

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проекту (роботи)

бакалавр

(ступінь вищої освіти)

на тему РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ
АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ API
SOFTWARE DEVELOPMENT FOR AUTOMATED API TESTING

Виконав(ла): студент(ка) 4 курсу, групи КНТ-121

Спеціальності 121 Інженерія програмного

(код і найменування спеціальності)

забезпечення

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

МУРАТОВ Д.С.

(ПРИЗВИЩЕ та ініціали)

Керівник ГЛАДКОВА О.М.

(ПРИЗВИЩЕ та ініціали)

Рецензент ДЬЯЧУК Т.С.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування закладу вищої освіти)

Факультет КНТ
 Кафедра програмних засобів
 Ступінь вищої освіти бакалавр
 Спеціальність 121 Інженерія програмного забезпечення
 (код і найменування)
 Освітня програма (спеціалізація) Інженерія програмного забезпечення
 (назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
 “ ” 2025 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

МУРАТОВА Дмитра Сергійовича
 (ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розробка програмного забезпечення для автоматизованого тестування API. Software Development for Automated API Testing

керівник проєкту (роботи) к.т.н., ГЛАДКОВА Ольга Миколаївна,
 (науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “ 28 ” квітня 2025 року № 209

2. Строк подання студентом проєкту (роботи) 02 червня 2025 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області та постановка завдання. 2. Дослідження та вибір інструментарію розробки. 3. Програмна реалізація API-тестів. 4. Результати та інструкції по використанню.

5. Перелік графічного матеріалу (з з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)

Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	ГЛАДКОВА О.М., доцент		
Нормоконтроль	КАЛІНІНА М.В., ст. викладач		

7. Дата видачі завдання « 14 » квітня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій розробки.	2 тиждень	Розділ 2
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3-4 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження програмного забезпечення.	5 тиждень	Розділ 4
7	Оформлення пояснювальної записки та документів до неї.	6 тиждень	Додатки
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студент(ка)

_____ Дмитро МУРАТОВ
(підпис) (Ім'я ПРИЗВИЩЕ)

Керівник проєкту (роботи)

_____ Ольга ГЛАДКОВА
(підпис) (Ім'я ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 70 с., 6 табл., 20 рис., 2 дод., 15 джерел.

API-ТЕСТУВАННЯ, АВТОМАТИЗАЦІЯ, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, ВЕБЗАСТОСУНОК, POSTMAN, JAVASCRIPT.

Об'єкт дослідження – процес автоматизованого тестування програмного забезпечення.

Предмет дослідження – створення та використання API автотестів для перевірки функціональності вебзастосунку.

Мета роботи – розробка програмного рішення для автоматизованого тестування API запитів для тестування вебзастосунків.

Матеріали, методи та технічні засоби: об'єктно-орієнтований підхід, REST-архітектура, інструмент Postman, мова програмування JavaScript (Postman scripting).

Результати. Реалізовано набір автотестів для API-запитів до вебзастосунку в середовищі Postman. Автотести перевіряють коректність відповідей сервера, статус-коди, структуру JSON-об'єктів і повідомлення про помилки.

Висновки. Розроблені автотести автоматизують процес API-тестування, знижують трудомісткість перевірок, підвищують швидкість проходження регресійних тестів та ефективність виявлення помилок, сприяє підтримці високої якості програмного продукту.

Галузь використання – контроль якості вебзастосунків у сфері програмної інженерії, зокрема під час інтеграційного та регресійного тестування на проектах із використанням REST API.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 70 pages, 6 tables, 20 figures, 2 appendixes, 15 sources.

API TESTING, AUTOMATION, SOFTWARE, WEB APPLICATION, POSTMAN, JAVASCRIPT.

Object of study is the process of automated software testing.

Subject of study – creation and use of API automated tests to verify the functionality of a web application.

The purpose of the work is to develop a software solution for automated API testing of web applications.

Materials, methods, and tools: object-oriented approach, REST architecture, Postman tool, JavaScript programming language (Postman scripting).

Results. A set of automated tests for API requests to a web application was implemented using Postman. The tests verify the correctness of server responses, status codes, JSON object structure, and error messages.

Conclusions. The developed automated tests streamline the API testing process, reduce the labor intensity of testing, increase the speed of regression testing and the efficiency of defect detection, and support the maintenance of high software quality.

The field of use is a quality assurance of web applications in the field of software engineering, particularly during integration and regression testing of projects utilizing REST APIs.

ЗМІСТ

	С.
Перелік скорочень та умовних позначок	8
Вступ.....	9
1 Аналіз предметної області та постановка завдання	11
1.1 Забезпечення якості вебзастосунків	11
1.2 API як об'єкт тестування: особливості, типи, стандарти REST	12
1.3 Види тестування API: ручне, автоматизоване, інтеграційне	14
1.4 Висновки за розділом та постановка завдання	16
2 Дослідження та вибір інструментарію розробки	18
2.1 Порівняння інструментів для API-тестування: Postman, REST Assured, SoapUI тощо	18
2.2 Огляд можливостей JavaScript-скриптів у Postman	19
2.3 Інструменти для запуску і звітності	20
2.4 Опис тестового об'єкта (вебзастосунку) та обраних API-запитів	21
2.5 Висновки за розділом 2	21
3 Програмна реалізація API-тестів	23
3.1 Архітектура API-тестів	23
3.2 Структура API-запитів	24
3.3 Реалізація тестів	25
3.4 Висновки до розділу 3	33
4 Результати та інструкції по використанню	34
4.1 Огляд результатів автотестування	34
4.3 Ризики, обмеження і переваги використаного підходу	35
4.4 Інструкція з запуску тестів вручну (через Postman)	35
4.5 Рекомендації щодо розширення або інтеграції у CI/CD	37
4.6 Висновки до розділу 4	38
Висновки	39
Перелік джерел посилання	41

Додаток А Текст програми.....	43
Додаток Б Слайди презентації	64

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API – Application Programming Interface;
IDE – Integrated Development Environment,
GUI – графічний інтерфейс користувача.

ВСТУП

У сучасному світі, де цифрові технології стрімко розвиваються, якість програмного забезпечення стає критично важливою. Особливо це стосується вебзастосунків, які активно використовують API (Application Programming Interface) для взаємодії між різними компонентами системи. Забезпечення надійності та стабільності таких API є ключовим завданням у процесі розробки програмного забезпечення.

Традиційні методи тестування, засновані на ручному виконанні тестів, не завжди ефективні в умовах швидкого циклу розробки та частих змін у коді. Вони можуть бути трудомісткими, схильними до помилок і не забезпечують необхідної швидкості зворотного зв'язку. У цьому контексті автоматизоване тестування API стає незамінним інструментом для розробників і команд забезпечення якості. Згідно з Katalon Studio, автоматизоване тестування API дозволяє швидше випускати продукти, забезпечує краще покриття тестами та знижує витрати на обслуговування [1].

Серед інструментів для автоматизованого тестування API особливе місце займає Postman. Цей інструмент надає зручний інтерфейс для створення, виконання та автоматизації тестів API. Використовуючи JavaScript для написання тестових скриптів, Postman дозволяє створювати гнучкі та потужні тести, які можуть бути інтегровані в процес безперервної інтеграції та доставки (CI/CD). Як зазначено на офіційному сайті Postman, платформа Postman включає кілька функцій для тестування API, які допомагають командам оптимізувати робочі процеси тестування та виявляти проблеми протягом усього життєвого циклу API [2].

Метою даної дипломної роботи є розробка програмного рішення для автоматизованого тестування API вебзастосунків з використанням Postman. У роботі буде розглянуто процес створення тестів, їх виконання.

Для досягнення поставленої мети в дипломній роботі передбачається виконання таких завдань:

- проаналізувати існуючі підходи та інструменти для автоматизованого тестування API;
- дослідити інструменти для написання автотестів;
- визначити набір API-запитів для тестування;
- розробити набір автотестів;
- виконати тестування API-запитів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1 Забезпечення якості вебзастосунків

У сучасному цифровому середовищі вебзастосунки відіграють ключову роль у забезпеченні взаємодії між користувачами та сервісами. Зростаюча складність таких застосунків, обумовлена використанням мікросервісної архітектури, інтеграцією з сторонніми сервісами та необхідністю забезпечення високої продуктивності, створює нові виклики для забезпечення їх якості. Забезпечення надійності, безпеки та стабільності вебзастосунків стає критично важливим завданням для розробників та команд забезпечення якості.

Однією з основних проблем є необхідність швидкого виявлення та усунення дефектів у ранніх стадіях розробки. Традиційне тестування через графічний інтерфейс користувача (GUI) часто є недостатньо ефективним, оскільки воно не дозволяє повністю охопити всі можливі сценарії взаємодії між компонентами системи. Крім того, GUI-тести можуть бути крихкими та вимагати значних зусиль для підтримки при зміні інтерфейсу [3].

У цьому контексті API-тестування виступає як ефективний підхід до забезпечення якості вебзастосунків. API (Application Programming Interface) дозволяє різним компонентам системи взаємодіяти між собою, і тестування цих інтерфейсів забезпечує перевірку логіки бізнес-процесів, незалежно від користувацького інтерфейсу. Це дозволяє виявляти помилки на рівні взаємодії між сервісами, що особливо важливо в умовах мікросервісної архітектури [4].

Крім того, API-тестування сприяє підвищенню ефективності процесу розробки, оскільки дозволяє автоматизувати перевірки та інтегрувати їх у процес безперервної інтеграції та доставки (CI/CD). Це забезпечує швидкий зворотний зв'язок для розробників та знижує ризик впровадження дефектів у продуктивне середовище [2].

Таким чином, впровадження API-тестування є важливим кроком у забезпеченні високої якості вебзастосунків, дозволяючи ефективно виявляти та усувати дефекти, забезпечувати стабільність системи та підвищувати задоволеність користувачів.

1.2 API як об'єкт тестування: особливості, типи, стандарти REST

API є інтерфейсом, який дозволяє різним програмним компонентам взаємодіяти між собою. У контексті вебзастосунків, API забезпечує зв'язок між клієнтською частиною (frontend) та серверною логікою (backend), що дозволяє реалізовувати складні функціональні можливості та забезпечувати гнучкість системи.

Основні особливості API як об'єкта тестування включають [5]:

- відсутність графічного інтерфейсу – API-тестування здійснюється на рівні передачі повідомлень, що дозволяє перевіряти функціональність без необхідності взаємодії з користувацьким інтерфейсом;
- стандартизовані протоколи та формати – більшість сучасних API використовують протокол HTTP/HTTPS та формати обміну даними JSON або XML, що спрощує процес тестування та інтеграції;
- підтримка різних методів запитів – API підтримує різні методи HTTP-запитів, такі як GET, POST, PUT, DELETE, що дозволяє реалізовувати різноманітні операції над ресурсами;
- REST-архітектура – більшість сучасних веб-API побудовані на основі REST (Representational State Transfer) архітектури, яка передбачає використання стандартних методів HTTP та забезпечує масштабованість і гнучкість системи.

Типи тестування API включають:

- функціональне тестування – перевірка того, чи API виконує свої функції відповідно до специфікації;

- інтеграційне тестування – перевірка взаємодії API з іншими компонентами системи;
- тестування безпеки – оцінка захищеності API від несанкціонованого доступу та вразливостей;
- тестування продуктивності – оцінка швидкодії та стійкості API під навантаженням;
- тестування надійності – перевірка здатності API відновлюватися після збоїв та обробляти помилки.

Порівняння API-тестування (через Postman) та GUI-тестування наведені в таблиці 1.1 [2], [6].

Таблиця 1.1 – Порівняння API-тестування через Postman та GUI-тестування

Критерій	API-тестування (Postman)	GUI-тестування
Рівень тестування	Логіка та взаємодія між сервісами	Кінцевий інтерфейс користувача
Швидкість виконання	Висока	Повільна (залежить від візуального рендерингу)
Залежність від UI	Незалежне	Повністю залежне
Стабільність	Висока (менше змін у запитах)	Низька (зміни в UI ламають тести)
Обсяг покриття	Бізнес-логіка, валідації, обробка даних	Відображення елементів, взаємодія з формами
Застосовність для автоматизації	Легко інтегрується в CI/CD	Потребує спеціальних інструментів (Selenium, Cypress)
Вартість підтримки	Низька	Висока
Зручність у написанні	Висока (особливо з використанням Postman scripting)	Нижча (потрібно відслідковувати зміни в DOM)

Таким чином, API-тестування є критично важливим елементом забезпечення якості вебзастосунків, оскільки дозволяє виявляти та усувати

дефекти на ранніх етапах розробки, забезпечувати стабільність системи та підвищувати задоволеність користувачів.

1.3 Види тестування API: ручне, автоматизоване, інтеграційне

Тестування API може здійснюватися різними методами, кожен з яких має свої переваги та обмеження [7].

Ручне тестування передбачає використання інструментів, таких як Postman або curl, для надсилання запитів до API та аналізу відповідей. Цей метод є корисним на початкових етапах розробки або для одноразових перевірок, але неефективний для масштабного тестування через високу трудомісткість та ймовірність людських помилок.

Автоматизоване тестування використовується для регулярного та повторюваного тестування API. Автоматизовані тести можуть бути інтегровані в процес безперервної інтеграції та доставки (CI/CD), що дозволяє швидко виявляти та усувати дефекти. Інструменти для автоматизованого тестування включають Postman (з використанням скриптів на JavaScript), Rest Assured, SoapUI та інші.

Інтеграційне тестування спрямоване на перевірку взаємодії API з іншими компонентами системи або сторонніми сервісами. Це дозволяє виявляти проблеми, пов'язані з інтеграцією, та забезпечувати узгоджену роботу всієї системи.

Тестування API може здійснюватися різними методами, кожен з яких має свої переваги та недоліки. В таблиці 1.2 наведено їх порівняння [7]-[8].

Вибір відповідного методу тестування залежить від конкретних вимог проекту, етапу розробки та ресурсів команди. У більшості випадків доцільно поєднувати різні методи для досягнення максимальної ефективності тестування.

Таблиця 1.2 – Порівняння видів тестування

Вид тестування	Опис	Переваги	Недоліки
Ручне тестування	Виконується вручну через інструменти типу Postman або curl	Гнучкість, швидке налаштування, добре для разової перевірки	Висока трудомісткість, складність повторного виконання
Автоматизоване	Скрипти, які перевіряють API-протоколи автоматично	Швидкість, інтеграція в CI/CD, багаторазове використання	Потребує початкових витрат часу на налаштування
Інтеграційне	Перевірка зв'язків між компонентами (наприклад, API + база даних)	Виявлення дефектів на стиках сервісів	Складніше в реалізації, потребує доступу до середовища
Тестування безпеки	Перевірка доступу, автентифікації, витоків даних	Важливе для відповідності стандартам і захисту користувачів	Потребує спеціалізованих знань
Продуктивності	Тестування навантаження, часу відповіді, витривалості API	Дає уявлення про масштабованість	Потребує специфічних інструментів (JMeter, Gatling)

Автоматизоване тестування API стало невід'ємною частиною сучасного процесу розробки програмного забезпечення, особливо в умовах використання гнучких методологій, таких як Agile та DevOps. Основні переваги автотестування наведені нижче [7].

Швидке виявлення дефектів – автоматизовані тести дозволяють швидко виявляти помилки на ранніх етапах розробки, що знижує вартість їх усунення та підвищує якість продукту.

Підвищення ефективності – автоматизація тестування зменшує трудомісткість процесу, дозволяючи командам зосередитися на розробці нових функцій та вдосконаленні продукту.

Інтеграція в CI/CD – автотести можуть бути інтегровані в процес безперервної інтеграції та доставки, що забезпечує постійний контроль якості та швидке впровадження змін.

Покращення документації – створення автотестів сприяє кращому розумінню функціональності API та служить додатковою документацією для команди.

Масштабованість – автоматизоване тестування дозволяє ефективно перевіряти великі обсяги даних та складні сценарії, що особливо важливо для масштабних систем.

1.4 Висновки за розділом та постановка завдання

У першому розділі було здійснено огляд основних проблем, що виникають під час забезпечення якості вебзастосунків, зокрема в умовах зростаючої складності, мікросервісної архітектури та потреби в швидкому зворотному зв'язку. Було проаналізовано роль API як ключового елемента взаємодії між компонентами вебсистем, розглянуто особливості API як об'єкта тестування, а також класифіковано основні типи тестування API.

Особливу увагу приділено порівнянню API-тестування з тестуванням через графічний інтерфейс користувача (GUI), що дало змогу підкреслити переваги API-підходу – зокрема його швидкість, стабільність, незалежність від змін у UI, зручність автоматизації та ефективність виявлення дефектів на ранніх етапах розробки.

Крім того, було порівняно різні види API-тестування – ручне, автоматизоване, інтеграційне, а також протестування безпеки та продуктивності, що дозволило оцінити доцільність застосування кожного з них у різних контекстах. Огляд засвідчив, що автоматизоване API-тестування – особливо з використанням інструментів на кшталт Postman – є оптимальним вибором для забезпечення надійної, гнучкої та масштабованої системи перевірки працездатності вебзастосунків.

Впровадження автотестування є ключовим фактором успішної розробки та підтримки якісного програмного забезпечення в сучасних умовах. Таким чином, було сформульовано мету – розробка програмного рішення для автоматизованого тестування API вебзастосунків.

Для досягнення цієї мети поставлено такі завдання:

- проаналізувати існуючі підходи та інструменти для автоматизованого тестування API;
- дослідити інструменти для написання автотестів;
- визначити набір API-запитів для тестування;
- розробити набір автотестів;
- виконати тестування API-запитів;
- оцінити ефективність запропонованого рішення.

2 ДОСЛІДЖЕННЯ ТА ВИБІР ІНСТРУМЕНТАРІЮ РОЗРОБКИ

2.1 Порівняння інструментів для API-тестування: Postman, REST Assured, SoapUI тощо

Для реалізації автотестів API розробники мають у розпорядженні широкий спектр інструментів таких як: Postman [2], REST Assured [9], SoapUI [10], JMeter [11]. Кожен із них має свої переваги, недоліки, особливості застосування та цільову аудиторію. У таблиці нижче представлено порівняння найпопулярніших рішень.

Таблиця 2.1 – Порівняння найпопулярніших рішень

Інструмент	Мова скриптів	Графічний інтерфейс	Автоматизація (CI/CD)	Особливості використання
Postman	JavaScript	Так	Через Newman	Легкий старт, GUI, створення колекцій, підтримка pre/post scripts
REST Assured	Java	Ні	Так	Інтегрується з JUnit/TestNG, потужний DSL для Java API
SoapUI	Groovy	Так	Так	Тестування як REST, так і SOAP, підтримка складних сценаріїв
JMeter	Java	Так	Так	Основний фокус – продуктивність, може запускати HTTP API

Серед розглянутих рішень для даного проєкту обрано Postman, що поєднує гнучкість, простоту у використанні та потужні можливості написання сценаріїв. Це інструмент, який підтримує як ручне, так і автоматизоване тестування API, дозволяючи створювати тести навіть без глибоких знань мов програмування.

Переваги використання Postman у цьому проєкті [2]:

- графічний інтерфейс для швидкого створення і перевірки запитів;
- підтримка скриптів для валідації відповідей на JavaScript;
- збереження колекцій, змінних середовища, параметризація;
- можливість запуску тестів у CI/CD через Newman;
- велика спільнота, документація, інтеграція з іншими інструментами.

Postman дозволяє сконцентруватися безпосередньо на логіці тестування, не витрачаючи час на налаштування складних середовищ чи фреймворків [12].

2.2 Огляд можливостей JavaScript-скриптів у Postman

Однією з головних переваг Postman є можливість використання JavaScript для реалізації pre-request та test scripts. Це дозволяє [13]:

- перевіряти статус-коди, заголовки, вміст відповіді;
- здійснювати логічні перевірки (наприклад, порівняння значень полів JSON);
- зберігати змінні з однієї відповіді для подальших запитів;
- реалізовувати цикли, умовні перевірки, кастомні повідомлення про помилки;
- логувати значення у вкладці Console.

Приклад простого скрипта валідації статус-коду [13]:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
```

```
});
```

Postman надає можливість створювати тести та логіку підготовки запитів за допомогою вбудованого механізму скриптів. У ролі мови програмування використовується JavaScript, що є єдиною офіційно підтримуваною мовою для створення pre-request та test scripts у Postman. Цей вибір зумовлений як технічними, так і стратегічними перевагами [14].

Переваги вибору JavaScript як мови скриптів у Postman наведені нижче.

JavaScript є однією з найпопулярніших мов програмування у світі, що робить її зрозумілою та доступною більшості розробників і тестувальників (Stack Overflow Developer Survey).

Завдяки динамічній природі JavaScript можна створювати прості та складні логічні перевірки, працювати з об'єктами, масивами, умовними конструкціями, тощо.

Всі змінні середовища, тіла запитів, заголовки, статус-коди – доступні через API Postman Sandbox саме в JavaScript [13].

Часто використовувані блоки можна легко копіювати між запитами, вставляти функції з бібліотек, створювати модульні конструкції.

Можливості JavaScript у Postman дозволяють зробити тестування гнучким і адаптивним під потреби конкретного API [14].

2.3 Інструменти для запуску і звітності

Collection Runner. Вбудований у Postman інструмент для послідовного запуску тестів у колекціях. Дозволяє переглядати результати, зберігати лог, але менш зручний для автоматизації [15].

Newman. CLI-інструмент від Postman, що дозволяє запускати колекції через термінал або інтегрувати у CI/CD. Переваги:

- автоматизований запуск через npm або docker;
- підтримка репортів у форматах JSON, HTML, JUnit;

- параметризація середовищ, зчитування глобальних змінних;
- можна запускати з GitHub Actions, Jenkins, GitLab CI тощо.

Команда для запуску: `newman run collection.json -e environment.json -r cli.html` [15].

2.4 Опис тестового об'єкта (вебзастосунку) та обраних API-запитів

У рамках даного дослідження обрано умовний вебзастосунок (або реально діючий, якщо вже визначено), що має RESTful API. Наприклад, це може бути сервіс, який містить наступні базові запити:

- GET /users – отримання списку користувачів;
- GET /users/:id – отримання конкретного користувача;
- POST /users – створення нового користувача;
- PUT /users/:id – оновлення даних користувача;
- DELETE /users/:id – видалення користувача.

API-запити обрані так, щоб охопити повний CRUD-функціонал і реалізувати найбільш типові сценарії для автоматизованого тестування.

2.5 Висновки за розділом 2

У другому розділі дипломної роботи було здійснено аналіз сучасних інструментів, які використовуються для автоматизованого тестування API, зокрема Postman, REST Assured, SoapUI та інших. На основі порівняльної характеристики було обґрунтовано доцільність використання Postman як основного інструменту розробки автотестів у даному проєкті. Його ключовими перевагами є простота використання, інтеграція з CI/CD через Newman, підтримка параметризації, збереження середовищ, а також скриптування на мові JavaScript.

Було також досліджено роль JavaScript як основної мови програмування у середовищі Postman. Саме ця мова використовується для

створення pre-request та test scripts, які дозволяють реалізувати гнучку та масштабовану логіку перевірок у межах колекцій запитів.

Окрему увагу приділено огляду інструментів запуску та звітності – Newman та Collection Runner, які забезпечують можливість використання створених автотестів у ручному та автоматизованому режимах. Також було описано тестовий об'єкт – вебзастосунок із REST API, на прикладі якого реалізовуватимуться сценарії CRUD-операцій.

Загалом, результати цього розділу створили необхідне теоретичне та практичне підґрунтя для переходу до наступного етапу – програмної реалізації API-тестів, що буде висвітлено в наступному розділі.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ API-ТЕСТІВ

3.1 Архітектура API-тестів

У рамках даної роботи було реалізовано набір автотестів для API-запитів до вебзастосунку, що дозволяє користувачам створювати й надсилати кастомізовані картки привітань. Основна мета реалізації – перевірка коректності роботи API, що обслуговує функціонал конструктора карток (card builder): кольори, шрифти, фони, обкладинки та стікери.

Тестова архітектура базується на використанні:

- Postman як середовища створення, запуску і налагодження запитів;
- JavaScript (Postman scripting) для написання скриптів перевірки;
- Postman Visualizer для табличного відображення даних відповіді;
- Environment variables для управління базовим URL та параметрами.

На рисунку 3.1 представлена діаграма взаємодії основних компонентів.

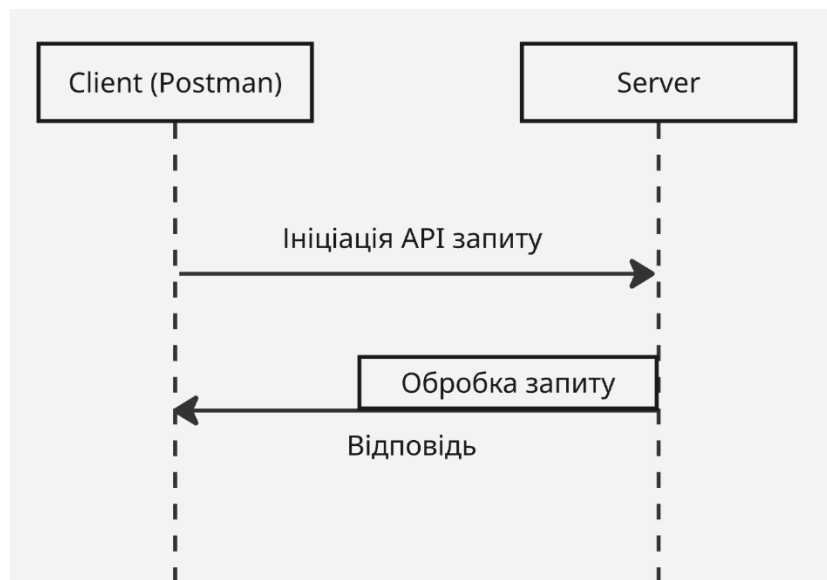


Рисунок 3.1 – Діаграма взаємодії основних компонентів

Postman працює як інструмент, який дозволяє взаємодіяти безпосередньо з API вебсайту, міняючи користувацький інтерфейс. Коли користувач надсилає запит через Postman, формується HTTP-запит, що надходить на сервер вебсайту. Сервер, отримавши запит, обробляє його згідно з вбудованою логікою, звертається до бази даних за необхідною інформацією або вносить зміни, після чого формує відповідь. Ця відповідь повертається назад у Postman у вигляді структурованих даних, зазвичай у форматі JSON. Таким чином, Postman дозволяє тестувати функціонування API, перевіряти правильність відповідей, статус-коди та загальну стабільність бекенду. Це дає змогу виявляти помилки, проводити автоматизоване тестування та переконуватись у правильності обробки запитів ще до того, як їх буде інтегровано у вебінтерфейс.

Усі запити згруповано у колекцію, структура яка наведена в наступному розділі. Для кожного з них реалізовані pre-request, test-скрипти, а також налаштовано змінні оточення.

3.2 Структура API-запитів

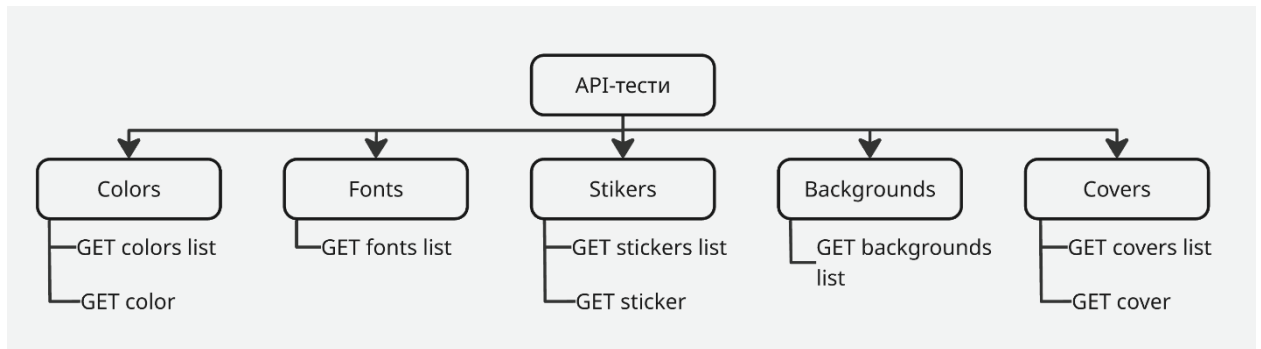
У межах дипломної роботи реалізовано структуру API-тестів у середовищі Postman, організовану за принципом колекцій. Для кожної категорії функціональності конструктор листівок (card builder) – кольори, темплейти, фони, шрифти, стікери – створено окремі запити в структурованих папках.

Колекція також містить:

- змінні середовища (наприклад: baseUrl, RandomColourId);
- pre-request scripts для генерації даних;
- тестові скрипти для валідації відповідей;
- окремий environment-файл для stage-оточення (https://stage-app.givingli.com/api_docs/...).

До тестового набору (колекції) включено такі запити по відповідним ендпоінтам, які представлені на рисунку 3.2:

- GET /api/v3/colors;
- GET /api/v3/colors/{id};
- GET /api/v3/fonts/list;
- GET /api/v3/backgrounds/list;
- GET /api/v4/covers;
- GET /api/v4/covers/{id};
- GET /api/v4/stickers;
- GET /api/v4/stickers/{id}.



Рисунко 3.2 – Структура колекції API-тестів

3.3 Реалізація тестів

У таблиці 3.1 представлено перелік API-ендпоінтів, згрупованих за категоріями, які використовуються для взаємодії з конструктором карток. Кожна категорія відповідає за певний тип візуального елемента, наприклад, кольори, шрифти, фони, обкладинки або стікери.

Для категорії «Кольори» передбачено два запити: перший (GET /api/v3/colors) дозволяє отримати загальний список доступних кольорів, а другий (GET /api/v3/colors/{id}) – отримати детальну інформацію про конкретний колір за його ідентифікатором.

Шрифти представлені одним запитом GET /api/v3/fonts/list, який повертає повний список шрифтів, доступних у системі.

Фони обробляються аналогічно через запит GET /api/v3/backgrounds/list, який надає перелік усіх фонових зображень.

Обкладинки мають два ендпоінти: GET /api/v4/covers повертає всі доступні обкладинки, тоді як GET /api/v4/covers/{{coverId}} дозволяє отримати деталі окремої обкладинки за її ідентифікатором.

Для стікерів також реалізовано два запити: загальний (GET /api/v4/stickers) для отримання списку всіх стікерів і конкретизований (GET /api/v4/stickers/{{stickerId}}) для отримання інформації про конкретний стікер.

Ці ендпоінти використовуються для динамічного наповнення інтерфейсу конструктора карток елементами дизайну, забезпечуючи користувачеві широкий вибір стилістичних опцій.

Таблиця 3.1 – Реалізовані API-ендпоінти

Категорія	Endpoint	Призначення
Кольори	GET /api/v3/colors	Отримання списку кольорів
	GET /api/v3/colors/{{id}}	Отримання конкретного кольору
Шрифти	GET /api/v3/fonts/list	Отримання списку шрифтів
Фони	GET /api/v3/backgrounds/list	Отримання списку фонових зображень
Обкладинки	GET /api/v4/covers	Отримання всіх обкладинок
	GET /api/v4/covers/{{coverId}}	Деталі конкретної обкладинки
Стікери	GET /api/v4/stickers	Отримання списку стікерів
	GET /api/v4/stickers/{{stickerId}}	Деталі конкретного стікера

Кожен з цих ендпоінтів може бути протестований на предмет:

- правильності статус-кодів відповіді (очікується 200 OK);
- структури та типів даних у відповіді (наприклад, наявність полів `id`, `name`, `url` тощо);
- відповідності даних очікуваним форматам (наприклад, перевірка, що поле `color` містить валідний HEX-код).

Також можна реалізувати негативні тести, наприклад, перевірити відповіді API на запити з неіснуючими `id` або з некоректними параметрами.

Для GET `list` – перевірка:

- статусу відповіді (200);
- структури масиву `result`;
- типів даних (`id`, `name`, `color`, тощо).

Для GET `by id` – перевірка:

- генерації `id` (`random`);
- наявності ключових полів у відповіді;
- типів і значень (наприклад, HEX-код, не порожні строки).

Розглянемо приклад реалізації перевірок для запитів, пов'язаних із кольоровими палітрами. API-ендпоінт `/api/v3/colors` повертає масив об'єктів з полями `id` та `color` (рис. 3.3).

Спочатку виконується перевірка того, що відповідь сервера має статус-код 200, що свідчить про успішне виконання запиту. Потім відповідь розбирається як JSON-об'єкт і перевіряється, чи містить він поле `result`, яке має бути масивом.

Далі перевіряються всі елементи цього масиву. Для кожного з них здійснюється низка тестів:

- чи містить елемент обов'язкові поля `id` і `color`;
- чи є `id` цілим додатним числом;
- чи є `color` рядком у форматі коректного HEX-коду кольору (наприклад, `#FF5733` або `#FF5733AA`).

Цей скрипт дозволяє переконатися, що структура й вміст відповіді відповідають очікуваним вимогам, і що всі значення коректно типізовані й валідовані. Він є частиною автоматизованого тестування API, що допомагає виявляти помилки ще на етапі розробки або змін у бекенді.



Рисунок 3.3 – Діаграма алгоритму перевірки запиту colors

На рисунку 3.4 зображена діаграма тестування API-запиту, що повертає інформацію про певний колір за випадковим ідентифікатором. Перед надсиланням запиту в блоці pre-request генерується випадкове число від 1 до 100, яке зберігається у змінну середовища RandomColourId. Ця змінна автоматично підставляється у запит, дозволяючи перевіряти різні кольори без ручного втручання.

Після отримання відповіді виконується низка перевірок. Найперше, перевіряється, чи сервер відповів зі статус-кодом 200, що вказує на успішну

обробку. Далі перевіряється, що відповідь містить об'єкт з полем `result`, і що цей об'єкт має ключі `id` та `color`. Ідентифікатор має бути числом, яке не є від'ємним, а значення кольору – непорожнім рядком. Також перевіряється, що заголовок `Content-Type` відповіді містить значення `application/json`, що підтверджує коректний формат даних.

У завершальній частині коду здійснюється обробка поля `result`, яке, всупереч попередній перевірці на об'єкт, тут розглядається як масив. Кожен елемент масиву проходить додаткову валідацію: перевіряється наявність полів `id` і `color`, цілісність значення `id` та правильність запису кольору у форматі HEX. Це може свідчити або про суперечність у структурі відповіді API, або про дублювання коду з іншого запиту, який повертає масив, а не окремий об'єкт.



Рисунок 3.4 – Діаграма алгоритму перевірки запиту `color`

Далі, наведмо у таблиці 3.2 функції та блоки коду, що були реалізовані для тестування отриманої відповіді від сервера на запит кольорів (код для всіх тестів повністю представлені в додатку А).

Таблиця 3.2 – Функції тестування API відповіді на запит кольорів

Назва функції/Блок коду	Опис
<code>pm.environment.set("RandomColourId", Math.floor(Math.random() * 100) + 1);</code>	У pre-request генерує випадкове число від 1 до 100 і зберігає його у змінну середовища RandomColourId.
<code>pm.test("Response status code is 200", function () {...});</code>	Перевіряє, що статус відповіді сервера дорівнює 200, тобто запит був успішним.
<code>pm.test("Response has the required fields - id and color", function () {...});</code>	Перевіряє, що тіло відповіді є об'єктом і містить поле result, яке включає ключі id і color.
<code>pm.test("Id is a non-negative integer", function () {...});</code>	Перевіряє, що поле id є числом і має значення більше або дорівнює нулю.
<code>pm.test("Color is a non-empty string", function () {...});</code>	Перевіряє, що поле color є непорожнім рядком.
<code>pm.test("Content-Type header is application/json", function () {...});</code>	Перевіряє, що заголовок відповіді Content-Type містить application/json.
<code>let jsonData = pm.response.json();</code>	Розбирає тіло відповіді як JSON-об'єкт і зберігає його у змінну jsonData.
<code>pm.test("Ответ содержит массив 'result'", function () {...});</code>	Перевіряє, що у відповіді є поле result, яке є масивом.
<code>jsonData.result.forEach((item, index) => {...});</code>	Перебирає всі елементи масиву result і виконує тести для кожного об'єкта.
<code>pm.test("Элемент \${index + 1}: содержит 'id' и 'color'", function () {...});</code>	Перевіряє, що кожен об'єкт у масиві має поля id і color.
<code>pm.test("Элемент \${index + 1}: 'id' – целое положительное число", function () {...});</code>	Перевіряє, що значення id – це ціле число і більше нуля.
<code>pm.test("Элемент \${index + 1}: 'color' – валидный HEX-код", function () {...});</code>	Перевіряє, що значення поля color є валідним HEX-кодом (наприклад, #FFAA00 або #FFAA00CC).

На рисунку 3.5 зображена діаграма тестування запиту по ендпоінту GET /api/v3/fonts/list). Функціональні перевірки цього скрипта:

- статус відповіді – 200 OK;
- структура – ключ result існує і є масивом;
- для кожного елемента перевіряє наявність 11 ключових полів (id, is_premium, title, is_cursive, тощо);
- для кожного елемента перевіряє відповідність типів (number, boolean, string).



Рисунок 3.5 – Діаграма алгоритму перевірки запиту fonts

Діаграма тестування запиту на перевірку фонів зображена на рисунку 3.6. Спочатку перевіряється, що сервер повертає статус-код 200, тобто запит був виконаний успішно. Далі аналізується структура відповіді – вона має містити поле `result`, яке повинно бути масивом з хоча б одним елементом.

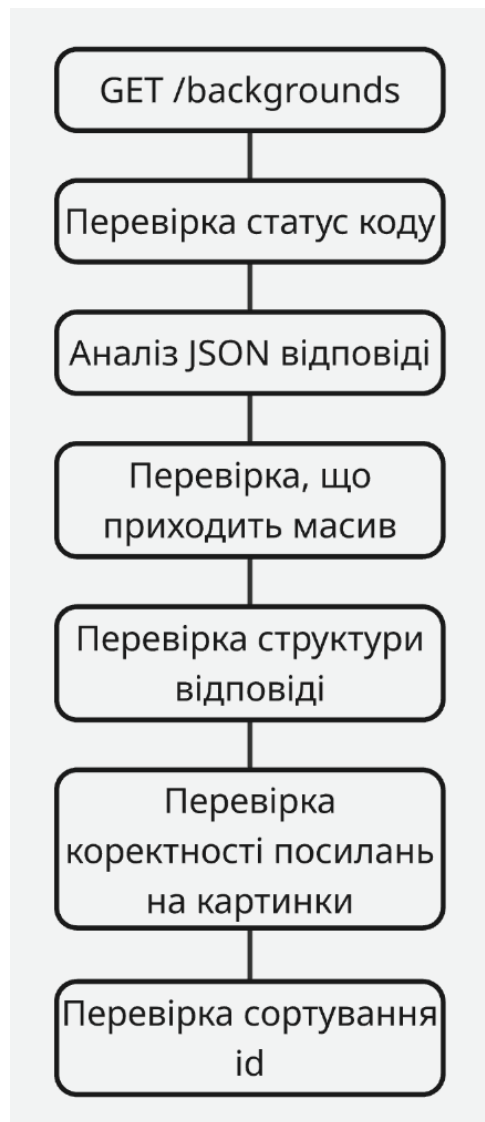


Рисунок 3.6 – Діаграма алгоритму перевірки запиту `backgrounds`

Кожен об'єкт у цьому масиві проходить детальну перевірку на наявність ключових полів: `id`, `is_dark`, `is_premium`, `group_id`, `position`, `for_desktop` та `image`, а також на відповідність їх типів – наприклад, `id` і

group_id мають бути числами, is_dark і for_desktop – булевими значеннями, а image – окремим вкладеним об'єктом.

Окремо перевіряється структура об'єкта image. Він має містити два поля – small і original, і обидва мають бути рядками. Для обох значень також перевіряється, чи є вони коректними URL-адресами за допомогою регулярного виразу.

На завершення, скрипт перевіряє, що значення id у кожному елементі масиву впорядковані за зростанням – тобто кожне наступне значення більше за попереднє. Такий підхід забезпечує як структурну, так і логічну перевірку відповіді API, дозволяючи переконатись у коректності даних, які будуть відображатися на стороні клієнта.

3.4 Висновки до розділу 3

У третьому розділі описано реалізацію API автотестів у середовищі Postman з використанням JavaScript. Розглянуто архітектуру рішення, протестовані запити, логіку перевірок та типові шаблони скриптів. Автотести дозволяють оперативно перевірити стабільність і якість API-функціоналу конструктора карток. Застосування цього підходу забезпечує прозорість і повторюваність перевірок, що сприяє підтримці високої якості вебзастосунку.

Реалізовані запити перевіряються як на наявність HTTP 200, так і на коректність структури відповіді (JSON), типів даних, відповідності URL, форматів дат, сортування id, та змісту масивів.

4 РЕЗУЛЬТАТИ ТА ІНСТРУКЦІЇ ПО ВИКОРИСТАННЮ

4.1 Огляд результатів автотестування

Ефективність автоматизованого тестування API у межах цієї роботи оцінюється за наступними критеріями:

- покриття функціоналу: протестовано всі ключові API-запити, пов'язані з конструктором карток (кольори, шрифти, фони, стікери, обкладинки);
- якість перевірок: реалізовано глибоку перевірку структури об'єктів, типів даних, валідацію URL, форматів дат та сортування;
- наявність негативних перевірок: тестування включає сценарії перевірки неправильних типів даних, відсутності полів, пустих масивів;
- зручність підтримки: тести організовані в колекції, використовують змінні середовища, а також шаблони для перевірок;
- можливість масштабування: завдяки використанню Postman і JavaScript, тести можна легко розширювати або адаптувати під нові версії API.

Таблиця 4.1 – Порівняння з ручним тестуванням

Параметр	Ручне тестування	Автоматизоване тестування (Postman)
Швидкість виконання	Повільне	Висока
Вірогідність помилок	Висока (людський фактор)	Низька
Повторюваність	Вимагає ручного втручання	Повністю автоматичне
Масштабованість	Обмежена	Висока
Витрати часу	Великі	Мінімальні після початкової реалізації

4.3 Ризики, обмеження і переваги використаного підходу

Переваги:

- швидкий запуск і налагодження без складної інфраструктури;
- глибока інтеграція з JSON-структурами та гнучкістю JavaScript;
- наявність візуалізації та документації прямо в Postman.

Обмеження:

- неможливість тестування інтеграцій між API та UI;
- обмеження JavaScript-середовища (немає сторонніх бібліотек);
- відсутність повноцінного фреймворку для сценаріїв на рівні end-to-end.

Ризики:

- залежність від структури API (при зміні – потрібно оновлювати всі перевірки);
- обмежена перевірка продуктивності без сторонніх інструментів (JMeter, k6).

4.4 Інструкція з запуску тестів вручну (через Postman)

Крок 1. Встановити Postman з офіційного сайту: https://www.postman.com/.

Крок 2. Імпортувати колекцію автотестів (файл формату `.json`):

* Відкрити Postman > Import > Upload Files > обрати файл `card_builder_tests.json`.

Крок 3. Імпортувати Environment (файл `.json` з `baseUrl`, токеном тощо).

Крок 4. Вибрати Environment у правому верхньому куті Postman.

Крок 5. Запустити колекцію або окремі тести через кнопку "Run" або використати Collection Runner. На рисунку 4.1-4.3 наведено результати запуску деяких автотестів.

Крок 6. Переглядати результати перевірок у вкладках "Tests" та "Console".

Крок 7. Для візуалізації результатів – перейти до вкладки Visualize у відповіді.

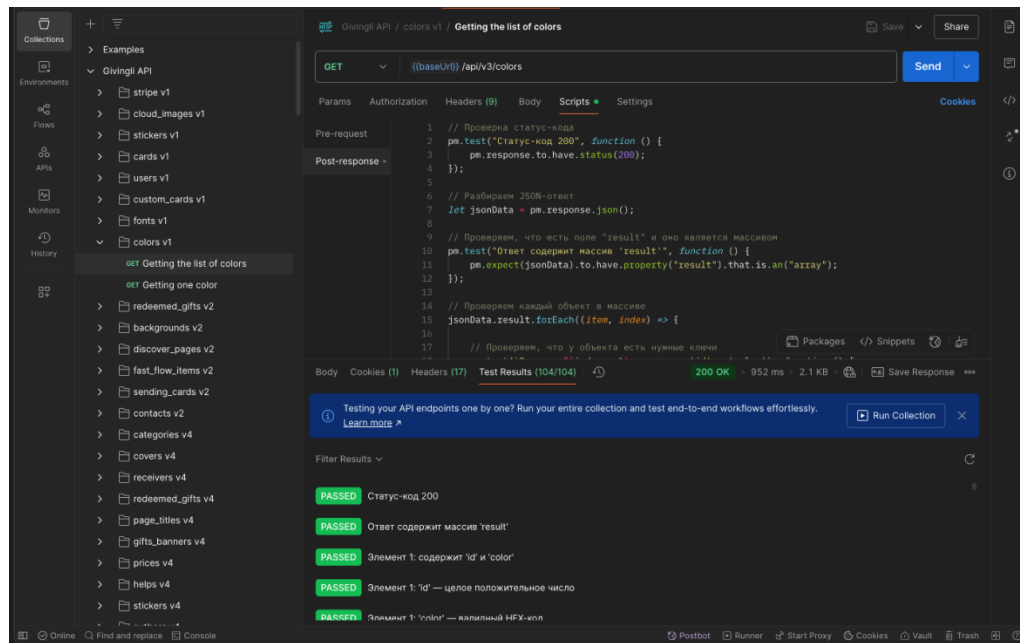


Рисунок 4.1 – Результат тестування API-запиту, що повертає інформацію про кольори

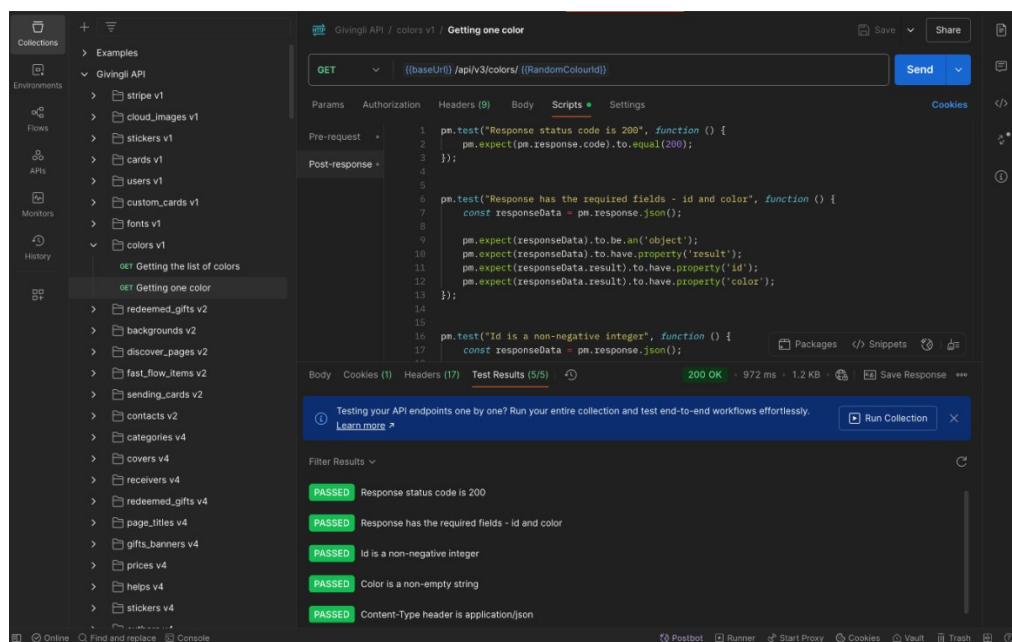


Рисунок 4.2 – Результат тестування API-запиту, що повертає інформацію про окремий колір

На рисунку 4.3 наведено результат запуску автотеста, що повертає інформацію про стікери.

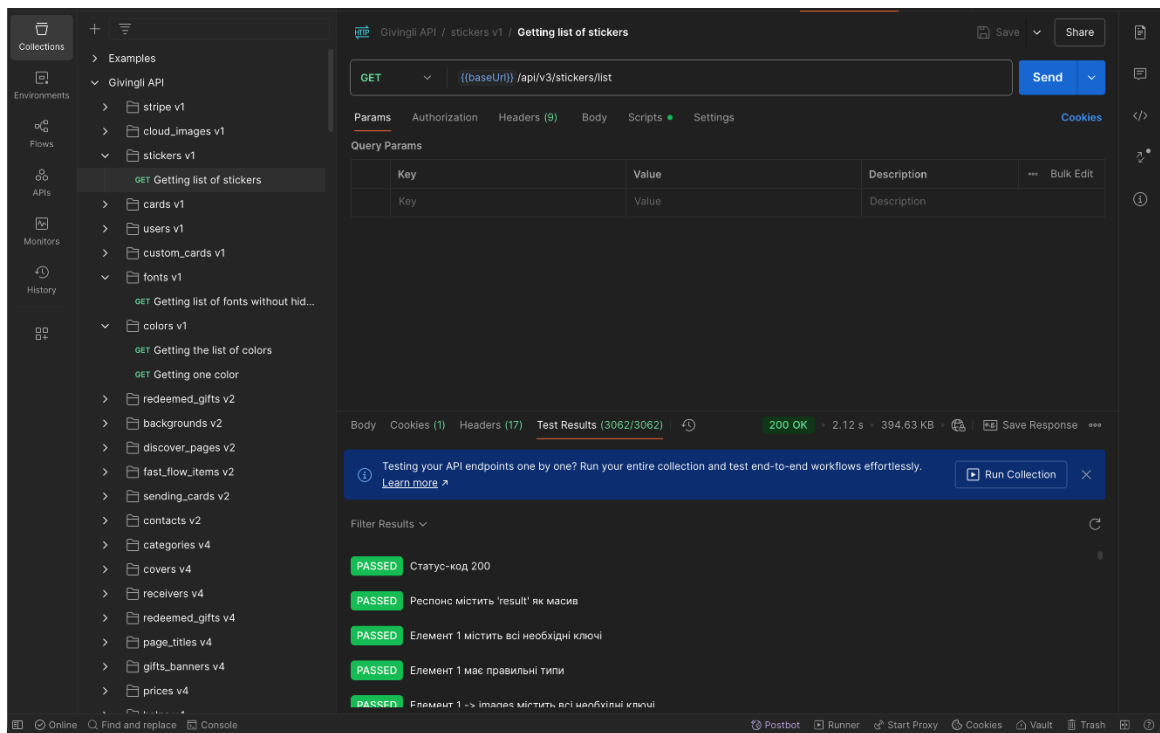


Рисунок 4.3 – Результат тестування API-запиту, що повертає інформацію про стікери

4.5 Рекомендації щодо розширення або інтеграції у CI/CD

Для запуску тестів у CI/CD можна використати Newman.

```
```bash
```

```
npm install -g newman
```

```
newman run card_builder_tests.json -e environment.json -r cli,html
```

Інтеграція можлива з GitHub Actions, GitLab CI, Jenkins

Рекомендується реалізувати тестування негативних сценаріїв та валідації авторизації.

Згенеровані HTML-звіти можуть зберігатись як артефакти пайплайну.

#### **4.6 Висновки до розділу 4**

Автоматизоване API-тестування з використанням Postman дозволило реалізувати гнучкий, масштабований і доступний інструмент перевірки функціональності критично важливих компонентів вебзастосунку. В розділі розглянуті переваги, ризики та обмеження обраного способу тестування. Описана інструкція запуску тестів та наведені результати виконання. Виходячи з результатів очевидно що спосіб має переваги у швидкості прогонки особливо для регресійного тестування застосунку.

## ВИСНОВКИ

У межах даної дипломної кваліфікаційної роботи бакалавра було досліджено, спроектовано та реалізовано рішення для автоматизованого тестування частину API вебзастосунку, що надає користувачам можливість надсилати кастомізовані картки привітань. А саме частину яка відповідає за відображення та перевірку доступності всіх айтемів (фонтів, стікерів, тощо) для побудови кастомізованої картки (card builder).

На етапі аналізу предметної області були виявлені основні проблеми забезпечення якості сучасних вебзастосунків, зокрема в контексті мікросервісної архітектури та потреби в ефективному виявленні дефектів ще до релізу. Встановлено, що API-тестування є критично важливою складовою забезпечення якості, оскільки воно дозволяє перевіряти логіку застосунку незалежно від користувацького інтерфейсу.

У другому розділі проведено порівняльний аналіз інструментів автоматизованого тестування API. На основі критеріїв зручності, гнучкості та доступності було обґрунтовано вибір Postman як основного інструменту. Також досліджено можливості JavaScript-скриптів у Postman та описано інструменти запуску (Newman, Collection Runner).

Третій розділ присвячено безпосередній реалізації автотестів. Було протестовано ключові API-запити конструктора карток: кольори, шрифти, фони, стікери, обкладинки. Реалізовано перевірки статус-кодів, структури JSON-об'єктів, типів даних, правильності форматів URL та дат, а також сортування і логіку ідентифікаторів. В окремих випадках використано Postman Visualizer для зручного подання результатів.

У четвертому розділі проведено оцінку ефективності запропонованого підходу, розглянуто його переваги, обмеження та ризики, а також надано практичну інструкцію щодо запуску тестів і рекомендації щодо інтеграції у CI/CD. Також наведені результати виконання автотестів.

Таким чином, розроблене рішення дозволяє:

- підвищити надійність вебзастосунку;
- зменшити витрати часу на рутинні (регресійні) перевірки;
- забезпечити швидкий зворотний зв'язок при зміні API;
- стандартизувати процес валідації API-функціоналу.

Отримані результати демонструють доцільність використання Postman як ефективного інструменту для побудови системи автоматизованого тестування API та відкривають перспективи для подальшої інтеграції таких рішень у повноцінні QA-процеси.

Переваги обраного підходу:

- простота реалізації і налагодження автотестів у Postman;
- швидкий запуск без налаштування середовища;
- наявність можливостей візуалізації даних;
- можливість інтеграції у CI/CD у майбутньому через Newman.

## ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. What Are The Main Advantages to API Testing? [Electronic resource]. – Access mode: <https://katalon.com/resources-center/blog/api-testing-advantages>.
2. What is API testing? [Electronic resource]. – Access mode: <https://www.postman.com/api-platform/api-testing>.
3. What is API testing? Everything you need to know. [Electronic resource]. – Access mode: <https://www.techtarget.com/searchapparchitecture/definition/API-testing>.
4. What is API Testing? Understand its Role in Quality Assurance. [Electronic resource]. – Access mode: <https://www.kiwiqa.com/what-is-api-testing-in-quality-assurance>.
5. API Testing. [Electronic resource]. – Access mode: [https://en.wikipedia.org/wiki/API\\_testing](https://en.wikipedia.org/wiki/API_testing).
6. API vs UI Testing: Which One Should You Pick. [Electronic resource]. – Access mode: <https://testsigma.com/blog/api-vs-ui-testing>.
7. Top 10 API Testing Tools in 2025: A Complete Guide. [Electronic resource]. – Access mode: <https://www.hypertest.co/api-testing/top-10-api-testing-tools>.
8. API Testing Tutorial: What is API Test Automation? [Electronic resource]. – Access mode: <https://www.guru99.com/api-testing.html>.
9. REST-assured [Electronic resource]. – Access mode: <https://rest-assured.io>.
10. Getting Started with REST Testing in SoapUI [Electronic resource]. – Access mode: <https://www.soapui.org/docs/rest-testing>.
11. Advantages and Disadvantages of Using JMeter [Electronic resource]. – Access mode: <https://automatenow.io/advantages-and-disadvantages-of-using-jmeter>.
12. Write scripts to test API response data in Postman [Electronic

resource]. – Access mode: <https://learning.postman.com/docs/tests-and-scripts/write-scripts/test-scripts>.

13. Process data and script workflows using Postman JavaScript objects [Electronic resource]. – Access mode: <https://learning.postman.com/docs/tests-and-scripts/write-scripts/postman-sandbox-api-reference>.

14. An overview of JavaScript testing [Electronic resource]. – Access mode: <https://gomakethings.com/an-overview-of-javascript-testing>.

15. Newman [Electronic resource]. – Access mode: <https://www.npmjs.com/package/newman#using-newman-cli>.

**ДОДАТОК А**  
**Текст програми**

### A.1 Файл GET `{{baseUrl}}/api/v3/colors`

```
// Проверка статус-кода
pm.test("Статус-код 200", function () {
 pm.response.to.have.status(200);
});

// Разбираем JSON-ответ
let jsonData = pm.response.json();

// Проверяем, что есть поле "result" и оно является массивом
pm.test("Ответ содержит массив 'result'", function () {
 pm.expect(jsonData).to.have.property("result").that.is.an("array");
});

// Проверяем каждый объект в массиве
jsonData.result.forEach((item, index) => {

 // Проверяем, что у объекта есть нужные ключи
 pm.test(`Элемент ${index + 1}: содержит 'id' и 'color'`, function () {
 pm.expect(item).to.have.property("id");
 pm.expect(item).to.have.property("color");
 });

 // Проверяем, что id - целое положительное число
 pm.test(`Элемент ${index + 1}: 'id' – целое положительное число`, function () {
 pm.expect(item.id).to.be.a("number").and.satisfy(Number.isInteger);
 pm.expect(item.id).to.be.above(0);
 });

 // Проверяем, что color - строка с HEX-кодом цвета (#RRGGBB или #RRGGBBAA)
 pm.test(`Элемент ${index + 1}: 'color' – валидный HEX-код`, function () {
 pm.expect(item.color).to.be.a("string").and.match(/^#([0-9A-Fa-f]{6}|[0-9A-Fa-f]{8})$/);
 });
});
```

## A.2 Файл GET {{baseUrl}}/api/v3/colors/{{RandomColourId}}

pre-request

```
pm.environment.set("RandomColourId", Math.floor(Math.random() * 100) + 1);
```

post-request

```
pm.test("Response status code is 200", function () {
 pm.expect(pm.response.code).to.equal(200);
});
```

```
pm.test("Response has the required fields - id and color", function () {
 const responseData = pm.response.json();
```

```
 pm.expect(responseData).to.be.an('object');
 pm.expect(responseData).to.have.property('result');
 pm.expect(responseData.result).to.have.property('id');
 pm.expect(responseData.result).to.have.property('color');
});
```

```
pm.test("Id is a non-negative integer", function () {
 const responseData = pm.response.json();
```

```
 pm.expect(responseData.result).to.be.an('object');
 pm.expect(responseData.result.id).to.be.a('number').and.to.satisfy((id) => id >=
0, "Id should be a non-negative integer");
});
```

```
pm.test("Color is a non-empty string", function () {
 const responseData = pm.response.json();
```

```
 pm.expect(responseData.result).to.be.an('object');
```

```
 pm.expect(responseData.result.color).to.be.a('string').and.to.have.lengthOf.at.least(
1, "Color should not be empty");
});
```

```
pm.test("Content-Type header is application/json", function () {
```

```

 pm.expect(pm.response.headers.get("Content-
Type")).to.include("application/json");
});
// Разбираем JSON-ответ
let jsonData = pm.response.json();

// Проверяем, что есть поле "result" и оно является массивом
pm.test("Ответ содержит массив 'result'", function () {
 pm.expect(jsonData).to.have.property("result").that.is.an("array");
});

// Проверяем каждый объект в массиве
jsonData.result.forEach((item, index) => {

 // Проверяем, что у объекта есть нужные ключи
 pm.test(`Элемент ${index + 1}: содержит 'id' и 'color'`, function () {
 pm.expect(item).to.have.property("id");
 pm.expect(item).to.have.property("color");
 });

 // Проверяем, что id - целое положительное число
 pm.test(`Элемент ${index + 1}: 'id' – целое положительное число`, function
 () {
 pm.expect(item.id).to.be.a("number").and.satisfy(Number.isInteger);
 pm.expect(item.id).to.be.above(0);
 });

 // Проверяем, что color - строка с HEX-кодом цвета (#RRGGBB или
 #RRGGBBAA)
 pm.test(`Элемент ${index + 1}: 'color' – валидный HEX-код`, function () {
 pm.expect(item.color).to.be.a("string").and.match(/^#([0-9A-Fa-f]{6}|[0-9A-
Fa-f]{8})$/);
 });

});

```

### А.3 Файл GET {{baseUrl}}/api/v3/fonts/list

```
// Проверяем, что статус ответа 200 OK
pm.test("Status code is 200", function () {
 pm.response.to.have.status(200);
});

// Проверяем структуру ответа
pm.test("Response structure is valid", function () {
 let responseJson = pm.response.json();

 // Убедимся, что ответ содержит ключ 'result', который является массивом
 pm.expect(responseJson).to.have.property("result").that.is.an("array");

 // Проверяем, что каждый элемент массива имеет корректную структуру
 responseJson.result.forEach((item, index) => {
 pm.test(`Validating item at index ${index}`, function () {
 const requiredFields = {
 id: "number",
 is_premium: "boolean",
 title: "string",
 is_cursive: "boolean",
 resource_file: "string",
 resource_file_file_name: "string",
 resource_file_content_type: "string",
 system_title: "string",
 group_id: "number",
 position: "number",
 is_hidden: "boolean",
 };

 // Проверяем наличие всех обязательных полей
 Object.keys(requiredFields).forEach((key) => {
 // Проверяем наличие свойства
 pm.expect(item).to.have.property(key);

 // Проверяем соответствие типов данных
 pm.expect(typeof item[key]).to.eql(
 requiredFields[key],
);
 });
 });
 });
});
```

```

 `Invalid type for property: ${key} (expected: ${requiredFields[key]},
got: ${typeof item[key]}) for item with id: ${item.id || "unknown"}`
);
 });
 });
 });
 });
var template = `
<style type="text/css">
 .tftable {font-size:14px;color:#333333;width:100%;border-width: 1px;border-
color: #87ceeb;border-collapse: collapse;}
 .tftable th {font-size:18px;background-color:#87ceeb;border-width:
1px;padding: 8px;border-style: solid;border-color: #87ceeb;text-align:left;}
 .tftable tr {background-color:#ffffff;}
 .tftable td {font-size:14px;border-width: 1px;padding: 8px;border-style:
solid;border-color: #87ceeb;}
 .tftable tr:hover {background-color:#e0ffff;}
</style>

<table class="tftable" border="1">
 <tr>
 <th>ID</th>
 <th>Is Premium</th>
 <th>Title</th>
 <th>Is Cursive</th>
 <th>Resource File</th>
 <th>File Name</th>
 <th>Content Type</th>
 <th>System Title</th>
 <th>Group ID</th>
 <th>Position</th>
 <th>Is Hidden</th>
 </tr>

 {{#each response.result}}
 <tr id=row_{{@key}}>
 <td>{{id}}</td>
 <td>{{is_premium}}</td>
 <td>{{title}}</td>
 <td>{{is_cursive}}</td>
 <td>{{resource_file}}</td>

```

```

 <td>{{resource_file_file_name}}</td>
 <td>{{resource_file_content_type}}</td>
 <td>{{system_title}}</td>
 <td>{{group_id}}</td>
 <td>{{position}}</td>
 <td>{{is_hidden}}</td>
 </tr>
 {{/each}}
</table>
`;

```

```

function constructVisualizerPayload() {
 return {response: pm.response.json()}
}
pm.visualizer.set(template, constructVisualizerPayload());

```

#### A.4 Файл GET {{baseUrl}}/api/v3/backgrounds/list

```

// Проверка, что статус ответа 200 OK
pm.test("Status code is 200", function () {
 pm.response.to.have.status(200);
});

// Проверка, что ответ содержит массив result
pm.test("Response contains 'result' array", function () {
 let responseJson = pm.response.json();
 pm.expect(responseJson).to.have.property("result").that.is.an("array");
 pm.expect(responseJson.result.length).to.be.above(0);
});

// Проверка структуры каждого элемента в result
pm.test("Each item in 'result' has correct structure", function () {
 let responseJson = pm.response.json();

 responseJson.result.forEach((item, index) => {
 pm.test(`Validating item at index ${index}`, function () {
 const requiredFields = {
 id: "number",
 is_dark: "boolean",
 is_premium: "boolean",

```

```

 group_id: "number",
 position: "number",
 for_desktop: "boolean",
 image: "object"
 };

 Object.keys(requiredFields).forEach((key) => {
 pm.expect(item).to.have.property(key);
 pm.expect(typeof item[key]).to.eql(
 requiredFields[key],
 `Invalid type for property: ${key} (expected: ${requiredFields[key]},
got: ${typeof item[key]})`
);
 });
});
});
});

// Проверка структуры image
pm.test("Each item has a valid 'image' object", function () {
 let responseJson = pm.response.json();

 responseJson.result.forEach((item, index) => {
 pm.test(`Validating 'image' structure at index ${index}`, function () {
 pm.expect(item).to.have.property("image").that.is.an("object");

 const requiredImageFields = {
 small: "string",
 original: "string"
 };

 Object.keys(requiredImageFields).forEach((key) => {
 pm.expect(item.image).to.have.property(key);
 pm.expect(typeof item.image[key]).to.eql(
 requiredImageFields[key],
 `Invalid type for image property: ${key} (expected:
${requiredImageFields[key]}, got: ${typeof item.image[key]})`
);
 });
 });
 });
});
});

```

```
});
```

```
// Проверка, что ссылки в image корректные URL
```

```
pm.test("Each image URL is valid", function () {
 let responseJson = pm.response.json();
 let urlRegex = /^https?:\V\[\^\s/$.?\#].[\^\s]*$/; // Простая проверка на URL

 responseJson.result.forEach((item, index) => {
 pm.test(`Validating image URLs at index ${index}`, function () {
 pm.expect(item.image.small).to.match(urlRegex, `Invalid URL for small
image at index ${index}`);
 pm.expect(item.image.original).to.match(urlRegex, `Invalid URL for
original image at index ${index}`);
 });
 });
});
```

```
// Проверка, что 'id' увеличивается по порядку
```

```
pm.test("ID values are in ascending order", function () {
 let responseJson = pm.response.json();
 let ids = responseJson.result.map(item => item.id);

 for (let i = 1; i < ids.length; i++) {
 pm.expect(ids[i]).to.be.above(ids[i - 1], `ID ${ids[i]} is not greater than
${ids[i - 1]}`);
 }
});
```

### A.5 Файл GET `{{baseUrl}}/api/v3/backgrounds/{{backgroundId}}`

```
// Проверка, что статус ответа 200 OK
```

```
pm.test("Status code is 200", function () {
 pm.response.to.have.status(200);
});
```

```
// Проверка, что ответ содержит объект result
```

```
pm.test("Response contains 'result' object", function () {
 let responseJson = pm.response.json();
 pm.expect(responseJson).to.have.property("result").that.is.an("object");
});
```

```
// Проверка структуры result
pm.test("'result' object has correct structure", function () {
 let responseJson = pm.response.json();
 let item = responseJson.result;

 const requiredFields = {
 id: "number",
 is_dark: "boolean",
 is_premium: "boolean",
 group_id: "number",
 position: "number",
 for_desktop: "boolean",
 image: "object"
 };

 Object.keys(requiredFields).forEach((key) => {
 pm.expect(item).to.have.property(key);
 pm.expect(typeof item[key]).to.eql(
 requiredFields[key],
 `Invalid type for property: ${key} (expected: ${requiredFields[key]}, got:
${typeof item[key]})`
);
 });
});

// Проверка структуры image
pm.test("'image' object has correct structure", function () {
 let responseJson = pm.response.json();
 let image = responseJson.result.image;

 pm.expect(image).to.be.an("object");

 const requiredImageFields = {
 large: "string"
 };

 Object.keys(requiredImageFields).forEach((key) => {
 pm.expect(image).to.have.property(key);
 pm.expect(typeof image[key]).to.eql(
 requiredImageFields[key],

```

```

 `Invalid type for image property: ${key} (expected:
 ${requiredImageFields[key]}), got: ${typeof image[key]}`
);
 });
});

```

// Проверка, что ссылка в image корректный URL

```

pm.test("'image.large' URL is valid", function () {
 let responseJson = pm.response.json();
 let image = responseJson.result.image;
 let urlRegex = /^https?:\V\[\^\s/$.?\#].[\^\s]*$/; // Простая проверка на URL

 pm.expect(image.large).to.match(urlRegex, "Invalid URL for large image");
});

```

#### **A.6 Файл GET {{baseUrl}}/api/v4/covers**

// Проверяем, что статус ответа 200 OK

```

pm.test("Status code is 200", function () {
 pm.response.to.have.status(200);
});

```

// Проверяем, что ответ содержит массив result

```

pm.test("Response structure is valid", function () {
 let responseJson = pm.response.json();

 pm.expect(responseJson).to.have.property("result").that.is.an("array");

```

// Проверяем, что массив не пустой

```

 pm.expect(responseJson.result.length).to.be.above(0);
});

```

// Проверяем структуру каждого элемента в массиве result

```

pm.test("Each item in result has the correct structure", function () {
 let responseJson = pm.response.json();

 responseJson.result.forEach((item, index) => {
 pm.test(`Validating item at index ${index}`, function () {
 const requiredFields = {
 id: "number",

```

```

 group_id: "number",
 is_premium: "boolean",
 images: "object",
 position: "number",
 };

 // Проверяем наличие всех обязательных полей
 Object.keys(requiredFields).forEach((key) => {
 pm.expect(item).to.have.property(key);
 pm.expect(typeof item[key]).to.eql(
 requiredFields[key],
 `Invalid type for property: ${key} (expected: ${requiredFields[key]},
got: ${typeof item[key]}) for item with id: ${item.id || "unknown"}`
);
 });
});
});
});

// Проверяем структуру объекта images внутри каждого item
pm.test("Each item has valid images structure", function () {
 let responseJson = pm.response.json();

 responseJson.result.forEach((item, index) => {
 pm.test(`Validating images structure at index ${index}`, function () {
 pm.expect(item).to.have.property("images").that.is.an("object");

 const requiredImageFields = {
 small: "string",
 medium: "string",
 aspect_ratio: "number",
 large: "string"
 };

 Object.keys(requiredImageFields).forEach((key) => {
 pm.expect(item.images).to.have.property(key);
 pm.expect(typeof item.images[key]).to.eql(
 requiredImageFields[key],
 `Invalid type for images property: ${key} (expected:
${requiredImageFields[key]}, got: ${typeof item.images[key]}) for item with id:
${item.id || "unknown"}`
);
 });
 });
 });
});

```

```

);
 });
});
});
});

// Проверяем, что ссылки в images являются корректными URL
pm.test("Each image URL is valid", function () {
 let responseJson = pm.response.json();
 let urlRegex = /^https?:\V\[\^\s/$.?\#\].[\^\s]*$/; // Простая проверка на URL

 responseJson.result.forEach((item, index) => {
 pm.test(`Validating image URLs at index ${index}`, function () {
 pm.expect(item.images.small).to.match(urlRegex, `Invalid URL for small
image at index ${index}`);
 pm.expect(item.images.medium).to.match(urlRegex, `Invalid URL for
medium image at index ${index}`);
 pm.expect(item.images.large).to.match(urlRegex, `Invalid URL for large
image at index ${index}`);
 });
 });
});
});

```

### **A.7 Файл GET {{baseUrl}} /api/v4/covers/{id}**

```

// Проверка, что статус ответа 200 OK
pm.test("Status code is 200", function () {
 pm.response.to.have.status(200);
});

// Проверка, что ответ содержит объект result
pm.test("Response contains 'result' object", function () {
 let responseJson = pm.response.json();
 pm.expect(responseJson).to.have.property("result").that.is.an("object");
});

// Проверка структуры result
pm.test("'result' object has correct structure", function () {
 let responseJson = pm.response.json();
 let item = responseJson.result;

```

```

const requiredFields = {
 id: "number",
 group_id: "number",
 is_premium: "boolean",
 images: "object"
};

Object.keys(requiredFields).forEach((key) => {
 pm.expect(item).to.have.property(key);
 pm.expect(typeof item[key]).to.eql(
 requiredFields[key],
 `Invalid type for property: ${key} (expected: ${requiredFields[key]}, got:
 ${typeof item[key]})`
);
});

// Проверка структуры images
pm.test("'images' object has correct structure", function () {
 let responseJson = pm.response.json();
 let images = responseJson.result.images;

 pm.expect(images).to.be.an("object");

 const requiredImageFields = {
 large: "string",
 aspect_ratio: "number"
 };

 Object.keys(requiredImageFields).forEach((key) => {
 pm.expect(images).to.have.property(key);
 pm.expect(typeof images[key]).to.eql(
 requiredImageFields[key],
 `Invalid type for images property: ${key} (expected:
 ${requiredImageFields[key]}, got: ${typeof images[key]})`
);
 });
});

// Проверка, что ссылки в images корректные URL

```

```

pm.test("'images' URLs are valid", function () {
 let responseJson = pm.response.json();
 let images = responseJson.result.images;
 let urlRegex = /^https?:\V\[\^\s/$.?\#].[\^\s]*$/; // Простая проверка на URL

 pm.expect(images.large).to.match(urlRegex, "Invalid URL for large image");
});

```

## A.8 Файл GET {{baseUrl}}/api/v4/stickers

```

// Проверка, что статус ответа 200 OK
pm.test("Status code is 200", function () {
 pm.response.to.have.status(200);
});

// Проверка, что ответ содержит массив result
pm.test("Response contains 'result' array", function () {
 let responseJson = pm.response.json();
 pm.expect(responseJson).to.have.property("result").that.is.an("array");
 pm.expect(responseJson.result.length).to.be.above(0, "Result array is empty");
});

// Проверка структуры каждого элемента в result
pm.test("Each item in 'result' has correct structure", function () {
 let responseJson = pm.response.json();

 responseJson.result.forEach((item, index) => {
 pm.test(`Validating item at index ${index}`, function () {
 const requiredFields = {
 id: "number",
 is_premium: "boolean",
 group_id: "number",
 images: "object",
 is_gif: "boolean",
 position: "number",
 is_frame: "boolean",
 is_hidden: "boolean",
 updated_at: "string"
 };
 });
 });
}

```

```

 Object.keys(requiredFields).forEach((key) => {
 pm.expect(item).to.have.property(key);
 pm.expect(typeof item[key]).to.eql(
 requiredFields[key],
 `Invalid type for property: ${key} (expected: ${requiredFields[key]},
got: ${typeof item[key]})`
);
 });

 // Проверяем tags отдельно
 pm.expect(item).to.have.property("tags");

 // Преобразуем tags в массив, если он объект
 let tagsArray = Array.isArray(item.tags) ? item.tags :
Object.values(item.tags);

 // Проверяем, что tags теперь массив
 pm.expect(tagsArray).to.be.an("array");

 // Проверяем структуру тегов (если есть)
 tagsArray.forEach((tag, tagIndex) => {
 pm.expect(tag).to.be.an("object");
 pm.expect(tag).to.have.property("id").that.is.a("number");
 pm.expect(tag).to.have.property("title").that.is.a("string").and.not.empty;
 });
 });
});

// Проверка структуры images
pm.test("Each item has a valid 'images' object", function () {
 let responseJson = pm.response.json();

 responseJson.result.forEach((item, index) => {
 pm.test(`Validating 'images' structure at index ${index}`, function () {
 pm.expect(item).to.have.property("images").that.is.an("object");

 const requiredImageFields = {
 original: "string",
 large: "string",
 medium: "string",

```

```

 medium2x: "string",
 small: "string",
 aspect_ratio: "number"
 };

 Object.keys(requiredImageFields).forEach((key) => {
 pm.expect(item.images).to.have.property(key);
 pm.expect(typeof item.images[key]).to.eql(
 requiredImageFields[key],
 `Invalid type for images property: ${key} (expected:
 ${requiredImageFields[key]}, got: ${typeof item.images[key]})`
);
 });
});
});
});

// Проверка, что ссылки в images корректные URL
pm.test("Each image URL is valid", function () {
 let responseJson = pm.response.json();
 let urlRegex = /^https?:\V\[\^\s/$.?\#\].[\^\s]*$/; // Простая проверка на URL

 responseJson.result.forEach((item, index) => {
 pm.test(`Validating image URLs at index ${index}`, function () {
 pm.expect(item.images.original).to.match(urlRegex, `Invalid URL for
 original image at index ${index}`);
 pm.expect(item.images.large).to.match(urlRegex, `Invalid URL for large
 image at index ${index}`);
 pm.expect(item.images.medium).to.match(urlRegex, `Invalid URL for
 medium image at index ${index}`);
 pm.expect(item.images.medium2x).to.match(urlRegex, `Invalid URL for
 medium2x image at index ${index}`);
 pm.expect(item.images.small).to.match(urlRegex, `Invalid URL for small
 image at index ${index}`);
 });
 });
});

// Проверка, что 'id' увеличивается по порядку
pm.test("ID values are in ascending order", function () {
 let responseJson = pm.response.json();

```

```

let ids = responseJson.result.map(item => item.id);

// Проверяем, отсортирован ли массив по возрастанию
let sortedIds = [...ids].sort((a, b) => a - b);
pm.expect(ids).to.eql(sortedIds, `IDs are not sorted correctly: ${ids}`);
});

// Проверка, что 'position' увеличивается по порядку
pm.test("Position values are in ascending order", function () {
 let responseJson = pm.response.json();
 let positions = responseJson.result.map(item => item.position);

 // Проверяем, отсортирован ли массив по возрастанию
 let sortedPositions = [...positions].sort((a, b) => a - b);
 pm.expect(positions).to.eql(sortedPositions, `Positions are not sorted correctly:
 ${positions}`);
});

// Проверка, что 'updated_at' имеет корректный формат даты
pm.test("Each item has valid 'updated_at' format", function () {
 let responseJson = pm.response.json();
 let dateRegex = /^(\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}\.\d{3}(Z|+\d{2}:\d{2}))$/; // Поддержка обоих
 форматов

 responseJson.result.forEach((item, index) => {
 pm.expect(item.updated_at).to.match(dateRegex, `Invalid date format for
 'updated_at' at index ${index}`);
 });
});

```

### A.9 Файл GET {{baseUrl}}/api/v4/stickers/{id}

```

// Проверка, что статус ответа 200 ОК
pm.test("Status code is 200", function () {
 pm.response.to.have.status(200);
});

// Проверка, что ответ содержит объект result
pm.test("Response contains 'result' object", function () {

```

```

 let responseJson = pm.response.json();
 pm.expect(responseJson).to.have.property("result").that.is.an("object");
 });

 // Проверка структуры result
 pm.test("'result' object has correct structure", function () {
 let responseJson = pm.response.json();
 let item = responseJson.result;

 const requiredFields = {
 id: "number",
 is_premium: "boolean",
 images: "object",
 is_gif: "boolean",
 updated_at: "string"
 };

 // Проверяем стандартные поля (кроме tags)
 Object.keys(requiredFields).forEach((key) => {
 pm.expect(item).to.have.property(key);
 pm.expect(typeof item[key]).to.eql(
 requiredFields[key],
 `Invalid type for property: ${key} (expected: ${requiredFields[key]}, got:
 ${typeof item[key]})`
);
 });

 // Проверяем tags отдельно
 pm.expect(item).to.have.property("tags");

 // Преобразуем tags в массив, если это объект
 if (!Array.isArray(item.tags) && typeof item.tags === "object") {
 item.tags = Object.values(item.tags);
 }

 // Проверяем, что tags теперь массив
 pm.expect(item.tags).to.be.an("array");
 });

 // Проверка структуры images
 pm.test("'images' object has correct structure", function () {

```

```

let responseJson = pm.response.json();
let images = responseJson.result.images;

pm.expect(images).to.be.an("object");

const requiredImageFields = {
 original: "string",
 large: "string",
 medium: "string",
 medium2x: "string",
 small: "string",
 aspect_ratio: "number"
};

Object.keys(requiredImageFields).forEach((key) => {
 pm.expect(images).to.have.property(key);
 pm.expect(typeof images[key]).to.eql(
 requiredImageFields[key],
 `Invalid type for images property: ${key} (expected:
 ${requiredImageFields[key]}, got: ${typeof images[key]})`
);
});

// Проверка, что ссылки в images корректные URL
pm.test("'images' URLs are valid", function () {
 let responseJson = pm.response.json();
 let images = responseJson.result.images;
 let urlRegex = /^https?:\V\[\^s/$.?\#\].[\^s]*$/; // Простая проверка на URL

 pm.expect(images.original).to.match(urlRegex, "Invalid URL for original image");
 pm.expect(images.large).to.match(urlRegex, "Invalid URL for large image");
 pm.expect(images.medium).to.match(urlRegex, "Invalid URL for medium image");
 pm.expect(images.medium2x).to.match(urlRegex, "Invalid URL for medium2x image");
 pm.expect(images.small).to.match(urlRegex, "Invalid URL for small image");
});

// Проверка, что 'updated_at' имеет корректный формат даты

```

```
pm.test("'updated_at' has valid format", function () {
 let responseJson = pm.response.json();
 let dateRegex = /^d{4}-d{2}-
\d{2}T\d{2}:\d{2}:\d{2}\.\d{3}(\+d{2}:d{2}|Z)$/; // ISO 8601 формат с UTC и
таймзоной
```

```
 pm.expect(responseJson.result.updated_at).to.match(dateRegex, "Invalid date
format for 'updated_at'");
});
```

```
// Проверка структуры 'tags'
```

```
pm.test("'tags' array has valid structure", function () {
```

```
 let responseJson = pm.response.json();
```

```
 let tags = responseJson.result.tags;
```

```
 pm.expect(tags).to.be.an("array");
```

```
 tags.forEach((tag, index) => {
```

```
 pm.expect(tag).to.have.property("id").that.is.a("number");
```

```
 pm.expect(tag).to.have.property("title").that.is.a("string").and.not.empty;
```

```
 });
```

```
});
```

**ДОДАТОК Б**  
**Слайди презентації**

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ  
Національний університет «Запорізька політехніка»

Дипломна кваліфікаційна робота бакалавра  
на тему: Розробка програмного забезпечення для  
автоматизованого тестування API

Виконав ст.гр. КНТ-121

Дмитро МУРАТОВ

Керівник к.т.н., доц. кафедри ПЗ

Ольга ГЛАДКОВА

Рисунок Б.1 – Слайд №1

## Мета та завдання

Мета роботи – розробка програмного рішення для автоматизованого тестування API запитів для тестування вебзастосунків.

Завдання:

- проаналізувати існуючі підходи та інструменти для тестування API;
- дослідити інструменти для написання автотестів;
- визначити набір API-запитів для тестування;
- розробити набір автотестів;
- виконати тестування API-запитів.

Рисунок Б.2 – Слайд №2

# Тестування API через середовище або графічний інтерфейс

Критерій	API-тестування (Postman)	GUI-тестування
Рівень тестування	Логіка та взаємодія між сервісами	Кінцевий інтерфейс користувача
Швидкість виконання	Висока	Повільна (залежить від візуального рендерингу)
Залежність від UI	Незалежне	Повністю залежне
Стабільність	Висока (менше змін у запитах)	Низька (зміни в UI ламають тести)
Обсяг покриття	Бізнес-логіка, валідації, обробка даних	Відображення елементів, взаємодія з формами
Застосовність для автоматизації	Легко інтегрується в CI/CD	Потребує спеціальних інструментів (Selenium, Cypress)
Вартість підтримки	Низька	Висока
Зручність у написанні	Висока (особливо з використанням Postman scripting)	Низька (потрібно відслідковувати зміни в DOM)

3

Рисунок Б.3 – Слайд №3

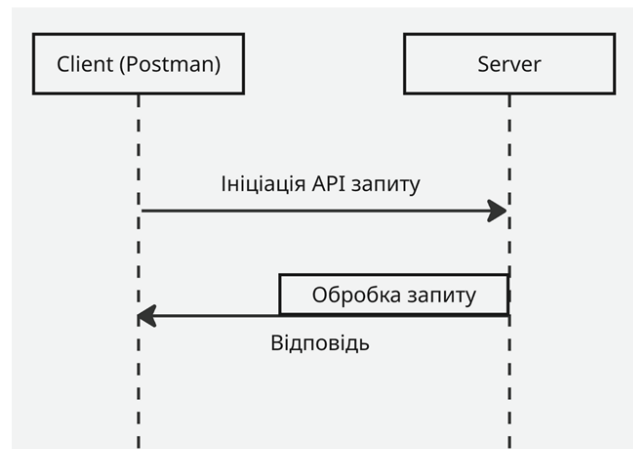
# Порівняння інструментарію

Інструмент	Мова скриптів	Графічний інтерфейс	Автоматизація (CI/CD)	Особливості використання
Postman	JavaScript	Так	Через Newman	Легкий старт, GUI, створення колекцій, підтримка pre/post scripts
REST Assured	Java	Ні	Так	Інтегрується з JUnit/TestNG, потужний DSL для Java API
SoapUI	Groovy	Так	Так	Тестування як REST, так і SOAP, підтримка складних сценаріїв
Insomnia	JavaScript (hooks)	Так	Частково	Зручний для інтерактивної роботи, слабша автоматизація
JMeter	Java	Так	Так	Основний фокус — продуктивність, може запускати HTTP API

4

Рисунок Б.4 – Слайд №4

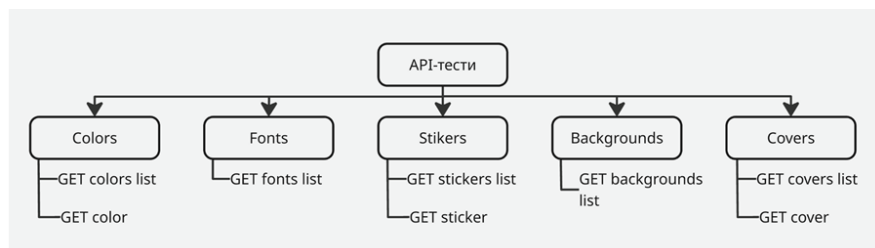
## Архітектура програмного рішення



5

Рисунок Б.5 – Слайд №5

## Структура API-тестів



6

Рисунок Б.6 – Слайд №6

# Діаграми послідовності запиту

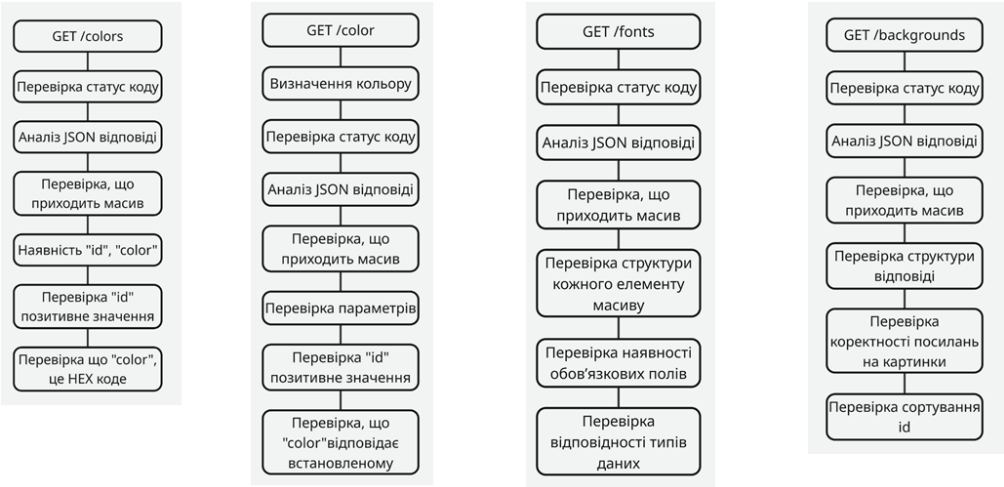


Рисунок Б.7 – Слайд №7

# Реалізовані функції

Назва функції/Блок коду	Опис
pm.environment.set("RandomColourId", Math.floor(Math.random() * 100) + 1);	У pre-request генерує випадкове число від 1 до 100 і зберігає його у змінну середовища RandomColourId.
pm.test("Response status code is 200", function () {...});	Перевіряє, що статус відповіді сервера дорівнює 200, тобто запит був успішним.
pm.test("Response has the required fields - id and color", function () {...});	Перевіряє, що тіло відповіді є об'єктом і містить поле result, яке включає ключі id і color.
pm.test("Id is a non-negative integer", function () {...});	Перевіряє, що поле id є числом і має значення більше або дорівнює нулю.
pm.test("Color is a non-empty string", function () {...});	Перевіряє, що поле color є непорожнім рядком.
pm.test("Content-Type header is application/json", function () {...});	Перевіряє, що заголовок відповіді Content-Type містить application/json.
let jsonData = pm.response.json();	Розбирає тіло відповіді як JSON-об'єкт і зберігає його у змінну jsonData.
pm.test("Ответ содержит массив 'result'", function () {...});	Перевіряє, що у відповіді є поле result, яке є масивом.
jsonData.result.forEach((item, index) => {...});	Перебирає всі елементи масиву result і виконує тести для кожного об'єкта.
pm.test("Элемент \${index + 1}: содержит 'id' и 'color'", function () {...});	Перевіряє, що кожен об'єкт у масиві має поля id і color.
pm.test("Элемент \${index + 1}: 'id' — целое положительное число", function () {...});	Перевіряє, що значення id — це ціле число і більше нуля.
pm.test("Элемент \${index + 1}: 'color' — валидный HEX-код", function () {...});	Перевіряє, що значення поля color є валідним HEX-кодом (наприклад, #FFAA00 або #FFAA00CC).

Рисунок Б.8 – Слайд №8

# Виконання тестів

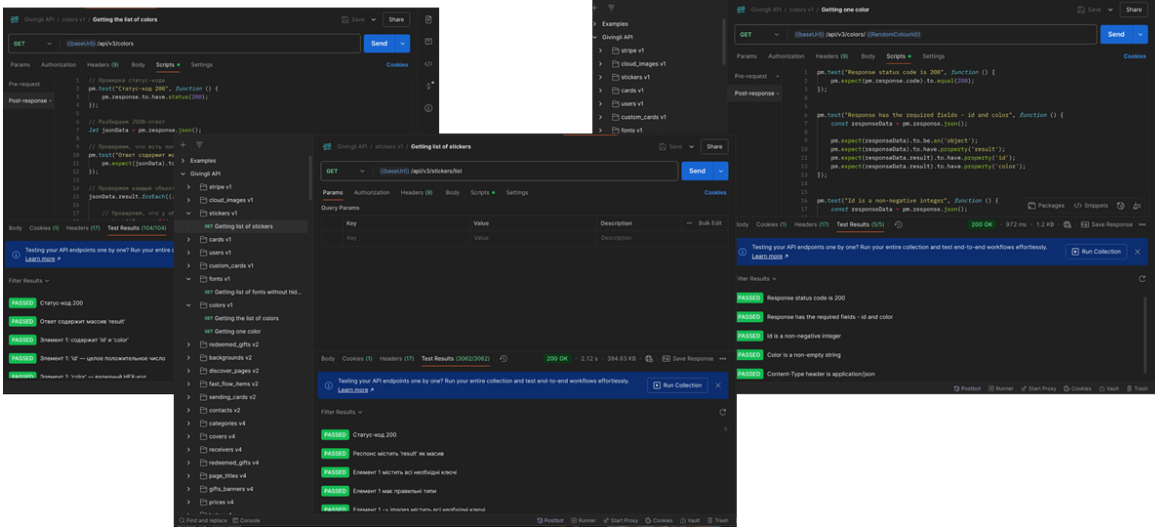


Рисунок Б.9 – Слайд №9

# Порівняння з ручним тестуванням

Параметр	Ручне тестування	Автоматизоване тестування (Postman)
Швидкість виконання	Повільне	Висока
Вірогідність помилок	Висока (людський фактор)	Низька
Повторюваність	Вимагає ручного втручання	Повністю автоматичне
Масштабованість	Обмежена	Висока
Витрати часу	Великі	Мінімальні після початкової реалізації

Рисунок Б.10 – Слайд №10

## Висновки

У межах даної дипломної кваліфікаційної роботи бакалавра було досліджено, спроектовано та реалізовано рішення для автоматизованого тестування частину API вебзастосунку, що надає користувачам можливість надсилати кастомізовані картки привітань. А саме частину функціональності, яка відповідає за відображення та перевірку доступності всіх айтемів (фонтів, стікерів, тощо) для побудови кастомізованої картки (card builder).

Розроблене рішення дозволяє: підвищити надійність вебзастосунку; зменшити витрати часу на рутинні (регресійні) перевірки; забезпечити швидкий зворотний зв'язок при зміні API; стандартизувати процес валідації API-функціоналу.