Міністерство освіти і науки України

Луцький національний технічний університет
(повне найменування закладу вищої освіти)

Факультет комп'ютерних та інформаційних технологій
(повне найменування факультету)

Кафедра комп'ютерної інженерії та безпеки
(повне найменування кафедри)

КВАЛІФІКАЦІЙНА РОБОТА ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

РОЗРОБКА ТА ОПТИМІЗАЦІЯ СИСТЕМИ ТЕСТУВАННЯ МЕРЕЖЕВОЇ ІНФРАСТРУКТУРИ REST API З ВИКОРИСТАННЯМ АПАРАТНО-ПРОГРАМНИХ ЗАСОБІВ

DEVELOPMENT AND OPTIMIZATION OF A NETWORK INFRASTRUCTURE TESTING SYSTEM FOR REST API USING HARDWARE AND SOFTWARE TOOLS

спеціальність 123 Комп'ютерна інженерія
(шифр і назва спеціальності)

освітня програма Комп'ютерна інженерія
(назва освітньої програми)

Виконав: здобувач вищої освіти
групи КІ-41
Куліков Назар Олександрович

(підпис)

Керівник:
к.т.н., доцент
Багнюк Наталія Володимирівна

(підпис)

Кваліфікаційну роботу
допущено до захисту
« 04 » червня 2025 р.
Гарант освітньої програми:
к.т.н., доцент
Лавренчук Світлана Василівна

(підпис)

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *комп'ютерної інженерії та безпеки*
Ступінь вищої освіти: *бакалавр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *123 Комп'ютерна інженерія*
Освітня програма: *«Комп'ютерна інженерія»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ доц. Т. ТЕРЛЕЦЬКИЙ
« 10 » 01 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Куліков Назар Олександрович
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *Розробка та оптимізація системи тестування мережевої інфраструктури REST API з використанням апаратно- програмних засобів*

Керівник роботи *к.т.н., доц. Багнюк Наталія Володимирівна*

затверджені наказом закладу вищої освіти від «04» січня 2025 року № 11/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи 10.06.2025р.

3. Вихідні дані до роботи *джерелом розробки є науково-технічна література та публікації в періодичних виданнях з даного питання, опубліковані зарубіжні та вітчизняні роботи в даній області та різні інтернет-ресурси технічного спрямування.*

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити):
Вступ
Аналіз теоретичних основ та підходів до тестування Rest API
Вибір архітектури та розробка фреймворку для автоматичного тестування
Реалізація та перевірка роботи API-тестів із генерацією звіту
Висновки

5. Перелік графічного (ілюстративного) матеріалу:

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз проблеми за темою роботи та постановка завдань дослідження	*Багнюк Н.В., доцент*		
Теоретичне дослідження підходів до тестування APIта розробка фреймворку	*Багнюк Н.В., доцент*		
Практична реалізація тестів та генерація звітів Allure	*Багнюк Н.В., доцент*		
Нормоконтроль	*Багнюк Н.В., доцент*		
Гарант ОП	*Лавренчук С.В., доцент*		
Показник запозичень тексту	***___%***		
Академічна доброчесність	*Міскевич О.І.,ст.викладач*		

7. Дата видачі завдання 10.01.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	*Огляд літератури із досліджуваної проблеми, аналіз предметної області та наявних рішень*	до 10.02.2025 р.	Виконано
2.	*Теоретичне дослідження підходів до тестування APIта розробка фреймворку*	до 02.03.2025 р.	Виконано
3.	*Практична реалізація тестів та генерація звітів Allure*	до 02.04.2025 р.	Виконано
4.	*Висновки та пропозиції*	до 10.04.2025 р.	Виконано
5.	*Формування списку використаних джерел*	до 15.04.2025 р.	Виконано
6.	*Формування додатків*	до 02.05.2025 р.	Виконано
7.	*Оформлення ілюстративного матеріалу*	до 10.05.2025 р.	Виконано
8.	*Представлення остаточного варіанту кваліфікаційної роботи керівникові*	до 15.05.2025 р.	Виконано
9.	*Нормоконтроль*	до 30.05.2025 р.	Виконано
10	*Інструментальна перевірка на академічний плагіат*	до 03.06.2025 р	Виконано
11.	*Здача кваліфікаційної роботи та всіх супровідних документів на кафедру*	до 10.06.2025 р.	Виконано

Здобувач вищої освіти _____ (підпис) Куліков Н.О. (прізвище, ініціали)

Керівник кваліфікаційної роботи _____ (підпис) Багнюк Н.В. (прізвище, ініціали)

АНОТАЦІЯ

Куліков Н.О. Розробка та оптимізація системи тестування мережевої архітектури REST API з використанням апаратно-програмних засобів.

Кваліфікаційна робота бакалавра ОП «Комп'ютерна інженерія» спеціальності 123 Комп'ютерна інженерія.

Луцький національний технічний університет. Луцьк, 2025.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, списку використаних джерел і додатків.

У першому розділі представлено теоретичні відомості щодо API як архітектурного рішення, розглянуто основні типи та формати обміну даними, принципи побудови REST API, особливості клієнт-серверної взаємодії, HTTP-запити, коди відповідей, а також підходи до тестування API. Надано огляд сучасних інструментів, які використовуються для реалізації автоматизованих перевірок. Наведено обґрунтування вибору мови програмування Java, інструментів Maven, TestNG, Rest Assured та Allure.

У другому розділі здійснено розробку власного фреймворку для тестування REST API. Описано архітектуру фреймворку, структуру проєкту, конфігурацію середовища та логіку взаємодії з API. Реалізація проєкту виконувалася в середовищі розробки IntelliJ IDEA.

Третій розділ присвячено написанню тестових сценаріїв. Реалізовано функціональні тести, проведено аналіз результатів, представлено автоматичні звіти про виконання тестів, а також оцінено ефективність побудованої системи перевірок.

Ключові слова: REST API, автоматизоване тестування, Java, TestNG, Rest Assured, Allure, HTTP, IntelliJ IDEA.

ANNOTATION

Kulikov N. Development and optimization of a network infrastructure testing system for REST API using hardware and software tools. Manuscript.

Bachelor's qualification work in the educational program «Computer Engineering», specialty 123 Computer Engineering.

Lutsk National Technical University. Lutsk, 2025.

The qualification work consists of an introduction, three chapters, conclusions, a list of references, and appendices.

The first chapter presents theoretical information about API as an architectural solution. It examines the main types and data exchange formats, the principles of building REST APIs, the specifics of client-server interaction, HTTP requests and response codes, as well as approaches to API testing. A review of modern tools used for automated testing is also provided.

The second chapter describes the development of a custom framework for REST API testing. It provides justification for the use of the Java programming language and tools such as Maven, TestNG, Rest Assured, and Allure. The chapter outlines the framework architecture, project structure, environment configuration, and the logic of interaction with the API. The project was implemented in the IntelliJ IDEA development environment.

The third chapter focuses on writing test scenarios. Functional and negative tests were implemented, test results were analyzed, and automated test execution reports were generated. The effectiveness of the developed testing system was also evaluated.

Keywords: REST API, automated testing, Java, TestNG, Rest Assured, Allure, HTTP, IntelliJ IDEA.

ЗМІСТ

ВСТУП

У сучасному світі, де веб-технології стрімко розвиваються, REST API стали ключовим інструментом для організації взаємодії між окремими частинами програмних систем. Вони дозволяють ефективно поєднувати клієнтські та серверні компоненти, забезпечуючи швидкий обмін даними у масштабованих інформаційних середовищах. З огляду на широке використання мікросервісної архітектури, особливо важливо мати надійну систему тестування REST API, яка дозволить виявити помилки ще до розгортання продукту та зменшити ризики, пов’язані з некоректною взаємодією між сервісами. Актуальність теми полягає в тому, що тестування REST API є критично важливим етапом у забезпеченні якості сучасних програмних продуктів. Правильна побудова системи тестування дозволяє значно зменшити витрати часу на пошук та усунення помилок, забезпечити стабільність роботи сервісів та підвищити загальну надійність інформаційної системи.

Метою кваліфікаційної роботи є створення та оптимізація системи тестування REST API з використанням апаратно-програмних засобів. Рішення повинно забезпечити ефективну перевірку коректності взаємодії сервісів у рамках мережевої архітектури. Для реалізації поставленої мети обрано мову програмування Java та бібліотеки TestNG, Rest Assured, Allure, а також систему збірки Maven. Тестове середовище створюється у середовищі розробки IntelliJ IDEA.

Об’єктом дослідження є мережева інфраструктура сучасних інформаційних систем, яка забезпечує взаємодію між окремими сервісами. Предметом дослідження виступає архітектура REST API та методи її тестування в умовах програмної взаємодії між сервісами.

Для досягнення поставленої мети в межах кваліфікаційної роботи необхідно виконати такі завдання: реалізувати систему тестування REST API із застосуванням сучасних інструментів, розробити архітектуру фреймворку, що дозволяє централізовано керувати конфігурацією, логуванням та викликами API,

дослідити особливості REST API, форматів обміну даними (JSON, XML), принципів HTTP-взаємодії та видів тестування, візуалізувати результати виконаних тестів за допомогою інтеграції з Allure, спроектувати структуру проєкту, яка забезпечує масштабованість та повторне використання коду.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ПОБУДОВИ ТА ТЕСТУВАННЯ API

1.1 Поняття API: визначення, класифікація, призначення

API(Application Programming Interface) – це набір визначених правил і протоколів, що дозволяють окремим компонентам або системам взаємодіяти між собою. На рисунку 1.1 зображено, як працює клієнт-серверна архітектура.

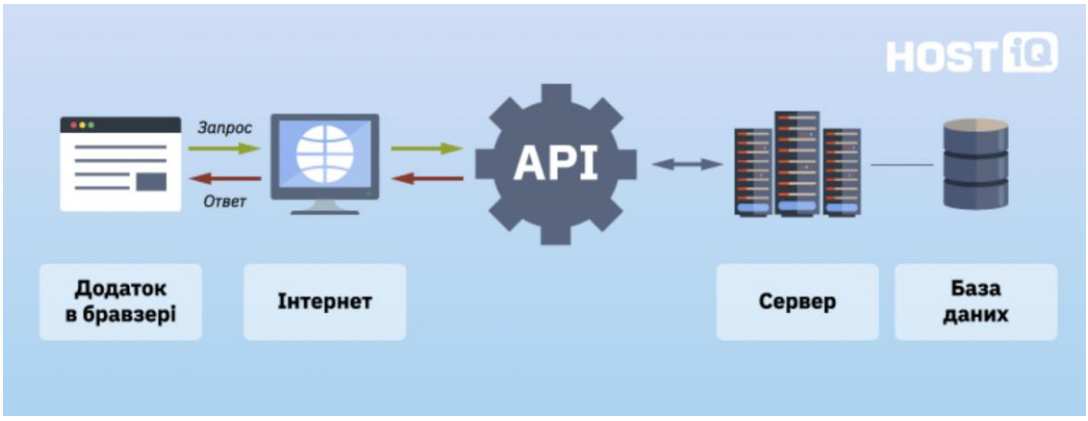

Рисунок 1.1 – Схема роботи API [1]

Використання API має низку вагомих переваг. Економічна доцільність полягає в тому, що розробники можуть використовувати вже готові інтерфейси, не витрачаючи час та ресурси на реалізацію базової функціональності з нуля. Це дозволяє скоротити строки розробки, зменшити витрати на проєктування та тестування, а також зосередити увагу на розв’язанні більш складних бізнес-завдань. Передбачуваність та уніфікація користувацького досвіду забезпечуються завдяки повторному використанню знайомих API-функцій. Користувачам не потрібно адаптуватися до нових принципів роботи кожного разу, оскільки багато дій та логік стають стандартними – наприклад, форми входу, кнопки «Подобається», кошики покупок тощо. Потенціал для монетизації API відкриває нові бізнес-моделі для розробників. Замість того, щоб надавати інтерфейси у відкритому доступі, компанії можуть пропонувати API як окремий продукт на комерційній основі. За умови наявності попиту, це дозволяє

отримувати стабільний прибуток від використання API сторонніми розробниками чи компаніями.

Простими словами, API – це посередник між програмами, який задає правила «спілкування». Тому в його назві закладено поняття інтерфейс – межа між об'єктами. Одній програмі не важливо, як працює інша. Розробники просто користуються інтерфейсом: надсилають запит і отримують зворотну відповідь [1].

1.2 Види та архітектура API

Залежно від рівня доступу та цільового призначення, API класифікуються на кілька основних типів:

1) публічні (відкриті) API – це інтерфейси, доступ до яких можуть отримати стороні розробники без обмежень. Такі API створюються з метою популяризації продукту або комерціалізації через платний доступ. Прикладом є API сервісу Google Maps;

2) приватні (внутрішні) API – використовуються виключно всередині однієї організації. Їх застосовують для інтеграції внутрішніх систем, сервісів або мікросервісів, які не мають бути доступними за межами компанії. Такий підхід підвищує безпеку, контроль над архітектурою та гнучкість у розробці внурішніх рішень;

3) партнерські API – надаються обмеженому колу зовнішніх партнерів або клієнтів на основі угод про співпрацю. Вони створюються з метою забезпечення взаємодії між системами компаній, які мають бізнес-зв'язки. Наприклад, API для обміну даними між внутрішньою базою даних підприємства та CRM-системою або поштовим сервісом зовнішнього постачальника.

У сучасому світі виділяють три основні типи архітектур API: RPC, REST та SOAP. Їх часто називають форматами або стилями API, оскільки кожен із них має власні особливості, підходи до побудови запитів та сфери застосування.

RPC (Remote Proceduce Call) – передбачає виклик віддалених процедур , що імітують виклики локальних функцій. Підходить для простих дій, але менш гнучкий у маштабованих розподілених системах.

SOAP (Simple Object Access Protocol) – орієнтований на суворо структуровану взаємодію за допомогою XML. Часто використовується в корпоративних системах, де критично важливі безпека, транзакційність та відповідність стандартам.

REST (Representational State Transfer) – найпоширеніший архітектурний стиль, що базується на HTTP-протоколі та використовує стандартизовані методи (GET, POST, PUT, DELETE). Відзначається простотою, масштабованістю та гнучкістю у використанні.

1.3 Принципи роботи Rest API та взаємодія клієнт-сервер

Rest API – це інтерфейс, який забезпечує взаємодію між клієнтською та серверною частинами веб-додатків. Він реалізується поверх протоколу HTTP (HyperText Transfer Protocol) – основного протоколу для обміну інформацією у веб-середовищі. HTTP працює за моделлю запит-відповідь, де клієнт надсилає запит, а сервер формує й повертає відповідь.

Коли користувач взаємодіє з веб-застосунком, наприклад, заповнює форму або переглядає сторінку з даними, ці дії ініціюють HTTP запити до сервера. Rest Api виступає у ролі посередника, який стандартизує такі запити та забезпечує доступ до ресурсів у межах веб-додатку. Типовий веб-застосунок складається з таких компонентів: frontend (клієнтська частина) – відповідає за інтерфейс користувача і backend (серверна частина) виконує бізнес-логіку, обробляє запити, працює з базою даних.

REST API виступає як мостовий інтерфейс між цими частинами. Наприклад, при натисканні користувачем кнопки «Переглянути замовлення», frontend формує HTTP-запит типу GET на адресу /api/orders. Сервер, отримавши

цей запит, обробляє його, виконує необхідні операції (наприклад, витягує дані з бази) та повертає відповідь у форматі JSON.

Основні принципи REST:

– клієнт-серверна архітектура: клієнт і сервер є незалежними компонентами, що дозволяє розвивати їх окремо, а також підвищує масштабованість і підтримуваність;

– відсутність стану (Stateless): сервер не зберігає інформацію про стан клієнта між запитами,кожен HTTP-запит містить всю необхідну інформацію для його обробки;

– кешованість (Cacheable): відповіді можуть містити інформацію, що дозволяє клієнту кешувати ресурси та зменшувати навантаження на сервер;

– універсальний інтерфейс (Uniform Interface): всі ресурси обробляються однаково через стандартизовані методи та URI-адреси, що спрощує взаємодію між системами та підвищує прозорість API;

– ієрархічна адресація ресурсів (URI): кожен ресурс у REST має унікальний і зрозумілий ідентифікатор у вигляді URI (Uniform Resource Identifier) [2].

HTTP (HyperText Transfer Protocol) – це протокол прикладного рівня, який використовується для обміну даними між клієнтом і сервером у мережі Інтернет. Він працює за принципом «запит-відповідь»: клієнт надсилає HTTP-запит, а сервер повертає відповідь із відповідним статус-кодом та вмістом.

У Rest API використовуються такі основні HTTP-методи:

– GET – отримання ресурів;

– POST – створення нових ресурсів;

– PUT – повне оновлення ресурсу;

– PATCH – часткове оновлення;

– DELETE – видалення ресурсу.

Кожна відповідь сервера супроводжується HTTP статус-кодом – це числовий код, який вказує на результат виконання запиту. Він є невіддільною

частиною Rest API, оскільки дозволяють клієнтській частині визначити, чи був запит успішним, чи сталася помилка.

Найпопулярніші статус-коди:

– 200 (OK) – запит успішно виконано;

– 201 (Created) – ресурс успішно створено;

– 204 (No Content) – успішне виконання без відповіді;

– 400 (Bad Request) – помилка клієнта, некоректний запит;

– 401 Unauthorized – відсутня або невірна авторизація;

– 403 Forbidden – доступ заборонено;

– 404 Not Found – ресурс не знайдено;

– 500 Internal Server Error – помилка на боці сервера.

1.4 Формати обміну даними в API

Rest API передбачає активний обмін даними між клієнтом і сервером. Для цього використовуються спеціальні формати, які повинні бути легкими для обробки, універсальними, читабельними як машиною, так і людиною. Найпоширенішими серед них є JSON та XML.

JSON (JavaScript Object Notation) – це текстовий файл, призначений для обміну структурованими даними між різними додатками. Завдяки своїй простоті та сумісності з більшістю мов програмування, JSON є найпопулярнішим форматом у контексті Rest API. Основна структура JSON – це колекція пар ключ-значення, де:

– ключ – це ім’я властивості;

– значення – може бути числовим, рядком, булевим значенням , масивом або іншим об’єктом [3].

На рисунку 1.2 зображено приклад JSON файлу.

Рисунок 1.2 – Приклад JSON , який містить інформацію про користувача [3]

У Rest API клієнт надсилає запит на сервер і, як відповідь отримує JSON- об'єкт із необхідними даними, які потім обробляються застосунокм.

XML (Extensible Markup Language) – це мова розмітки, яка використовується для зберігання та передачі структурованих даних. XML є більш формальним та описовим форматом у порівнянні з JSON і забезпечує розширені можливості, зокрема валідацію структури документа. На рисунку 1.3 зображено приклад XML файлу [4].

Рисунок 1.3 – Приклад XML [5]

У API основним форматом передачі даних є JSON, оскільки він простий, зручний у використанні та ефективний для взаємодії клієнт-сервер. Формат XML, хоча й менш поширений у відкритих API, все ще актуальний для міжсистемного

обміну даними, особливо у корпоративних та стандартизованих середовищах, де критично важлива валідація структури документа.

1.5 Основи тестування API

У сучасному процесі розробки програмного забезпечення тестування API є ключовим етапом забезпечення якості, особливо в системах з мікросервісною архітектурою, де велика кількість сервісів взаємодіють один з одним через інтерфейси. Тестування API дозволяє перевірити правильність логіки, стабільність відповіді, обробку помилок і відповідність специфікаціям – ще до того, як буде реалізовано користувацький інтерфейс.

Підходи до тестування API:

1) ручне тестування (Manual Testing) полягає у відправленні запитів до API за допомогою інструментів (наприклад, Postman, Insomnia) з подальшою перевіркою відповіді вручну. Цей підхід корисний для первинного дослідження API, перевірки сценаріїв або при відсутності автотестів;

2) автоматизоване тестування (Automated Testing) – здійснюється з використанням спеціалізованих фреймворків (Rest Assured, JUnit/TestNG, Karate, тощо), що дозволяє програмно надсилати запити, аналізувати відповіді, порівнювати результати з очікуваними і формувати звіти. Автоматизація підвищує продуктивність тестування, забезпечує повторюваність і зменшує людський фактор;

3) тестування контрактів (Contract Testing) – використовується для перевірки того, що API відповідає очікуваному контракту (опису структури запитів і відповідей). Це особливо важливо при взаємодії незалежних команд розробників клієнта і сервера;

4) мокування (Mocking) – дає змогу створити фіктивний сервер або заглушки, які імітують поведінку реального API. Це корисно для тестування залежностей без повної реалізації всіх компонентів.

Типи перевірок при тестуванні API:

1) функціональне тестування (Functional Testing) перевіряє, чи відповідає API вимогам, тобто, чи правильно обробляються запити, чи повертаються коректні дані, чи підтримуються всі очікувані сценарії взаємодії;

2) негативне тестування (Negative Testing) потрібно для того, щоб переконатися, що API правильно реагує на некоректні запити. Наприклад, неправильні параметри, відсутні обов'язкові поля, спроби доступу без авторизації тощо;

3) інтеграційне тестування (Integration Testing) – перевірка взаємодії API з іншими компонентами системи або сторонніми сервісами, наприклад, базою даних, платіжним шлюзом або email-сервером;

4) тестування продуктивності (Performance Testing) – вимірюється час відповіді сервера, стабільність роботи при великій кількості запитів, масштабованість API;

5) тестування безпеки (Security Testing) – перевірка авторизації, автентифікації, захисту від SQL-ін'єкцій, XSS, доступу до приватних ресурсів, перевірка заголовків CORS тощо.

Тестування API є критично важливим для забезпечення стабільності програмного забезпечення, особливо у високонавантажених та розподілених системах. Поєднання функціонального, негативного, інтеграційного та безпекового тестування дозволяє виявити широкий спектр помилок ще до етапу розгортання. Сучасні автоматизовані підходи дають змогу підвищити ефективність, повторюваність і прозорість перевірок.

1.6 Вибір архітектури мови та інструментарію для фреймворку

У межах реалізації фреймворку для автоматизованого тестування REST API прийнято рішення використовувати мову програмування Java та набір відповідних інструментів − Rest Assured, TestNG, Allure, а також SLF4J у поєднанні з Log4j2 для логування. Такий вибір зумовлений рядом технічних, функціональних та практичних переваг, що забезпечують ефективну побудову, виконання й аналіз тестів API.

Java була обрана як основна мова розробки через її стабільність, поширеність у сфері автоматизованого тестування, широкі можливості інтеграції та підтримку великої кількості бібліотек. Вона є кросплатформенною, має потужну екосистему та активну спільноту, що спрощує процес налагодження, пошуку рішень і розширення проєкту. Завдяки об’єктно-орієнтованому підходу Java дозволяє будувати модульну архітектуру тестів, що важливо для масштабованості системи. До того ж, Java підтримується у сучасних середовищах розробки, зокрема IntelliJ IDEA, у якому і здійснювалася реалізація проєкту.

Для безпосередньої роботи з HTTP-запитами використано бібліотеку Rest Assured, яка надає простий, зручний та лаконічний синтаксис для побудови REST-запитів та перевірки відповідей. Цей інструмент є спеціалізованим рішенням для тестування API саме на Java, що дозволяє уникати зайвого кодування при виконанні рутинних дій, таких як обробка заголовків, параметрів, авторизації чи JSON-тіл запитів. Завдяки Rest Assured стало можливим легко реалізовувати як функціональні, так і негативні тести, що дозволило максимально покрити можливі сценарії використання API.

Для організації структури тестів, їхнього групування та конфігурації застосовано TestNG – фреймворк, який забезпечує розширене управління тестовими наборами, підтримку параметризації, залежностей між тестами, а також налаштування умов ініціалізації та завершення (setup/teardown) [6].

Однією з ключових причин вибору TestNG стала його сумісність із Rest Assured і зручна інтеграція з Maven, що значно полегшує процес налаштування проєкту.

Щоб зробити результати тестування наочними та зручними для аналізу, у фреймворк інтегровано Allure Report систему для генерації HTML-звітів із деталізацією кроків тестів, відображенням результатів виконання, прикріпленням логів, параметрів запитів і відповідей. Allure дозволяє в реальному часі переглядати структуру тестів, їхній статус, час виконання та потенційні проблеми, що є надзвичайно корисним для валідації якості API.

Важливою частиною будь-якого автоматизованого фреймворку є система логування, яка допомагає відстежувати процес виконання тестів, виявляти помилки й діагностувати проблеми. Для логування в межах проєкту використано SLF4J зручний фасад для багатьох популярних систем логування, що забезпечує єдиний API і гнучкість у виборі реалізації. У даному фреймворку в якості конкретної реалізації використовується Log4j2, що дозволяє гнучко налаштовувати рівні логів, формати повідомлень, а також зберігати журнали виконання в окремому лог-файлі. Конфігураційний файл log4j2.xml розміщено в ресурсах проєкту й відповідає за формат виводу та збереження логів у структурованому вигляді [7].

Вибір Java у поєднанні з Rest Assured, TestNG, Allure, SLF4J і Log4j2 забезпечив гнучку, надійну та прозору платформу для реалізації системи тестування REST API, яка відповідає сучасним вимогам до автоматизації, масштабованості, моніторингу та звітності.

РОЗДІЛ 2
РОЗРОБКА ФРЕЙМВОРКУ ДЛЯ ТЕСТУВАННЯ REST API

2.1 Структура проєкту та вибір середовища роботи

Для реалізації фреймворку для тестування Rest API обрано середовище IntelliJ IDEA, яке забезпечує повну підтримку мови Java, інтеграцію з Maven, зручне керування залежностями, роботу з Git та візуальні засоби для налагодження й запуску тестів. Завдяки зручному інтерфейсу, розширеному автодоповненню, шаблонам коду та підтримці плагінів, IntelliJ IDEA значно спрощує процес розробки та супроводу автоматизованого фреймворку.

Проєкт створено як Maven-проєкт – це дозволяє керувати бібліотеками та залежностями централізовано за допомогою файлу pom.xml, а також забезпечує чітку структуру директорій. Такий підхід є стандартом для Java-проєктів, що підвищує гнучкість та масштабованість рішення.

На рисунку 2.1 зображено структуру фреймворка у вигляді системи каталогів , кожен з яких виконує певну функцію.

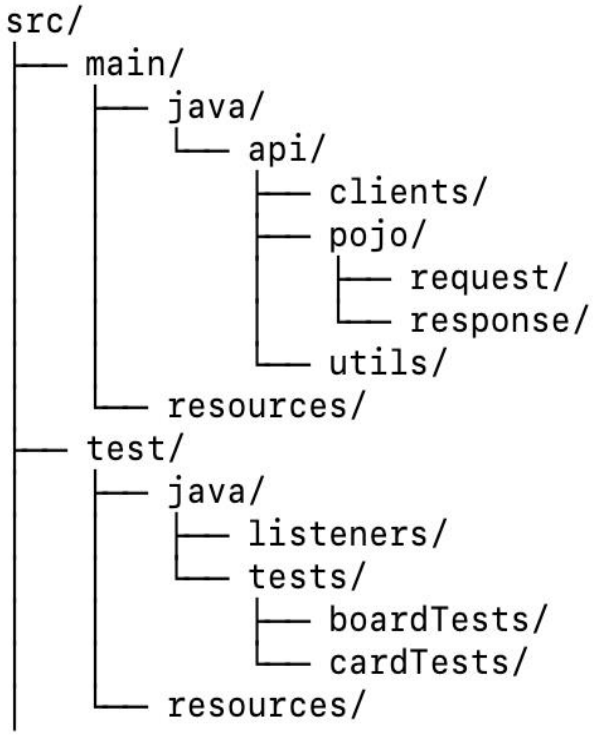

Рисунок 2.1 – Структура фреймворка

У каталозі src/main/java/api/ зосереджена основні компоненти фреймворку, що забезпечує взаємодію з API. Зокрема, у підкаталозі clients/ розміщуються класи, які реалізують HTTP-запити до відповідних ендпоінтів. Це дозволяє централізовано керувати викликами API та забезпечити повторне використання коду.

У папці pojo/ знаходяться модельні класи, що відповідають за структуру даних у форматі JSON. Ці класи поділено на дві підгрупи: request/ – для формування тіл запитів, і response/ – для зчитування та обробки відповідей. Такий поділ дозволяє уникати конфліктів між різними типами структур та забезпечує кращу читабельність коду [8].

Каталог utils/ містить допоміжні класи, що використовуються для конфігурації Rest Assured, зчитування параметрів, формування заголовків або базової ініціалізації клієнтів. Його створено з метою винесення повторюваної логіки з тестових або клієнтських класів.

Папка src/main/resources/ зберігає конфігураційні файли та налаштування логування. Вона використовується для зберігання параметрів, які мають бути доступні у середовищі виконання, зокрема конфігурацій логера Log4j2, API-ключів, базових URL та інших системних значень.

У каталозі src/test/java/ містяться всі тести, розділені на логічні блоки. У папці listeners/ розміщуються слухачі (listeners) в TestNG – спеціальні класи, які реагують на події під час виконання тестів (наприклад, початок, завершення, падіння). Це дозволяє інтегрувати додаткові функції, такі як логування, звітність або динамічні конфігурації.

Основні тести розміщено в папці tests/, яка містить два підкаталоги, що відповідають за окремі групи тестів. Така модульна побудова дозволяє зручно масштабувати фреймворк, ізолювати тест-кейси та уникати надмірної зв’язаності між модулями.

Папка src/test/resources/ призначена для конфігураційних файлів, пов’язаних із виконанням тестів – зокрема, конфігураційний XML-файл TestNG для визначення структури запуску, параметрів, груп тестів і слухачів.

Такий підхід до структурування дозволяє досягнути високого рівня модульності, повторного використання коду, зручності навігації та розширюваності фреймворку. Крім того, він відповідає загальноприйнятим практикам організації Java-проєктів у галузі автоматизованого тестування.

2.2 Підключення необхідних залежностей у файлі pom.xml

Одним із ключових етапів побудови автоматизованого фреймворку тестування REST API є підключення відповідних бібліотек і плагінів за допомогою системи керування залежностями Maven. Файл pom.xml виконує роль конфігураційного центру, в якому визначаються версії, бібліотеки, плагіни для компіляції, запуску тестів, формування звітів і логування.

Для реалізації фреймворку використано такі основні групи інструментів:

– Rest Assured – для виконання HTTP-запитів;

– TestNG – для структурування та запуску тестів;

– Allure – для генерації звітів;

– Log4j2 у зв’язці зі SLF4J – для логування;

– Jackson – для серіалізації/десеріалізації JSON;

– Lombok – для зменшення шаблонного коду.

А також допоміжні залежності, як-от Hamcrest і Typesafe Config.

На рисунку 2.2 зображено фрагмент коду, що підключає бібліотеку Rest Assured у pom.xml.

```
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <version>${restassured.version}</version>
</dependency>
```

Рисунок 2.2 – Фрагмент коду залежності Rest Assured

Rest Assured забезпечує зручний DSL для побудови HTTP-запитів (GET, POST, PUT, DELETE) та перевірки відповідей. Він у фреймворку використовується як основна бібліотека взаємодії з API [9].

На рисунку 2.3 зображено фрагмент коду, що підключає бібліотееку TestNJ у pom.xml.

```
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>${testng.version}</version>
    <scope>test</scope>
</dependency>
```

Рисунок 2.3 – Фрагмент коду залежності TestNJ

TestNG дає змогу структурувати тести, використовувати анотації @Test, @BeforeSuite, @AfterClass, групувати, запускати XML-наборами й інтегрувати слухачів (listeners).

На рисунку 2.4 зображено фрагмент коду , що підключає Allure у pom.xml.

```
<dependency>
    <groupId>io.qameta.allure</groupId>
    <artifactId>allure-testng</artifactId>
    <version>${allure-testng.version}</version>
</dependency>

<dependency>
    <groupId>io.qameta.allure</groupId>
    <artifactId>allure-java-commons</artifactId>
    <version>${allure-testng.version}</version>
</dependency>

<dependency>
    <groupId>io.qameta.allure</groupId>
    <artifactId>allure-rest-assured</artifactId>
    <version>2.29.0</version>
</dependency>
```

Рисунок 2.4 – Фрагмент коду залежності Allure

Allure інтегрується з TestNG і Rest Assured для створення деталізованих HTML-звітів. Вони містять повну інформацію про виконані тести, включаючи кроки зі сценаріїв, параметри запитів, відповіді сервера, тривалість виконання та статус кожного тесту. Завдяки підтримці анотацій @Step, звіт наочно відображає логіку виконання, що значно полегшує аналіз результатів і діагностику помилок.

На рисунку 2.5 зображено фрагмент коду , що дає змогу логування через SL4J і Log4j2 у pom.xml.

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>${log4j.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-slf4j2-impl</artifactId>
    <version>${log4j.version}</version>
</dependency>
```

Рисунок 2.5 – Фрагмент коду залежності SLF4J і Log4j2

Для логування фреймворк використовує SLF4J як фасад, а Log4j2 як конкретну реалізацію. Це дозволяє зручно реєструвати повідомлення як у консоль, так і у файл, що полегшує відстеження процесу виконання тестів і виявлення проблем. Конфігураційний файл log4j2.xml визначає формат логів, рівень деталізації, шлях до файлу журналу, а також структуру виводу, що забезпечує контроль над відображенням важливої інформації у зручному вигляді. Такий підхід підвищує прозорість процесу тестування та сприяє ефективній діагностиці помилок.

На рисунку 2.6 зображено фрагмент коду , що додає бібліотеку Jackson.

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>${jackson.version}</version>
</dependency>
```

Рисунок 2.6 – Фрагмент коду залежності Jackson

На рисунку 2.7 зображено інші залежності, які використано для оптимізації коду. Бібліотека Lombok (org.projectlombok:lombok) дозволяє автоматично генерувати шаблонний код, такий як гетери, сетери, конструктори, методи toString(), equals() та hashCode() завдяки спеціальним анотаціям (наприклад, @Data, @Builder, @NoArgsConstructor, @AllArgsConstructor). Це значно скорочує обсяг ручного коду, підвищує його читабельність і зменшує кількість помилок, пов’язаних із рутинною реалізацією допоміжних методів.

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>${lombok.version}</version>
    <scope>provided</scope>
</dependency>

<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest</artifactId>
    <version>2.2</version>
</dependency>
```

Рисунок 2.7 – Фрагмент коду залежності Typesafe Config , Lombook і Hamcrest

На рисунку 2.8 зображено фрагмент коду, плагіну у Maven. Він забезпечує запуск тестів на основі файлу testng.xml і збереження результатів у папку target/allure-results, звідки Allure формує HTML-звіт. Окрім цього, плагін maven-

surefire-plugin дозволяє налаштовувати параметри виконання тестів, наприклад, вивід логів, повторні запуски, використання системних властивостей тощо. Такий підхід дозволяє централізовано керувати процесом тестування та гарантує коректну інтеграцію з системами побудови і звітності.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.2.5</version>
    <configuration>
        <suiteXmlFiles>
            <suiteXmlFile>src/test/resources/testng.xml</suiteXmlFile>
        </suiteXmlFiles>
        <systemPropertyVariables>
            <allure.results.directory>${project.build.directory}/allure-results</allure.results.directory>
        </systemPropertyVariables>
    </configuration>
</plugin>
```

Рисунок 2.8 – Фрагмент кудо плагіна Surefire Plugin

Таким чином, файл pom.xml налаштований як центральний конфігураційний елемент, який повністю забезпечує роботу фреймворку: від виконання запитів до API та логування до створення звітів про результати тестів у зручному форматі. Кожна залежність була додана з урахуванням специфіки фреймворку, що дозволило досягти високої якості, читабельності та автоматизації процесу тестування REST API.

2.3 Організація конфігурації проєкту

Для забезпечення гнучкості та зручного управління параметрами тестового фреймворку конфігураційні значення винесено в окремий файл application.conf. Такий підхід дозволяє централізовано змінювати ключові параметри без потреби редагувати програмний код, що є рекомендованою практикою при розробці підтримуваних систем.

На рисунку 2.9 зображено файл application.conf, що розміщується в директорії src/main/resources/ і зчитується в момент ініціалізації тестового середовища. У ньому містяться значення, необхідні для взаємодії з API Trello:

– apiKey – унікальний ідентифікатор користувача, який дозволяє API Trello автентифікувати запити;

– apiToken – токен доступу, що разом із ключем використовується для авторизації запитів від імені користувача;

– baseUrl – базова адреса, до якої додаються ендпоінти API під час формування повного HTTP-запиту.

```
apiKey="9bc750cdbd48124b11a1456cbabb9fd3"
apiToken="ATTAff2e76e98e4ee2116c395e26ec355b728e68b8f54a193c5a295fdbbbb99087d8EA513534"

baseUrl="https://api.trello.com"
```

Рисунок 2.9 – Файл application.conf

Для забезпечення зручного доступу до конфігураційних параметрів, таких як ключ API, токен доступу та базова URL-адреса, у фреймворку реалізовано інтерфейс ConfigProvider (рис. 2.10), що розміщується в пакеті api.utils. Основним його призначенням є централізоване зчитування конфігураційних значень із зовнішнього конфігураційного файлу application.conf та надання до них статичного доступу з будь-якої частини проєкту.

Такий підхід дозволяє уникнути дублювання коду для зчитування параметрів у різних класах і забезпечує централізований доступ до конфігурацій. Це значно спрощує тестування, оскільки для зміни параметрів достатньо відредагувати лише один файл. При зміні середовища, наприклад із тестового на бойове, достатньо підставити відповідний конфігураційний файл без зміни логіки самого фреймворку. Така організація покращує підтримку проєкту та підвищує його гнучкість.

```java
package api.utils;

import com.typesafe.config.Config;
import com.typesafe.config.ConfigFactory;

public interface ConfigProvider {  2 usages

    Config config = readConf();  3 usages

    static Config readConf() { return ConfigFactory.load( resourceBasename: "application.conf"); }

    String API_KEY = config.getString( s: "apiKey");  1 usage
    String API_TOKEN = config.getString( s: "apiToken");  1 usage
    String BASE_URL = config.getString( s: "baseUrl");  2 usages
}
```

Рисунок 2.10 – Реалізований інтерфейс ConfigProvider

У межах цього інтерфейсу створено статичний метод readConf(), який використовує можливості бібліотеки Typesafe Config для завантаження конфігурацій. Зокрема, метод викликає ConfigFactory.load ("application.conf"), що дозволяє зчитати вміст зазначеного файлу та зберегти його у змінній config. Далі на основі цього об’єкта визначено три публічні константи: API_KEY, API_TOKEN та BASE_URL, які відповідно містять значення параметрів apiKey, apiToken і baseUrl, що зберігаються у файлі конфігурації.

Використання інтерфейсу як контейнера для конфігураційних значень дозволяє досягти максимальної простоти доступу до параметрів у будь-якому класі – достатньо імпортувати ConfigProvider і звернутися до відповідного поля. Такий підхід спрощує структуру коду, усуває потребу в дублюванні логіки зчитування параметрів у різних частинах фреймворку та сприяє підтримці принципів одного джерела істини (single source of truth).

Реалізація ConfigProvider є важливим елементом інфраструктури фреймворку, який забезпечує ефективну та безпечну роботу з конфігураційними даними та дозволяє зберігати їх незалежно від основного програмного коду.

Для централізованого формування запитів до Rest API у фреймворку реалізовано окремий базовий клієнт BaseRestTestClient, що використовується як

шаблон для створення запитів до різних ендпоінтів. Рисунок 2.11 демонструє повну реалізацію класу.

```
package api;

import io.qameta.allure.restassured.AllureRestAssured;
import io.restassured.RestAssured;
import io.restassured.filter.log.RequestLoggingFilter;
import io.restassured.filter.log.ResponseLoggingFilter;
import io.restassured.http.ContentType;
import io.restassured.specification.RequestSpecification;

import static api.utils.ConfigProvider.*;
import static io.restassured.RestAssured.given;

public class BaseRestTestClient {  2 usages  1 inheritor

    protected final RequestSpecification requestSpec;  6 usages

    public BaseRestTestClient(String url) {  1 usage
        RestAssured.filters(new RequestLoggingFilter(), new ResponseLoggingFilter());

        requestSpec = given().baseUri(url)
                .contentType(ContentType.JSON)
                .queryParam( s: "key", API_KEY)
                .queryParam( s: "token", API_TOKEN)
                .log().all()
                .filter(new AllureRestAssured());

        if (url.matches( regex: "^(https)://.*$")) {
            requestSpec.relaxedHTTPSValidation();
        }
    }
}
```

Рисунок 2.11 – Реалізація класу BaseRestClient

У наведеному фрагменті коду реалізовано базову конфігурацію REST-клієнта, яка охоплює декілька важливих компонентів. Насамперед додаються глобальні фільтри логування запитів і відповідей (RequestLoggingFilter та

ResponseLoggingFilter) до RestAssured, що дозволяє виводити HTTP-трафік у консоль під час виконання тестів. Далі створюється об’єкт RequestSpecification за допомогою методу given(), у якому задається базова адреса сервера (що передається через параметр конструктора), встановлюється заголовок Content-Type: application/json, а також додаються параметри автентифікації (key та token). Їхні значення зчитуються з конфігураційного класу ConfigProvider, який, у свою чергу, отримує їх із зовнішнього файлу application.conf. У цю ж специфікацію вбудовано інтеграцію з Allure через фільтр AllureRestAssured, що дозволяє автоматично виводити вхідні та вихідні HTTP-запити до тестових звітів. Крім того, реалізовано перевірку на використання HTTPS-протоколу: якщо передана адреса містить відповідну схему, до конфігурації requestSpec додається виклик relaxedHTTPSValidation(), що дозволяє ігнорувати помилки сертифікатів у тестовому середовищі. Завдяки такому підходу всі запити до API у фреймворку базуються на єдиній структурі, що підвищує зручність написання тестів, уніфікує конфігурацію запитів та забезпечує централізоване логування і звітність.

Для забезпечення прозорості виконання тестів, а також для зручного моніторингу запитів і відповідей, у рамках фреймворку реалізовано систему логування на основі бібліотеки Log4j2. Конфігурація цієї системи визначається у спеціальному XML-файлі log4j2.xml, який розміщується у каталозі ресурсів (src/main/resources/) та автоматично підвантажується під час запуску тестів. У даному файлі описано два основних типи виводу логів: у консоль та у текстовий лог-файл. Логи, що виводяться у консоль, форматуються у вигляді повідомлень із зазначенням часу, потоку виконання, рівня повідомлення, джерела логування та тексту повідомлення.

Окремо налаштовано вивід логів у файл target/logs/api-tests.log (рис. 2.12), де інформація записується з форматуванням за датою та часом, зберігаючи повний журнал виконання тестів. Запис у файл здійснюється у режимі додавання до існуючого вмісту, без перезапису, що забезпечує збереження попередніх логів. У конфігурації визначено кореневий логер із рівнем info, що дозволяє

фільтрувати повідомлення і виводити лише найважливішу інформацію – починаючи з інформаційних повідомлень і вище. Такий підхід дозволяє з одного боку – уникати зайвого шуму у звітах, а з іншого – зберігати повну картину виконання тестів для подальшого аналізу або налагодження. Завдяки використанню логування у фреймворку досягається краща контрольованість процесу тестування, спрощується пошук помилок та забезпечується можливість збереження історії запусків.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
        </Console>

        <File name="FileLogger" fileName="target/logs/api-tests.log" immediateFlush="false" append="true">
            <PatternLayout>
                <Pattern>%d{yyyy-MM-dd HH:mm:ss} [%t] %-5level %logger{36} - %msg%n</Pattern>
            </PatternLayout>
        </File>
    </Appenders>

    <Loggers>
        <Root level="info">
            <AppenderRef ref="Console"/>
            <AppenderRef ref="FileLogger"/>
        </Root>
    </Loggers>
</Configuration>
```

Рисунок 2.12 – Реалізація файлу log4j2.xml

Для організації та запуску тестових сценаріїв у фреймворку використовується конфігураційний файл testng.xml, який є стандартною частиною інфраструктури при використанні тестового фреймворку TestNG. Цей файл зберігається в директорії src/test/resources/ і використовується Maven- плагіном Surefire для автоматизованого виконання тестів під час збірки проєкту. У структурі testng.xml (рис. 2.13) визначено тестовий набір під назвою API Test Suite, у межах якого виконується один тестовий блок Trello API Tests. Усередині цього блоку вказано, що потрібно виконати всі тести, які знаходяться у пакеті api.tests, де сконцентровано основні класи з перевірками. Рівень деталізації виконання тестів задається через атрибут verbose="1", що дозволяє

отримати стислий, але інформативний лог про хід запуску. Паралельне виконання тестів відключено (через параметр parallel="false"), що дає змогу запускати перевірки послідовно та уникнути конфліктів при роботі з API, які не підтримують паралельні запити. Завдяки використанню testng.xml забезпечується централізований контроль над структурою запуску тестів, можливість гнучко змінювати конфігурацію без втручання в код, а також сумісність з плагінами Allure і Maven, що спрощує побудову звітності й автоматизацію.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd" >
<suite name="API Test Suite" verbose="1" parallel="false">
    <test name="Trello API Tests">
        <packages>
            <package name="api.tests"/>
        </packages>
    </test>
</suite>
```

Рисунок 2.13 – Вміст tentnj.xml файлу

2.4 Створення базової структури тестів

Клас TestInit (рис. 2.14) відіграє важливу роль у структурі тестового фреймворку та реалізує механізм базової ініціалізації для всіх тестових класів, що виконують перевірки роботи API. Він розміщується у пакеті tests і виступає у ролі базового батьківського класу, від якого успадковуються конкретні класи з тестами. Основне призначення TestInit – підготувати тестове середовище перед кожним тестовим методом, а також завершити перевірки після виконання кожного з них.

Ключовим елементом класу є метод setUp(), позначений анотацією @BeforeMethod з бібліотеки TestNG. Ця анотація вказує на те, що метод має виконуватися перед кожним тестом, який запускається в рамках сесії. У цьому

методі створюється новий об’єкт класу ApiBoardClient, що відповідає за взаємодію з ендпоінтами API, пов’язаними з операціями над дошками в системі Trello (наприклад, створення, видалення, оновлення тощо). Ініціалізація apiBoardClient у кожному тесті забезпечує ізольованість тестових сценаріїв, дозволяючи уникнути конфліктів між ними, а також унеможливлює повторне використання одного й того ж клієнта в кількох тестах, що може призвести до помилок або некоректного збереження стану.

```
package tests;

import api.clients.ApiBoardClient;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.asserts.SoftAssert;

public class TestInit {  2 inheritors
    protected ApiBoardClient apiBoardClient;  6 usages
    protected SoftAssert softAssert;  4 usages

    @BeforeMethod
    public void setUp(){
        apiBoardClient = new ApiBoardClient();
        softAssert = new SoftAssert();
    }

    @AfterMethod
    public void tearDown(){
        softAssert.assertAll();
    }
}
```

Рисунок 2.14 – Реалізація класу TestInit

Другим важливим елементом методу setUp() є створення об’єкта класу SoftAssert. На відміну від класичного підходу з використанням Assert, об’єкт SoftAssert дозволяє накопичувати всі помилки перевірок, не зупиняючи виконання тесту при першій невдачі. Це дає змогу отримати повну картину про всі відхилення у відповіді сервера або поведінці API в межах одного сценарію тестування. Таке рішення є особливо корисним у тих випадках, коли необхідно

перевірити декілька умов одночасно й побачити, які саме з них не виконалися, замість того, щоб переривати тест після першої помилки.

Після завершення кожного тестового методу викликається метод tearDown(), позначений анотацією @AfterMethod. У цьому методі виконується виклик softAssert.assertAll(), який запускає перевірку усіх накопичених умов і фіксує результат тесту. Якщо хоча б одна з перевірок не пройшла, тест вважається невдалим і відповідна інформація відображається у звіті. Такий підхід дозволяє гнучко контролювати результативність виконання кожного сценарію та зберігати максимум інформації про поведінку API при кожному запуску.

Використання класу TestInit у якості батьківського класу для всіх тестів дозволяє дотримуватися принципу інверсії управління та забезпечує централізовану ініціалізацію всіх необхідних об’єктів перед тестами. Це спрощує масштабування тестової бази, знижує кількість дублювання коду, підвищує читабельність і підтримуваність фреймворку. Додатково, у разі необхідності внесення змін до початкової логіки ініціалізації, достатньо змінити лише цей один клас – і всі похідні тести автоматично підлаштуються під нову логіку.

У процесі реалізації фреймворку відбувається взаємодія між локальною системою розробника та віддаленим сервером Trello за допомогою мережевого протоколу HTTP. Такий підхід відображає не лише програмну логіку тестування, але й апаратну взаємодію, оскільки серверна частина API працює на фізичних або віртуалізованих обчислювальних ресурсах. У цьому контексті система тестування REST API розглядається як апаратно-програмний комплекс, що функціонує у реальному мережевому середовищі [10].

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ТА ДЕМОНСТРАЦІЯ РОБОТИ ТЕСТІВ REST API

3.1 Перший тест створення і видалення дошки

Першим етапом практичної перевірки роботи створеного фреймворку є тестування одного з базових функціоналів API Trello – створення нової дошки з подальшим її видаленням. Цей сценарій дозволяє переконатися в коректній роботі API-ендпоінтів, що відповідають за створення ресурсів (POST) та їхнє видалення (DELETE). Також перевіряється, чи відбувається правильна передача параметрів авторизації, формування тіла запиту та коректна обробка відповіді.

Для забезпечення зручності підготовки вхідних даних та обробки відповідей при взаємодії з API, у межах фреймворку реалізовано дві спеціалізовані модельні структури – BoardBuilder (рис. 3.1) та BoardResponse (рис. 3.2), які розміщено у відповідних пакетах api.pojo.request і api.pojo.response. Вони відповідають за представлення JSON-структур у вигляді Java-об’єктів, що значно спрощує процес серіалізації й десеріалізації даних під час виконання тестів.

```java
package api.pojo.request;

import lombok.Builder;
import lombok.Data;

import static org.apache.commons.lang3.RandomStringUtils.randomAlphabetic;

@Data  no usages
@Builder
public class BoardBuilder {
    @Builder.Default
    private String name = "Board " + randomAlphabetic( count: 3);
    @Builder.Default
    private String desc = "Default description " + randomAlphabetic( count: 3);
    @Builder.Default
    private Boolean defaultLabels = true;
    @Builder.Default
    private Boolean defaultLists = true;
    @Builder.Default
    private String prefs_permissionLevel = "public";
    @Builder.Default
    private String prefs_voting = "public";
}
```

Рисунок 3.1 – Клас BoardBuilder

Клас BoardBuilder призначено для формування тіла запиту на створення нової дошки у Trello. Він побудований з використанням бібліотеки Lombok, що дозволяє скоротити кількість шаблонного коду за рахунок автоматичної генерації гетерів, сетерів і конструктора. За допомогою анотації @Builder реалізовано патерн будівельник, який дає змогу гнучко створювати об’єкти з різними параметрами. Поля name, desc, prefs_permissionLevel, prefs_voting, defaultLabels і defaultLists ініціалізуються типовими значеннями, до яких додаються випадкові алфавітні символи з метою унікалізації назв для кожного тесту. Таким чином, BoardBuilder дозволяє легко та програмно формувати тіло POST-запиту у вигляді об’єкта, який потім автоматично перетворюється у JSON при відправленні через Rest Assured.

```java
package api.pojo.response;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data  no usages
@JsonIgnoreProperties(ignoreUnknown = true)
@AllArgsConstructor
@NoArgsConstructor
public class BoardResponse {
    private String id;
    private String name;
    private String desc;
    private Boolean closed;
    private String idOrganization;
    private Boolean pinned;
    private String shortUrl;
}
```

Рисунок 3.2 – Клас BoardResponse

З іншого боку, клас BoardResponse використовується для відображення відповіді сервера після створення або оновлення дошки. Завдяки анотаціям @Data, @AllArgsConstructor і @NoArgsConstructor (також з Lombok) реалізовано повноцінну модель з усіма необхідними методами для зберігання даних. Крім того, анотація @JsonIgnoreProperties(ignoreUnknown = true) дозволяє ігнорувати всі зайві поля у відповіді, які не описані у класі, що робить його більш стійким до змін структури відповіді з боку API.

У подальших тестах обидва класи активно використовуються: BoardBuilder – при формуванні запиту на створення дошки, а BoardResponse – при зчитуванні результатів відповіді, перевірці коректності назви, статусу, ідентифікатора та інших параметрів. Таким чином, використання POJO-класів дозволяє побудувати читабельні, масштабовані та підтримувані тести, забезпечуючи чисте відокремлення логіки запиту і перевірок від низькорівневої роботи з JSON.

Наступним кроком реалізації першого тесту стало створення спеціалізованого клієнта ApiBoardClient, який інкапсулює логіку роботи з API Trello, зокрема з ресурсом дошка. Клас розміщується у пакеті api.clients та успадковується від раніше створеного базового класу BaseRestTestClient, що дозволяє використовувати спільну конфігурацію HTTP-запитів, зокрема базову адресу, авторизацію, логування та інтеграцію з Allure. Конструктор ApiBoardClient встановлює базовий шлях для всіх запитів (BASE_URL + "/1"), що відповідає актуальній версії API Trello.

У межах класу реалізовано два методи, кожен з яких відображає окрему дію над ресурсом дошка. Метод createNewBoard (рис. 3.3) відповідає за надсилання POST-запиту на створення нової дошки. Він приймає як аргументи об’єкт BoardBuilder, який містить тіло запиту у вигляді серіалізованого JSON, та очікуваний код відповіді (як правило, 200 або 201). У тілі методу викликається ланцюг Rest Assured: виконується встановлення тіла запиту, вказується метод POST, а далі – валідація статус-коду та десеріалізація відповіді у вигляді об’єкта BoardResponse. Завдяки цьому стає можливим зручно працювати з

ідентифікатором, назвою або іншими полями відповіді без необхідності додаткової обробки JSON вручну.

Другий метод (рис. 3.3) deleteBoard реалізує операцію видалення дошки за її ідентифікатором. Метод формує DELETE-запит до відповідного ендпоінту /boards/{id}, де id – це динамічна змінна. Також виконується перевірка відповідного статус-коду (expectedStatusCode), що дає змогу переконатися у коректному завершенні операції.

```
@Step("Create New Trello Board with name {boardBody.name}") 2 usages
public BoardResponse createNewBoard(BoardBuilder boardBuilder, int expectedStatusCode) {
    return requestSpec
            .body(boardBuilder) RequestSpecification
            .when()
            .post( s: "/boards") Response
            .then() ValidatableResponse
            .statusCode(expectedStatusCode)
            .extract() ExtractableResponse<Response>
            .as(BoardResponse.class);
}

@Step("Delete Trello board with ID: {boardID}. Expected status code {expectedStatusCode}") 1 usage
public void deleteBoard(String boardId, int expectedStatusCode) {
    requestSpec
            .when() RequestSpecification
            .delete( s: "/boards/" + boardId) Response
            .then() ValidatableResponse
            .statusCode(expectedStatusCode);
}
```

Рисунок 3.3 – Методи створенні і видалення дошки

Обидва методи позначено анотацією @Step з бібліотеки Allure, що дозволяє відображати кожен крок тесту у звіті з відповідним описом. Наприклад, у випадку створення дошки у звіті буде зазначено назву дошки, а при видаленні її унікальний ідентифікатор. Це значно покращує зручність аналізу звітів, особливо у разі падіння тестів.

Після створення клієнта ApiBoardClient та реалізації відповідних методів, на їх основі розроблено безпосередній тестовий сценарій, розміщений у класі CreateBoardAndDeleteTest. Цей клас розташовується у пакеті tests.boardTests і наслідує TestInit, що забезпечує попередню ініціалізацію об’єктів API-клієнта та

об’єкта типу SoftAssert. Така спадковість дозволяє уникнути дублювання коду та централізовано керувати базовими передумовами тестування.

У межах тесту реалізовано один метод – createBoard(), який виконує повний цикл перевірки роботи API: створення нової дошки, перевірку відповіді сервера, видалення створеної дошки та перевірку факту її видалення. На початковому етапі формується новий об’єкт BoardBuilder, що містить згенеровані значення для назви, опису та інших параметрів дошки. Цей об’єкт передається у метод createNewBoard(), який виконує запит до API та повертає відповідь у вигляді об’єкта BoardResponse.

Після отримання відповіді виконується перевірка за допомогою м’якого твердження (softAssert.assertNotNull()), що ідентифікатор новоствореної дошки не є null, тобто вона успішно створена. Наступним кроком є видалення цієї ж дошки через метод deleteBoard(), після чого викликається допоміжна перевірка assertBoardDeleted(), яка гарантує, що ресурс був коректно видалений і більше не існує у системі Trello.

Для підтвердження результатів виконання тесту здійснено додаткову перевірку у візуальному інтерфейсі Trello для створення (рис. 3.4) і видалення дошки (рис. 3.5).

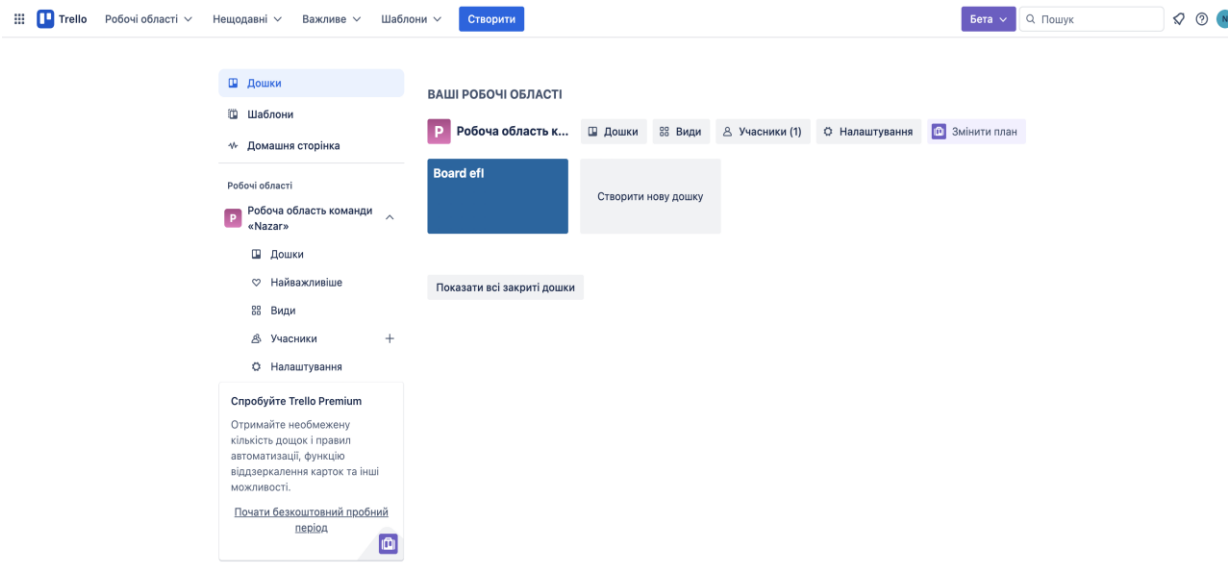

Рисунок 3.4 – Створена дошка за допомогою POST запиту

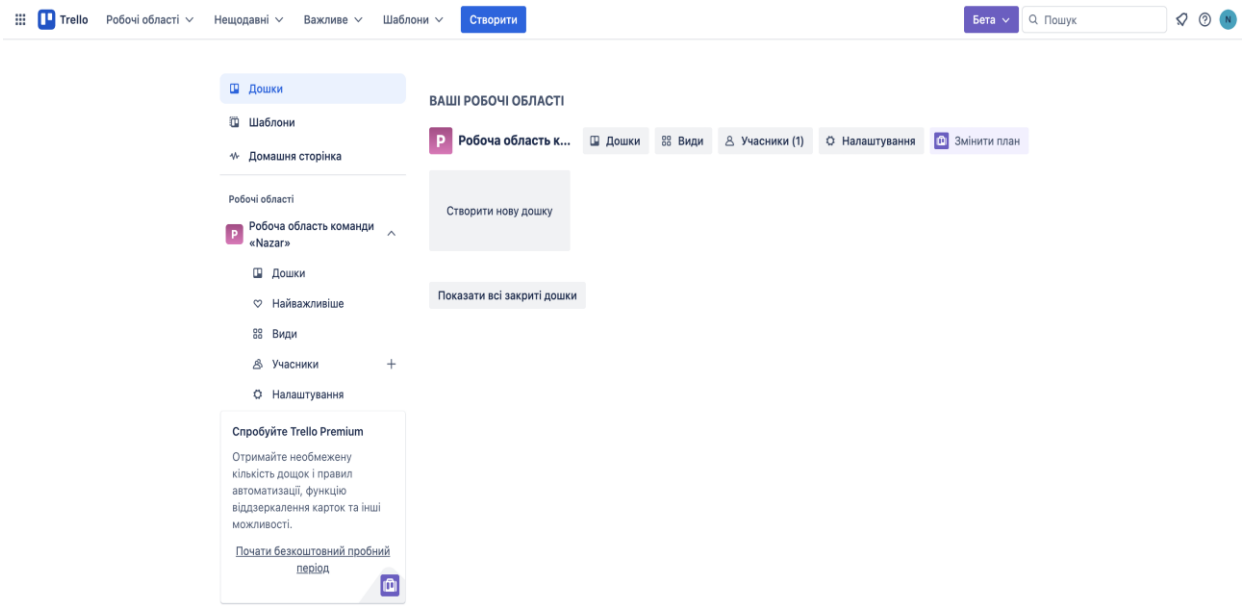

Рисунок 3.5 – Видалена дошка за допомогою DELETE запиту

Під час виконання першого методу createNewBoard() (рис. 3.6) на фізичному рівні відбувається обмін даними між двома апаратними об’єктами – локальною обчислювальною системою (ноутбуком користувача) та віддаленим сервером Trello. Ноутбук виконує роль клієнтського пристрою, на якому запущене середовище розробки IntelliJ IDEA та реалізований фреймворк автоматизованого тестування. У момент запуску тесту формується HTTP-запит, який проходить через мережевий стек операційної системи, конвертується у TCP/IP-пакети та передається мережею Ethernet або Wi-Fi до маршрутизатора.

Далі ці пакети передаються мережею Інтернет до фізичної серверної інфраструктури Trello – кластеру серверів, розташованих у хмарному дата-центрі, які виконують роль обробника REST API запитів. Сервер отримує запит, обробляє його на рівні прикладного програмного забезпечення, зберігає відповідні дані в базі та формує відповідь з кодом 201 Created. Ця відповідь знову пакується в TCP/IP, передається мережею до клієнтського пристрою та обробляється на рівні фреймворку, що працює на ноутбуці. Таким чином, реалізовано повноцінний обмін між двома апаратно незалежними вузлами в моделі клієнт-сервер, що відповідає концепціям комп’ютерних мереж і розподілених обчислень.

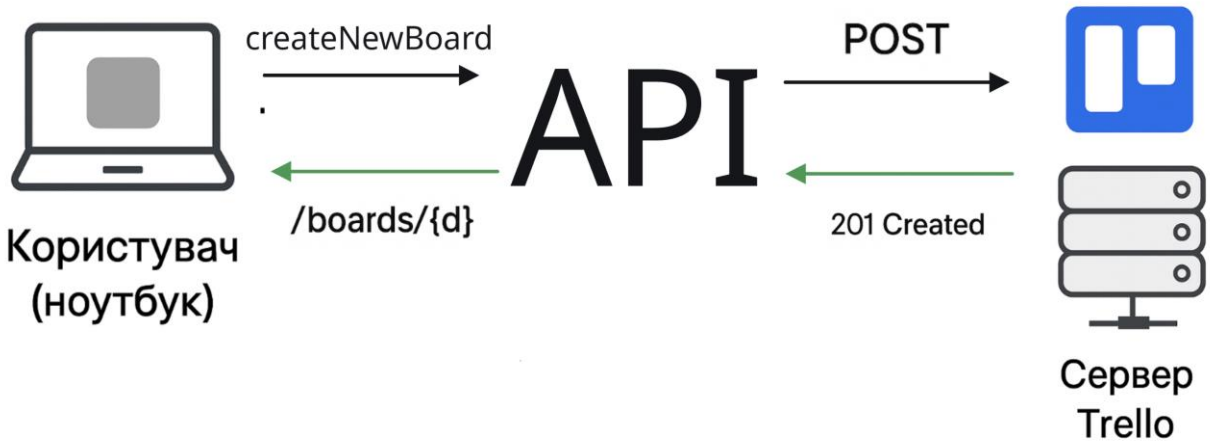

Рисунок 3.6 – Апаратна взаємодія між пристроєм та сервером Trello під час виконання методу createNewBoard()

Під час виконання другого методу (рис 3.7) deleteBoard() у фреймворку ініціюється запит на видалення раніше створеної дошки Trello за її унікальним ідентифікатором. З технічної точки зору, апаратна частина взаємодії полягає у тому, що користувацький пристрій (у нашому випадку ноутбук), на якому запущено тестовий фреймворк, надсилає HTTP-запит типу DELETE до API-сервера Trello.

Цей запит проходить мережевим маршрутом до віддаленого сервера, який фізично розміщено у хмарній інфраструктурі. Сервер Trello обробляє запит, перевіряє авторизаційні параметри, наявність запитуваного ресурсу, і за умови валідності надсилає відповідь зі статус-кодом 200 OK, що сигналізує про успішне видалення дошки.

На боці клієнта (ноутбука) ця відповідь обробляється фреймворком, і в тесті відбувається перевірка очікуваного статусу, що дозволяє підтвердити успішність операції. Така взаємодія ілюструє типову модель клієнт-серверного обміну в сучасних мережевих середовищах, де апаратне забезпечення локального пристрою використовується для ініціації, передачі й обробки HTTP-запитів до віддалених хмарних сервісів.

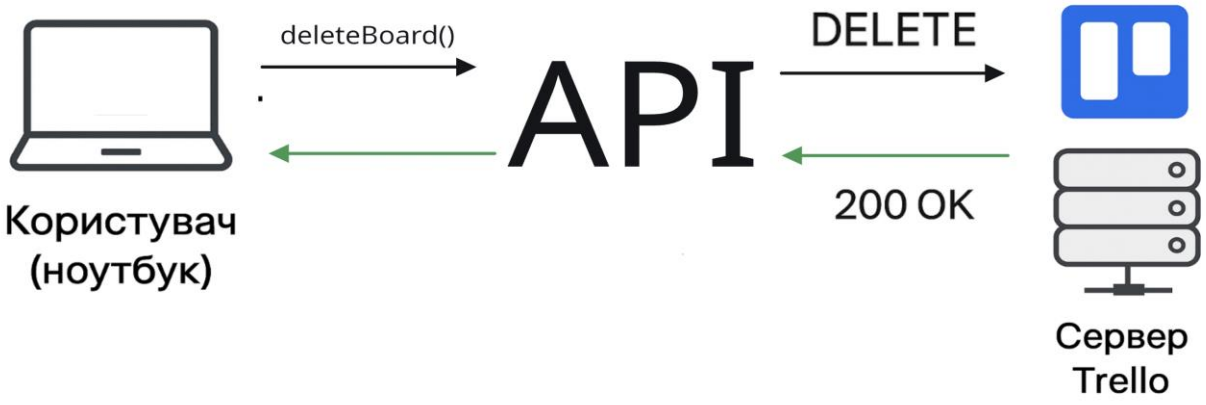

Рисунок 3.7 – Апаратна взаємодія пристроя та сервер Trello під час виконання методу deleteBoard()

3.2 Другий тест на додавання списку у дошку

Наступним етапом перевірки працездатності реалізованого фреймворку стало тестування функціональності додавання списку (list) до створеної дошки. Цей сценарій дозволяє перевірити коректність взаємодії з API Trello на рівні вкладених ресурсів, зокрема – створення залежного об’єкта типу список усередині вже наявного ресурсу дошка.

Для реалізації зазначеного сценарію клас ApiBoardClient доповнено новим методом (рис. 3.8).

```
@Step("Add list in the board with{boardId} .Expected status code {expectedStatusCode} ")
public void addList(String boardId, String name, int expectedStatusCode) {
    requestSpec
            .queryParam( s: "name", name) RequestSpecification
            .when()
            .post( s: "/boards/" + boardId + "/lists") Response
            .then() ValidatableResponse
            .statusCode(expectedStatusCode);
}
```

Рисунок 3.8 – Реалізація методу для додавання листа до дошки

Метод addList реалізує додавання нового списку до раніше створеної дошки Trello. Як показано на схемі, під час виклику цього методу користувач (із локального пристрою, наприклад ноутбука) формує HTTP POST-запит до ендпоінта /boards/{boardId}/lists, де boardId – унікальний ідентифікатор цільової дошки. Назва списку передається у вигляді параметра name.

Запит надсилається до API-сервера Trello (рис. 3.9), який обробляє його та, у разі успіху, повертає статус-код 201 Created, що підтверджує створення нового ресурсу. Даний код відповіді перевіряється у тесті як ознака коректної роботи ендпоінта. Таким чином, метод addList демонструє взаємодію клієнта з апаратною частиною – сервером Trello, яка забезпечує зберігання та обробку даних. Це підкреслює важливість серверної інфраструктури як компонента мережевої архітектури системи.

Для перевірки коректності роботи реалізованого методу addList() створено окремий тестовий клас CreateListInBoardTest. У межах цього тесту відбувається повний сценарій взаємодії з API Trello створення нової дошки, додавання до неї списку та перевірка валідності ключових параметрів відповіді. Наведений нижче фрагмент коду демонструє реалізацію тестового методу (рис. 3.9).

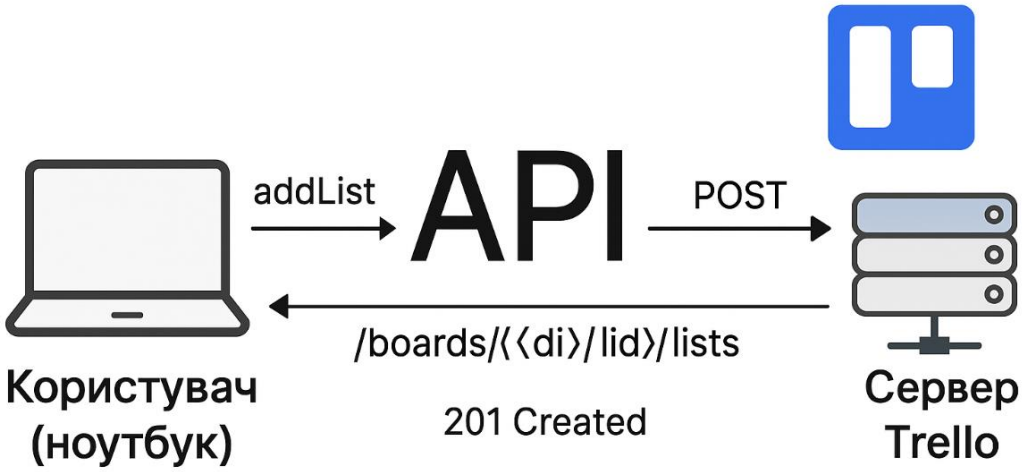

Рисунок 3.9 – Апаратна взаємодія пристроя та сервер Trello під час виконання методу addList

Тест виконується у кілька етапів. Створюється об’єкт BoardBuilder, який генерує унікальні дані для нової дошки (назву, опис, параметри видимості тощо). За допомогою клієнта ApiBoardClient викликається метод createNewBoard(), який надсилає POST-запит на створення дошки. Після отримання відповіді виконується перевірка на наявність ідентифікатора (boardResponse.getId()), що підтверджує успішне створення ресурсу. Далі, на основі ідентифікатора створеної дошки, виконується виклик методу addList(), який додає список із назвою, ідентичною назві дошки.

Таким чином, у рамках одного тесту перевіряється не лише створення основного ресурсу, а й успішна взаємодія з вкладеним ресурсом – списком. Крім автоматизованих перевірок, також здійснено ручну перевірку результатів на офіційному веб-сайті Trello. Після виконання тесту здійснено авторизований вхід у систему, де візуально підтверджено наявність створеної дошки та доданого до неї списку. Це дозволило переконатися в коректності обробки запиту сервером і фактичному створенні об’єктів у графічному інтерфейсі користувача.

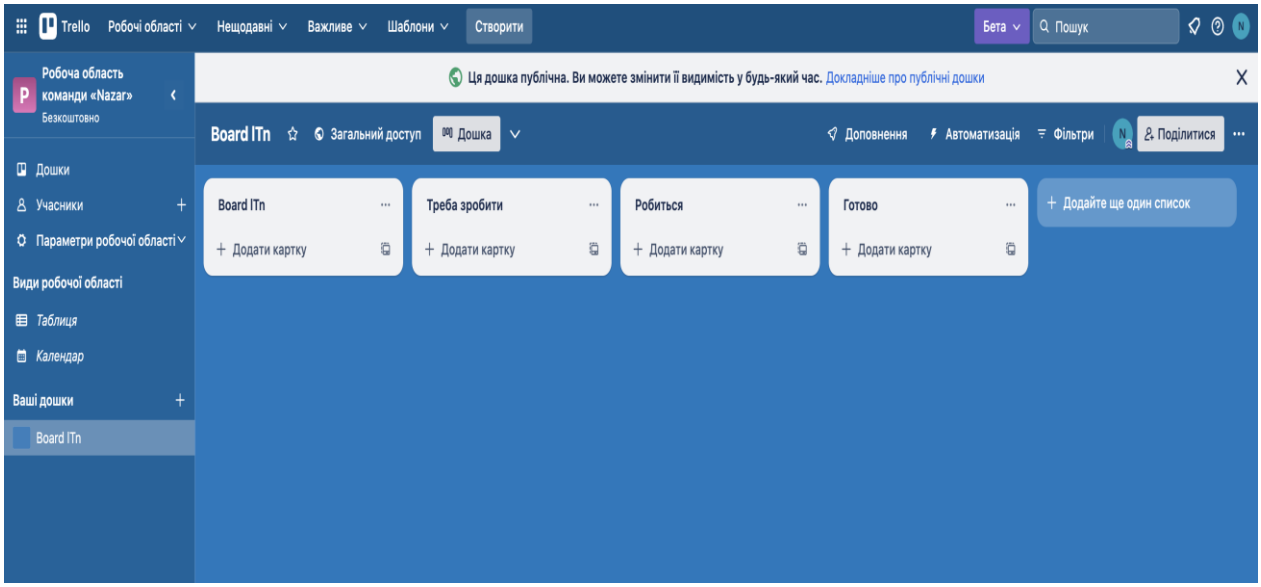

Рисунок 3.10 – Створена дошка зі списком

3.3 Генерація автоматичного звіту Allure

Після виконання автоматизованих тестів одним із ключових етапів є формування наочного та структурованого звіту, який відображає результати кожного кроку, використовувані дані, очікувану та фактичну поведінку API. Для цього у фреймворку інтегровано систему звітності Allure – популярний інструмент для побудови гнучких і візуально зручних тестових звітів. У процесі виконання тестів бібліотека Allure автоматично формує проміжні результати у вигляді .json файлів у директорії target/allure-results. Для генерації повного HTML-звіту на основі цих файлів необхідно виконати два послідовних кроки за допомогою Maven. Для запуску тестів і очищення проєкту потрібно у консоль вести цю команду mvn clean test. Ця команда очищає попередні результати (clean) і виконує тести (test). Після її завершення в каталозі target створюється структура даних для Allure.

Далі використовується спеціальний плагін Allure, інтегрований у систему збірки Maven. Після запуску тестів, результати зберігаються у форматі, що підтримується Allure, і на основі цих даних формується HTML-звіт, який можна переглядати у веб-браузері (рис. 3.11).

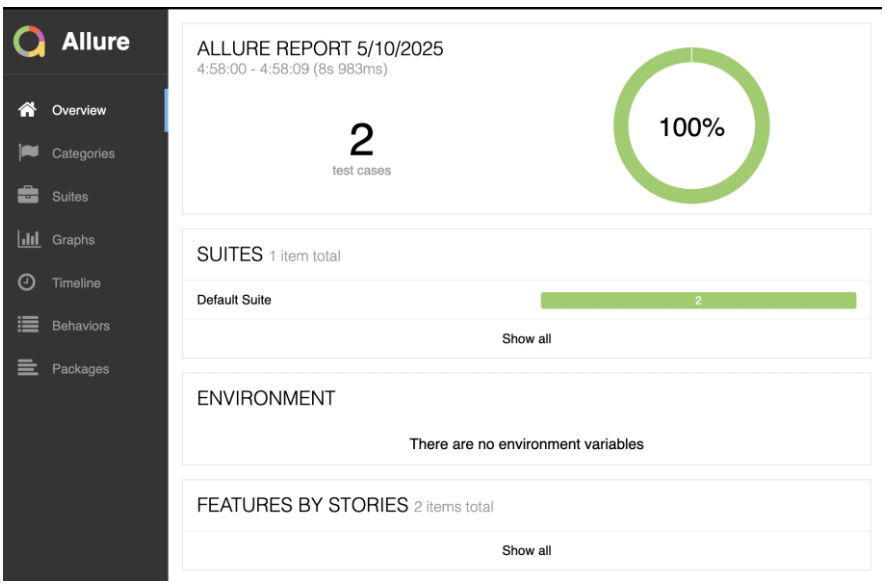

Рисунок 3.11 – Звіт Allure

У цьому звіті для кожного тесту доступна така інформація, як статус виконання (успішно/неуспішно), очікувані та фактичні значення, детальний опис кожного кроку з використанням анотацій @Step, час виконання, а також HTTP-запити та відповіді при інтеграції з REST API.

Таким чином, звіт Allure забезпечує повну наочність процесу тестування, значно спрощує аналіз помилок і дозволяє формувати технічну документацію високої якості для супроводу та передачі проєкту.

ВИСНОВКИ

За результатами виконання кваліфікаційної роботи досягнуто поставлену мету – розроблено та оптимізовано систему автоматизованого тестування REST API з використанням сучасних апаратно-програмних засобів. Робота включала теоретичне обґрунтування, побудову фреймворку, реалізацію тестів та генерацію звітності.

Здійснено аналіз основних понять, класифікацій та архітектурних стилів API. Розглянуто особливості протоколу HTTP, форматів обміну даними (JSON, XML), а також підходи до тестування REST API. Це дозволило сформувати глибоке розуміння принципів взаємодії клієнт-сервер і необхідних елементів тестування в умовах мікросервісної архітектури.

Спроектовано та реалізовано фреймворк для тестування REST API з використанням мови програмування Java, системи збірки Maven, бібліотеки Rest Assured, фреймворку TestNG, засобів логування SLF4J та Log4j2, а також системи генерації звітів Allure. Створена архітектура забезпечує зручну структуру, централізовану конфігурацію, повторне використання компонентів і можливість масштабування.

Виконано реалізацію низки API-тестів, які перевіряють базові сценарії роботи з Trello API: створення дошки, додавання списку, видалення ресурсів, перевірка статус-кодів та обробки відповіді. Для кожного запиту були застосовані відповідні методи HTTP (POST, DELETE), налаштування параметрів авторизації та серіалізація/десеріалізація даних через POJO-класи.

Візуалізовано результати виконання тестів за допомогою Allure-звітів. Звіти містять детальну інформацію про кожен тестовий сценарій, кроки з анотаціями, вхідні дані, статус виконання, час та HTTP-лог. Це дозволяє зручно аналізувати помилки та підтверджувати правильність функціонування API на всіх етапах тестування.

Як бачимо з отриманих результатів, розроблена система демонструє високу ефективність і стабільність при виконанні тестів. Вона може бути

використана як основа для побудови автоматизованого контролю якості REST API у будь-якому проєкті, що передбачає інтеграцію через HTTP-запити.

На основі проведеної роботи можемо зробити висновок, що запропоноване рішення відповідає сучасним вимогам до автоматизованого тестування, є гнучким, масштабованим та придатним до інтеграції в загальний процес розробки ПЗ.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Що таке API: простими словами про складне | Блог хостера HOSTiQ.ua. Блог хостера HOSTiQ.ua. URL: https://surl.li/uxtqdv (дата звернення: 10.03.2025).

2. Що таке rest api: основні принципи та практики застосування. FoxmindEd. URL: https://surli.cc/xyvbza (дата звернення: 14.03.2025).

3. Working with JSON - Learn web development | MDN. MDN Web Docs. URL: https://surl.li/qkukft (дата звернення: 24.03.2025).

4. JSON vs XML - Difference Between Data Representations - AWS. Amazon Web Services, Inc. URL: https://surl.lu/rpjxlj (дата звернення: 27.03.2025).

5. What is XML? - XML File Explained - AWS. Amazon Web Services, Inc. URL: https://surl.gd/xijwxl (дата звернення: 29.03.2025).

6. Testing це сучасний фреймворк для автоматизації тестування. FoxmindEd. URL: https://surl.li/lmdege (дата звернення: 3.04.2025).

7. Log4j – Apache Log4j 2 - Apache Log4j 2. Apache Logging Services. URL: https://surl.li/tsffmb (дата звернення: 08.04.2025).

8. GeeksforGeeks. POJO vs Java Beans - GeeksforGeeks. GeeksforGeeks. URL: https://surl.li/fvbrfy (дата звернення: 11.04.2025)

9. Rest assured. Cambridge Dictionary | English Dictionary, Translations & Thesaurus. URL: https://surl.lu/preoku (дата звернення: 13.04.2025).

10. REST Assured. REST Assured. URL: https://surli.cc/vcjlkc (дата звернення: 15.04.2025).

ДОДАТКИ

Додаток А

Консольний вивід під час першого автоматизованого тесту

```
Request method:   POST
Request URI:
      https://api.trello.com/1/boards?key=9bc750cdbd48124b11a1456cbabb9fd3&token=A
TTAff2e76e98e4ee2116c395e26ec355b728e68b8f54a193c5a295fdbbbb99087d8EA513534
Proxy:                  <none>
Request params:   <none>
Query params:     key=9bc750cdbd48124b11a1456cbabb9fd3

      token=ATTAff2e76e98e4ee2116c395e26ec355b728e68b8f54a193c5a295fdbbbb99087d8EA
513534
Form params:      <none>
Path params:      <none>
Headers:          Accept=*/*
                        Content-Type=application/json
Cookies:          <none>
Multiparts:       <none>
Body:
{
    "name": "Board ROE",
    "desc": "Default description Pfl",
    "defaultLabels": true,
    "defaultLists": true,
    "prefs_permissionLevel": "public",
    "prefs_voting": "public"
}
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 1692
Connection: keep-alive
Date: Sat, 10 May 2025 01:27:53 GMT
{
    "id": "681eab992f836c5f7b2353fc",
    "name": "Board ROE",
    "desc": "Default description Pfl",
    "descData": null,
    "closed": false,
    "idOrganization": "681b8fb64b9d29244549fcf9",
    "idEnterprise": null,
    "pinned": false,
    "url": "https://trello.com/b/p7QtN8dB/board-roe",
    "shortUrl": "https://trello.com/b/p7QtN8dB",
    "prefs": {
        "permissionLevel": "public",
        "hideVotes": false,
        "voting": "public",
        "comments": "members",
        "invitations": "members",
        "selfJoin": true,
        "cardCovers": true,
        "showCompleteStatus": true,
        "cardCounts": false,
        "isTemplate": false,
        "cardAging": "regular",
        "calendarFeedEnabled": false,
        "hiddenPluginBoardButtons": [

        ],
        "switcherViews": [
            {
```

```
                "viewType": "Board",
                "enabled": true
            },
            {
                "viewType": "Table",
                "enabled": true
            },
            {
                "viewType": "Calendar",
                "enabled": false
            },
            {
                "viewType": "Dashboard",
                "enabled": false
            },
            {
                "viewType": "Timeline",
                "enabled": false
            },
            {
                "viewType": "Map",
                "enabled": false
            }
        ],
        "autoArchive": null,
        "background": "blue",
        "backgroundColor": "#0079BF",
        "backgroundDarkColor": null,
        "backgroundImage": null,
        "backgroundDarkImage": null,
        "backgroundImageScaled": null,
        "backgroundTile": false,
        "backgroundBrightness": "dark",
        "sharedSourceUrl": null,
        "backgroundBottomColor": "#0079BF",
        "backgroundTopColor": "#0079BF",
        "canBePublic": true,
        "canBeEnterprise": true,
        "canBeOrg": true,
        "canBePrivate": true,
        "canInvite": true
    },
    "labelNames": {
        "green": "",
        "yellow": "",
        "orange": "",
        "red": "",
        "purple": "",
        "blue": "",
        "sky": "",
        "lime": "",
        "pink": "",
        "black": "",
        "green_dark": "",
        "yellow_dark": "",
        "orange_dark": "",
        "red_dark": "",
        "purple_dark": "",
        "blue_dark": "",
        "sky_dark": "",
        "lime_dark": "",
        "pink_dark": "",
        "black_dark": "",
        "green_light": "",
        "yellow_light": "",
```

```
        "orange_light": "",
        "red_light": "",
        "purple_light": "",
        "blue_light": "",
        "sky_light": "",
        "lime_light": "",
        "pink_light": "",
        "black_light": ""
    },
    "limits": {

    }
}
Request method:   DELETE
Request URI:
      https://api.trello.com/1/boards/681eab992f836c5f7b2353fc?key=9bc750cdbd48124
b11a1456cbabb9fd3&token=ATTAff2e76e98e4ee2116c395e26ec355b728e68b8f54a193c5a295fdb
bbb99087d8EA513534
Proxy:                  <none>
Request params:   <none>
Query params:     key=9bc750cdbd48124b11a1456cbabb9fd3

      token=ATTAff2e76e98e4ee2116c395e26ec355b728e68b8f54a193c5a295fdbbbb99087d8EA
513534
Form params:      <none>
Path params:      <none>
Headers:          Accept=*/*
                        Content-Type=application/json
Cookies:          <none>
Multiparts:       <none>
Body:
{
    "name": "Board ROE",
    "desc": "Default description Pfl",
    "defaultLabels": true,
    "defaultLists": true,
    "prefs_permissionLevel": "public",
    "prefs_voting": "public"
}
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 16
Connection: keep-alive
{
    "_value": null
}
Request method:   GET
Request URI:
      https://api.trello.com/1/boards681eab992f836c5f7b2353fc?key=9bc750cdbd48124b
11a1456cbabb9fd3&token=ATTAff2e76e98e4ee2116c395e26ec355b728e68b8f54a193c5a295fdbb
bb99087d8EA513534
Proxy:                  <none>
Request params:   <none>
Query params:     key=9bc750cdbd48124b11a1456cbabb9fd3

      token=ATTAff2e76e98e4ee2116c395e26ec355b728e68b8f54a193c5a295fdbbbb99087d8EA
513534
Form params:      <none>
Path params:      <none>
Headers:          Accept=*/*
                        Content-Type=application/json
Cookies:          <none>
Multiparts:       <none>
Body:
{
```

```
    "name": "Board ROE",
    "desc": "Default description Pfl",
    "defaultLabels": true,
    "defaultLists": true,
    "prefs_permissionLevel": "public",
    "prefs_voting": "public"
}
Request method:   GET
Request URI:
      https://api.trello.com/1/boards681eab992f836c5f7b2353fc?key=9bc750cdbd48124b
11a1456cbabb9fd3&token=ATTAff2e76e98e4ee2116c395e26ec355b728e68b8f54a193c5a295fdbb
bb99087d8EA513534
Proxy:                  <none>
Request params:   <none>
Query params:     key=9bc750cdbd48124b11a1456cbabb9fd3

      token=ATTAff2e76e98e4ee2116c395e26ec355b728e68b8f54a193c5a295fdbbbb99087d8EA
513534
Form params:      <none>
Path params:      <none>
Headers:          Accept=*/*
                        Content-Type=application/json
Cookies:          <none>
Multiparts:       <none>
Body:
{
    "name": "Board ROE",
    "desc": "Default description Pfl",
    "defaultLabels": true,
    "defaultLists": true,
    "prefs_permissionLevel": "public",
    "prefs_voting": "public"
}
HTTP/1.1 403 Forbidden
Server: CloudFront
Date: Sat, 10 May 2025 01:27:55 GMT
Content-Type: text/html
Content-Length: 915
Connection: close
X-Cache: Error from cloudfront
Via: 1.1 d9e9226e7f1bd505e314379bb60fd416.cloudfront.net (CloudFront)
X-Amz-Cf-Pop: WAW51-P3
X-Amz-Cf-Id: k9Nb4CZ7EiZwmyRu_yikz8wzRRRAfvcxJMhgiJTMYzUWsADzV2MTaA==

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"/>
    <title>ERROR: The request could not be satisfied</title>
  </head>
  <body>
    <h1>403 ERROR</h1>
    <h2>The request could not be satisfied.</h2>
    <hr noshade="noshade" size="1px"/>

Bad request.
We can't connect to the server for this app or website at this time. There might
be too much traffic or a configuration error. Try again later, or contact the app
or website owner.
    <br clear="all"/>

If you provide content to customers through CloudFront, you can find steps to
troubleshoot and help prevent this error by reviewing the CloudFront
documentation.
    <br clear="all"/>
    <hr noshade="noshade" size="1px"/>
```

```
    <pre>
Generated by cloudfront (CloudFront)
Request ID: k9Nb4CZ7EiZwmyRu_yikz8wzRRRAfvcxJMhgiJTMYzUWsADzV2MTaA==
</pre>
    <address/>
  </body>
</html>

================================================
Default Suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
================================================
```

Додаток Б

Консольний вивід під час другого автоматизованого тесту

```
}
Request method:   POST
Request URI:
      https://api.trello.com/1/boards?key=9bc750cdbd48124b11a1456cbabb9fd3&token=A
TTAff2e76e98e4ee2116c395e26ec355b728e68b8f54a193c5a295fdbbbb99087d8EA513534
Proxy:                  <none>
Request params:   <none>
Query params:     key=9bc750cdbd48124b11a1456cbabb9fd3

      token=ATTAff2e76e98e4ee2116c395e26ec355b728e68b8f54a193c5a295fdbbbb99087d8EA
513534
Form params:      <none>
Path params:      <none>
Headers:          Accept=*/*
                        Content-Type=application/json
Cookies:          <none>
Multiparts:       <none>
Body:
{
    "name": "Board Ajk",
    "desc": "Default description Ega",
    "defaultLabels": true,
    "defaultLists": true,
    "prefs_permissionLevel": "public",
    "prefs_voting": "public"
}
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 1692
Connection: keep-alive
{
    "id": "681eb45837deaa6f95025977",
    "name": "Board Ajk",
    "desc": "Default description Ega",
    "descData": null,
    "closed": false,
    "idOrganization": "681b8fb64b9d29244549fcf9",
    "idEnterprise": null,
    "pinned": false,
    "url": "https://trello.com/b/Fv7tCgzf/board-ajk",
    "shortUrl": "https://trello.com/b/Fv7tCgzf",
    "prefs": {
        "permissionLevel": "public",
        "hideVotes": false,
        "voting": "public",
        "comments": "members",
        "invitations": "members",
        "selfJoin": true,
        "cardCovers": true,
        "showCompleteStatus": true,
        "cardCounts": false,
        "isTemplate": false,
        "cardAging": "regular",
        "calendarFeedEnabled": false,
        "hiddenPluginBoardButtons": [

        ],
        "switcherViews": [
            {
```

```
                "viewType": "Board",
                "enabled": true
            },
            {
                "viewType": "Table",
                "enabled": true
            },
            {
                "viewType": "Calendar",
                "enabled": false
            },
            {
                "viewType": "Dashboard",
                "enabled": false
            },
            {
                "viewType": "Timeline",
                "enabled": false
            },
            {
                "viewType": "Map",
                "enabled": false
            }
        ],
        "autoArchive": null,
        "background": "blue",
        "backgroundColor": "#0079BF",
        "backgroundDarkColor": null,
        "backgroundImage": null,
        "backgroundDarkImage": null,
        "backgroundImageScaled": null,
        "backgroundTile": false,
        "backgroundBrightness": "dark",
        "sharedSourceUrl": null,
        "backgroundBottomColor": "#0079BF",
        "backgroundTopColor": "#0079BF",
        "canBePublic": true,
        "canBeEnterprise": true,
        "canBeOrg": true,
        "canBePrivate": true,
        "canInvite": true
    },
    "labelNames": {
        "green": "",
        "yellow": "",
        "orange": "",
        "red": "",
        "purple": "",
        "blue": "",
        "sky": "",
        "lime": "",
        "pink": "",
        "black": "",
        "green_dark": "",
        "yellow_dark": "",
        "orange_dark": "",
        "red_dark": "",
        "purple_dark": "",
        "blue_dark": "",
        "sky_dark": "",
        "lime_dark": "",
        "pink_dark": "",
        "black_dark": "",
        "green_light": "",
        "yellow_light": "",
```

```
        "orange_light": "",
        "red_light": "",
        "purple_light": "",
        "blue_light": "",
        "sky_light": "",
        "lime_light": "",
        "pink_light": "",
        "black_light": ""
    },
    "limits": {

    }
}
Request method:   POST
Request URI:
      https://api.trello.com/1/boards/681eb45837deaa6f95025977/lists?key=9bc750cdb
d48124b11a1456cbabb9fd3&token=ATTAff2e76e98e4ee2116c395e26ec355b728e68b8f54a193c5a
295fdbbbb99087d8EA513534&name=Board%20Ajk
Proxy:                  <none>
Request params:   <none>
Query params:     key=9bc750cdbd48124b11a1456cbabb9fd3

      token=ATTAff2e76e98e4ee2116c395e26ec355b728e68b8f54a193c5a295fdbbbb99087d8EA
513534
                        name=Board Ajk
Form params:      <none>
Path params:      <none>
Headers:          Accept=*/*
                        Content-Type=application/json
Cookies:          <none>
Multiparts:       <none>
Body:
{
    "name": "Board Ajk",
    "desc": "Default description Ega",
    "defaultLabels": true,
    "defaultLists": true,
    "prefs_permissionLevel": "public",
    "prefs_voting": "public"
}
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 182
Connection: keep-alive
{
    "id": "681eb4588f33669fc3f4b160",
    "name": "Board Ajk",
    "closed": false,
    "color": null,
    "idBoard": "681eb45837deaa6f95025977",
    "pos": 8192,
    "type": null,
    "datasource": {
        "filter": false
    },
    "limits": {

    }
}

================================================
Default Suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
================================================
```