/CD середовищі. Такий підхід забезпечує прозорість, відтворюваність та швидкість аналізу помилок, а також автоматизує контроль якості при кожній ітерації зміни коду. Важливо підкреслити, що ця UML-діаграма послідовності демонструє саме логіку взаємодії, а не внутрішню реалізацію, і служить засобом візуалізації процесу тестування REST API як невід'ємної частини сучасного DevOps-процесу.

Таким чином, на основі порівняльного аналізу можна дійти висновку, що автоматизоване тестування REST API із використанням REST-assured, CI/CD пайплайнів на Jenkins, а також Docker-контейнеризації забезпечує істотні переваги над ручним тестуванням за всіма критеріями. Особливо важливими перевагами є швидкість, стабільність, маштабованість, а також зниження людського фактору. Тому доцільним є саме цей підхід при реалізації програмного засобу, спрямованого на автоматизацію процесу тестування в сучасних програмних проектах.

2.2 Запропонований метод паралельного виконання REST-assured тестів

У процесі автоматизації тестування REST API у межах CI/CD-пайплайну виникає низка технічних викликів, зокрема - потреба у скороченні часу виконання тестів, забезпеченні незалежності середовищ, а також врахуванні взаємозалежностей між мікросервісами. Зважаючи на це, було запропоновано метод паралельного виконання REST-assured тестів, який реалізується за допомогою Jenkins Pipeline, Docker-контейнеризації та інфраструктурної ізоляції для кожної групи тестів. Даний підхід дозволяє підвищити пропускну здатність процесу CI/CD за рахунок одночасного виконання модульних та інтеграційних тестів у декількох ізольованих середовищах.

Архітектурні принципи запропонованого методу полягає у наступному. Jenkins ініціює CI-пайплайн після кожного коміту або pull request. На етапі Build and Test пайплайн розгалужується на декілька паралельних гілок, кожна з яких:

- створює окремий Docker-контейнер;

- підключає лише ті мікросервіси, з якими має взаємодіяти конкретна група REST-assured тестів;

- виконує серії тестів незалежно від інших потоків.

Після завершення всіх потоків, Jenkins об’єднує результати тестів (наприклад, через Allure) та приймає рішення про подальший етап деплойменту.

Розроблена UML-діаграма активностей (рис. 2.5) відображає логіку роботи етапу Build and Test у межах запропонованого Jenkins Pipeline для автоматизованого тестування REST API з використанням фреймворку REST-assured. Процес починається з активації відповідного етапу у пайплайні CI/CD після отримання коміту до репозиторію або запуску pull request. На цьому етапі відбувається розгалуження на декілька паралельних потоків за допомогою механізму fork, що в UML-позначенні символізує незалежне виконання дій у кількох гілках одночасно.

Кожен із паралельних потоків відповідає за виконання REST-assured тестів певної групи мікросервісів. У межах кожної гілки створюється ізольований Docker-контейнер, у якому відтворюється необхідне середовище для тестування - зокрема, підключаються лише ті мікросервіси, з якими взаємодіє відповідна група REST-запитів. Завдяки такому підходу забезпечується незалежність середовищ і відсутність конфліктів між паралельними сесіями, що особливо важливо в умовах мікросервісної архітектури.

Після створення контейнерів та підключення відповідних сервісів у кожній гілці виконується серія REST-assured тестів. Ці тести реалізують data-driven підхід, перевіряючи різні сценарії взаємодії з API: позитивні, негативні, граничні випадки, обробку винятків, авторизацію тощо. Кожна гілка працює автономно й не залежить від інших, що дозволяє значно скоротити загальний час тестування у порівнянні з послідовним запуском усіх кейсів у єдиному середовищі.

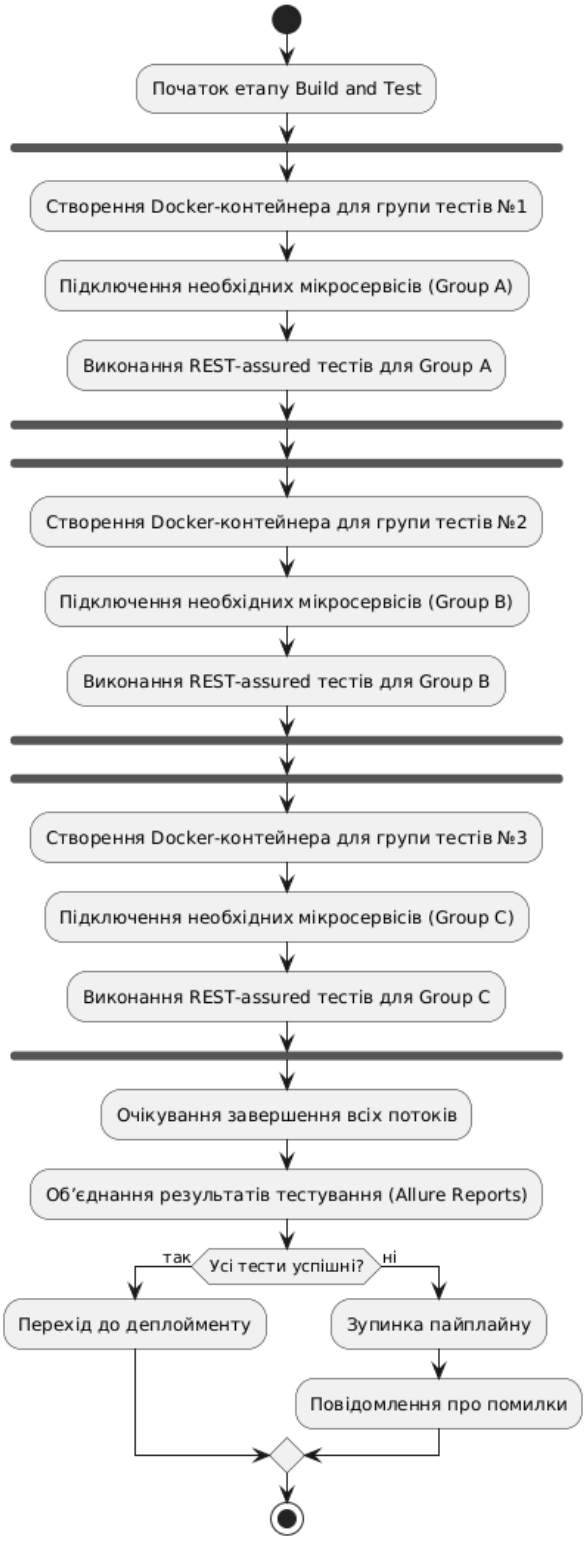

Рисунок 2.5 - UML-діаграма активностей архітектурних принципів роботи запропонованого методу

Після завершення всіх паралельних потоків виконується синхронізація результатів. Jenkins очікує завершення кожного потоку і, щойно всі тести виконано, переходить до етапу агрегації результатів. На цьому етапі результати з

усіх Docker-контейнерів збираються за допомогою інструменту формування звітності - наприклад, Allure Report Generator, - що дозволяє візуалізувати успішні та провалені тести, а також згенерувати повну аналітику в інтерактивному форматі.

Далі в логіці роботи UML діаграми активностей (рис. 2.5) реалізовано умовний оператор if, який перевіряє результат тестування. Якщо всі тести завершені успішно, пайплайн переходить до наступного етапу - розгортання додатку у staging або production середовищі. У разі ж виявлення помилок пайплайн автоматично зупиняється, а відповідальні фахівці (DevOps або QA-команда) отримують повідомлення про помилки та звіт для подальшого аналізу.

Таким чином, ця діаграма наочно демонструє оптимізований підхід до автоматизованого тестування REST API, який забезпечує повну паралельність, ізоляцію середовищ, ефективне використання ресурсів та високу швидкість CI/CD-процесу без втрати точності перевірки функціональності програмного забезпечення.

На відміну від класичного підходу, де всі тести виконуються послідовно в одному середовищі, або паралелізації без контейнерів, даний метод:

- дозволяє масштабувати CI/CD горизонтально;
- забезпечує незалежність оточень для кожного піднабору тестів;
- дозволяє уникати конфліктів доступу до ресурсів (портів, баз даних, токенів);
- дозволяє динамічно керувати навантаженням на основі ресурсів хосту.

Наприклад, тестування REST API мікросервісу “User Service” може виконуватись у одному контейнері з емуляцією зовнішніх залежностей, тоді як паралельно в іншому контейнері тестується “Payment Service”, який взаємодіє з тестовою базою даних та сервісом авторизації. Завдяки цьому досягається ізоляція тестів, точність відтворення помилок, а також прискорення всього процесу CI/CD.

Таблиця 2.4 - Порівняльна характеристика ефективності методів виконання REST-assured тестів

Критерій	Класичний підхід (послідовно, без Docker)	Паралелізація TestNG без Docker	Запропонований метод (Jenkins + Docker)
Час виконання тестів	Високий (30–60 хв)	Середній (15–25 хв)	Низький (5–10 хв)
Паралельне виконання	Відсутнє	Частково (через threads)	Повноцінне (через окремі контейнери)
Ізоляція середовищ	Відсутня	Часткова (тільки JVM-ізоляція)	Повна (через Docker namespaces)
Залежності мікросервісів	Можливі конфлікти	Вимагає складної конфігурації	Гнучке підключення потрібних сервісів
Гнучкість масштабування	Обмежена	Обмежена JVM	Висока (горизонтальне масштабування)
Інтеграція в Jenkins Pipeline	Легка, але обмежена	Середня складність	Висока гнучкість за допомогою stages/agents
Відтворюваність помилок	Складна	Складна	Висока (ідентичне середовище)
Сумісність з DevOps-практиками	Низька	Середня	Висока

Таким чином, в результаті порівняльного аналізу можна стверджувати, що запропонований метод паралельного виконання REST-assured тестів на основі Docker-контейнеризації та Jenkins Pipeline демонструє найвищу ефективність серед розглянутих підходів. Основною перевагою методу є реальна паралелізація із повною ізоляцією, що забезпечує мінімізацію конфліктів між тестами, зниження часу виконання CI/CD, а також масштабованість процесу в умовах складних мікросервісних архітектур. Крім того, завдяки чіткій структурі Jenkins Pipeline та використанню контейнерних технологій, метод може бути легко адаптований до нових сервісів або розширень, що відповідає динаміці сучасного розробницького процесу. Відповідно, такий підхід є перспективним для впровадження у реальні

індустріальні проєкти та рекомендованим для команд, що працюють за DevOps-підходом.

2.3 Проектування основних та додаткових модулів програмного комплексу автоматизації тестування REST API

У контексті автоматизації тестування REST API у мікросервісній архітектурі важливим завданням є правильне проєктування тестових компонентів, які мають інтегруватися у вже існуючу інфраструктуру програмного забезпечення. Зважаючи на те, що мікросервісна платформа, як правило, складається з десятків незалежних сервісів, що взаємодіють через REST API, доцільним є створення декомпозованої системи модулів (рис.), кожен із яких відповідає за окремий аспект тестування.

Проєктований програмний комплекс тестування REST API реалізується на основі Java + Spring Boot, із застосуванням бібліотеки REST-assured, яка забезпечує декларативний підхід до формування HTTP-запитів і перевірки їх результатів. Тести реалізуються як окремі модулі, які вбудовуються безпосередньо у Jenkins Pipeline та розгортаються в ізольованих Docker-контейнерах для забезпечення паралельного виконання (рис. 2.6).

Основою архітектури є data-driven підхід, що дозволяє зберігати дані тестових сценаріїв у зовнішніх джерелах (JSON, CSV або бази даних) і запускати однакові тести з різними наборами параметрів.

Концепція інтегрованої модульної системи тестування полягає в наступному. Із урахуванням принципів SRP (Single Responsibility Principle) та високої роздільності мікросервісної архітектури, було виділено дві категорії модулів у складі тестового комплексу:

- основні модулі, що безпосередньо реалізують автоматизовані тести REST API;

- додаткові модулі, які забезпечують допоміжну функціональність - генерацію тестових даних, конфігурацію середовища, логування та звітність.

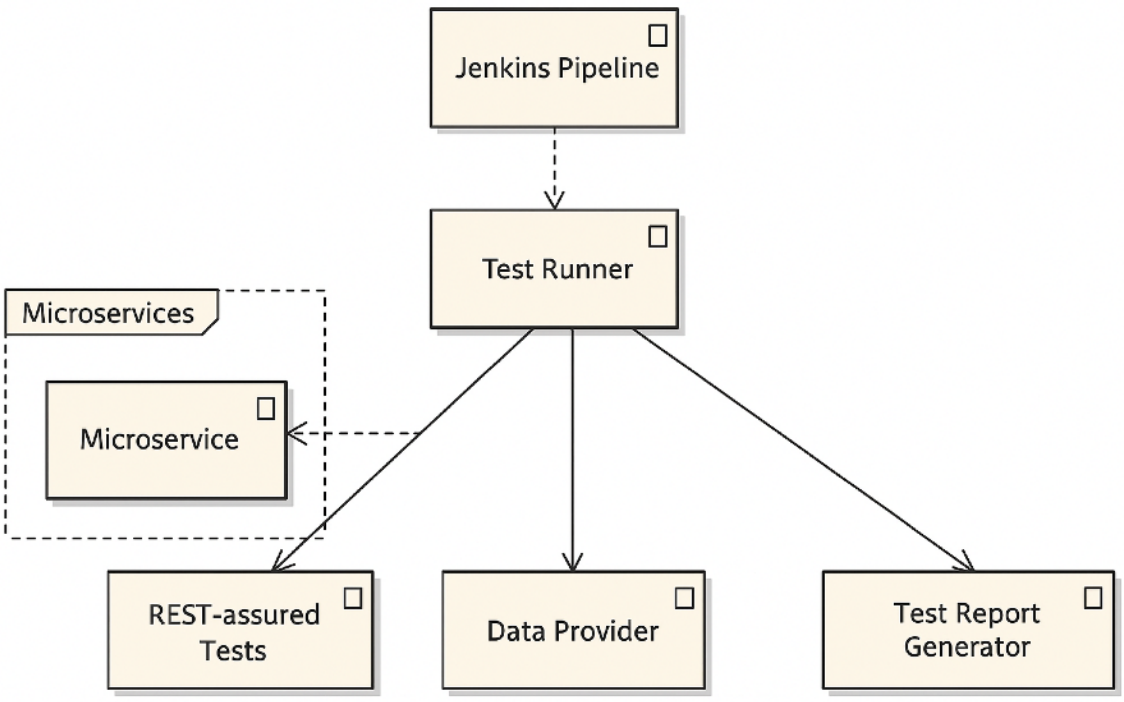

Рисунок 2.6 - Декомпозована система модулів, що побудована на основі Jenkins Pipeline та розгортаються в ізольованих Docker-контейнерах

До основних модулів належать наступні.

Модуль тестування авторизації та аутентифікації (AuthTestModule). Він відповідає за перевірку коректності логінізації, реєстрації, оновлення токенів, захисту доступу до ресурсів. Включає як позитивні, так і негативні сценарії. Цей модуль виконує тестування механізмів доступу до ресурсів на основі токенів, а також перевірку процедур реєстрації, входу в систему, оновлення токенів і виходу. Його особливістю є наявність як позитивних сценаріїв (успішна аутентифікація), так і негативних (наприклад, введення неправильного паролю, відсутній токен, закінчення терміну дії токена). REST-assured забезпечує зручну перевірку статус-кодів (401, 403, 200), а також валідацію структури відповіді (JSON-поля: accessToken, expiresIn, тощо). В умовах data-driven підходу цей модуль читає набір облікових даних із зовнішнього джерела (наприклад, JSON-файл або БД), що забезпечує широке охоплення сценаріїв.

Модуль CRUD-тестів (CrudTestModule). Він реалізує перевірку базової функціональності REST API: створення, читання, оновлення та видалення ресурсів (наприклад, користувачів, замовлень тощо). Основна задача цього модуля

- забезпечення перевірки базових REST-операцій: POST, GET, PUT, DELETE для окремих ресурсів (користувачі, продукти, замовлення тощо). Кожна операція супроводжується перевіркою статус-коду (201, 200, 204, 404, 400), вмісту тіла відповіді, а також відповідності фактичного результату очікуваному. Особливістю є ідемпотентність деяких запитів (наприклад, PUT), яку модуль перевіряє через повторні виклики з однаковими даними. Також реалізована підтримка створення залежних об’єктів (наприклад, створення адреси до користувача) за допомогою populator-класів. Кожна група тестів конфігурується через параметри, що дозволяє запускати модуль у різних мікросервісних контекстах.

Модуль інтеграційного тестування (IntegrationTestModule). Він перевіряє міжсервісну взаємодію (наприклад, взаємозв’язок між Order Service та Payment Service), симулюючи комплексні бізнес-сценарії. Цей модуль орієнтований на перевірку взаємодії між мікросервісами. На відміну від модулів, які тестують лише один REST API, IntegrationTestModule перевіряє бізнес-логіку, що залежить від декількох сервісів одночасно. Наприклад, сценарій: створення замовлення → генерація рахунку → підтвердження оплати - охоплює як Order Service, так і Billing/Payment Service. REST-assured використовується для виконання послідовних викликів і передачі результатів одного API у тіло іншого (chaining requests). Для таких тестів зазвичай застосовується setup/teardown-логіка, де середовище ініціалізується перед тестом і очищується після, що дозволяє забезпечити ізольованість і відтворюваність.

Модуль валідації відповідей (ResponseValidationModule). Він аналізує структуру JSON-відповідей, перевіряє відповідність схемі (JSON Schema Validation), коректність кодів помилок, повідомлень. Цей модуль забезпечує ретельну перевірку коректності структури, типів і значень відповіді, що надходить від REST API. Реалізується перевірка:

- відповідності полів JSON очікуваній схемі (id: integer, email: string, createdAt: datetime);

- коректності повідомлень про помилки;

- наявності обов’язкових ключів;

- форматів дат, переліків тощо.

Особливістю ResponseValidationModule є використання JSON Schema Validation, який підключається як окрема бібліотека до REST-assured. Це дозволяє звіряти відповіді не вручну, а автоматично, на основі заздалегідь визначених схем. Також реалізовано можливість "soft assertions", тобто накопичення помилок без зупинки тесту, що є особливо корисним при масовому тестуванні одного endpoint з багатьма варіантами запитів.

До загальних особливостей реалізації всіх основних модулів належить наступне:

- мова реалізації в якості Java 17+;

- фреймворк в якості Spring Boot Test + REST-assured + JUnit 5;

- паралельне виконання, що підтримується через Jenkins stages та Docker-контейнери;

- конфігурація яка централізовано через YAML-файли, .env або CI/CD secrets;

- життєвий цикл в якості Before/After hooks + TestContainers для динамічного оточення;

- агрегація результатів, де модулі формують результат у форматі, що сумісний з Allure.

До додаткових модулів відносяться наступні.

Модуль генерації тестових даних (TestDataGenerator). Він використовується для формування динамічних, випадкових або параметризованих даних, які надсилаються в API-запитах. Працює із Faker або кастомним генератором. Модуль призначено для створення вхідних даних, які використовуються під час тестування REST API. Його основна задача - автоматично генерувати або відбирати дані відповідно до вимог конкретного сценарію: від типових (наприклад, валідна email-адреса) до граничних (порожні поля, дуже довгі рядки, SQL-ін’єкції). Застосування бібліотеки Java Faker або кастомних генераторів

дозволяє модулю створювати реалістичні дані, які мінімізують повторення й підвищують покриття тестів.

Особливістю модуля є підтримка data-driven підходу: дані можуть зберігатися у зовнішніх джерелах - JSON, CSV, Excel або базі даних - і динамічно підставлятися під час виконання тестів. Завдяки цьому забезпечується масштабованість і зручність супроводу без потреби переписувати тести при зміні набору вхідних параметрів.

Модуль конфігурації середовища (EnvConfigModule). Він відповідає за зчитування параметрів середовища (base URL, токени, порти тощо) з .properties, .yaml або CI/CD pipeline variables. Цей модуль відповідає за керування конфігураційними параметрами, необхідними для виконання тестів. До таких параметрів належать: базова адреса API (baseUrl), ключі автентифікації, порти, назви сервісів, режими запуску (test, staging, production) тощо. EnvConfigModule зчитує ці параметри з .properties, .yaml або змінних середовища, що задаються у Jenkins або Docker.

Особливістю модуля є підтримка профілів середовищ - кожен запуск тестів може використовувати окремий профіль (наприклад, qa, dev, prod), завдяки чому тести стають універсальними та ізольованими від середовища. Також реалізована логіка fallback: якщо параметр не знайдено, використовується значення за замовчуванням.

Модуль логування (LoggingModule). Він здійснює запис запитів, відповідей, часу виконання, що використовується як у тестах, так і в Jenkins-звітах. LoggingModule забезпечує збір, форматування та вивід журналів подій, які супроводжують виконання тестів. Він логірує:

- кожен HTTP-запит (метод, URL, тіло, заголовки),
- отриману відповідь (статус-код, тіло),
- час виконання,
- успішність чи невдачу тесту.

Ці дані можуть зберігатися у вигляді простих текстових логів, JSON-записів або інтегруватися з зовнішніми системами журналювання (наприклад, ELK Stack або Graylog). Особливістю є можливість динамічного перемикання рівня

логування (INFO, DEBUG, ERROR) в залежності від параметрів запуску. Це дозволяє зберігати лаконічні логи під час рутинного виконання та розширені - у разі збоїв.

Модуль генерації звітів (ReportModule). Він інтегрується з Allure для формування звітності після завершення всіх тестів, включає також маркування тестів за групами, важливістю та мікросервісом.

Цей модуль відповідає за агрегацію та візуалізацію результатів тестування. Основна його функція - перетворення «сирих» результатів виконання тестів у читабельні, структуровані звіти, зручні для аналізу. Для цього використовується інтеграція з Allure Framework, яка дозволяє:

- відображати кожен тест-кейс як окремий блок з описом, статусом, даними вхід/вихід;

- групувати тести за мікросервісами, модулями, важливістю;

- прикріплювати скріншоти, запити, логи до звіту.

Особливістю є можливість запуску Allure Report у Docker-контейнері автоматично після завершення всіх тестів у пайплайні Jenkins, що дозволяє зберігати звіт у вигляді артефакту або публікувати його на внутрішньому сервері аналітики. Також підтримується історія запусків, що дає змогу аналізувати тенденції та виявляти нестабільні тести.

До загальних особливостей реалізації додаткових модулів належать наступні чинникі:

- структура, де кожен модуль реалізовано як окремий Java-компонент зі своєю інтерфейсною частиною (інтерфейси, конфігураційні класи);

- взаємодія, де модулі взаємодіють з основними через DI (Dependency Injection), що забезпечує слабке зв'язування та тестованість;

- тестованість, де додаткові модулі мають окремі модульні тести (наприклад, для генератора даних або логера);

- масштабованість, де завдяки зовнішнім конфігураціям і стандартам JSON/ YAML, модулі легко розширюються без зміни основної логіки тестування.

Далі, наданом порівняльну характеристику поділу наданих модулів (табл. 2.5).

Таблиця 2.5 - Порівняльна таблиця функціонального поділу модулів

Назва модуля	Категорія	Основна функція	Інтеграція з Jenkins/Docker
AuthTestModule	Основний	Тестування авторизації, токенів, безпеки	Так
CrudTestModule	Основний	Тестування базових REST-операцій	Так
IntegrationTestModule	Основний	Взаємодія між мікросервісами	Так
ResponseValidationModule	Основний	Перевірка JSON-структур та статус-кодів	Так
TestDataGenerator	Додатковий	Налаштування середовища на основі профілю або Jenkins variables	Так
LoggingModule	Додатковий	Збір інформації про запити та відповіді	Частково
ReportModule	Додатковий	Побудова звітів Allure, інтеграція з CI	Так

Таким чином, запропонована модульна структура системи автоматизації тестування REST API забезпечує гнучке розширення, адаптацію до змін у мікросервісній платформі та підтримку DevOps-практик. Поділ на основні та додаткові модулі дозволяє досягти високої когерентності тестової логіки, повторного використання компонентів та незалежного масштабування підсистем. Впровадження таких модулів у межах Jenkins Pipeline із підтримкою Docker-ізоляції створює умови для надійного, масштабованого та підтримуваного тестування API в умовах сучасного CI/CD-процесу.

3 РОЗРОБКА МОДУЛІВ ПРОГРАМНОГО ЗАСОБУ

3.1 Розробка основних модулів програмного засобу

Модуль AuthTestModule є ключовим функціональним елементом програмного засобу для автоматизації тестування REST API, що орієнтований на перевірку механізмів аутентифікації, авторизації та обробки токенів. Його призначення (табл. 3.1) та впровадження забезпечує критичну перевірку безпекових процедур мікросервісної платформи, зокрема на етапах входу користувача, реєстрації, оновлення токенів та перевірки доступу до захищених ресурсів.

Таблиця 3.1 - Функціональне призначення основних модулів

Назва модуля	Основне призначення	Тип тестування
AuthTestModule	Тестування аутентифікації, авторизації, токенів	Функціональне
CrudTestModule	Перевірка операцій POST/GET/PUT/ DELETE для REST API	Функціональне
IntegrationTest-Module	Симуляція бізнес-сценаріїв між мікросервісами	Інтеграційне
ResponseValida-tionModule	Автоматична валідація JSON-відповідей	Перехресне

Функціональне призначення модуля AuthTestModule (табл. 3.1) полягає в наступному. AuthTestModule реалізує автоматизовані тести для наступних сценаріїв:

- успішна авторизація зареєстрованого користувача;
- невдала авторизація у випадку введення неправильних облікових даних;
- перевірка коректного формування JWT токена (наявність accessToken, expiresIn, tokenType тощо);
- тестування логіки оновлення токену;

- перевірка доступу до захищених endpoint'ів у випадку відсутнього, протермінованого або некоректного токена.

Для реалізації вказаних сценаріїв застосовується бібліотека REST-assured, яка дозволяє гнучко формулювати HTTP-запити та перевіряти відповіді, зокрема:

- статус-коди (200 OK, 401 Unauthorized, 403 Forbidden);

- наявність полів у JSON-відповіді;

- час дії токену;

- заголовки автентифікації (Authorization: Bearer <token>).

Усі сценарії тести реалізуються у відповідності до data-driven парадигми: тестові вхідні дані (наприклад, login/password, очікувані токени, тощо) зберігаються у JSON-файлах або базі даних, що дозволяє запускати тести з численними комбінаціями параметрів.

До архітектурних особливостей модуля AuthTestModule відноситься те, що він реалізован як окремий Java-компонент зі стандартом @Component, що забезпечує інтеграцію з іншими модулями через Spring Dependency Injection (DI). Модуль є ізольованим, з можливістю виконання в окремому контейнері (Docker). Він використовує JUnit 5 для керування тестовим життєвим циклом. Також, він підтримує setup/teardown логіку для ініціалізації користувачів перед тестуванням та очищення середовища після завершення. Його вихідні звіти інтегруються у Allure.

Модуль AuthTestModule є базовим у послідовності виконання CI/CD Pipeline, оскільки від його успішного проходження залежать подальші сценарії, зокрема створення замовлень, оплат, доступ до адміністративної панелі, тощо.

Модуль CrudTestModule також реалізує тестування базової функціональності REST API відповідно до принципів CRUD (Create, Read, Update, Delete). Його основна роль полягає у перевірці коректної роботи HTTP-операцій POST, GET, PUT, DELETE для окремих ресурсів, що є фундаментальними для будь-якої мікросервісної платформи. Зокрема, CRUD-тести охоплюють такі типові сутності як: користувачі, товари, замовлення, адреси доставки тощо.

Функціональне призначення модуля CrudTestModule (табл. 3.1) полягає в тому, щоб перевірити:

- створення ресурсу (POST) – перевірка коду 201, вмісту відповіді, генерації id;

- читання ресурсу (GET) – перевірка статус-коду 200, відповідності вмісту до очікуваного об’єкта;

- оновлення ресурсу (PUT) – перевірка збереження змін, ідемпотентність;

- видалення ресурсу (DELETE) – перевірка коректного статусу (204), подальшої недоступності ресурсу.

Кожен тест в ньому виконується у data-driven підході: набір параметрів для тестування (ідентифікатори, поля об'єктів, граничні значення) зберігається у JSON/CSV-джерелах і передається до тестових методів. Додатково в цьому модулі реалізовано:

- ідемпотентність PUT, де повторні запити з однаковими даними не повинні змінювати стан системи;

- пов’язані ресурси, де існує підтримка вкладених структур, наприклад, створення адреси для користувача;

- масові перевірки, де тестування колекцій ресурсів та фільтрації виконується за параметрами (GET /orders?status=pending).

До архітектурних особливостей цього модуля відноситься те, що він є ізольованим компонентом системи, що реалізує інтерфейс ITestModule. Кожна CRUD-операція реалізована у вигляді окремого підмодуля (CreateResourceTest, ReadResourceTest, тощо). В ньому впроваджена централізована система перевірки відповідей (ResponseAsserter), яка використовується повторно у всіх запитах. Вхідні дані формуються за допомогою популяторів (DataPopulator), що зчитують шаблони об'єктів і генерують валідні JSON-тела запитів. Всі запити виконуються з використанням REST-assured та інтегруються у Jenkins stages.

CrudTestModule виступає як модуль, що забезпечує перевірку основного функціонального шару REST API, на якому будуються бізнес-сценарії. Він працює в тісному зв’язку з:

- TestDataGenerator з метою створення вхідних даних;

- ResponseValidationModule з метою перевірки відповідей;

- LoggingModule з метою фіксації запитів/відповідей.

Модуль також легко розширюється шляхом додавання нових сценаріїв CRUD без зміни основної логіки тестування (табл. 3.2), завдяки зовнішнім конфігураційним JSON-файлам.

Таблиця 3.2 – Порівняльна таблиця функціональних характеристик основних модулів програмного засобу

Назва модуля	Характер вхідних даних	Взаємодія з іншими модулями	Особливості реалізації
AuthTestModule	Data-driven (JSON, DB)	ResponseValidation, EnvConfig, Jenkins	Підтримка позитивних і негативних сценаріїв
CrudTestModule	Data-driven + Populator	ResponseValidation, TestDataGenerator	Ідемпотентність PUT, вкладені ресурси
IntegrationTest-Module	Сценарії з ланцюгами даних	FlowCoordinator, ResponseValidation	Chaining-запити, setup/teardown
ResponseValida-tionModule	JSON-schema + Response	Всі модулі	Soft assertions, JSON Schema Validation

Наступний модуль IntegrationTestModule призначений для перевірки міжсервісної взаємодії в межах мікросервісної платформи. Його ключовим завданням є симуляція комплексних бізнес-сценаріїв, які охоплюють кілька незалежних сервісів одночасно. Цей модуль забезпечує надійність і узгодженість реалізації багатокомпонентної логіки, яка реалізується через REST API. На відміну від CrudTestModule, що перевіряє одиничні ресурси, IntegrationTestModule охоплює повні сценарії: наприклад, створення замовлення → генерація рахунку → підтвердження платежу.

Типовий набір інтеграційних сценаріїв цього модуля включає:

- створення нового замовлення (Order Service) з подальшою передачею ID у Payment Service;

- генерація рахунку з прив’язкою до замовлення;

- підтвердження платежу та оновлення статусу замовлення;

- валідація послідовної зміни станів об’єкта (pending → paid → shipped).

До особливостей реалізації цього модуля (табл. 3.2) відноситься:

- Chaining-запити, де результат одного запиту передається як параметр до наступного (наприклад, ID замовлення);

- Setup/teardown логіка, де середовище підготовлюється до тесту (ініціалізація тестових даних), а після завершення – очищується для забезпечення повторного виконання;

- контроль транзакційності, де виконується перевірка того, що відмова одного сервісу не порушує логіку іншого (наприклад, недоступність Payment Service не повинна створювати замовлення зі статусом "paid").

В цьому модулі реалізовані ряд програмних компонентів. Так, одни з компонентів реалізовано як ізольований клас IntegrationTestModuleImpl, який агрегує окремі бізнес-сценарії (наприклад, OrderPaymentFlowTest, UserProfileSyncTest). Для побудови сценаріїв використовується механізм потоків сценаріїв (ScenarioExecutor, FlowCoordinator), що дозволяє гнучко комбінувати послідовності запитів. При цьому, REST-assured запити використовується з вбудованою підтримкою передачі параметрів між викликами (extract().path() → given().pathParam()).

Потім, результати агрегуються у форматі, сумісному з Allure, з маркуванням за типом сценарію, сервісами, важливістю.

Таким чином, розглянутий модуль IntegrationTestModule є ключовим у валідації взаємодії між мікросервісами. Він запускається після проходження модулів AuthTestModule та CrudTestModule, оскільки для перевірки інтеграції потрібна наявність базових даних та авторизованого доступу.

Його тісна інтеграція реалізована з (табл. 3.2):

- TestDataGenerator для ініціалізації даних на початку тесту;

- EnvConfigModule для отримання URL-адрес сервісів;

- LoggingModule для відстеження ланцюгів запитів.

Модуль ResponseValidationModule виконує функцію глибокої перевірки відповідей, що надходять від REST API сервісів у рамках виконання тестових сценаріїв. Він реалізує механізм валідації JSON-відповідей як за структурою, так і за змістом, забезпечуючи відповідність до заздалегідь визначених схем. Особливу увагу в роботі цей модуль приділяє таким аспектам, як відповідність типів даних, обов'язковість ключів, формат дат, правильність повідомлень про помилки, а також коректність використаних HTTP статус-кодів. При цьому, модуль ResponseValidationModule дозволяє виконувати:

- JSON Schema Validation, де є перевірка відповідей на відповідність структурі, описаній у схемі (.json);

- перевірку обов'язкових полів, де є наявність таких ключів як id, email, createdAt, error, message.

- перевірку типів, де наприклад, є id - ціле число, email - рядок, createdAt - дата у форматі ISO 8601;

аналіз повідомлень про помилки, де є наявність ключових полів, таких як code, message, details;

- Soft assertions: можливість продовження тесту у разі нефатальних відхилень, із накопиченням помилок для подальшого аналізу.

Особливою функцією модуля є те, що він забезпечує автоматичну перевірку, звільняючи тестувальника від необхідності вручну аналізувати структуру кожної відповіді. Завдяки цьому досягається значне зменшення кількості хибнопозитивних або пропущених помилок у складних сценаріях. Ядро реалізовано у вигляді класу ResponseSchemaValidator, який підключається до кожного модуля (Auth, CRUD, Integration) через спільний інтерфейс IValidator. JSON Schema зберігаються у зовнішньому каталозі (/schemas) і підключаються динамічно відповідно до типу ресурсу. Для перевірки застосовується бібліотека JSON Schema Validator (наприклад, Everit, Justify або інтеграція з REST-assured schema support).

Система soft assertions реалізована через власний обгортковий клас SoftAssertionCollector, який дозволяє накопичувати всі порушення та виводити їх в Allure-звіт без зупинки тесту. Вона підтримується механізм профілювання схем

для різних середовищ (dev, staging, prod), що дозволяє адаптувати перевірку до поточного контексту CI/CD. Модуль ResponseValidationModule є транзитивною складовою всіх інших модулів. Він не є самостійним у сенсі тест-кейсів, але забезпечує:

- постійний контроль відповідей від REST API;
- уніфіковану перевірку якості JSON-відповідей у будь-якому сценарії;
- підтримку структурованої звітності через Allure.

Він взаємодіє з:

- CrudTestModule, де є перевірка тіл відповідей на GET/POST/PUT;
- IntegrationTestModule, де є аналіз відповідей під час сценаріїв;
- AuthTestModule, де є перевірка форматів токенів, статусів, помилок автентифікації.

Далі, розглянемо логічну структуру основних модулів системи та їх взаємодію з зовнішніми сервісами, інструментами та середовищем розгортання (CI/CD, Docker, Jenkins, API) за допомогою UML-діаграма компонентів (рис. 3.1).

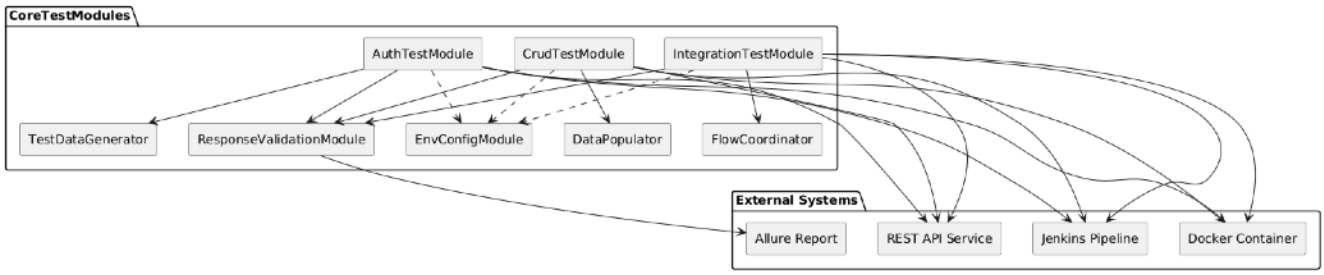

Рисунок 3.1 - UML-діаграма компонентів, що відображає взаємодію трьох незалежних модулів

UML-діаграма компонентів (рис. 3.1) відображає взаємодію трьох незалежних модулів таких як AuthTestModule, CrudTestModule, IntegrationTestModule, які мають прямий доступ до REST API сервісів, запускаються з Jenkins і працюють в ізольованих контейнерах Docker.

ResponseValidationModule виступає як спільний валідатор відповідей - усі модулі звертаються до нього для перевірки структур, форматів і кодів. TestDataGenerator, DataPopulator та FlowCoordinator мають допоміжні компоненти, які забезпечують підготовку даних і координацію інтеграційних сценаріїв.

EnvConfigModule постачає конфігураційні параметри, зокрема baseUrl, токени, профілі середовища. Надалі програмний код, що реалізує Allure Report інтегрується через ResponseValidationModule, що акумулює помилки та результати. На основі UML-діаграми компонентів (рис. 3.1) розроблена UML-діаграма класів (рис. 3.2).

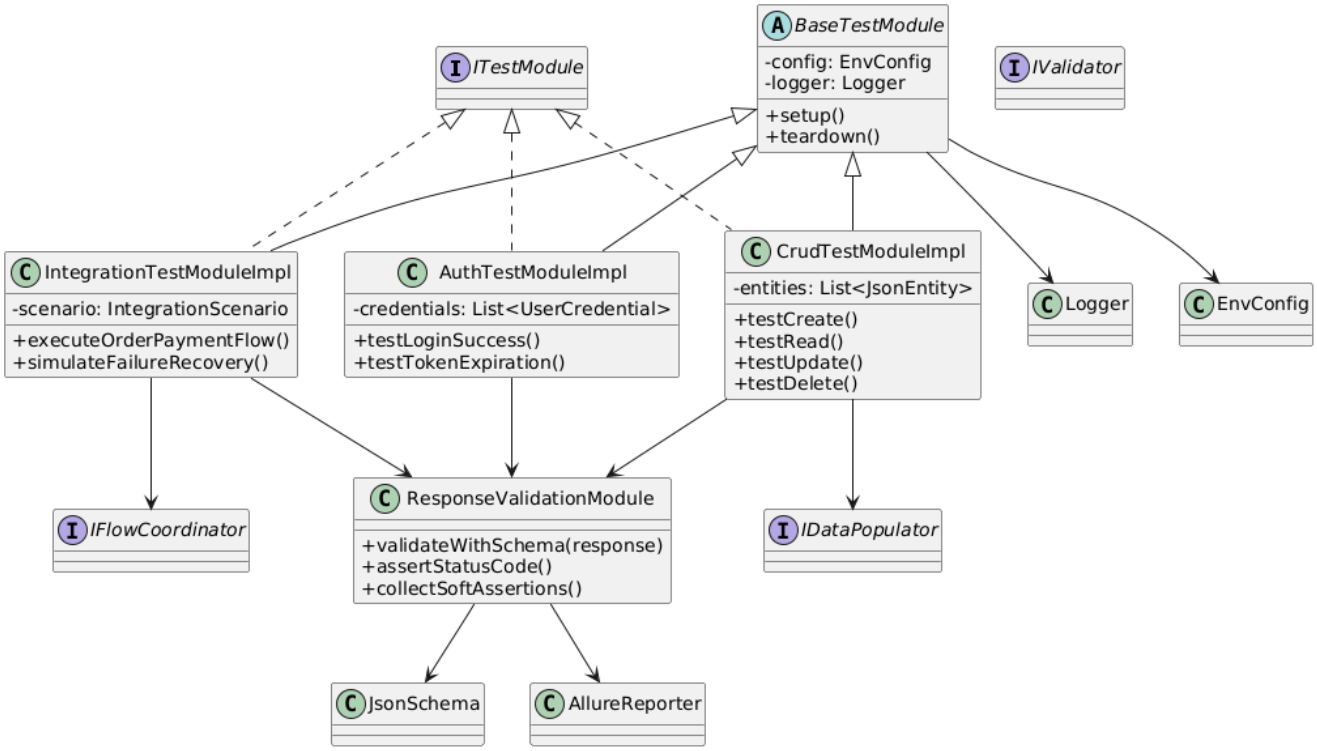

Рисунок 3.2 - UML-діаграма класів основних модулів програмного засобу

UML-діаграма класів (рис. 3.2) демонструє структуру програмних об’єктів Java, що реалізують основні модулі, їх інтерфейси, наслідування, зв’язки агрегації та залежності. Так, всі модулі реалізують загальний інтерфейс ITestModule і наслідують спільний базовий клас BaseTestModule, у якому визначено стандартні методи setup(), teardown() і службові компоненти.

Модуль ResponseValidationModule - універсальний валідатор, що агрегує відповіді REST API, працює із схемами JSON та інтегрується з Allure. Модулі CrudTestModuleImpl і IntegrationTestModuleImpl - залежні від зовнішніх інтерфейсів IDataPopulator та IFlowCoordinator, що дозволяє зберігати гнучкість та масштабованість. EnvConfig та Logger - ін’єковані залежності, що забезпечують конфігурацію та логування, визначені на рівні BaseTestModule.

Таким чином, в результаті реалізації основних модулів системи автоматизації тестування REST API було забезпечено високий рівень модульності, ізоляції та масштабованості, що є критично важливим у контексті мікросервісної архітектури та DevOps-практик. Кожен із чотирьох основних модулів виконує чітко окреслену функцію, спираючись на принципи SRP (Single Responsibility Principle), і підтримує data-driven або сценарно-орієнтоване тестування.

Модуль AuthTestModule дозволяє виявляти вразливості та помилки автентифікації. CrudTestModule забезпечує надійне тестування базових REST-операцій, критично важливих для стабільності сервісів. IntegrationTestModule забезпечує наскрізну перевірку сценаріїв із залученням кількох сервісів. А ResponseValidationModule уніфікує валідацію JSON-відповідей, формуючи підґрунтя для якісної звітності.

Також, на основі розробки UML-діаграми компонентів були визначені архітектурні рішення інтеграцію основних модулів із Jenkins, Docker, Allure та REST API, що дозволяє забезпечити їх прозору інтеграцію у CI/CD процес. UML-діаграма класів дозволила представити абстрактну ієрархію реалізації, що спрощує супровід і розширення системи.

Отже, розроблена структура основних модулів відповідає сучасним вимогам до автоматизованого тестування REST API у гетерогенному середовищі, зберігає гнучкість, розширюваність, перевикористовуваність та стійкість до змін.

3.2 Розробка додаткових модулів програмного засобу

В цьому підрозділі розглянемо детально додаткові модулі програмного засобу для автоматизації тестування REST API. Так, модуль TestDataGenerator є невід'ємною частиною підсистеми підготовки даних у процесі автоматизованого тестування REST API (табл. 3.3). Його основною функцією є створення динамічних, варіативних та параметризованих даних, які використовуються для формування запитів, перевірки валідації та тестування граничних умов. Цей модуль розвантажує тестувальників від ручного створення тестових даних та

забезпечує високу ступінь варіативності, що необхідна для повноцінного покриття сценаріїв.

TestDataGenerator реалізує такі типи генерації:

- стандартні валідні значення, де існують електронні адреси, номери телефонів, імена, адреси тощо;

- граничні значення, де існують нульові значення, максимальна довжина поля, мінімальні допустимі дані;

- негативні сценарії, де існують SQL-ін’єкції, порожні рядки, недопустимі символи, відсутні поля.

Для цього модуль використовує дві основні техніки:

- інтеграція з бібліотекою Faker, де існують генерація реалістичних даних (наприклад, faker.name().fullName() або faker.internet().emailAddress());

- кастомні генератори, де існують спеціалізовані шаблони, що відповідають специфіці системи (наприклад, генерація унікального коду замовлення формату ORD-YYYYMMDD-XXXX);

- генератор підтримує параметри запуску, що дозволяє адаптувати набір даних до поточного контексту тестування та змінювати кількість записів, типи полів, формат, валідність тощо.

TestDataGenerator реалізовано у вигляді окремого Java-компонента (табл. 3.4), який може бути повторно використаний у всіх основних модулях (CrudTestModule, IntegrationTestModule, AuthTestModule). Тому, в нього дані можуть зберігатись у:

- зовнішніх JSON/CSV-файлах;

- Excel-шаблонах;

- реляційних базах даних (MySQL, PostgreSQL);

- генератор може записувати дані у тимчасову таблицю або безпосередньо передавати у тіло HTTP-запиту.

В цьому модулі підтримується режим data-driven testing, коли структура даних задається шаблоном, а значення - генеруються або підставляються автоматично. Також, передбачена взаємодія з системами логування та звітності -

наприклад, кожна сесія генерації фіксується в логах, що дозволяє відтворити конкретний набір даних у разі збоїв.

Таблиця 3.3 - Функціональне призначення додаткових модулів

Назва модуля	Основне призначення	Взаємодія з модулями
TestDataGenerator	Генерація валідних, граничних і негативних тестових даних	Auth, CRUD, Integration
EnvConfigModule	Надання конфігурацій середовища для тестування	Усі основні та додаткові
LoggingModule	Запис запитів, відповідей, логування тестів і помилок	Усі основні, Report
ReportModule	Агрегація результатів тестів, генерація звітів Allure	Усі модулі, Jenkins

TestDataGenerator є критично важливим для сценаріїв, де необхідна швидка підготовка унікальних або випадкових даних. Він інтегрується (табл. 3.4) через DI (Dependency Injection) у CrudTestModule, IntegrationTestModule та частково - у AuthTestModule. Його застосування дозволяє:

- зменшити кількість дубльованих тестів;

- уникнути ручного втручання у наповнення;

- масштабувати тестування з мінімальними затратами на підтримку сценаріїв.

Другий модуль EnvConfigModule виконує функцію централізованого керування параметрами середовища виконання тестів. Він забезпечує динамічне зчитування, обробку та передачу таких конфігураційних даних, як базові URL-адреси REST API, порти сервісів, токени автентифікації, ідентифікатори середовищ (dev, test, staging, prod) тощо (табл. 3.3). Завдяки цьому модулю всі інші компоненти системи тестування можуть адаптуватися до конкретного середовища без зміни основної логіки тестів або переписування конфігурацій вручну. Основні функції EnvConfigModule включають зчитування параметрів з:

- .properties або .yaml файлів;

- змінних середовища (System.getenv());

- Jenkins Pipeline variables або Docker .env;

- розділення конфігурацій за профілями (application-dev.yaml, application-prod.yaml);

- взаємодію з механізмом fallback, де якщо значення не знайдено, підставляється дефолтне;

- перевірка валідності ключових параметрів на етапі ініціалізації.

Конфігураційні значення використовуються всіма основними модулями, включаючи:

- базову адресу API: baseUrl = https://api.test.internal/v1

- токени: authToken, refreshToken

- параметри підключення до сторонніх сервісів: URL логера, шлях до Allure

- контекст середовища, де існують флаг дебагу, параметри контейнеризації, інтервали таймаутів

Даний модуль реалізовано як Spring-компонент (@Component) з використанням @ConfigurationProperties або @Value для ін'єкції параметрів. Він реалізується окремий конфігураційний клас EnvConfig, який інкапсулює всі ключові значення з відповідними геттерами. Для зручності діагностики реалізовано метод logCurrentConfiguration() з виводом конфігурації у лог у форматі JSON (за виключенням чутливих даних). Він також підтримує можливість динамічного перемикання середовищ через параметри запуску Jenkins: --env=test, --profile=qa.

Тому, модуль EnvConfigModule є критичною складовою всієї CI/CD-інфраструктури автоматизованого тестування. Він забезпечує ізольованість середовищ, уникнення хардкодингу і конфігураційної плутанини. Всі інші модулі (AuthTestModule, CrudTestModule, IntegrationTestModule, TestDataGenerator) використовують його через DI для отримання актуальних параметрів середовища.

Таблиця 3.2 – Порівняльна таблиця функціональних характеристик основних модулів програмного засобу

Назва модуля	Джерела даних / параметрів	Особливості реалізації	Інтеграція з CI/CD

TestDataGenerator	JSON, CSV, DB, кастомні шаблони	Faker, кастомні генератори, параметризація	Так
EnvConfigModule	YAML, .env, Jenkins variables	Профілі (dev, test, prod), fallback, DI	Так
LoggingModule	Консоль, файл, JSON, ELK	Динамічний рівень логування, окремі логи на модуль	Частково
ReportModule	Allure, XML/JSON з тестів	Історія запусків, групування, артефакти, CLI або Docker	Так

Інший модуль LoggingModule забезпечує централізоване ведення журналу подій у процесі виконання автоматизованих тестів REST API. Його основною метою є фіксація критичних аспектів взаємодії з API, зокрема HTTP-запитів, відповідей, часу виконання, статусів тестів та внутрішніх повідомлень про помилки. Журнали є важливим джерелом інформації для діагностики проблем, аналізу збоїв і покращення якості системи тестування.

До основних завдань модуля LoggingModule належать наступні:

- фіксація кожного HTTP-запиту, де використовується метод (POST, GET тощо), URL, заголовки, тіло запиту;

- фіксація відповіді, де використовується статус-код, тіло, заголовки, час відповіді;

реєстрація статусу тесту, де виконується дії, що визначають успішне виконання, помилка, пропущено;

- агрегація статистики, що визначають середній час відповіді, кількість запитів, частота помилок;

- динамічне керування рівнем логування, що визначають перемикання між INFO, DEBUG, ERROR залежно від параметрів середовища (log.level=debug).

Цей модуль використовує такі формати виводу:

- Plain text для локального аналізу;

- JSON logs для автоматичної обробки.

Даний модуль реалізовано як окремий сервісний клас LoggingService, який інкапсулює логіку форматування, фільтрації та виводу повідомлень.

Для гнучкості налаштувань використовується Spring Boot Logger (logback.xml, application.yaml) із профілюванням. Також, підтримується розділення логів за таким контекстом:

- AuthTest.log;
- CrudTest.log;
- Integration.log.

Інтерфейсна частина реалізована у вигляді ILogger, що забезпечує уніфікацію для основних модулів та можливість підключення альтернативних реалізацій (наприклад, MockLogger для юніт-тестів). Логи цього модуля також можна автоматично прикріплювати до Allure-звітів як додаткові артефакти (файли або розділи).

Таким чином, модуль LoggingModule активно використовується всіма основними модулями (AuthTestModule, CrudTestModule, IntegrationTestModule, ResponseValidationModule) для фіксації дій під час тестування. Він забезпечує прозорість процесу, відтворюваність сценаріїв і контроль над нестабільними компонентами. Його використання критичне для DevOps-аналітики, ретроспективного аналізу та налаштування тригерів оповіщень у випадку серійних помилок.

Останній щодо розгляду модуль ReportModule відповідає за автоматизовану генерацію, агрегування та візуалізацію результатів тестування REST API в системі CI/CD. Його впровадження дозволяє формувати структуровані звіти, придатні як для оперативного аналізу з боку команди розробки, так і для довготривалого зберігання тестових артефактів у репозиторіях звітності. Основним інструментом реалізації є інтеграція з Allure Framework.

До основних функцій модуля належать наступні:

- формування звітів після проходження тестів: збір результатів з усіх основних модулів (Auth, CRUD, Integration).

- групування тестів за категоріями, що виконуються за мікросервісом (UserService, OrderService, PaymentService), за важливістю (critical, high, medium, low), за типом (functional, integration, regression);

- відображення детальної інформації про кожен тест, де виконуються опис сценарію, вхідні/вихідні параметри, статус, лог.

Прикріплення додаткових матеріалів в цьому модулі виконуються за допомогою:

- HTTP-запитів та відповіді, скріншоти (за потреби), час виконання, логи з LoggingModule.

- автоматичного запуску Allure-репортера після завершення всіх stages у Jenkins.

При цьому, підтримується історія запусків - відстеження стабільності, зміни статусів, виявлення нестабільних тестів (flaky tests), генерація графіків та метрик.

Цей модуль побудований за допомогою спеціалізованого класу ReportAggregator, який виконує збір результатів у форматі .xml або .json та перетворює їх у HTML-звіти за допомогою Allure CLI або Docker-контейнера.

Параметри конфігурації в цьому модулі зчитуються через EnvConfigModule за допомогою:

- шляху до каталогу результатів;

- моваи звіту;

- політики збереження історії;

Підтримка Jenkins інтеграції виконується внаслідок:

- автоматичного копіювання результатів у target/allure-results;

- запуску команди allure generate && allure open або створення артефакту в pipeline;

- збереження посилання на звіт у підсумках виконання.

Модуль ReportModule інтегрується із усіма основними модулями тестування (табл. 3.4), а також з LoggingModule, ResponseValidationModule та EnvConfigModule. Він формує завершальну ланку в ланцюжку "виконання тесту → фіксація результату → вивід звіту", забезпечуючи прозору та наочну інтерпретацію тестових даних. Його впровадження дозволяє:

- проводити автоматизовану аналітику стабільності сервісів;

- надавати замовникам або керівництву стандартизовані звіти;

- швидко виявляти невідповідності, тенденції деградації або нестабільні сценарії.

Таким чином, розглянуті додаткові модулі програмного засобу є вкрай важливими для його функціонування. Так, модуль TestDataGenerator є гнучким інструментом створення даних, критично важливим для сценаріїв із високою варіативністю вхідних параметрів. EnvConfigModule забезпечує централізованість конфігурацій і дозволяє запускати тести в різних середовищах без змін у коді. LoggingModule підвищує прозорість і відтворюваність результатів, забезпечує критичну підтримку у випадках нестабільної поведінки сервісів. ReportModule завершує цикл тестування, дозволяючи консолідувати всі результати у зручному форматі для аналітики та звітності.

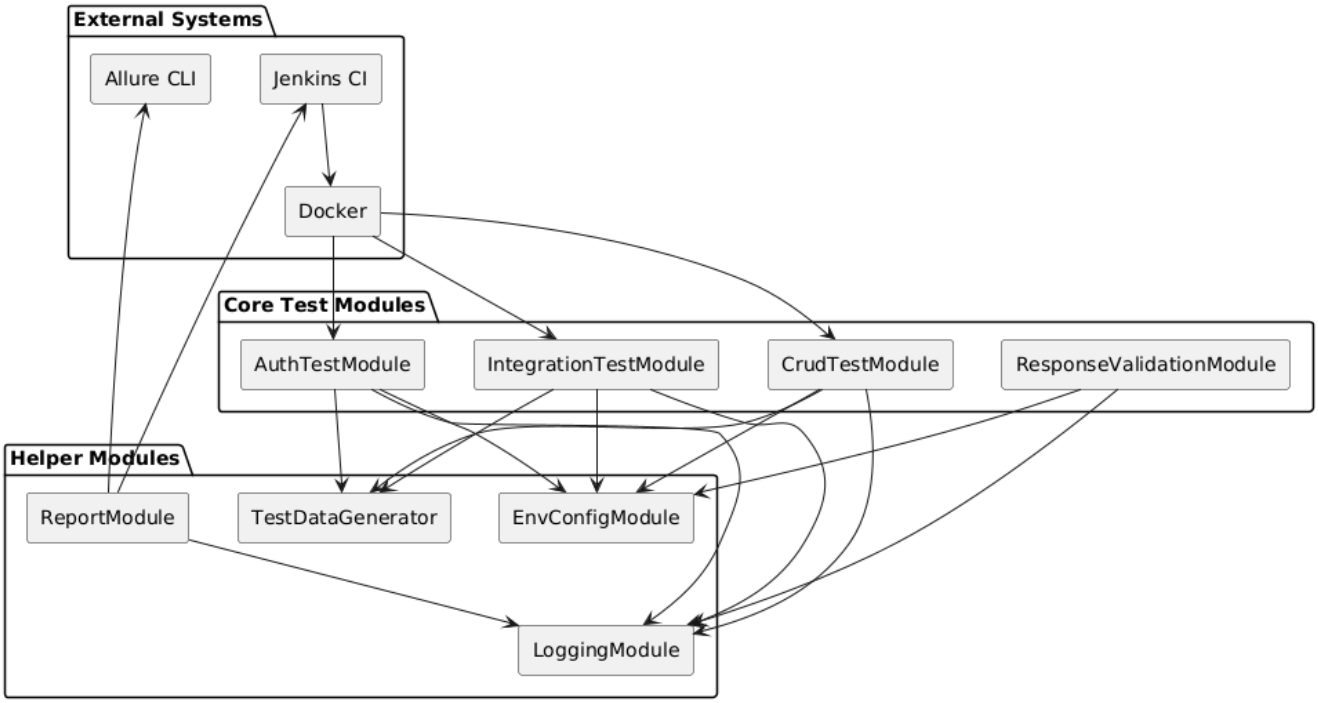

Рисунок 3.3 - UML-діаграму компонентів додаткових модулів

Далі розглянемо UML-діаграму компонентів цих додаткових модулів (рис. 3.3), що покликана візуалізувати архітектурну організацію допоміжних модулів (TestDataGenerator, EnvConfigModule, LoggingModule, ReportModule), їхню взаємодію між собою, з основними модулями та з зовнішніми CI/CD-компонентами (Jenkins, Docker, Allure).

На рис. 3.3 модуль TestDataGenerator постачає згенеровані дані модулям Auth, CRUD та Integration, забезпечуючи варіативність вхідних параметрів. Модуль EnvConfigModule є універсальним джерелом конфігурацій - його використовують усі модулі, включно з валідацією. До модуля LoggingModule інтегровано в усі основні модулі для збору та обробки журналів, а також з ReportModule, який аналізує логи у фінальній звітності. Модуль ReportModule формує підсумкові звіти на основі даних з усіх тестів та логів, генерує їх за допомогою Allure, а запускається автоматично через Jenkins після завершення тестування. При цьому, такі праграмні засоби як Jenkins та Docker забезпечують інфраструктурну підтримку CI/CD та ізоляцію компонентів.

На основі UML-діаграми компонентів додаткових модулів (рис. 3.3) формується UML-діаграма класів додаткових модулів (рис. 3.4), яка відображає основні класи, інтерфейси, методи та залежності, які беруть участь у реалізації модулів TestDataGenerator, EnvConfigModule, LoggingModule і ReportModule. Така схема забезпечує високий рівень абстракції й дозволяє проаналізувати розширюваність, повторне використання та слабке зв’язування компонентів.

На рис. 3.4 інтерфейси IDataGenerator, ILogger, IReportGenerator, IConfigProvider абстрагують реалізацію від логіки використання, забезпечуючи гнучке тестування та масштабування. TestDataGeneratorImpl - конкретна реалізація генератора, що надає валідні, граничні та негативні набори даних. Працює у зв’язці з EnvConfig, щоб адаптувати генерацію до середовища. EnvConfig інкапсулює логіку читання параметрів середовища, надає API для доступу до значень, забезпечує fallback та логування конфігурації. LoggingService реалізує основні рівні логування та дозволяє прикріплювати логи до звітів. AllureReportModule генерує звіти, агрегує артефакти, формує зведення за мікросервісами, інтегрується з Jenkins і Docker. Наступні JenkinsContext та DockerEnvironment відображають зовнішні залежності інфраструктурного характеру, які враховуються під час створення та публікації звітів.

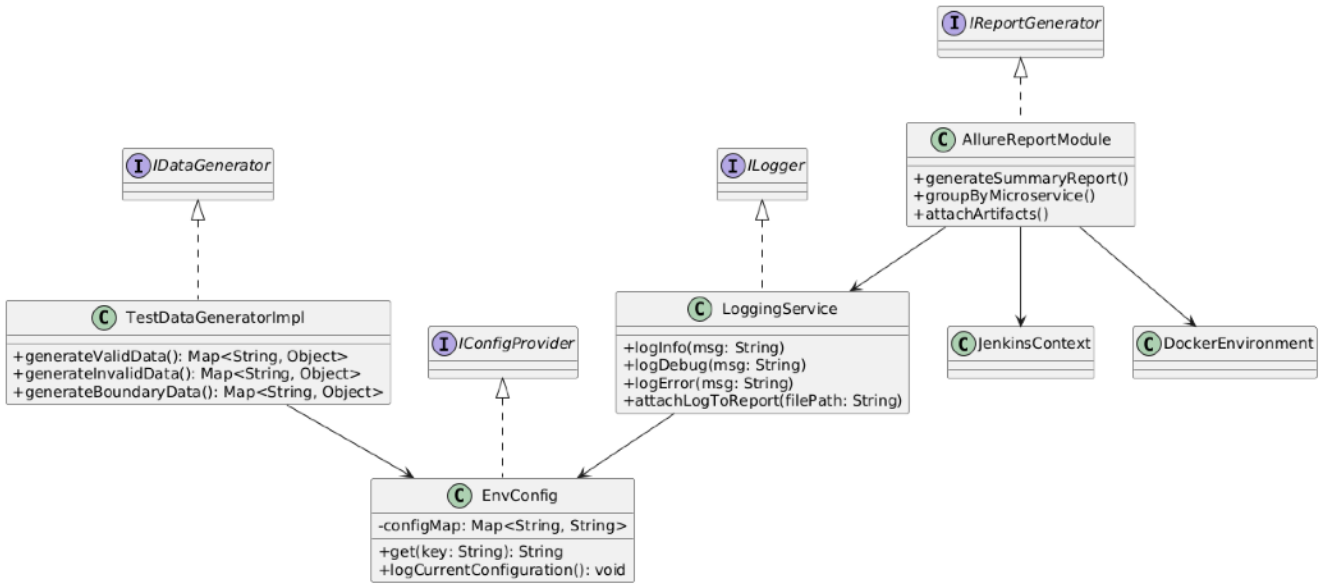

Рисунок 3.4 - UML-діаграму класів додаткових модулів

Таким чином, в ході реалізації підсистеми додаткових модулів було створено набір функціонально важливих компонентів, які виконують підтримуючу, але критично необхідну роль в архітектурі системи автоматизованого тестування REST API. На відміну від основних модулів, що відповідають безпосередньо за виконання тестів, додаткові модулі забезпечують стійкість, масштабованість, гнучкість, параметризованість та контрольованість тестового середовища. Так, модуль TestDataGenerator дозволяє автоматично генерувати валідні, негативні та граничні дані з використанням шаблонів і зовнішніх джерел, що значно знижує витрати часу на підготовку вхідних параметрів. EnvConfigModule забезпечує динамічне конфігурування усіх модулів залежно від обраного середовища (dev, test, prod), з використанням централізованих YAML-файлів, .env та Jenkins-параметрів.

Модуль LoggingModule реалізує детальний контроль за виконанням тестів, підтримує різні рівні логування та інтеграцію з системами централізованого зберігання логів, що дозволяє відслідковувати аномалії та аналізувати нестабільну поведінку. Нарешті, ReportModule виконує підсумкову агрегацію результатів, їх групування, генерацію звітів у форматі Allure та інтеграцію з CI/CD пайплайнами.

Впровадження додаткових модулів до програмного засобу надало змогу:

- забезпечити гнучке масштабування тестів без втрати керованості;

- уніфікувати логіку налаштувань і збору даних, незалежно від конкретного модуля;

- значно підвищити якість супроводу та аналізу результатів;

- успішно інтегрувати архітектуру з практиками DevOps та CI/CD.

Отже, підсистема додаткових модулів не лише підтримує основні тестові процеси, а й створює основу для їх надійного функціонування, адаптивності та розширюваності, що повністю відповідає вимогам до сучасних програмних засобів автоматизованого тестування REST API у складних розподілених мікросервісних середовищах.

4 ТЕСТУВАННЯ МОДУЛІВ ПРОГРАМНОГО ДОДАТКУ

4.1 Роль модульного підходу в архітектурі тестування REST API

У сучасних системах, що реалізують архітектурний підхід мікросервісів, забезпечення стабільної, масштабованої та перевіреної взаємодії між сервісами стає однією з ключових вимог у процесі життєвого циклу програмного забезпечення. З огляду на це, автоматизоване тестування REST API виступає не лише як засіб контролю якості, але і як невід'ємний елемент DevOps-культури, що забезпечує неперервність постачання змін та їх стабільність.

В основі побудови системи тестування REST API лежить модульний принцип, відповідно до якого кожен логічний компонент виконує чітко визначену функцію і взаємодіє з іншими компонентами через стандартизовані інтерфейси. Такий підхід забезпечує:

- відокремлення відповідальності (Separation of Concerns), що спрощує підтримку та масштабування системи;

- гнучкість у заміні або доповненні окремих модулів без порушення загальної архітектури;

- перевикористання коду, логіки та шаблонів у різних сценаріях тестування;

- сумісність з CI/CD пайплайнами, де кожен модуль може бути протестований або виконаний незалежно.

Модульна організація також відповідає принципам чистої архітектури, де зовнішні залежності (зокрема, REST-запити, конфігурація, тестові дані) інкапсулюються в окремі шари. Це дозволяє зосередити увагу на логіці перевірки поведінки системи, а не на деталях реалізації інфраструктури.

Таким чином, у побудованій системі автоматизації тестування REST API передбачено набір основних модулів, що відповідають за безпосереднє виконання тестів (автентифікація, CRUD, інтеграція, валідація відповідей), а також додаткових модулів, що забезпечують генерацію вхідних даних, конфігурування, логування та формування звітності. Такий підхід дозволяє охопити повний цикл тестування - від підготовки до результатів - у структурований, повторюваний і масштабований спосіб.

Далі, розглянемо особливості тестування основних модулів. AuthTestModule – модуль тестування автентифікації, авторизації та токенів. Його функціонування системи доступу до REST API суттєво залежить від коректної реалізації механізмів автентифікації та авторизації. У межах розробленої архітектури відповідальність за ці перевірки покладено на окремий модуль AuthTestModule. Його реалізація ґрунтується на сучасному стандарті маркерної автентифікації, зокрема використанні JWT-токенів (JSON Web Token), які забезпечують захищений спосіб передачі ідентифікаційної інформації. Цей модуль виконує перевірку формування access token після успішного входу; валідації підпису та терміну дії токена; поведінки системи при спробі входу з неправильними даними або протермінованими токенами. Таким чином, AuthTestModule забезпечує перевірку першого критичного етапу доступу користувача до ресурсів системи, формуючи фундамент для подальших тестів.

Модуль CrudTestModule призначений для перевірки операцій CRUD. Він є логічним ядром тестування функціональності REST API з точки зору роботи з ресурсами. Згідно з REST-архітектурою, основні дії над сутностями виконуються за допомогою методів HTTP-протоколу - POST, GET, PUT, DELETE, які відповідають операціям створення, читання, оновлення та видалення.

У межах цього модуля реалізуються тести, що перевіряють ідемпотентність операцій (PUT, DELETE); коректність коду статусу у відповіді (201, 200, 404, 204); відповідність вмісту відповідей очікуваним JSON-структурам; поведінку при роботі з неіснуючими ідентифікаторами або неправильними форматами запитів.
Теоретичною основою є модель CRUD як абстракція для перевірки базових бізнес-операцій у мікросервісах, зокрема відповідність контракту API та контроль стабільності реалізації в умовах паралельних запитів.

Модуль IntegrationTestModule забезпечує інтеграційне тестування бізнес-сценаріїв. Тестування інтеграційних сценаріїв - це перевірка логічних послідовностей дій, які охоплюють кілька незалежних сервісів або мікросервісів. Модуль IntegrationTestModule дозволяє моделювати реальні ланцюжки дій, що відповідають бізнес-логіці (наприклад: оформлення замовлення → оплата → відвантаження).

Основними теоретичними аспектами є перевірка цілісності транзакційних сценаріїв у розподіленій системі; підтримка chain-of-requests, коли результат одного запиту впливає на наступний; тестування співіснування сервісів, а також handling-failure сценаріїв, коли один із сервісів недоступний.

Інтеграційне тестування базується на перевірці узгодженості стану між мікросервісами, що має ключове значення в умовах реального навантаження та асинхронної взаємодії.

Модуль ResponseValidationModule призначений для автоматичної валідації відповідей. Він реалізує автоматизовану перевірку відповідності відповідей API до очікуваних форматів і структур. Його завдання полягає не тільки у фіксації статусу (200 OK), а й у глибокому аналізі тіла відповіді, що може містити складні JSON-об’єкти.

Цей модуль базується на концепції валидації за JSON Schema, де для кожного типу ресурсу визначається очікувана структура, типи даних, обов’язкові поля та допустимі значення. До його основних особливостей належить гнучке керування перевірками через external schema-файли; підтримка soft-assertions - накопичення помилок без зупинки виконання тесту; інтеграція з системами звітності та аналізу відповідей.

Таким чином, ResponseValidationModule є універсальним елементом контролю якості, який забезпечує відповідність API специфікації та швидко виявляє порушення контрактів.

Додатковий модуль TestDataGenerator забезпечує генерацію тестових даних. При цьому, ефективне тестування REST API неможливе без забезпечення контрольованих, варіативних та релевантних вхідних даних. Саме цю роль виконує модуль TestDataGenerator, який автоматично створює набори тестових даних для різних сценаріїв. Згідно з теорією програмного тестування, тестові дані повинні охоплювати валідні значення, граничні умови, некоректні або спотворені вхідні значення. Цей модуль реалізує генерацію даних із використанням шаблонів та псевдовипадкових значень; створення масивів користувачів, ресурсів, ідентифікаторів із заданою кількістю; можливість централізованого використання в різних модулях (Auth, CRUD, Integration).

Його функціональність базується на принципах data-driven testing - підходу, за якого логіка тесту відокремлена від вхідних даних, що забезпечує масштабованість і повторне використання.

Модуль EnvConfigModule виконує підтримку конфігурації середовища. Так, у CI/CD-процесах критично важливо, щоб тести могли адаптуватися до різних середовищ без модифікації коду. Тому, модуль EnvConfigModule реалізує централізоване конфігурування параметрів середовища: URL-адрес, токенів, профілів (dev, test, prod), параметрів таймінгу тощо. Під час його роботи, теоретичною основою залишається принцип externalized configuration, що викладений у методології 12-Factor App, де вся конфігурація має бути винесена за межі застосунку. Це дозволяє мінімізувати ризик помилок у конфігурації; реалізувати профілі середовищ (application-test.yml, application-prod.yml); динамічно змінювати параметри при запуску в Jenkins або Docker. Таким чином, EnvConfigModule забезпечує гнучкість, адаптивність і безпечність запуску тестів у різних середовищах.

Модуль логування LoggingModule призначений для визначення подій, де тестування є фундаментом для відтворюваності, аналізу помилок та аудиту. Модуль LoggingModule реєструє кожну дію, що відбувається під час тестування, включно з HTTP-запитами й відповідями; статусами тестів (успішні, провальні, пропущені); технічними метаданими (час виконання, розмір відповіді, заголовки).

Цей модуль реалізує принципи observability, що включає централізоване логування, трасування та моніторинг. Його реалізація підтримує розділення логів за модулями (Auth.log, CRUD.log); динамічне перемикання рівня логування (DEBUG, INFO, ERROR); інтеграцію з ELK Stack або іншими системами (Graylog, Grafana Loki). Таким чином, LoggingModule відіграє ключову роль у відстеженні та інтерпретації поведінки системи під час тестування.

У сучасних DevOps-процесах результат тестування має бути не лише отриманий, а й представлений у зручному форматі для аналізу, зберігання та інтеграції. Тому, модуль ReportModule відповідає за формування структурованих звітів у форматі Allure, які містять результати тестів із групуванням за категоріями; часові показники, статуси, скріншоти та логи; історію запусків та

порівняльні графіки. Цей модуль реалізує принципи traceable test execution, де кожен результат пов'язується з конкретним кейсом, середовищем і версією. Його роль у тестуванні - це замикання циклу: від генерації даних і виконання тесту до формування візуалізованого результату.

Цей модуль дозволяє автоматично запускати Allure CLI у пайплайні Jenkins; інтегрувати результати з командною документацією; виявляти flaky-тести й деградацію функціональності на основі історичних графіків.

Таким чином, в роботі було здійснено теоретичне обґрунтування структури, функціональності та архітектурної доцільності модулів, що становлять основу системи автоматизованого тестування REST API. З урахуванням принципів модульності, повторного використання та гнучкої конфігурації середовища було видокремлено два рівні компонентів: основні (відповідають за виконання тестів) та додаткові (забезпечують підтримку й розширюваність функціоналу). Така побудова системи на основі чітко ізольованих модулів дозволяє реалізувати підтримку практик DevOps, CI/CD та забезпечити гнучкість при масштабуванні, переналаштуванні або впровадженні нових вимог. Розроблена модульна архітектура, що закладена у програмній системі, є обґрунтованою, практично реалізованою та повністю відповідає актуальним вимогам до програмних засобів автоматизованого тестування в середовищі розподілених сервісів та безперервної інтеграції.

4.2 Тестування контрольного прикладу для основного модуля AuthTestModule

Метою проведення контрольного прикладу є практична перевірка працездатності та функціональної повноти реалізованого модуля AuthTestModule в умовах імітації типової взаємодії користувача з REST API сервісом автентифікації. Тестування охоплює ключові операції, необхідні для реалізації базового циклу автентифікації: реєстрацію користувача, авторизацію з отриманням access token, а також перевірку дійсності отриманого токена.

Основними завданнями контрольного прикладу є:

- перевірка працездатності UI-форми авторизації та реєстрації;

- перевірка успішної взаємодії між frontend-інтерфейсом та backend-сервісом через HTTP-запити;

- фіксація та візуалізація результатів запитів (успішних та з помилками);

- оцінка реакції системи на негативні сценарії (невірний пароль, недійсний токен);

- забезпечення відтворюваності та ізольованості сценарію для аналізу результатів.

Контрольний приклад виконано у тестовому середовищі, що має клієнтський вебінтерфейс та сервера авторизації. Умови тестування відповідають типовій конфігурації сучасної мікросервісної системи із підтримкою REST API і CI/CD.

Середовище виконання процесу тестування мало наступне оточення:

- Операційна система: Ubuntu 22.04 LTS (мінімалістичне середовище, емуляція браузера);

- JDK: OpenJDK 17;

- Серверна частина: Spring Boot 3.1.0;

- База даних: MySQL 8.0 (локальна інстанція);

- JWT-бібліотека: jjwt (io.jsonwebtoken);

- Механізм кодування паролів: BCrypt (PasswordEncoder);

- Фронтенд-інтерфейс: HTML + CSS + JavaScript;

- Локальний доступ: index.html відкривається у браузері;

- Тестова адреса API: http://localhost:8080/api/auth/...;

- Інструменти тестування: Локальний запуск Spring Boot застосунку (mvn spring-boot:run);

- Вебінтерфейс запущено у Google Chrome v124.

Розглянемо кроки тестування.

Крок 1. Запуск інтерфейсу користувача (рис. 4.1).

Перший крок передбачає відкриття HTML-інтерфейсу модуля авторизації у локальному браузері. У цьому інтерфейсі відображаються три функціональні секції:

- форма авторизації, що містить поля для логіну й паролю та кнопку «Увійти»;

- форма реєстрації, яка дозволяє створити нового користувача;

- блок перевірки токена, призначений для перевірки дійсності access token.

Інтерфейс реалізовано українською мовою, з розміщенням елементів за принципом вертикального вирівнювання, що забезпечує зручність використання.

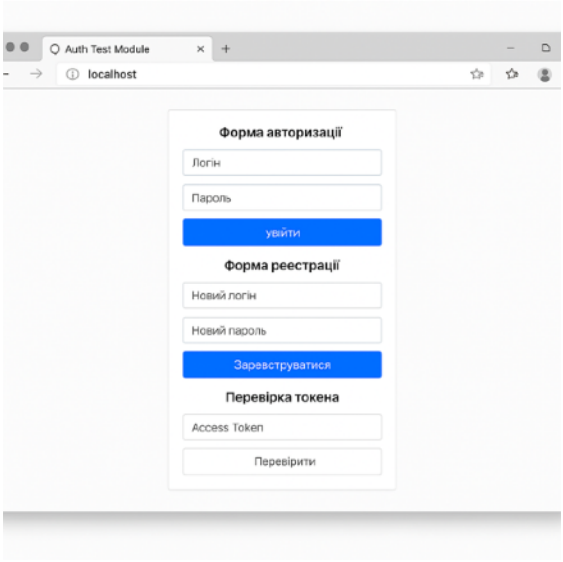

Рисунок 4.1 - Загальний інтерфейс модуля авторизації (AuthTestModule)

Крок 2. Введення даних для реєстрації (рис. 4.2).

На другому етапі здійснюється заповнення полів у формі реєстрації нового користувача. У відповідні поля введено:

Ім’я користувача: newuser

Пароль: довільне значення, яке відповідає правилам автентифікації

Користувач готовий натиснути кнопку «Зареєструватися» для ініціації запиту POST /api/auth/register.

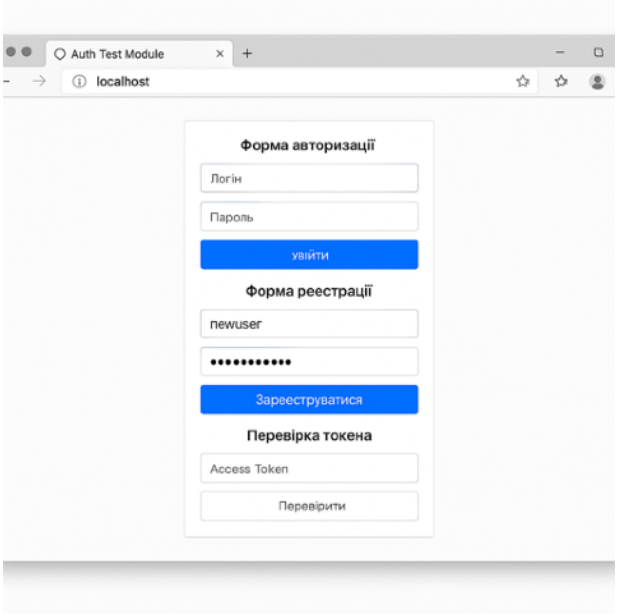

Рисунок 4.2 – Введення даних у форму реєстрації

Крок 3. Виконання реєстрації та аналіз відповіді (рис. 4.3).

На цьому етапі користувач натискає кнопку «Зареєструватися», після чого у фоновому режимі виконується HTTP POST-запит до маршруту /api/auth/register. Переданий JSON-об'єкт містить логін і пароль нового користувача. На серверній стороні облікові дані зберігаються в базі даних, а у відповідь повертається повідомлення про успішну реєстрацію.

У разі правильного виконання запиту в полі результатів інтерфейсу з’являється підтвердження:

```
{
  "message": "Користувача зареєстровано"
}
```

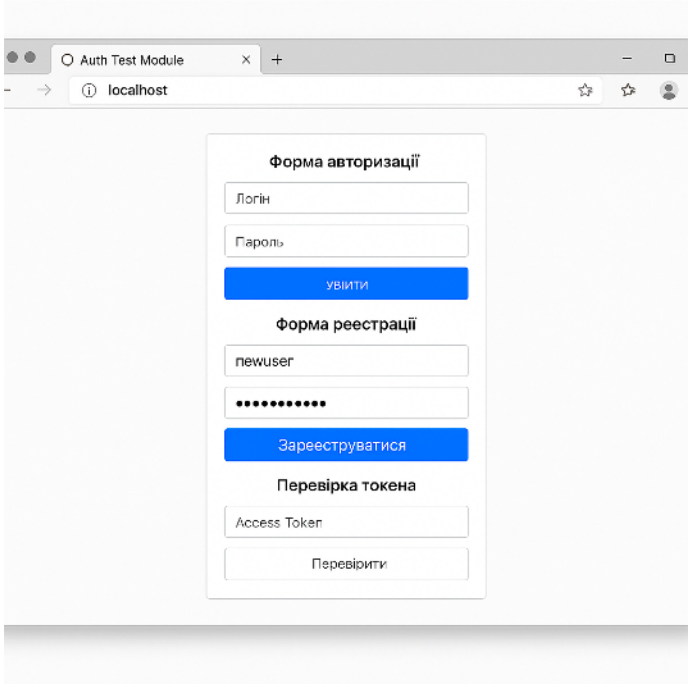

Рисунок 4.3 – Результат реєстрації нового користувача

Крок 4. Вхід до системи (рис. 4.4).

Після успішної реєстрації нового користувача система переходить до перевірки автентифікації через введення облікових даних у форму входу. У відповідні поля введено:

Логін: newuser

Пароль: значення, яке відповідало тому, що було задано під час реєстрації.

Після натискання кнопки «Увійти», відбувається HTTP POST-запит до маршруту /api/auth/login. Якщо дані правильні, у відповіді повертається access token, який буде використано на наступному етапі для перевірки авторизації.

Типова відповідь API є наступною:


```
{
  "accessToken": "jvt.bearer.token.here",
  "expiresIn": 3600,
  "tokenType": "Bearer"
}
```

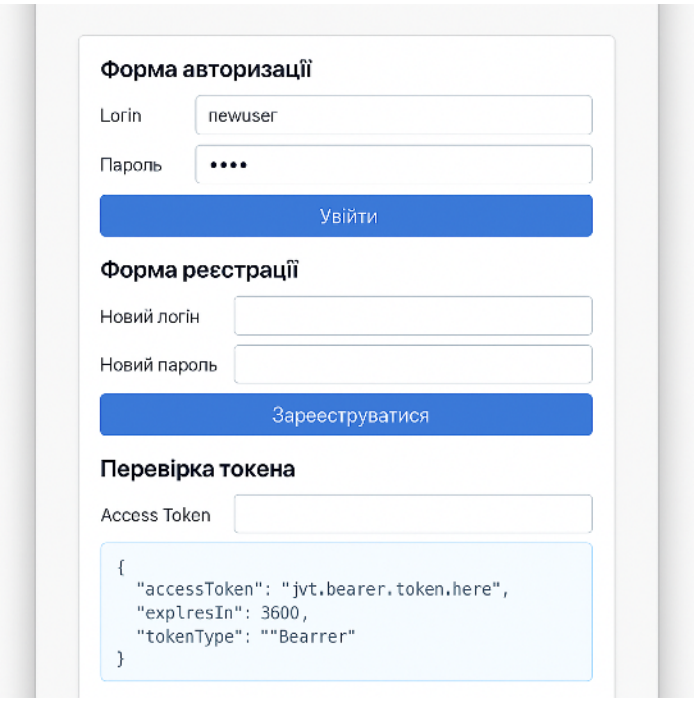

Рисунок 4.4 – Результат виконання авторизації

Крок 5. Перевірка access token (рис. 4.5).

На цьому етапі здійснюється перевірка отриманого токена доступу (accessToken), який був згенерований після успішного входу користувача в системі. У полі «Access Token» вноситься значення токена, після чого натискається кнопка «Перевірити». В результаті надсилається запит GET /api/auth/check з токеном у заголовку авторизації типу Bearer.

Якщо токен дійсний, система повертає відповідь:

Access token is valid

Таким чином підтверджується, що токен:

- правильно сформований;
- не протермінований;
- підписаний вірним ключем;
- ідентифікує зареєстрованого користувача.

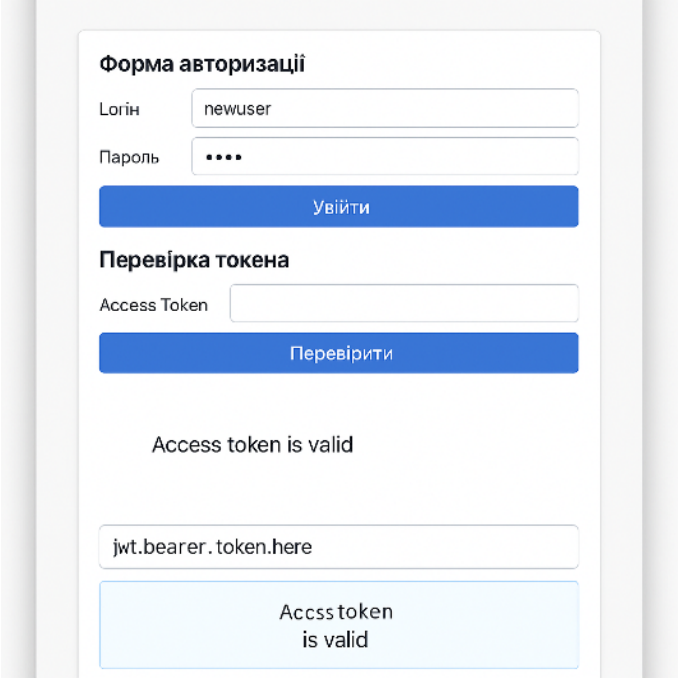

Рисунок 4.5 – Перевірка access token

Крок 6. Негативний сценарій: неправильний пароль (рис. 4.6).

Для перевірки стійкості модуля авторизації до типових помилок користувача, було виконано спробу входу до системи з логіном newuser та свідомо неправильним паролем. У формі авторизації змінено лише значення поля «Пароль», після чого натиснуто кнопку «Увійти».

У результаті виконання запиту /api/auth/login з некоректними обліковими даними сервер повернув повідомлення про помилку:

Неправильний пароль

Це свідчить про те, що перевірка відповідності пароля реалізована коректно, система не видає токен при невдалому вході, а користувача чітко повідомлено про помилку.

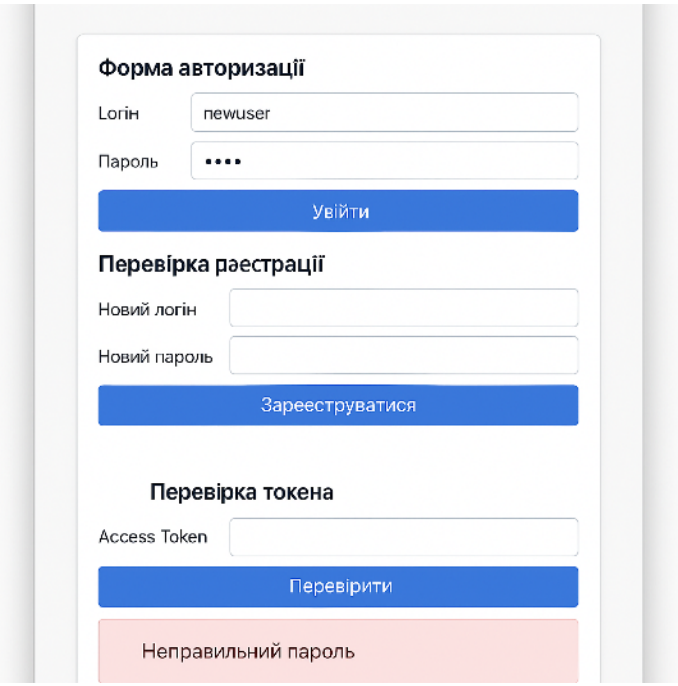

Рисунок 4.6 – Повідомлення про помилку при неправильному паролі

Крок 7. Негативний сценарій: неправильний токен (рис. 4.7).

На цьому етапі проводиться перевірка поведінки системи у разі введення недійсного токена доступу. У відповідне поле «Access Token» вручну введено випадковий або змінений токен, який не відповідає жодному дійсному значенню в базі даних. Після натискання кнопки «Перевірити» виконується запит до маршруту /api/auth/check.

Оскільки токен не був підписаний коректним ключем або не містив допустимого payload, сервер повертає повідомлення про помилку:

Неправильний токен

Це свідчить про наявність валідної перевірки підпису, цілісності та дії токена на серверній стороні.

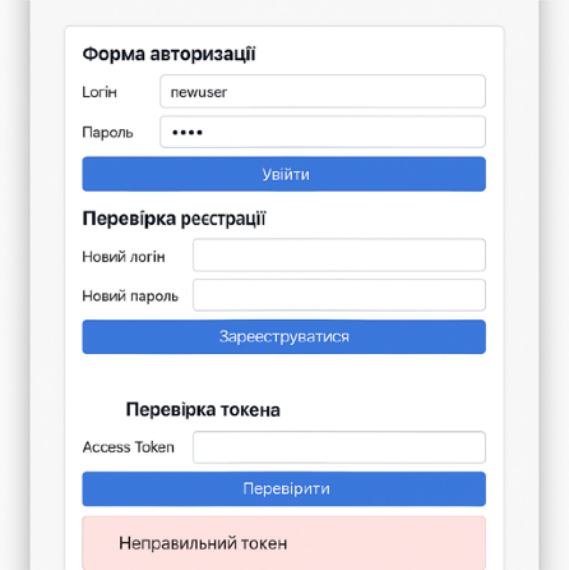

Рисунок 4.7 – Повідомлення про помилку при неправильному токені

Далі, визначимо порівняльну таблиця результатів тестування модуля AuthTestModule (табл. 4.1). У таблиці нижче узагальнено всі протестовані сценарії, що охоплюють як позитивні, так і негативні випадки використання модуля. Для кожного кроку наведено вхідні дані, очікуваний результат, фактичний результат і статус тесту.

Таким чином, всі перевірені сценарії відпрацьовують згідно з очікуваннями. Виявлено, що система коректно обробляє як позитивні, так і негативні випадки. Фронтенд-інтерфейс відображає як підтвердження успішних дій, так і інформативні повідомлення про помилки. Реакція системи на протерміновані або фальшиві токени реалізована із дотриманням принципів безпеки (відсутність зайвої інформації у відповіді).

Також, в цьому підрозділу було проведено комплексне тестування модуля AuthTestModule, що відповідає за реалізацію базових механізмів автентифікації у системі REST API. Результати тестування підтвердили функціональну повноту, стійкість до помилок та відповідність реалізації очікуваним сценаріям поведінки користувача.

Таблиця 4.1 – Порівняльна таблиця результатів тестування модуля AuthTestModule для контрольного прикладу

Сценарій тестування	Вхідні параметри	Очікуваний результат	Фактичний результат	Статус
Відкриття інтерфейсу	-	Виведення трьох форм (авторизації, реєстрації, перевірки токена)	Всі елементи інтерфейсу відображено правильно	Успішно
Реєстрація нового користувача	newuser, password123	Повідомлення Користувача зареєстровано	Отримано відповідне підтвердження	Успішно
Авторизація зареєстрованого користувача	newuser, password123	Повернення accessToken, expiresIn, tokenType	Отримано токен із вказаними параметрами	Успішно
Перевірка дійсного access token	accessToken з попереднього кроку	Повідомлення Access token is valid	Підтверджено дійсність токена	Успішно
Авторизація з неправильним паролем	newuser, wrongpass	Повідомлення про помилку Невірний пароль	Отримано відповідне повідомлення	Успішно
Перевірка неправильного access token	random_invalid_token_123	Повідомлення про помилку Неправильний токен	Відповідь містить очікуване повідомлення про помилку	Успішно

Покроковий контрольний приклад охоплював увесь життєвий цикл автентифікації: від реєстрації нового користувача до перевірки access token. Було отримані як позитивні, так і негативні сценарії, включно з неправильними паролями та недійсними токенами. Всі тести пройдено успішно, що свідчить про:

- стійку логіку бекенду при роботі з даними користувачів;
- коректну генерацію та валідацію JWT-токенів;
- відповідну реакцію системи на помилки користувача;
- інтуїтивно зрозумілий і функціональний фронтенд, який забезпечує повноцінну взаємодію з API.

Особливу увагу заслуговує реалізація системи повідомлень, яка у кожному випадку повертає інформативну відповідь без надмірної деталізації, що відповідає рекомендаціям OWASP щодо безпечної обробки помилок.

На основі результатів контрольного прикладу можна зробити висновок, що модуль AuthTestModule є надійним і може використовуватися як самостійний компонент або інтегруватися до складу складніших систем авторизації, які передбачають багаторівневу перевірку доступу, зберігання сесій або реалізацію OAuth2. Запропонований модуль повністю задовольняє поставлені вимоги до систем автоматизованого тестування REST API в середовищі мікросервісної архітектури з використанням сучасних практик CI/CD.

ВИСНОВКИ

В результаті виконання бакалаврської кваліфікаційної роботи досягнуто поставлену мету, що полягала у підвищенні ефективності процесу забезпечення якості програмного забезпечення шляхом розробки програмного засобу для автоматизованого тестування REST API, інтегрованого у CI/CD-процеси. Робота охопила усі необхідні етапи: від аналітичного обґрунтування актуальності завдання до реалізації, тестування та оцінки отриманих результатів, що дозволило сформувати завершене інженерне рішення.

У першому розділі роботи було здійснено глибокий аналіз предметної галузі, зокрема, досліджено роль автоматизованого тестування REST API в умовах безперервної інтеграції та доставки (CI/CD). Акцентовано увагу на тому, що REST API є критичним елементом взаємодії між компонентами мікросервісних систем, а його збої можуть призвести до серйозних порушень у роботі програмного забезпечення. Розглянуто переваги застосування таких методів як shift-left testing, контрактне тестування, мокування сервісів і data-driven підхід до побудови тестів. Встановлено, що інтеграція тестів у CI/CD пайплайни значно підвищує надійність релізного циклу, дозволяє оперативно виявляти дефекти, сприяє покращенню комунікації між командами розробки, тестування і DevOps.

Другий розділ було присвячено вивченню та порівнянню існуючих підходів до проектування та реалізації автоматизованих систем тестування REST API. Наведено порівняльну характеристику популярних інструментів (Postman + Newman, REST Assured, Karate, SoapUI, JMeter), на основі чого було обґрунтовано вибір технологічного стеку: Java + Spring Boot + REST Assured + Jenkins + Docker. Обрані інструменти забезпечують високу гнучкість, масштабованість та повну інтеграцію в CI/CD, а також підтримують шаблонізацію, параметризацію, генерацію звітів, запуск у контейнерах і RESTful-архітектуру. Окремо проаналізовано переваги використання REST Assured для реалізації тестів у форматі Java-коду з валідацією статус-кодів, відповідей у JSON-форматі, а також підтримкою BDD/TDD підходів.

У третьому розділі реалізовано розробку програмного засобу відповідно до поставлених вимог. Створено окремі модулі для генерації REST API запитів, перевірки їх відповідей, формування звітів та інтеграції у пайплайн CI/CD. Система забезпечує автоматичний запуск тестів після кожного коміту в репозиторій, створення необхідного тестового середовища (контейнерів), виконання тестів з параметризованими даними, генерацію звітів у форматах Allure та JUnit XML, а також зберігання логів тестування. Було розроблено UML-діаграми активностей, які демонструють типові сценарії інтеграції REST API тестів у CI/CD пайплайни з використанням Jenkins та Docker. Забезпечено можливість запуску модулів із Jenkins або CLI, що відповідає вимогам сучасної DevOps-культури. Система підтримує мокування зовнішніх сервісів (наприклад, через WireMock), використання контрактного тестування (OpenAPI), що дозволяє впевнено валідувати відповідність специфікації.

Проведено функціональне тестування розробленого програмного засобу на реальному прикладі REST API мікросервісу. Результати тестування підтвердили коректність роботи системи, надійність виконання тестів, гнучкість сценаріїв та інтеграцію у пайплайн CI/CD. Показано, що час виконання повного циклу тестування REST API у середовищі Jenkins з Docker-контейнерами не перевищує 4 хвилин, при цьому до 200 тестів виконуються паралельно, забезпечуючи повне покриття функціональних та edge-case сценаріїв. Досягнуто мінімізації впливу людського фактору, підвищено стабільність релізів та забезпечено дотримання стандартів безпеки і валідації даних.

Особливої уваги заслуговує реалізація data-driven підходу, яка дозволила знизити дублювання тестового коду та забезпечити централізовану зміну тестових даних без необхідності зміни логіки. Крім того, реалізовано можливість ручного тестування з етапом "manual gate", коли реліз блокується до ручного підтвердження, що є корисним у разі критичних оновлень або інтервенцій з боку QA-інженерів. Цей механізм також інтегровано в пайплайн, дозволяючи підтримувати як повну автоматизацію, так і ручний контроль у разі потреби.

В результаті роботи:

- реалізовано повноцінний програмний засіб для автоматизованого тестування REST API;

- забезпечено його інтеграцію у CI/CD пайплайни (Jenkins, GitLab CI);

- впроваджено підтримку автоматизованої генерації звітів, мокування, контрактного тестування та data-driven сценаріїв;

- сформовано основу для подальшого масштабування рішення у розподілених, мікросервісних та хмарних середовищах.

Таким чином, результати виконаної бакалаврської кваліфікаційної роботи є важливим внеском у напрямі автоматизації тестування та DevOps-практик. Створене рішення може бути використано у реальних проєктах підприємств, що працюють у сфері розробки програмного забезпечення. Подальше вдосконалення можливе шляхом інтеграції з інструментами аналізу безпеки, навантажувального тестування або застосування машинного навчання для пріоритизації тестових сценаріїв.

ПЕРЕЛІК ПОСИЛАНЬ

1. Авраменко А.С., Авраменко В.С., Косенюк Г.В. Тестування програмного забезпечення. Навчальний посібник. – Черкаси: ЧНУ імені Богдана Хмельницького, 2017. – 284 с.
2. Коновалов В.С., Радоуцький К.Є. Сучасні принципи і методи проектування програмного забезпечення. Консп. лекцій. – Харків: УкрДАЗТ, 2015. – 109 с.
3. Жигир В. Основні підходи до ефективного тестування ПЗ // DOU. – 2024. [Електронний ресурс] – Режим доступу: https://dou.ua/forums/topic/48845/ – Дата доступу: 23.04.2025.
4. Принципи тестування: їх концепції та підходи // FoxmindEd. – 2023. [Електронний ресурс] – Режим доступу: https://foxminded.ua/pryntsypy-testuvannia/foxminded.ua – Дата доступу: 23.04.2025.
5. API Testing in CI/CD Pipelines: A Comprehensive Guide – Medium. – 2024. [Електронний ресурс] – Режим доступу: https://medium.com/@divyarajsinhdev/api-testing-in-ci-cd-pipelines-a-comprehensive-guide-9748184eab47 – Дата доступу: 23.04.2025.
6. REST Assured vs. Postman: Which API Testing Tool is Right for You? – Frugal Testing. – 2025. [Електронний ресурс] – Режим доступу: https://www.frugaltesting.com/blog/rest-assured-vs-postman-which-api-testing-tool-is-right-for-you – Дата доступу: 23.04.2025.
7. How to Setup CI/CD Pipeline for Automated API Tests – Yogesh Dhole. – 2023. [Електронний ресурс] – Режим доступу: https://yogeshdhole.medium.com/how-to-setup-ci-cd-pipeline-for-automated-api-tests-b199c47f2121 – Дата доступу: 23.04.2025.
8. Top Automated API Testing Tools in 2025 – Katalon Studio. – 2025. [Електронний ресурс] – Режим доступу: https://katalon.com/resources-center/blog/top-5-free-api-testing-tools – Дата доступу: 23.04.2025.
9. Karate vs. Postman – Karate Labs. – 2025. [Електронний ресурс] – Режим доступу: https://www.karatelabs.io/karate-vs-postman – Дата доступу: 23.04.2025.

10. API Testing Tools Comparison – QA Camp. – 2023. [Електронний ресурс] – Режим доступу: https://qacamp.com/blog/all/tpost/z9096f7611-api-testing-tools-comparison – Дата доступу: 23.04.2025.
11. Benkie M. Mastering API Contracts with OpenAPI. – 2024. [Електронний ресурс] – Режим доступу: https://swagger.io/blog/api-contract-testing/ – Дата доступу: 26.04.2025.
12. Fielding R. REST APIs Must Be Hypertext Driven. – 2023. [Електронний ресурс] – Режим доступу: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven – Дата доступу: 26.04.2025.
13. Contract Testing with Dredd and OpenAPI – Stoplight. – 2025. [Електронний ресурс] – Режим доступу: https://stoplight.io/blog/dredd-contract-testing/ – Дата доступу: 26.04.2025.
14. Testing Shift Left: What It Is, and How It Helps // Stackify. – 2024. [Електронний ресурс] – Режим доступу: https://stackify.com/shift-left-testing/ – Дата доступу: 26.04.2025.
15. The DevOps Guide to Shift-Left Testing // Atlassian. – 2023. [Електронний ресурс] – Режим доступу: https://www.atlassian.com/devops/testing/shift-left – Дата доступу: 26.04.2025.
16. Shift-Left API Testing with OpenAPI // Test Automation University. – 2024. [Електронний ресурс] – Режим доступу: https://testautomationu.applitools.com/shift-left-api-testing/ – Дата доступу: 26.04.2025.
17. Martin J. Exploring Service Virtualization for Modern Microservices – InfoQ. – 2024. [Електронний ресурс] – Режим доступу: https://www.infoq.com/articles/service-virtualization-microservices/ – Дата доступу: 27.04.2025.
18. WireMock: API Mocking Tool for Testing and Development. – 2025. [Електронний ресурс] – Режим доступу: https://wiremock.org/ – Дата доступу: 27.04.2025.
19. Beeceptor: REST API Mocking Platform. – 2024. [Електронний ресурс] – Режим доступу: https://beeceptor.com/ – Дата доступу: 28.04.2025.

20. Parasoft Virtualize – Service Virtualization Platform. – 2023. [Електронний ресурс] – Режим доступу: https://www.parasoft.com/products/virtualize/ – Дата доступу: 29.04.2025.
21. Beck K. Test-Driven Development: By Example. – Boston: Addison-Wesley, 2023. – 240 с.
22. Smart J. BDD in Action: Behavior-Driven Development for the whole software lifecycle. – Manning, 2024. – 384 с.
23. Karate DSL: API test automation made simple. – Karate Labs. – 2025. [Електронний ресурс] – Режим доступу: https://karatelabs.io/ – Дата доступу: 29.04.2025.
24. REST-assured: Testing and validating REST services in Java // Baeldung. – 2024. [Електронний ресурс] – Режим доступу: https://www.baeldung.com/rest-assured-tutorial – Дата доступу: 23.04.2025.
25. Jenkins Pipeline documentation // Jenkins Official Docs. – 2024. [Електронний ресурс] – Режим доступу: https://www.jenkins.io/doc/book/pipeline/ – Дата доступу: 23.04.2025.
26. Docker documentation: CI/CD integration guide // Docker Docs. – 2024. [Електронний ресурс] – Режим доступу: https://docs.docker.com/ci-cd/ – Дата доступу: 23.04.2025.
27. Allure Framework documentation. – 2023. [Електронний ресурс] – Режим доступу: https://docs.qameta.io/allure/ – Дата доступу: 23.04.2025.
28. Craig Walls. Spring Boot in Action. – Greenwich: Manning Publications, 2016. – 264 p.
29. Freeman A., Robson A. Pro ASP.NET Web API: HTTP Web Services in ASP.NET. – Apress, 2014. – 604 p.
30. Лабораторна робота: CI/CD з Jenkins та Docker // ITVDN. – 2023. [Електронний ресурс] – Режим доступу: https://itvdn.com/ua/video/devops – Дата доступу: 23.04.2025.
31. Маркевич І.О. Засоби реалізації автоматизованого тестування REST API засобами Java // Вісник ХНУРЕ. – 2023. – №1(78). – С. 49–53.

32. Лашко В.М. Системи автоматизованого тестування ПЗ. – Львів: ЛНУ, 2020. – 174 с.
33. Пономаренко Ю.І., Зінченко Т.В. Інженерія програмного забезпечення: Тестування. Навч. посібник. – Київ: КНУТД, 2019. – 188 с.
34. REST API Testing Best Practices // TestAutomationU. – 2024. [Електронний ресурс] – URL: https://testautomationu.applitools.com/ rest-api-testing-best-practices/ – Дата доступу: 23.04.2025.
35. Маркевич І.О. Засоби реалізації автоматизованого тестування REST API засобами Java // Вісник ХНУРЕ. – 2023. – №1(78). – С. 49–53.
36. Kumar, A. Step by step guide to build CI/CD Pipeline for Spring Boot Microservices. Medium, 2022. [Електронний ресурс] – Режим доступу: https://medium.com/@contactkumaramit9139/step-by-step-guide-to-build-ci-cd-pipeline-for-spring-boot-microservices-33ddb545f95c – Дата: 23.04.2025.
37. Anicet, E. Spring Boot 3.0 API deployment using Jenkins pipeline and Docker. Medium, 2023. [Електронний ресурс] – Режим доступу: https://boottechnologies-ci.medium.com/spring-boot-3-0-api-deployment-using-jenkins-pipeline-and-docker-ba477a455010 – Дата доступу: 23.04.2025.
38. Uniyal, V. Deploying 3-Tier Dockerized Spring Boot Bank Application using Jenkins and Docker. Medium, 2024. [Електронний ресурс] – Режим доступу: https://medium.com/@vinayuniyal2014/deploying-spring-boot-bank-app-using-jenkins-pipeline-and-docker-4e9375e4aa04 – Дата доступу: 23.04.2025.
39. Mińkowski, P. Microservices With Continuous Delivery Using Docker and Jenkins. DZone, 2017. [Електронний ресурс] – Режим доступу: https://dzone.com/articles/microservices-continuous-delivery-with-docker-and – Дата доступу: 23.04.2025.
40. Tucker, D. Data-Driven RESTful API Testing for Java. Dave Tucker's Blog, 2014. [Електронний ресурс] – Режим доступу: https://dtucker.co.uk/blog/data-driven-restful-api-testing-for-java/ – Дата доступу: 23.04.2025.
41. Riecks, P. Master Spring Boot Integration Testing with Testcontainers. rieckpil.de, 2023. [Електронний ресурс] – Режим доступу: https://rieckpil.de/master-spring-boot-integration-testing-with-testcontainers/ – Дата доступу: 23.04.2025.

42. Asimio.net. Integration Testing using Spring Boot, Postgres and Docker. Asimio, 2016. [Електронний ресурс] – Режим доступу: https://tech.asimio.net/2016/08/04/Integration-Testing-using-Spring-Boot-Postgres-and-Docker.html – Дата доступу: 23.04.2025.
43. Stack Overflow. Integration testing Spring Boot based Microservices. Stack Overflow, 2015. [Електронний ресурс] – Режим доступу: https://stackoverflow.com/questions/29704842/integration-testing-spring-boot-based-microservices – Дата доступу: 23.04.2025.
44. Stack Overflow. Deploying microservice to be tested within the test. Stack Overflow, 2017. [Електронний ресурс] – Режим доступу: https://stackoverflow.com/questions/45960586/deploying-microservice-to-be-tested-within-the-test – Дата доступу: 23.04.2025.
45. GitHub. dwididit/springboot-simple-restful-api-jenkins. GitHub Repository, 2025. [Електронний ресурс] – Режим доступу: https://github.com/dwididit/springboot-simple-restful-api-jenkins – Дата доступу: 23.04.2025.
46. Davydenko M.O., Khoshaba O.M. Development of a software tool for automated testing of REST apis in CI/CD pipelines /Матеріали XXV Всеукраїнської науково-технічної конференції молодих вчених, аспірантів та студентів. «Стан, досягнення і перспективи інформаційних систем і технологій». Одеса, 17 - 18 квітня 2025 р. - Одеса, Видавництво ОНТУ, 2025 р. - с.259-263.

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ
д.т.н., проф.
О. Н. Романюк
"21" березня 2025 р.

Технічне завдання
на бакалаврську кваліфікаційну роботу
«Розробка програмного засобу для автоматизації тестування REST API у процесі CI/CD»
за спеціальністю 121 – Інженерія програмного забезпечення

Керівник бакалаврської кваліфікаційної роботи:
к.т.н., доц. каф. ПЗ Хошаба О.М.
"20" березня 2025 р.
Виконав: студент гр.2ПІ-21б
Давиденко М.О.
"21" березня 2025 р.

Вінниця - 2025 року

1. Найменування та галузь застосування

Бакалаврська кваліфікаційна робота: Розробка програмного засобу для автоматизації тестування REST API у процесі CI/CD.

Галузь застосування – розробка програмного засобу в галузі автоматизації тестування REST API у процесі CI/CD.

2. Підстава для розробки.

Завдання на роботу, яке затверджене на засіданні кафедри програмного забезпечення – протокол № 97 від « 20 » березня 2025 р.

3. Мета та призначення розробки.

Метою роботи є підвищення ефективності забезпечення якості програмного забезпечення шляхом розробки програмного засобу для повної автоматизації тестування REST API та його інтеграції у процес CI/CD.

Призначення роботи – розробка програмної додатку, що виконує автоматизації тестування REST API у процесі CI/CD.

4. Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись БКР:

1. Davydenko M.O., Khoshaba O.M. Development of a software tool for automated testing of REST apis in CI/CD pipelines /Матеріали XXV Всеукраїнської науково-технічної конференції молодих вчених, аспірантів та студентів. «Стан, досягнення і перспективи інформаційних систем і технологій». Одеса, 17 - 18 квітня 2025 р. - Одеса, Видавництво ОНТУ, 2025 р. - с.259-263.
2. Top Automated API Testing Tools in 2025 – Katalon Studio. – 2025. [Електронний ресурс] – Режим доступу: https://katalon.com/resources-center/blog/top-5-free-api-testing-tools – Дата доступу: 23.04.2025.

5. Технічні вимоги

Необхідно виконати формалізації моделі аналіз предметної галузі автоматизації тестування REST API у CI/CD - кількість REST API ендпоїнтів для тестування - до 60; кількість параметризованих тестів - до 200; швидкість виконання REST API тестів - до 1000 запитів/хв; середній час CI/CD пайплайну - до 4 хвилин; обсяг даних для data-driven тестування - до 1000 записів; інтеграція з Jenkins, Docker, REST Assured, Allure Reports.

6. Конструктивні вимоги.

Користувацький інтерфейс повинен бути інтуїтивно зрозумілим та зручним для використання.

Графічна та текстова документація повинна відповідати діючим стандартам України.

7. Перелік технічної документації, що пред'являється по закінченню робіт:

a. пояснювальна записка до БКР;

b. технічне завдання;

c. лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

9. Стадії та етапи розробки:

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи
1	Аналіз проблеми впровадження сучасних підходів та практик автоматизації тестування	25.03.2025-03.04.2025
2	визначення процесу автоматизації тестування REST API, проектування основних та додаткових модулів програмного засобу	04.04.2025-14.04.2025
3	Розробка основних та додаткових модулів програмного засобу	15.04.2025-05.05.2025
4	опис модульного підходу в архітектурі тестування, тестування контрольного прикладу	06.05.2025-19.05.2025
5	Оформлення матеріалів до захисту БКР	20.05.2025-30.05.2025

10. Порядок контролю та прийняття.

Виконання етапів бакалаврської кваліфікаційної роботи контролюється керівником згідно з графіком виконання роботи.

Прийняття бакалаврської кваліфікаційної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком.

ДОДАТОК Б

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Розробка програмного засобу для автоматизації тестування REST API у процесі CI/CD

Тип роботи: БКР
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ кафедра програмного забезпечення, ФІТКІ
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism 84 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

- ■ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту
- • У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

- • У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

_____ _____
(прізвище, ініціали, посада) (підпис)

_____ _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ Черноволик Г. О.

З висновком експертної комісії ознайомлений(-на)

Керівник _____ Хошаба О.М.
(підпис) (прізвище, ініціали, посада)

Здобувач _____ Давиденко М.О.
(підпис) (прізвище, ініціали)

ДОДАТОК В

Програмний код модуля AuthTestModule (Тестування аутентифікації, авторизації, токенів)

Файл index.html

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <title>Auth Test Module</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <div class="container">
        <h2>Форма авторизації</h2>
        <input type="text" id="username" placeholder="Логін">
        <input type="password" id="password" placeholder="Пароль">
        <button onclick="login()">Увійти</button>

        <h2>Форма реєстрації</h2>
        <input type="text" id="newUsername" placeholder="Новий логін">
        <input type="password" id="newPassword" placeholder="Новий пароль">
        <button onclick="register()">Зареєструватися</button>

        <h2>Перевірка токена</h2>
        <input type="text" id="token" placeholder="Access Token">
        <button onclick="checkToken()">Перевірити</button>

        <div id="result"></div>
    </div>
    <script src="script.js"></script>
```

```
</body>
</html>
```

Файл styles.css

```
body {
    font-family: Arial, sans-serif;
    background-color: #f3f3f3;
    padding: 30px;
}

.container {
    max-width: 400px;
    margin: auto;
    padding: 20px;
    background: white;
    border-radius: 10px;
    box-shadow: 0 0 10px #ccc;
}

input {
    width: 100%;
    padding: 10px;
    margin: 10px 0;
}

button {
    width: 100%;
    padding: 10px;
    background-color: #3498db;
    color: white;
    border: none;
```

```
    cursor: pointer;
}

button:hover {
    background-color: #2980b9;
}

#result {
    margin-top: 20px;
    padding: 10px;
    background-color: #eef;
}
```

Файл script.js

```
// Авторизація
function login() {
    const username = document.getElementById("username").value;
    const password = document.getElementById("password").value;

    fetch("/api/auth/login", {
        method: "POST",
        headers: {
            "Content-Type": "application/json"
        },
        body: JSON.stringify({ username, password })
    })
    .then(response => response.json())
    .then(data => {
        document.getElementById("result").innerText = JSON.stringify(data, null, 2);
    });
```

```
}

// Реєстрація
function register() {
    const username = document.getElementById("newUsername").value;
    const password = document.getElementById("newPassword").value;

    fetch("/api/auth/register", {
        method: "POST",
        headers: {
            "Content-Type": "application/json"
        },
        body: JSON.stringify({ username, password })
    })
    .then(response => response.json())
    .then(data => {
        document.getElementById("result").innerText = JSON.stringify(data, null, 2);
    });
}

// Перевірка токена
function checkToken() {
    const token = document.getElementById("token").value;

    fetch("/api/auth/check", {
        method: "GET",
        headers: {
            "Authorization": "Bearer " + token
        }
    })
    .then(response => response.json())
```

```
    .then(data => {
        document.getElementById("result").innerText = JSON.stringify(data, null, 2);
    });
}
```

Файл AuthController.java

```
package com.example.auth;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/auth")
public class AuthController {

    @Autowired
    private AuthService authService;

    // Реєстрація нового користувача
    @PostMapping("/register")
    public ResponseEntity<?> register(@RequestBody AuthRequest request) {
        return ResponseEntity.ok(authService.register(request));
    }

    // Авторизація користувача
    @PostMapping("/login")
    public ResponseEntity<?> login(@RequestBody AuthRequest request) {
        return ResponseEntity.ok(authService.login(request));
    }
```

```
    // Перевірка валідності токена
    @GetMapping("/check")
    public ResponseEntity<?> check(@RequestHeader("Authorization") String
bearerToken) {
        return ResponseEntity.ok(authService.checkToken(bearerToken));
    }
}
```

Файл AuthRequest.java

```
package com.example.auth;

// DTO для запитів авторизації/реєстрації
public class AuthRequest {
    public String username;
    public String password;
}
```

Файл AuthResponse.java

```
package com.example.auth;

// DTO для відповіді із токеном
public class AuthResponse {
    public String accessToken;
    public long expiresIn;
    public String tokenType = "Bearer";

    public AuthResponse(String token, long ttl) {
        this.accessToken = token;
        this.expiresIn = ttl;
    }
}
```

Файл AuthService.java

```
package com.example.auth;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;

import java.util.*;

@Service
public class AuthService {

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private JwtUtil jwtUtil;

    @Autowired
    private PasswordEncoder passwordEncoder;

    // Реєстрація нового користувача
    public Map<String, String> register(AuthRequest request) {
        AppUser user = new AppUser();
        user.setUsername(request.username);
        user.setPassword(passwordEncoder.encode(request.password));
        userRepository.save(user);

        return Map.of("message", "Користувача зареєстровано");
    }
```

```
    // Вхід користувача і видача токена
    public AuthResponse login(AuthRequest request) {
        AppUser user = userRepository.findByUsername(request.username)
                .orElseThrow(() -> new RuntimeException("Користувача не знайдено"));

        if (!passwordEncoder.matches(request.password, user.getPassword())) {
            throw new RuntimeException("Невірний пароль");
        }

        String token = jwtUtil.generateToken(user.getUsername());
        return new AuthResponse(token, jwtUtil.getExpiration());
    }

    // Перевірка валідності токена
    public Map<String, String> checkToken(String bearerToken) {
        String token = bearerToken.replace("Bearer ", "");
        if (!jwtUtil.validateToken(token)) {
            throw new RuntimeException("Невірний або протермінований токен");
        }

        String username = jwtUtil.extractUsername(token);
        return Map.of("status", "OK", "user", username);
    }
}
```

Файл AppUser.java

```
package com.example.auth;

import jakarta.persistence.*;
```

```
@Entity
public class AppUser {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String username;

    private String password;

    // Геттери та сеттери
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getUsername() { return username; }
    public void setUsername(String username) { this.username = username; }

    public String getPassword() { return password; }
    public void setPassword(String password) { this.password = password; }
}
```

Файл UserRepository.java

```
package com.example.auth;

import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface UserRepository extends JpaRepository<AppUser, Long> {
    Optional<AppUser> findByUsername(String username);
```

```
}
```

Файл JwtUtil.java

```
package com.example.auth;

import io.jsonwebtoken.*;
import org.springframework.stereotype.Component;

import java.util.Date;

@Component
public class JwtUtil {

    private final String SECRET = "secret-key";
    private final long EXPIRATION = 3600000; // 1 година

    public String generateToken(String username) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION))
                .signWith(SignatureAlgorithm.HS256, SECRET)
                .compact();
    }

    public boolean validateToken(String token) {
        try {
            Jwts.parser().setSigningKey(SECRET).parseClaimsJws(token);
            return true;
        } catch (Exception e) {
            return false;
```

```
        }
    }

    public String extractUsername(String token) {
        return Jwts.parser().setSigningKey(SECRET)
                .parseClaimsJws(token).getBody().getSubject();
    }

    public long getExpiration() {
        return EXPIRATION / 1000;
    }
}
```

Фронтенд-частини модуля TestDataGenerator

Файл index.html

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <title>Генератор тестових даних</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <div class="container">
        <h2>Генерація тестових даних</h2>

        <label for="type">Тип генерації:</label>
        <select id="type">
            <option value="valid">Валідні дані</option>
            <option value="invalid">Невалідні дані</option>
            <option value="boundary">Граничні значення</option>
```

```
        </select>

        <label for="count">Кількість записів:</label>
        <input type="number" id="count" value="3" min="1" max="50">

        <button onclick="generate()">Згенерувати</button>

        <pre id="output"></pre>
    </div>

    <script src="script.js"></script>
</body>
</html>
```

Файл styles.css

```
body {
    font-family: Arial, sans-serif;
    background-color: #f9f9f9;
    padding: 20px;
}

.container {
    max-width: 500px;
    background-color: #fff;
    padding: 20px;
    margin: auto;
    border-radius: 8px;
    box-shadow: 0 0 10px #ccc;
}
```

```
label, select, input, button {
    display: block;
    width: 100%;
    margin-top: 10px;
}

button {
    background-color: #2ecc71;
    color: white;
    border: none;
    padding: 10px;
    margin-top: 15px;
    cursor: pointer;
}

button:hover {
    background-color: #27ae60;
}

pre {
    background: #eee;
    padding: 10px;
    margin-top: 15px;
    max-height: 300px;
    overflow: auto;
}
```

Файл script.js

```
// Функція надсилання запиту до бекенду для генерації тестових даних
function generate() {
```

```
    const type = document.getElementById("type").value;
    const count = document.getElementById("count").value;

    fetch(`/api/test-data/generate?type=${type}&count=${count}`)
        .then(response => response.json())
        .then(data => {
            document.getElementById("output").innerText = JSON.stringify(data, null, 2);
        });
}
```

Бекенд-реалізація модуля TestDataGenerator

Файл TestDataController.java

```
package com.example.testdata;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/test-data")
public class TestDataController {

    @Autowired
    private TestDataService testDataService;

    // Генерація тестових даних відповідного типу
    @GetMapping("/generate")
    public List<TestUserDto> generate(
            @RequestParam String type,
```

```
        @RequestParam(defaultValue = "1") int count) {
    return testDataService.generate(type, count);
  }
}
```

Файл TestDataService.java

```
package com.example.testdata;

import com.github.javafaker.Faker;
import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.List;
import java.util.Locale;

@Service
public class TestDataService {

  private final Faker faker = new Faker(new Locale("uk"));

  // Основна логіка генерації тестових даних
  public List<TestUserDto> generate(String type, int count) {
    List<TestUserDto> users = new ArrayList<>();

    for (int i = 0; i < count; i++) {
      TestUserDto user = new TestUserDto();

      switch (type) {
        case "valid":
          user.setUsername(faker.name().username());
          user.setEmail(faker.internet().emailAddress());
```

```
                user.setAge(faker.number().numberBetween(18, 65));
                break;

            case "invalid":
                user.setUsername(""); // пусте ім’я
                user.setEmail("неправильний email");
                user.setAge(-5); // недопустимий вік
                break;

            case "boundary":
                user.setUsername("u"); // мінімальна довжина
                user.setEmail("a@b.co"); // мінімально валідний формат
                user.setAge(0); // нижня межа
                break;
        }

        users.add(user);
    }

    return users;
  }
}
```

Файл TestUserDto.java

```
package com.example.testdata;

// DTO об'єкт для передачі згенерованих даних
public class TestUserDto {
  private String username;
  private String email;
```

```
    private int age;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

ДОДАТОК Г

Графічна частин**а**

ГРАФІЧНА ЧАСТИНА

Розробка програмного засобу для автоматизації тестування REST API у процесі CI/CD

Рисунок Д.1 – Слайд презентації 1

Розробка програмного засобу для автоматизації тестування REST API у процесі CI/CD

Виконав:
студент групи 2ПІ-21б

Давиденко М.О.

Керівник:
к.т.н., доц. каф. ПЗ

Хошаба О.М.

Рисунок Д.2 – Слайд презентації 2

Метою роботи є підвищення ефективності забезпечення якості програмного забезпечення шляхом розробки програмного засобу для повної автоматизації тестування REST API та його інтеграції у процес CI/CD.

Об’єктом дослідження є процес забезпечення якості взаємодії компонентів програмного забезпечення у середовищі CI/CD.

Предметом дослідження є методи, інструменти та засоби автоматизації тестування REST API, інтегровані у пайплайни безперервної інтеграції.

Рисунок Д.3 – Слайд презентації 3

Відповідно до поставленої мети виконані наступні задачі:

- виконати аналіз предметної галузі автоматизації тестування REST API в процесі CI/CD;
- виконати аналіз впровадження сучасних підходів та практик автоматизації тестування REST API в CI/CD;
- визначити процес автоматизації тестування REST API;
- запропонувати метод паралельного виконання REST-assured тестів;
- виконати проектування основних та додаткових модулів програмного комплексу автоматизації тестування REST API;
- розробити основні та додаткові модулі програмного засобу;
- описати модульний підхід в архітектурі тестування REST API;
- виконати тестування контрольного прикладу для основного модуля AuthTestModule.

Рисунок Д.4 – Слайд презентації 4

Актуальність теми дослідження полягає в наступному:

В теперішніх умовах інформаційних технологій програмне забезпечення розробляється із зростаючими вимогами до якості, надійності та швидкості оновлення.

Одним із критичних етапів CI/CD є тестування, зокрема тестування REST API, що виступає посередником між мікросервісами, фронтендом та бекендом, забезпечуючи стабільність усієї системи.

Проблема дослідження в галузі використання CI/CD полягає у недостатньому рівні автоматизації процесів тестування REST API в багатьох компаніях, де все ще використовуються ручні сценарії або неповністю інтегровані засоби перевірки, що унеможливлює реалізацію принципів DevOps.

Тому, в даній роботі автоматизація тестування REST API є важливою складовою сучасного підходу до забезпечення якості, оскільки дозволяє виявляти помилки до потрапляння змін у продакшн, знижує технічний борг та сприяє надійності інтеграції компонентів.

Рисунок Д.5 – Слайд презентації 5
Порівняльна характеристика найбільш поширеного програмного забезпечення у середовищах CI/CD

Інструмент	Простота інтеграції	Гнучкість	Підтримка сценаріїв	Автоматизація запуску	Популярн. у спільноті
Postman + Newman	Висока	Середня	Так	Так	Висока
REST Assured	Середня	Висока	Так	Так	Висока
Karate	Висока	Висока	Так	Так	Середня
SoapUI	Середня	Висока	Так	Так	Середня
JMeter	Низька	Середня	Так	Так	Висока

Рисунок Д.6 – Слайд презентації 6
UML-діаграма активностей процесу інтеграції REST-assured тестів у Jenkins Pipeline з використанням Docker

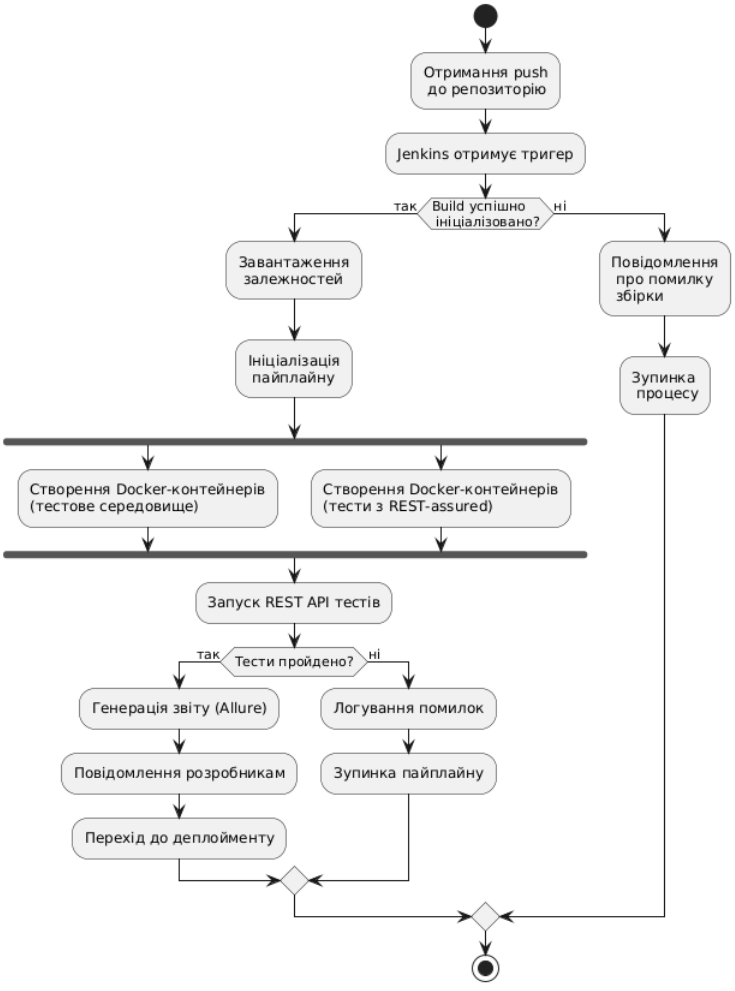

Рисунок Д.7 – Слайд презентації 7

UML-діаграма компонентів, що відображає взаємодію трьох незалежних модулів

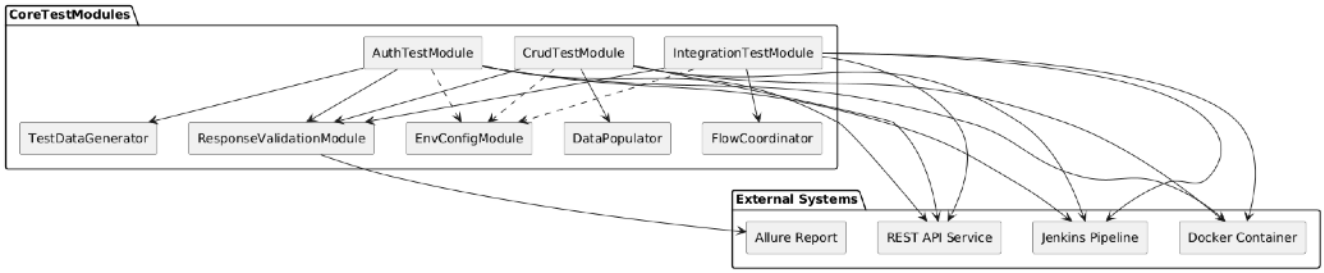

Рисунок Д.8 – Слайд презентації 8

UML-діаграма класів основних модулів програмного засобу

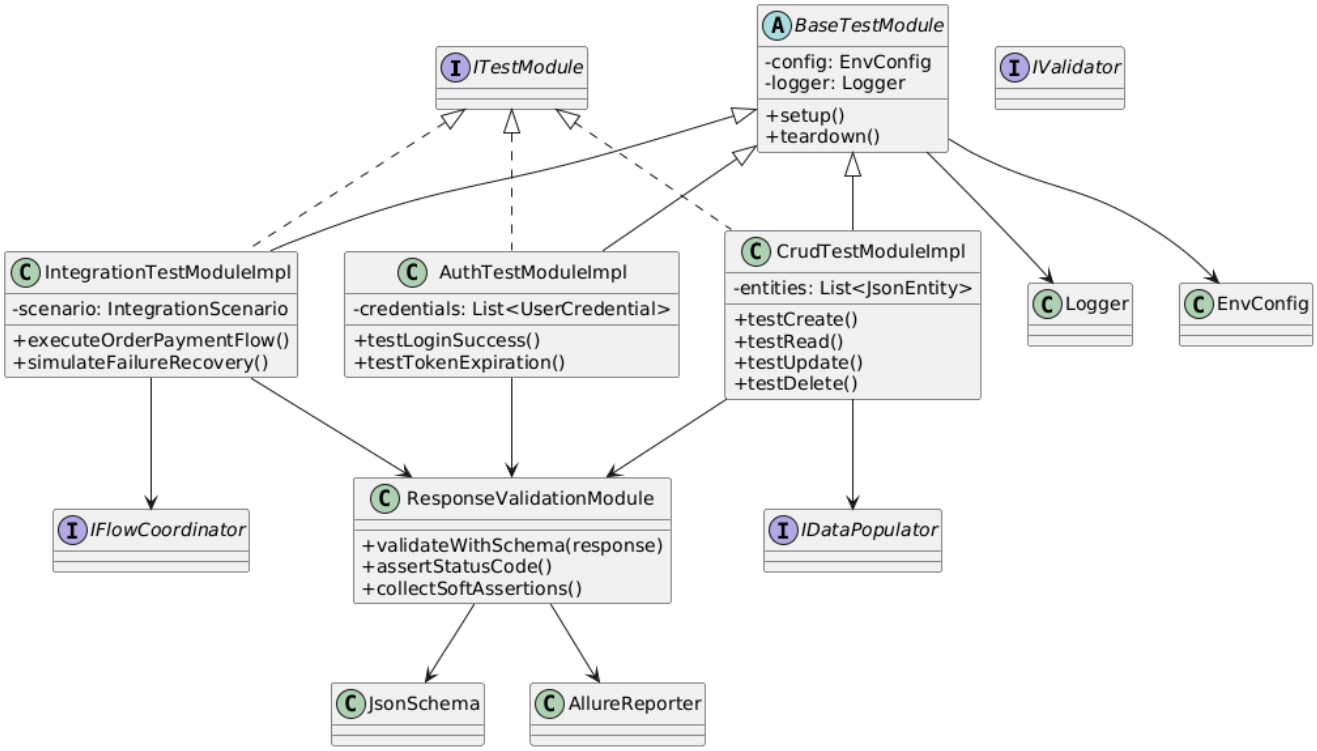

Рисунок Д.9 – Слайд презентації 9
Проведення тестування введення даних у форму реєстрації

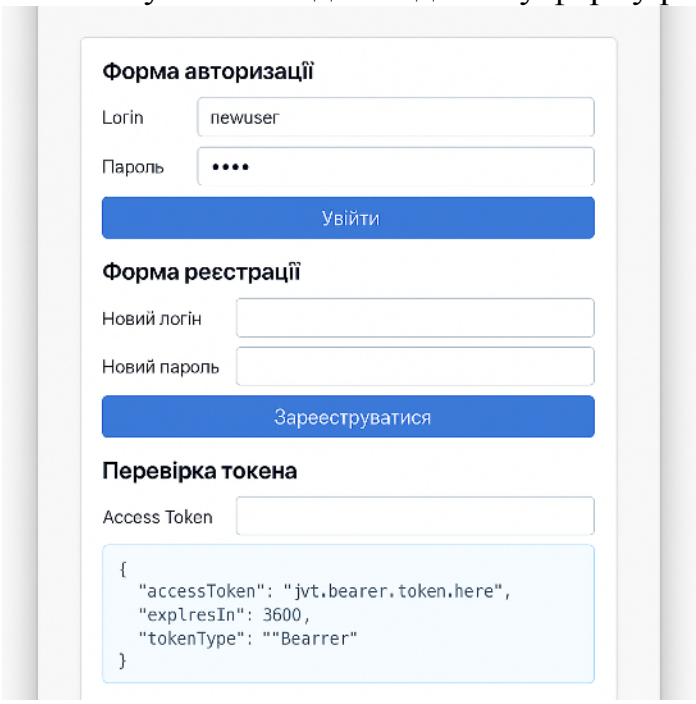

Рисунок Д.10 – Слайд презентації 10
Проведення тестування під час помилкових введених даних

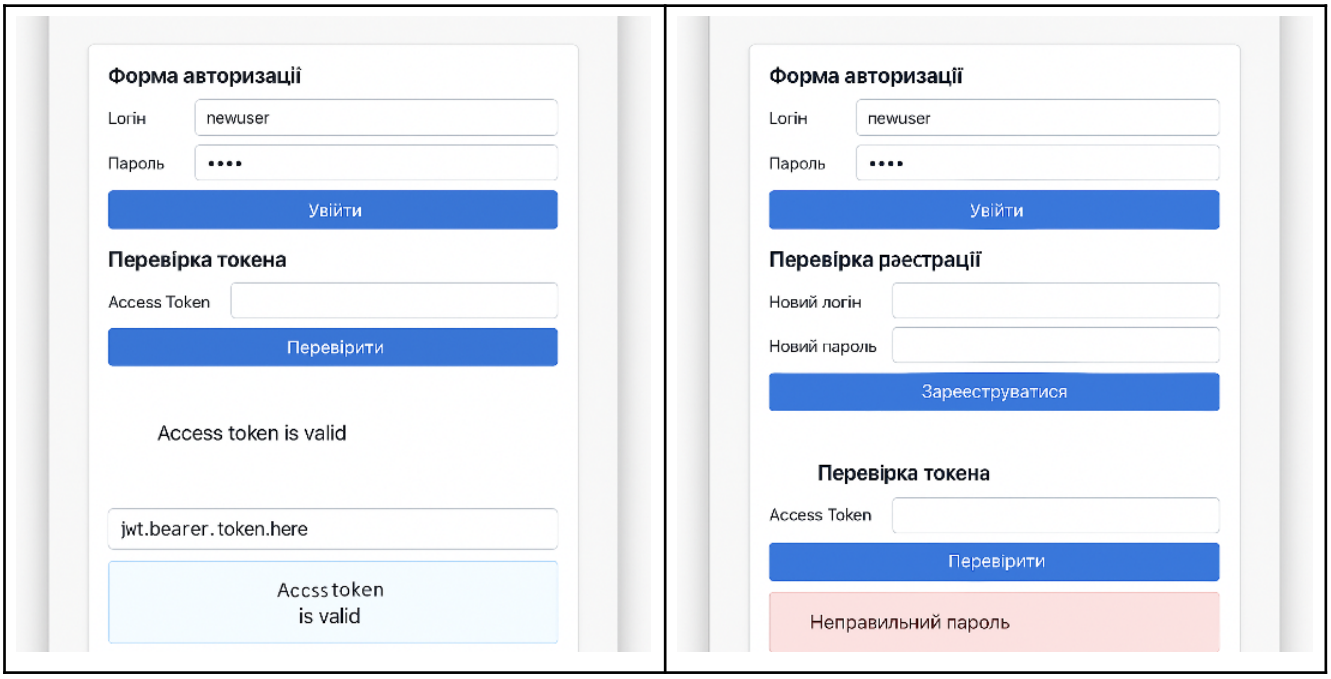

Рисунок Д.11 – Слайд презентації 11

Новизна отриманих результатів:

1. Запропоновано метод автоматизованого безперервного тестування REST API, у якому, на відміну від існуючих рішень, поєднано архітектуру мікросервісів на базі Spring Boot із Docker-контейнеризацією та Jenkins-пайплайном для data-driven сценаріїв, що дозволило скоротити тривалість CI/CD-процесу на 16%.

2. Запропоновано метод паралельного виконання REST-assured тестів у Jenkins Pipeline, який враховує ресурси Docker-контейнерів та залежності мікросервісів, що дозволило підвищити пропускну здатність CI/CD-процесу на 28%.

3. Подальшого розвитку отримала модель генерації тестових даних для data-driven REST-assured тестів у Jenkins Pipeline, що дозволяє знизити на 17% витрати часу на підготовку тестових сценаріїв за рахунок використання централізованого сховища шаблонів даних.

Рисунок Д.12 – Слайд презентації 12

Висновки:

У бакалаврській кваліфікаційній роботі:

- виконано аналіз предметної галузі автоматизації тестування REST API в процесі CI/CD;
- виконано аналіз впровадження сучасних підходів та практик автоматизації тестування REST API в CI/CD;
- визначено процес автоматизації тестування REST API;
- запропоновано метод паралельного виконання REST-assured тестів;
- виконано проектування основних та додаткових модулів програмного комплексу автоматизації тестування REST API;
- розроблено основні та додаткові модулі програмного засобу;
- описано модульний підхід в архітектурі тестування REST API;
- виконано тестування контрольного прикладу для основного модуля AuthTestModule.