

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 проєкту (роботи)

бакалавр

(ступінь вищої освіти)

на тему ПРОГРАМНА РЕАЛІЗАЦІЯ АВТОТЕСТІВ ДЛЯ ТЕСТУВАННЯ
КОНТАКТНОЇ КНИГИ
SOFTWARE IMPLEMENTATION OF AUTOTESTS TO TEST
THE CONTACTS BOOK

Виконав(ла): студент(ка) 4 курсу, групи КНТ-211
Спеціальності 122 Комп'ютерні науки

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Комп'ютерні науки

РОДІН Д.В.

(ПРІЗВИЩЕ та ініціали)

Керівник ГЛАДКОВА О.М.

(ПРІЗВИЩЕ та ініціали)

Рецензент ДЬЯЧУК Т.С.

(ПРІЗВИЩЕ 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 122 Комп'ютерні науки
 (код і найменування)
 Освітня програма (спеціалізація) Комп'ютерні науки
 (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)

РОДІНА Дмитра Віталійовича
 (ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Програмна реалізація автотестів для тестування контактної книги. Software Implementation of Autotests to Test the Contacts Book

керівник проєкту (роботи) к.т.н., ГЛАДКОВА Ольга Миколаївна,
 (науковий ступінь, вчене звання, ПРІЗВИЩЕ, 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.
2. Обґрунтування вибору засобів розробки. 3. Розробка програмного забезпечення. 4. Аналіз результатів та перспективи використання.

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., асистент		

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 середовища програмування та інших технологій	2 тиждень	Розділ 2
	розробки.		
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3-4 тижні	Розділ 3, 4
6	Аналіз результатів та дослідження програмного	5 тиждень	Розділ 4
	забезпечення.		
7	Оформлення пояснювальної записки та документів	6 тиждень	Додатки
	до неї.		
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студент(ка)

(підпис) Дмитро РОДІН
(Ім'я ПРИЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Ольга ГЛАДКОВА
(Ім'я ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 70 с., 3 табл., 19 рис., 2 дод., 17 джерел.

API-ТЕСТУВАННЯ, КОНТАКТНА КНИГА, CRUD-ОПЕРАЦІЇ, POSTMAN, JSON.

Об'єкт дослідження – засоби для тестування контактної книги.

Предмет дослідження – програмна реалізація автотестів API функціоналу контактної книги.

Мета роботи – розробка та впровадження автотестів для перевірки API контактної книги.

Матеріали, методи та технічні засоби: тестування RESTful API, середовище Postman, формат JSON, скрипти на JavaScript, інструменти Collection Runner, операційна система Windows.

Результати. Створено набір автотестів у Postman, що охоплюють операції створення, оновлення, видалення та перегляду контактів.

Висновки. Розроблені автоматизовані API-тести забезпечують стабільність роботи контактної книги, скорочує час перевірки змін і підвищує загальну якість програмного забезпечення.

Галузь використання – тестування REST API вебзастосунків у командах розробки та забезпечення якості.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 70 pages, 3 tables, 19 figures, 2 appendixes, 17 sources.

API TESTING, CONTACT BOOK, CRUD OPERATIONS, POSTMAN, JSON.

Object of study is a tools for testing a contact book.

Subject of study are software implementation of API test automation for contact book functionality.

The purpose of the work is to develop and implement of automated tests to verify the correctness and stability of the contact book API.

Materials, methods, and tools: RESTful API testing, Postman environment, JSON format, JavaScript scripts, Collection Runner tool, Windows operating system.

Results. A set of automated tests was created in Postman covering operations such as creating, updating, deleting, and retrieving contacts.

Conclusions. The developed automated API tests ensure the stability of the contact book, reduce the time required for regression testing, and improve overall software quality.

The field of use is a testing REST APIs of web applications by development and quality assurance teams.

ЗМІСТ

	С.
Перелік скорочень та умовних познач.....	8
Вступ.....	9
1 Аналіз предметної області.....	11
1.1 Особливості CRUD-операцій у контактних книгах	11
1.1.1 Операція Create.....	11
1.1.2 Операція Read.....	12
1.1.3 Операція Update	12
1.1.4 Операція Delete.....	12
1.1.5 Додаткові особливості CRUD-операцій	13
1.1.6 Роль CRUD у тестуванні контактних API	13
1.2 Стандарти RESTful API для роботи з контактами.....	14
1.3 Практики тестування API-книг контактів (валідація полів, авторизація, дублікати тощо).....	16
1.3.1 Валідація полів	16
1.3.2 Обробка дублікатів	17
1.3.3 Тестування граничних значень.....	18
1.4 Висновки за розділом 1	18
2 Обґрунтування вибору засобів розробки	19
2.1 Методології тестування API	19
2.2 Вибір інструментів для тестування REST API.....	20
2.3 Мова програмування скриптів у Postman	22
2.4 Засоби запуску та звітності	23
2.5 Висновки за розділом 2	26
3 Розробка програмного забезпечення.....	27
3.1 Реалізація автотестів для створення, оновлення, видалення та отримання контактів	27
3.2 Тестування граничних значень, помилкових даних, авторизації	33
3.3 Перевірка структури відповіді, статус-кодів, повідомлень	35

3.4 Генерація звітів та запуск тестів.....	38
3.5 Висновки за розділом 3	39
4 Аналіз результатів та перспективи використання	41
4.1 Результати запуску тестів.....	41
4.2 Помічені дефекти і пропозиції щодо поліпшення	44
4.3 Можливості масштабування автотестів.....	45
4.4 Інструкція використання автотестів.....	48
4.5 Висновки за розділом 4	50
Висновки	52
Перелік джерел посилання	54
Додаток А Текст програми.....	56
Додаток Б Слайди презентації	65

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API – Application Programming Interface;

CRUD – створення (Create), читання (Read), оновлення (Update) та видалення (Delete);

IDE – Integrated Development Environment;

REST – Representational State Transfer;

БД – база даних;

ПЗ – програмне забезпечення.

ВСТУП

У сучасному світі інформаційних технологій забезпечення якості програмного забезпечення стало ключовим етапом життєвого циклу розробки. Зі зростанням складності програмних систем та високими вимогами до швидкості розробки і стабільності, автоматизоване тестування набуває дедалі більшого значення. Особливо це актуально для вебзастосунків з API-архітектурою, які передбачають активну взаємодію з користувацькими або внутрішніми сервісами [1].

Одним із базових прикладів таких систем є контактна книга – застосунок для зберігання та керування контактною інформацією користувачів. Незважаючи на простоту функціоналу, саме такі системи містять класичні CRUD-операції (Create, Read, Update, Delete), які становлять основу для перевірки стабільності RESTful API [2]. Тестування таких операцій дозволяє виявити критичні помилки, забезпечити цілісність даних та перевірити коректну обробку запитів у різних сценаріях.

У цьому контексті популярним інструментом для тестування API є Postman – інтерактивне середовище, що дозволяє створювати, запускати та документувати HTTP-запити, а також реалізовувати автотести з використанням JavaScript. Завдяки простоті у використанні та потужному функціоналу, Postman став стандартом де-факто для API-тестування у багатьох командах розробки [3].

Актуальність теми зумовлена потребою у впровадженні автоматизованих підходів до перевірки стабільності програмних інтерфейсів, що скорочує витрати часу на ручне тестування, знижує ймовірність людських помилок та сприяє підвищенню якості ПЗ.

Мета роботи – розробити та реалізувати набір автотестів для перевірки API контактної книги.

Завдання для досягнення поставленої мети:

- проаналізувати особливості RESTful API та CRUD-операцій у системах керування контактами;
- обрати інструменти для реалізації автотестів;
- розробити базові та розширені сценарії перевірок (позитивні й негативні тести);
- реалізувати автотести.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Особливості CRUD-операцій у контактних книгах

Сучасні інформаційні системи, які передбачають збереження та обробку користувацьких даних, майже завжди реалізують базову модель CRUD – створення (Create), читання (Read), оновлення (Update) та видалення (Delete). У випадку контактної книги ці операції є фундаментальними для забезпечення роботи з записами, які містять імена, номери телефонів, електронні адреси та іншу персональну інформацію [2].

Реалізація CRUD-операцій зазвичай базується на архітектурі REST, яка дозволяє забезпечити зрозумілий, масштабований та стандартизований спосіб доступу до ресурсів через HTTP-методи [4].

1.1.1 Операція Create

Операція створення контакту реалізується через HTTP-метод POST. Стандартний запит надсилається на ендпоінт /contacts із JSON-тілом, що містить дані нового контакту.

Приклад запиту: POST /contacts. Та можлива відповідь на цей запит наведена нижче.

```
{  
  "name": "Олена Іваненко",  
  "email": "olena@example.com",  
  "phone": "+380931112233"  
}
```

У типових API, таких як JSONPlaceholder, цей метод створює новий запис, який потім можна переглянути або змінити. При цьому сервер повинен повертати статус-код 201 Created та відповідь із новим об'єктом [5].

1.1.2 Операція Read

Читання контактів може виконуватись у двох основних варіантах:

- отримання списку всіх контактів GET /contacts;
- отримання конкретного контакту GET /contacts/{id}.

Приклад відповіді.

```
{ "id": 105,  
  "name": "Олена Іваненко",  
  "email": "olena@example.com",  
  "phone": "+380931112233"}
```

Розширені REST API також підтримують фільтрацію, сортування та пагінацію. Наприклад: GET /contacts?name=Олена&page=2&limit=10. Ці можливості дозволяють ефективно працювати з великими наборами даних [6].

1.1.3 Операція Update

Оновлення контакту здійснюється методом PUT або PATCH. PUT зазвичай використовується для повного перезапису об'єкта. PATCH – для часткових оновлень, коли потрібно змінити лише окремі поля.

Приклад часткового оновлення: PATCH /contacts/105. При некоректному запиті або відсутності ресурсу сервер має повертати 400 Bad Request або 404 Not Found відповідно [4].

1.1.4 Операція Delete

Видалення контакту реалізується за допомогою DELETE /contacts/{id}. В ідеалі API має повертати 204 No Content у разі успішного видалення. Якщо ресурс відсутній – 404 Not Found.

Приклад відповіді: { "message": "Contact successfully deleted"}.

Деякі системи реалізують soft delete – коли запис не видаляється фізично, а лише позначається як "неактивний", що дозволяє відновлення або аудит [7].

1.1.5 Додаткові особливості CRUD-операцій

Дублювання даних: API контактної книги повинно передбачати перевірку унікальності полів, таких як email або номер телефону. Повторне створення одного і того ж запису має завершуватись статусом 409 Conflict [4].

Обмеження записів: У практиці API розробки іноді обмежують кількість контактів для одного користувача з міркувань продуктивності або тарифікації.

Аудит змін: Особливо в корпоративних рішеннях всі операції створення, оновлення та видалення мають логуватись для відстеження історії змін.

1.1.6 Роль CRUD у тестуванні контактних API

CRUD-операції є базисом для розробки тест-кейсів. Один із ключових підходів до API-тестування – побудова послідовних сценаріїв, що покривають життєвий цикл ресурсу [3]:

- створити новий контакт;
- перевірити його існування (GET /contacts/{id});
- оновити контакт (наприклад, змінити телефон);
- видалити контакт;
- переконатися, що контакт більше не існує.

Такі сценарії можуть бути автоматизовані за допомогою Postman або аналогічних інструментів і є основою для smoke- чи регресійного тестування [3].

1.2 Стандарти RESTful API для роботи з контактами

REST (Representational State Transfer) є найпоширенішим архітектурним стилем, що використовується для побудови веб-сервісів. Його популярність пояснюється простотою, зрозумілою логікою і сумісністю з HTTP-протоколом, який є основою сучасного вебу [6].

REST не є протоколом, а набором архітектурних обмежень, що визначають спосіб взаємодії клієнтів і серверів через HTTP. У контексті контактної книги RESTful API використовується для надання доступу до ресурсів (тобто контактів) у зрозумілій та структурованій формі.

До базових принципів REST належать:

- ідентифікація ресурсів – кожен ресурс має унікальний URI. Наприклад: /contacts – список всіх контактів, /contacts/105 – конкретний контакт;
- стандартизовані HTTP-методи: GET – отримання ресурсів, POST – створення нового ресурсу, PUT – повне оновлення існуючого ресурсу, PATCH – часткове оновлення, DELETE – видалення;
- статусність – кожен запит містить усю необхідну інформацію для його обробки, сервер не зберігає контекст попередніх запитів;
- кешування – відповіді можуть кешуватись для зменшення навантаження на сервер;
- єдність інтерфейсу (Uniform Interface) – забезпечує узгодженість API: клієнт не потребує специфічних знань про реалізацію сервера.

Ці принципи дозволяють створювати API, які є простими у використанні, масштабованими і незалежними від клієнтської платформи [8].

Іменування ендпоінтів є критично важливим для RESTful API. Правильна практика – використовувати іменники у множині для назв ресурсів: правильно: /contacts; неправильно: /getContact, /create-contact.

Ендпоінти мають бути ресурсо-орієнтованими, а не дійслівними (тобто не doSomething). Всі дії визначаються HTTP-методами.

Приклад типового ендпоінту: GET /contacts/105.

Більшість реальних API дозволяють запити зі складною логікою:

- фільтрація: GET /contacts?name=Олена;
- сортування: GET /contacts?sortBy=name&order=asc;
- пагінація: GET /contacts?page=2&limit=10.

Ці механізми покращують ефективність при роботі з великими обсягами даних і дозволяють клієнтам краще керувати відображенням інформації [6].

RESTful API повинно чітко використовувати HTTP-статуси для інформування клієнта про результат операції (таблиц 1.1).

Таблиця 1.1 – Основні HTTP-статуси

Статус-код	Назва	Значення
200	OK	Запит успішно оброблено
201	Created	Ресурс успішно створено
204	No Content	Запит оброблено, але без тіла відповіді
400	Bad Request	Некоректний запит
401	Unauthorized	Необхідна авторизація
403	Forbidden	Доступ заборонено
404	Not Found	Ресурс не знайдено
409	Conflict	Конфлікт (наприклад, дублікати)
500	Internal Server Error	Помилка сервера

Коректне використання статусів – важлива частина як проектування, так і тестування API [9].

Ідемпотентність означає, що багаторазове виконання одного й того самого запиту не змінює результату. У REST це означає:

- GET, PUT, DELETE – ідемпотентні;
- POST – неідемпотентний (кожен запит створює новий ресурс).

Безпечними вважаються ті методи, які не змінюють стан сервера – зазвичай це лише GET [8].

Для збереження зворотної сумісності REST API має підтримувати версії через URI `/api/v1/contacts` або через заголовки запиту асерт: `application/vnd.contacts.v1+json`.

Це дозволяє клієнтам поступово переходити на нові версії без ризику для існуючих інтеграцій [10].

1.3 Практики тестування API-книг контактів (валідація полів, авторизація, дублікати тощо)

Тестування API контактної книги охоплює перевірку коректності обробки даних, безпеки доступу та уникнення дублювання записів. Основні аспекти включають:

1.3.1 Валідація полів

Перевірка правильності введених даних є критичною для забезпечення цілісності інформації. Наприклад, при створенні контакту необхідно перевірити:

- обов'язкові поля – ім'я, електронна пошта, номер телефону;
- формат полів – електронна пошта повинна відповідати стандартному формату, номер телефону – містити лише цифри та допустимі символи.

Приклад запиту з невалідною електронною поштою: `POST /contacts`.

```
{
  "name": "Іван Петренко",
  "email": "ivan.petrenko[at]example.com",
  "phone": "+380501234567"
}
```

Очікувана відповідь:

```
{
```

```
"error": "Невірний формат електронної пошти."
}
```

У разі виявлення помилок валідації, API повинен повертати статус-код 422 Unprocessable Entity, що вказує на неможливість обробки запиту через семантичні помилки [11].

Забезпечення безпеки доступу до API є важливим аспектом. Необхідно перевірити:

- доступ без токена – запити без токена повинні повертати статус 401 Unauthorized;
- доступ з недійсним токеном – запити з неправильним або простроченим токеном також повинні повертати 401 Unauthorized;
- доступ з недостатніми правами – запити з токеном, що не має необхідних прав, повинні повертати 403 Forbidden.

Приклад запиту без токена: GET /contacts.

Використання токенів доступу, таких як JWT, є поширеною практикою для забезпечення безпеки API [8].

1.3.2 Обробка дублікатів

Уникнення створення дублікатів контактів є важливим для підтримки цілісності даних. API повинен перевіряти унікальність певних полів, таких як електронна пошта або номер телефону.

```
Приклад запиту на створення дубліката: POST /contacts
{
  "name": "Іван Петренко",
  "email": "ivan.petrenko@example.com",
  "phone": "+380501234567"
}
```

Якщо контакт з такою електронною поштою вже існує, API повинен повертати статус-код 409 Conflict з відповідним повідомленням [11].

1.3.3 Тестування граничних значень

Перевірка поведінки API при введенні граничних значень допомагає виявити потенційні помилки:

- максимальна довжина полів – перевірити, чи обробляє API поля з максимально допустимою довжиною;
- порожні поля – перевірити, чи API відхиляє запити з порожніми обов'язковими полями.

Приклад запиту з порожнім іменем:

POST /contacts

```
{  
  "name": "",  
  "email": "ivan.petrenko@example.com",  
  "phone": "+380501234567"  
}
```

Очікувана відповідь:

```
{  
  "error": "Поле 'name' не може бути порожнім."  
}
```

1.4 Висновки за розділом 1

У цьому розділі було розглянуто ключові особливості CRUD операцій та аспекти тестування API контактної книги, включаючи валідацію полів, авторизацію, обробку дублікатів та граничних значень. Дотримання стандартів RESTful API та використання спеціалізованих інструментів для тестування сприяє створенню надійного та безпечного API, що відповідає вимогам користувачів та забезпечує високу якість обслуговування.

2 ОБГРУНТУВАННЯ ВИБОРУ ЗАСОБІВ РОЗРОБКИ

2.1 Методології тестування API

Тестування API є невід'ємною частиною забезпечення якості програмного забезпечення, особливо у вебзастосунках, де значна частина логіки реалізована на стороні сервера та взаємодіє з клієнтами через HTTP-запити. У випадку з контактною книгою, яка реалізує стандартний набір CRUD-операцій (створення, читання, оновлення, видалення), тестування API дозволяє перевірити як базову функціональність, так і надійність обробки нестандартних ситуацій [12].

Методології тестування API можна умовно поділити на ручне та автоматизоване. Ручне тестування передбачає безпосереднє виконання HTTP-запитів через інструменти типу Postman, Curl або браузер, і дозволяє швидко перевірити основну функціональність, особливо на ранніх етапах розробки. Проте з розвитком продукту та зростанням кількості тестів ручне виконання стає неефективним, потребує багато часу та є джерелом людських помилок. У зв'язку з цим дедалі ширше застосовується автоматизоване тестування [12].

Автоматизація API-тестів дозволяє повторно виконувати сотні перевірок у будь-який момент – наприклад, перед релізом, при інтеграції нових функцій або після змін у базі даних. Автотести підвищують стабільність проєкту, виявляючи регресії, які могли залишитися непоміченими. Вони легко інтегруються у процес безперервної інтеграції та доставки (CI/CD), а також дозволяють формувати звіти про покриття функціональності [12].

Основними типами API-тестів є:

- функціональні тести – перевірка очікуваної поведінки при коректних запитах (наприклад, створення нового контакту);

- негативні тести – перевірка поведінки при некоректних або неповних даних (наприклад, спроба створити контакт без обов’язкового поля);
- тестування статус-кодів – перевірка, що система повертає відповідний HTTP-статус відповідно до ситуації (200 OK, 201 Created, 400 Bad Request, 401 Unauthorized тощо) [4];
- тестування вмісту відповіді – перевірка структури JSON-об’єкта, типів даних, логічних залежностей між полями;
- тестування граничних значень – аналіз поведінки системи при введенні мінімальних, максимальних або порожніх значень;
- тестування безпеки та авторизації – перевірка, чи забороняється доступ до приватних даних без токена або з недійсними правами доступу [12].

Якісне тестування також передбачає використання моків (mock-server), коли реальна бізнес-логіка ще не реалізована, або перевірки контрактів API (наприклад, за допомогою JSON Schema або OpenAPI), що дозволяє автоматично виявляти невідповідність між документацією та реальною поведінкою сервера [6].

Таким чином, ефективна методологія тестування API базується на автоматизації, сценарному покритті CRUD-операцій, перевірці граничних випадків та постійному відстеженні змін через звітність. У даній роботі вибір зосереджено на функціональних автотестах для REST API контактної книги, реалізованих у середовищі Postman, з подальшою інтеграцією до процесів тестування.

2.2 Вибір інструментів для тестування REST API

Ефективне тестування REST API передбачає використання спеціалізованих інструментів, які дозволяють виконувати запити, перевіряти відповіді, створювати автотести та формувати звіти. Вибір інструменту має

базуватись на вимогах до зручності, функціональності, підтримки автоматизації та інтеграції в CI/CD.

Серед найбільш поширених засобів для API-тестування можна виділити Postman, REST Assured, SoapUI, Swagger та Insomnia [13]. Кожен із них має свої переваги та обмеження наведені в таблиці 2.1.

Таблиця 2.1 – Порівняння інструментів для API-тестування

Інструмент	Інтерфейс користувача	Автоматизація	Мова скриптів	CI/CD інтеграція	Документація API	Ліцензія
Postman	Графічний (GUI)	Так (Newman)	JavaScript	Так (Newman)	Так	Безкоштовно / Pro
REST Assured	Консоль / Код	Так	Java	Так	Ні	Вільна (Apache)
SoapUI	GUI скрипти +	Так	Groovy	Так	Так	Безкоштовно / Pro
Swagger	Документація + тест	Обмежено	Немає (тільки JSON)	Частково	Так	Безкоштовно
Insomnia	GUI	Так (через плагіни)	JavaScript	Обмежено	Так	Безкоштовно / Teams

Серед усіх інструментів, найбільш збалансованим для реалізації автотестів у межах даної роботи виявився Postman, оскільки він:

- має інтуїтивно зрозумілий інтерфейс і низький поріг входу;
- дозволяє створювати запити вручну або генерувати колекції для автоматизації;
- підтримує JavaScript-скрипти на етапах pre-request та tests, що дозволяє реалізовувати логіку перевірок;
- має вбудовану функцію Collection Runner та сумісність з Newman – CLI-інструментом, який дозволяє інтегрувати тести в CI/CD;
- підтримує змінні, середовища, параметризацію та генерацію динамічних даних [12];
- дозволяє формувати звіти у форматах JSON/HTML [14].

REST Assured, попри свою потужність, потребує знання Java та розробки в середовищі кодування, що не є доцільним у рамках дипломного проєкту, де перевага надається швидкості реалізації та візуальній доступності результатів.

SoapUI хоча й підтримує складні сценарії, є важчим у освоєнні і менш придатним для REST-first архітектури, що домінує у сучасних вебсервісах. Insomnia поступається Postman за кількістю навчальних матеріалів та зручністю скриптування.

Таким чином, вибір на користь Postman обґрунтований його популярністю, широкою спільнотою користувачів, активною підтримкою, розширюваністю через Newman та відповідністю всім технічним вимогам, що ставляться до API-тестування в рамках дипломної роботи.

2.3 Мова програмування скриптів у Postman

Postman – один із найбільш популярних інструментів для тестування REST API, який підтримує можливість написання скриптів для автоматизації перевірок, генерації динамічних даних та обробки відповідей. Основною мовою, яку використовує Postman для створення таких скриптів, є JavaScript – широко розповсюджена, гнучка та легка у використанні мова сценаріїв [15].

Скрипти в Postman поділяються на два основні типи:

- pre-request Script – виконується до надсилання HTTP-запиту;
- tests – виконується після отримання відповіді на запит.

Ці скрипти дозволяють реалізовувати додаткову логіку: від створення випадкових даних до перевірки відповідей API, що робить процес тестування більш гнучким і близьким до реального сценарного покриття.

Також доступні вбудовані змінні та методи Postman API (pm.*), які дозволяють:

- зберігати змінні (локальні, глобальні, середовища);
- логувати повідомлення;

- виконувати умовні перевірки;
- працювати з датами, таймерами, циклами.

Переваги використання JavaScript у Postman:

- гнучкість – завдяки умовам, циклами та логіці, можна адаптувати автотести до будь-яких сценаріїв;
- легкість у навчанні – JavaScript має простий синтаксис та велику базу прикладів у документації Postman;
- повторне використання - можна створювати колекції запитів і додавати до них спільні скрипти;
- динамічне налаштування - значення заголовків, параметрів і тіла запиту можна змінювати в режимі виконання;
- сумісність з Newman – скрипти, написані у Postman, коректно виконуються при запуску колекції у Newman, що важливо для автоматизації в CI/CD.

JavaScript-сценарії у Postman не потребують встановлення окремого середовища або компіляції, а виконуються прямо у вікні запиту, що значно спрощує процес створення та налагодження автотестів. Крім того, велика спільнота користувачів та постійна підтримка з боку Postman сприяють появі великої кількості готових прикладів, бібліотек і розширень [16].

JavaScript у Postman – це не просто мова для написання простих перевірок, а повноцінний інструмент для створення динамічних, адаптивних і ефективних автотестів. Його використання дає можливість реалізувати як прості smoke-тести, так і складні перевірки зі змінними, умовами та логікою. У рамках цієї роботи JavaScript став основою для написання перевірок до кожного ендпоінту API контактної книги.

2.4 Засоби запуску та звітності

Для забезпечення ефективного виконання та аналізу результатів API-тестів важливо мати не лише тест-кейси, а й зручні засоби запуску та

генерації звітності. Postman, як інструмент для тестування API, пропонує декілька механізмів виконання автотестів, як у графічному інтерфейсі, так і через командний рядок, що дозволяє легко інтегрувати тести в автоматизовані процеси (CI/CD) [17].

Collection Runner – це вбудований інструмент Postman, що дозволяє запускати одну або кілька колекцій запитів з можливістю повторення, параметризації та завантаження даних із CSV/JSON-файлів.

Основні можливості Collection Runner:

- запуск колекцій з попередньо визначеними середовищами та змінними;
- підтримка таблиць тестових даних (data-driven testing);
- візуальне представлення результатів (успішні/невдалі тести);
- просте налагодження локально без командного рядка.

Хоча цей спосіб зручний на етапі розробки та перевірки окремих сценаріїв, він не є оптимальним для регулярного використання у CI/CD, оскільки не підтримує автоматичне формування звітів у зручному форматі (наприклад, HTML).

Newman – це офіційний інструмент командного рядка (CLI), створений командою Postman, який дозволяє запускати колекції запитів і генерувати звіти поза графічним інтерфейсом. Він встановлюється через Node.js та підтримує повну інтеграцію з Postman-колекціями.

Основні можливості Newman [14]:

- запуск тестів безпосередньо з терміналу `newman run collection.json -e environment.json`;
- формування звітів у форматах JSON, HTML, JUnit, CLI;
- можливість інтеграції в GitLab CI, Jenkins, GitHub Actions, Azure DevOps та інші пайплайни;
- підтримка глобальних та середовищних змінних;
- зручний для роботи на сервері або в хмарі (наприклад, у Docker-контейнерах).

Приклад запуску з HTML-звітом: `newman run collection.json -r html,cli --reporter-html-export report.html`.

Завдяки Newman API-тести з Postman можуть бути виконані автоматично після кожного коміту в репозиторій або при зміні коду на сервері. Це дозволяє впровадити практику безперервного тестування та підтримувати стабільність коду впродовж усього життєвого циклу проекту.

Автотести, реалізовані у Postman та запущені через Newman, можна легко включити до пайплайнів CI/CD. У типовому сценарії тести додаються до `.yaml` або `build.gradle`-конфігурації, і при кожному оновленні репозиторію виконується автоматичний запуск [17]:

- у GitLab CI/CD – тести виконуються як частина test-джоба;
- у Jenkins – використовується Newman як shell-команда або Node-плагін;
- у GitHub Actions – тести запускаються в окремому job-блоці з Node-оточенням.

Такий підхід дозволяє виявляти помилки до потрапляння коду в продакшн, скорочує час на ручне тестування та підвищує якість кінцевого продукту.

Комбінація Postman + Newman надає повноцінний інструментарій для запуску API-тестів як вручну, так і в автоматизованому режимі. Collection Runner зручний для локальної перевірки, тоді як Newman дозволяє інтегрувати тести у CI/CD, формувати звіти, масштабувати тестування та підвищувати ефективність команди. У рамках даної дипломної роботи саме такий підхід використовується для реалізації автотестів API контактної книги.

2.5 Висновки за розділом 2

У другому розділі було розглянуто методологічні засади, інструменти та технології, що застосовуються для розробки автотестів REST API контактної книги.

Було проаналізовано методології тестування API, зокрема розмежовано ручне та автоматизоване тестування. Наведено основні типи API-тестів – функціональні, негативні, перевірка статус-кодів, граничних значень, а також перевірки структури відповідей і авторизації. Обґрунтовано доцільність автоматизації як базового підходу до забезпечення якості API.

Здійснено порівняльний аналіз найпопулярніших інструментів для тестування REST API: Postman, REST Assured, SoapUI, Swagger, Insomnia. З-поміж них обґрунтовано вибір Postman як найбільш зручного, доступного та функціонального інструменту, що поєднує GUI, автоматизацію, скриптування й інтеграцію з CI/CD.

Розглянуто можливості мови JavaScript як основного засобу створення логіки тестів у Postman. Показано приклади pre-request та test-скриптів, що дозволяють динамічно генерувати дані, перевіряти відповіді, працювати зі змінними та логічними умовами. Підкреслено гнучкість і доступність такого підходу.

Проаналізовано інструменти запуску та звітності. Collection Runner використовується для ручного запуску колекцій у середовищі Postman, тоді як Newman забезпечує автоматизований запуск у командному рядку з можливістю формування звітів у форматах JSON, HTML та інтеграції в CI/CD-процеси. Підтверджено доцільність використання Newman для регулярного тестування та впровадження безперервного контролю якості.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Реалізація автотестів для створення, оновлення, видалення та отримання контактів

На рисунку 3.1 зображена структура взаємодії компонентів системи тестування API контактної книги в Postman.

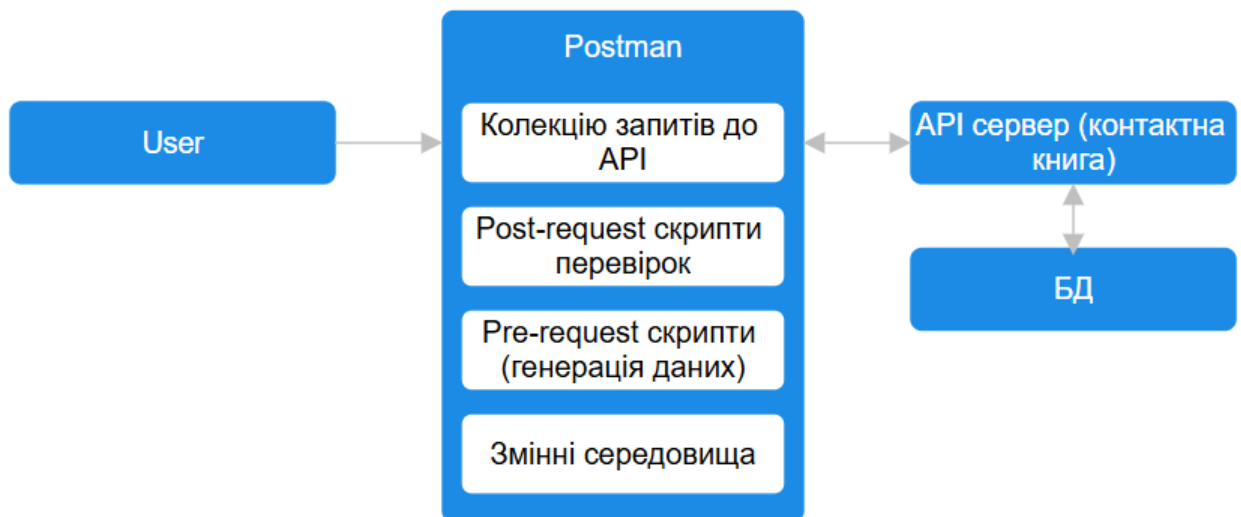


Рисунок 3.1 – Структура взаємодії компонентів системи тестування API

Користувач (User) – ініціює тестування, працює у середовищі Postman. Postman – основний інструмент для виконання тестів:

Колекція запитів до API – містить набір HTTP-запитів (GET, POST, PATCH, DELETE) до ендпоінтів контактної книги.

Post-request скрипти перевірок – автоматично виконуються після кожного запиту. Перевіряють статус-коди, структуру відповіді, правильність значень.

Pre-request скрипти (генерація даних) – виконуються перед запитом. Генерують, наприклад, випадкові номери телефонів або встановлюють токени.

Змінні середовища - використовуються для зберігання динамічних значень (базовий URL, токен, ID контакту тощо).

API сервер (контактна книга). Отримує запити з Postman. Повертає відповіді у форматі JSON. Спілкується з базою даних (БД) для обробки даних контактів.

База даних (БД). Зберігає інформацію про користувачів, контакти, CSV-імпорти, токени тощо. Відповідає на запити API-сервера при кожній операції (створення, читання, оновлення, видалення).

Реалізація автотестів для API контактної книги охоплює основні CRUD-операції: створення (Create), отримання (Read), оновлення (Update) та видалення (Delete) контактів. Усі тести реалізовані за допомогою інструменту Postman з використанням скриптів на мові JavaScript у вкладках Pre-request Script та Tests. На рисунку 3.2 зображена структура розробленої колекції API-запитів.

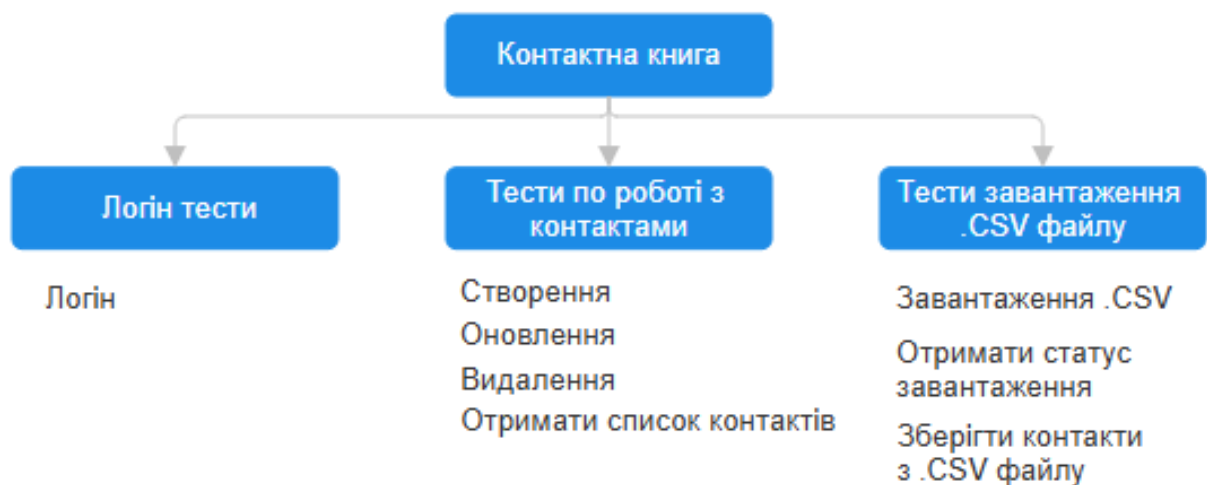


Рисунок 3.2 – Структура колекції API-запитів

Діаграма послідовності тесту авторизації користувача (рис. 3.3). Використовується запит POST /sessions/validate_code, який приймає код підтвердження та повертає інформацію про користувача разом з токеном авторизації.

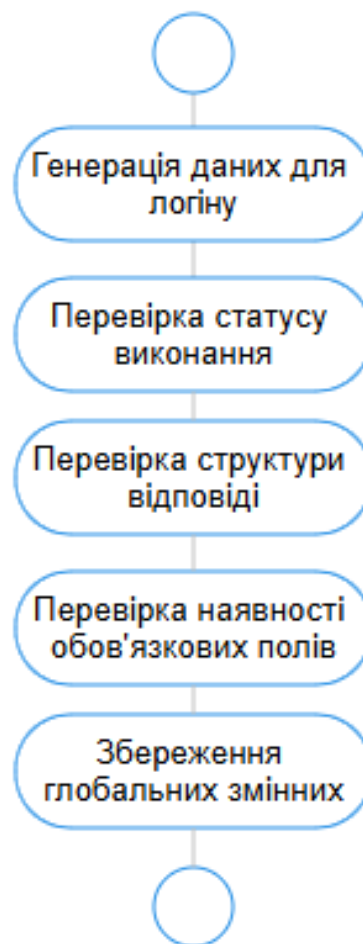


Рисунок 3.3 – Діаграма послідовності тесту авторизації користувача

У post-request скрипті реалізовано:

- збереження токена авторизації до глобальних змінних середовища Postman;
- логування результату в консоль.

У скрипті також реалізовано повну перевірку структури об'єкта користувача, який приходить у відповіді.

Перевіряються:

- статус-код 200 OK, що підтверджує успішну авторизацію;
- типи та формати даних основних полів: id, notifications, credits, chats_amount – як числа; phone, username, email – як рядки з регулярною перевіркою формату; birth_date, birth_date_only, created_at, updated_at – як дати у форматі ISO 8601;

- об'єкт `avatar` – має всі обов'язкові поля `small`, `medium`, `large`, `aspect_ratio` і правильні типи;
- повне ім'я `full_name` – перевіряється, чи включає `first_name` і `last_name`;
- логічні прапори (`boolean`): такі як `is_blocked`, `has_premium`, `sms_enabled`, `email_enabled` тощо – мають тип `boolean`;
- масиви: `kyc_elements_ids`, `kyc_events_ids`, `kyc_gift_lists_ids`, `kyc_brands_ids` – перевіряються як масиви.

Таким чином, цей скрипт не лише зберігає токен для подальших запитів, але й забезпечує глибоку валідацію користувацьких даних після авторизації, що є важливим етапом перевірки безпеки, коректності структури API та готовності користувача до взаємодії із застосунком.

Діаграма послідовності тесту для створення контактів (рис. 3.4). Використовується запит `POST /users_contacts` із динамічно згенерованими даними, зокрема випадковим номером телефону.

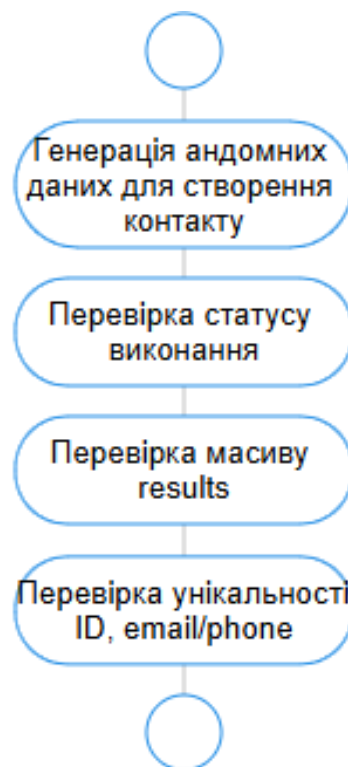


Рисунок 3.4 – Діаграма послідовності тесту для створення контакту

У pre-request script реалізовано логіку формування унікального номера з допустимих кодів операторів:

```

– const operatorCodes = ["050", "099", "063", "067", "096"];
– const randomOperatorCode =
operatorCodes[Math.floor(Math.random() * operatorCodes.length)];
– const randomSevenDigits = Math.floor(1000000 + Math.random() *
9000000);
– const phoneNumber = randomOperatorCode +
randomSevenDigits.toString();
– pm.environment.set("randomPhoneNumber", phoneNumber).

```

У тестах перевіряється:

- статус-код 200 OK;
- наявність поля id у відповіді;
- відповідність збережених полів очікуваним значенням.

Діаграма послідовності тесту для отримання всіх контактів (рис. 3.5).

Використовується запит GET /users_contacts. У відповіді перевіряється:

- статус-код 200;
- що поле result є масивом;
- унікальність id кожного контакту;
- наявність хоча б одного засобу зв'язку – phone або email;
- коректність формату дати в полях updated_at, birth_date, workiversary;
- правильність типів полів (id – число, email – рядок тощо);
- структура вкладеного об'єкта avatar.

Ці перевірки дозволяють переконатися, що API не лише повертає правильні дані, але й дотримується стандарту структури, необхідного для подальшої інтеграції.

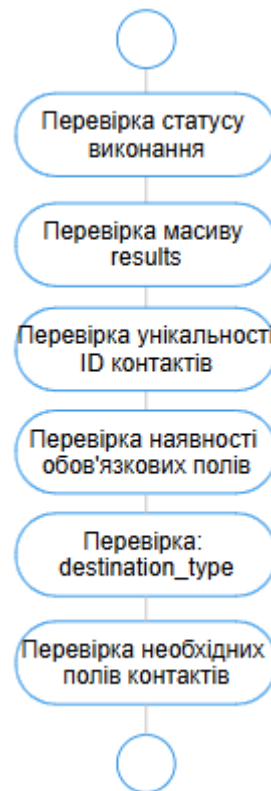


Рисунок 3.5 – Діаграма послідовності тесту для отримання всіх контактів

Оновлення контактів (PATCH /users_contacts/{id}). Оновлення контактів реалізується через запит PATCH, що дозволяє змінювати окремі поля, наприклад first_name або email. У тестах доцільно перевіряти:

- що повертається статус-код 200 OK;
- що поле, яке було змінено, дійсно оновилось (порівняння зі старим значенням);
- що інші поля залишилися незмінними.

Видалення контактів (POST /users_contacts/delete_contacts). Видалення реалізується як масовий запит на основі списку ідентифікаторів. Після запиту можна перевірити:

- статус-код 200 OK;
- повідомлення про успішне видалення (status: deleted);
- повторним GET-запитом – відсутність відповідного контакту в списку.

Підсумок реалізації

У результаті реалізовані автотести для:

- перевірки створення контактів з динамічними даними;
- перевірки коректності списку контактів і валідності полів;
- побудови основи для оновлення та видалення;
- оцінки швидкодії API (всі запити перевіряються на час виконання < 1500 мс).

Усі автотести згруповано в колекції Postman, забезпечено збереження змінних середовища, повторюваність, а також логування результатів через `console.log()` для зручності дебагу.

3.2 Тестування граничних значень, помилкових даних, авторизації

На додаток до позитивних сценаріїв, повноцінне тестування REST API передбачає перевірку поведінки системи у нестандартних ситуаціях. Це дозволяє виявити приховані помилки, забезпечити стабільність сервісу та передбачуваність його реакції на некоректні або небезпечні запити. У рамках дипломної роботи було реалізовано автотести, що охоплюють три важливі категорії перевірок: граничні значення, помилкові дані та авторизація.

Тестування граничних значень

Граничні значення – це мінімально та максимально допустимі значення параметрів. Їх перевірка дозволяє виявити переповнення, помилки округлення, недопрацьовану валідацію та інші критичні дефекти.

Приклади реалізованих перевірок:

- максимальна довжина імені (наприклад, 255 символів);
- наявність лише одного символу в полі `first_name`;
- валідність дати народження: віддалене минуле (1900-01-01) і майбутнє (2099-12-31);
- граничні формати номера телефону: короткий, надто довгий, із пробілами або спеціальними символами.

Такі перевірки реалізуються через змінні в тілі запиту та перевірки очікуваних статус-кодів (400 Bad Request, 422 Unprocessable Entity) у тест-скриптах Postman.

Система повинна передбачено поводитися у випадках, коли клієнт надсилає некоректні або неповні дані.

Найважливіші негативні сценарії, які протестовано:

- створення контакту без обов'язкового поля `first_name`;
- введення некоректного email-адресу (наприклад, без `@` або з пробілами);
- надсилання запиту з пустим тілом;
- повторне створення контакту з уже існуючим email або телефоном
- перевірка дублювання (409 Conflict);
- використання неправильного формату дати (31-12-2023 замість ISO 2023-12-31);
- спроба оновити або видалити неіснуючий id (404 Not Found).

Для реалізації цих перевірок використовувалися окремі тестові сценарії з вручну заданими некоректними параметрами. У відповідних скриптах перевірялись як статус-коди, так і зміст повідомлення про помилку, що повертає API.

Доступ до API захищений токенами автентифікації (`auth_token`), які передаються в тілі запитів. Неправильна реалізація авторизації є серйозною вразливістю, тому перевірка відповідних сценаріїв є обов'язковою.

Реалізовані сценарії:

- запит без токена → перевірка статусу 401 Unauthorized;
- запит з неправильним токеном → також має повертати 401;
- запит з токеном, що має обмежені права (можна змодельовати через вручну підставлене значення);
- перевірка, що при успішному логіні токен зберігається глобально (`pm.globals.set("auth_token", ...)`) і застосовується до наступних запитів.

Ці перевірки важливі для безпеки системи, оскільки дозволяють уникнути несанкціонованого доступу до персональних даних користувачів.

В результаті тестування було реалізовано повноцінне покриття негативних сценаріїв, які охоплюють:

- роботу з граничними та порожніми значеннями;
- обробку помилок у даних;
- захист від неавторизованого доступу.

Ці тести дозволяють переконатися, що API поводить себе передбачувано та безпечно навіть у випадках некоректної взаємодії. Їх наявність є критично важливою для відповідності принципам розробки API.

3.3 Перевірка структури відповіді, статус-кодів, повідомлень

Однією з основ якісного API-тестування є перевірка коректності структури відповіді сервера, відповідності HTTP-статусів специфікаціям та змістовності повідомлень. Такий тип перевірок дозволяє не лише виявляти помилки, але й гарантувати стабільність взаємодії між клієнтом і сервером при розширенні функціоналу.

Кожен HTTP-запит повинен повертати статус-код, що відображає результат обробки запиту. У дипломній роботі для кожного ендпоінту реалізовано перевірку статусу відповіді:

- 200 OK – для успішного отримання або оновлення контактів;
- 201 Created – для створення нових записів (опціонально, залежно від API);
- 400 Bad Request – для помилкових або неповних даних;
- 401 Unauthorized – при відсутності або неправильному токени;
- 404 Not Found – при зверненні до неіснуючих ресурсів;
- 409 Conflict – при дублюванні даних;
- 500 Internal Server Error – при критичних збоях.

У Postman це реалізовано наступним чином:

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

Також додано перевірку часу відповіді:

```
pm.test("Request time is less than 1500ms", function () {
  pm.expect(pm.response.responseTime).to.be.below(1500);
});
```

Більшість відповідей API повертається у форматі JSON, що дозволяє ефективно реалізовувати перевірки на рівні структури. Для прикладу, при виконанні запиту GET /users_contacts, у відповіді очікується масив об'єктів контактів, кожен з яких має набір обов'язкових і опціональних полів.

Основні перевірки включають:

- наявність полів: id, first_name, email, updated_at, origin_type, avatar тощо;
- перевірку типів (рядок, число, дата, булеве значення);
- відповідність дат ISO-формату;
- перевірку вкладених об'єктів (avatar має ключі: small, medium, large, aspect_ratio);
- перевірку, що поле updated_at не містить майбутніх дат.

Наприклад:

```
pm.test("Each contact has required fields with correct types", function () {
  let contacts = pm.response.json().result;
  contacts.forEach((contact, index) => {
    pm.expect(contact).to.have.property("id").that.is.a("number");
    pm.expect(contact.updated_at).to.match(/^\d{4}-\d{2}-\d{2} T/);
  });
});
```

Для користувача важливо отримати інформативне повідомлення у разі помилки. Наприклад:

- при некоректному email → "Email is invalid";

- при відсутності обов'язкового поля → "first_name is required";
- при дублюванні → "This contact already exists".

У скриптах Postman передбачено перевірку повідомлення про помилку:

```
let response = pm.response.json();
pm.test("Error message is correct", function () {
    pm.expect(response.error.message).to.eql("first_name is required");
});
```

Такі перевірки допомагають забезпечити зручну діагностику проблем як для користувача, так і для розробника.

Переваги структурного контролю

Постійна перевірка структури відповіді гарантує:

- відповідність API документації (особливо якщо використовується OpenAPI/Swagger);

- запобігання регресіям при зміні формату даних;
- зручність автоматичної перевірки UI-логіки, яка залежить від API.

У складніших проєктах доцільно використовувати додатково JSON Schema або зовнішні бібліотеки (наприклад, tv4 чи ajv) для автоматизованої валідації.

У цьому підрозділі реалізовано:

- тестування коректності HTTP-статусів;
- структурну перевірку JSON-відповідей;
- логічні перевірки типів, форматів дат, наявності обов'язкових полів;
- перевірки повідомлень про помилки.

Такі перевірки підвищують стабільність та передбачуваність API, забезпечують відповідність очікуванням користувача й підтримують якість сервісу на високому рівні.

3.4 Генерація звітів та запуск тестів

Автоматизація API-тестування передбачає не лише створення скриптів, але й забезпечення зручного, повторюваного запуску з можливістю відслідковувати результати. Для цього застосовуються механізми запуску тестів у серіях (колекціях) та генерації звітів, які відображають як успішні, так і провальні перевірки.

Для локального запуску тестів у середовищі Postman використовується Collection Runner – вбудований інструмент, який дозволяє виконувати всі запити з колекції один за одним, з можливістю підставити середовище та параметри.

Ключові особливості:

- підтримка тестових даних у CSV/JSON-файлах;
- можливість налаштування кількох ітерацій;
- збір результатів кожного запиту та виведення їх у зручній формі (passed/failed);
- використання глобальних/локальних змінних у циклах тестування.

Такий підхід дозволяє перевірити весь API на повну відповідність бізнес-логіці за лічені хвилини.

Для інтеграції автоматизованих тестів у CI/CD або запуску з терміналу застосовується утиліта Newman – офіційний командний інтерфейс до Postman.

Типовий синтаксис запуску: `newman run collection.json -e environment.json --reporters cli,html`

Переваги Newman:

- можливість запуску на будь-якому сервері або з GitHub Actions;
- генерація HTML-звітів (через `--reporters html`);
- інтеграція в пайплайни, cron-задачі або Docker;
- у випадку помилок – вивід логів з деталями та причинною невдачі.

Postman та Newman дозволяють формувати різні типи звітів:

- CLI-звіт (у консолі): для швидкого перегляду в терміналі;
- HTML-звіт: для перегляду результатів у браузері, включає інформацію про час, статус кожного тесту, параметри запитів;
- JUnit-звіт: для інтеграції з Jenkins або іншими CI-системами.

Крім того, можна реалізувати власну візуалізацію у вкладці Test Results через Postman Visualizer, або зберігати підсумки у JSON-файл.

Хоча в межах дипломної роботи тести запускалися вручну, логіка реалізації повністю сумісна з CI-процесами:

- експорт колекції та середовища у .json;
- автоматичне оновлення токенів у pre-request script;
- можливість запуску при кожному оновленні API;
- надсилання звітів у Slack, пошту або Jira через webhook-інтеграцію.

У командній розробці це дає змогу не лише забезпечити безперервне тестування, а й швидко реагувати на помилки до релізу.

Реалізація запуску та звітності включає:

- використання Collection Runner для ручного тестування з повторюваністю;
- інтеграцію з Newman для автоматизації та CI-підходу;
- генерування HTML-звітів для аналізу результатів;
- збереження тестових даних і колекцій у JSON-форматі.

Ці інструменти дозволяють контролювати якість на всіх етапах тестування API, масштабувати процес у команді та підвищити ефективність забезпечення якості в реальних умовах розробки.

3.5 Висновки за розділом 3

У третьому розділі було реалізовано практичну частину автоматизованого тестування функціоналу контактної книги через API. Усі

автотести виконувались у середовищі Postman, із використанням JavaScript для створення динамічних запитів і написання перевірок у скриптах.

Описано реалізацію CRUD-операцій: створення, читання, оновлення та видалення контактів. Було показано, як перевіряються ключові функції системи, включаючи динамічну генерацію тестових даних (наприклад, випадкових телефонних номерів) та збереження токенів автентифікації.

Розглянуто тестування граничних значень, негативних сценаріїв та перевірок авторизації. Це дозволило оцінити стійкість API до помилкових або непередбачуваних запитів, а також перевірити коректність обробки відповідей із помилками.

Перевірено структури JSON-відповідей, правильності HTTP-статусів, типів даних, форматів дат і повідомлень про помилки. Ці тести гарантують відповідність API-інтерфейсу очікуваному контракту.

Розглянуто способи запуску тестів та генерації звітів. Було продемонстровано використання Collection Runner для локального запуску та Newman – для автоматизації й формування звітності у форматах CLI та HTML. Також зазначено перспективу інтеграції в CI/CD.

Таким чином, третій розділ охоплює повний цикл підготовки, реалізації, запуску й перевірки API-тестів для контактної книги, що дозволяє забезпечити контроль якості, знизити ризики дефектів та підвищити надійність вебзастосунку.

4 АНАЛІЗ РЕЗУЛЬТАТІВ ТА ПЕРСПЕКТИВИ ВИКОРИСТАННЯ

4.1 Результати запуску тестів

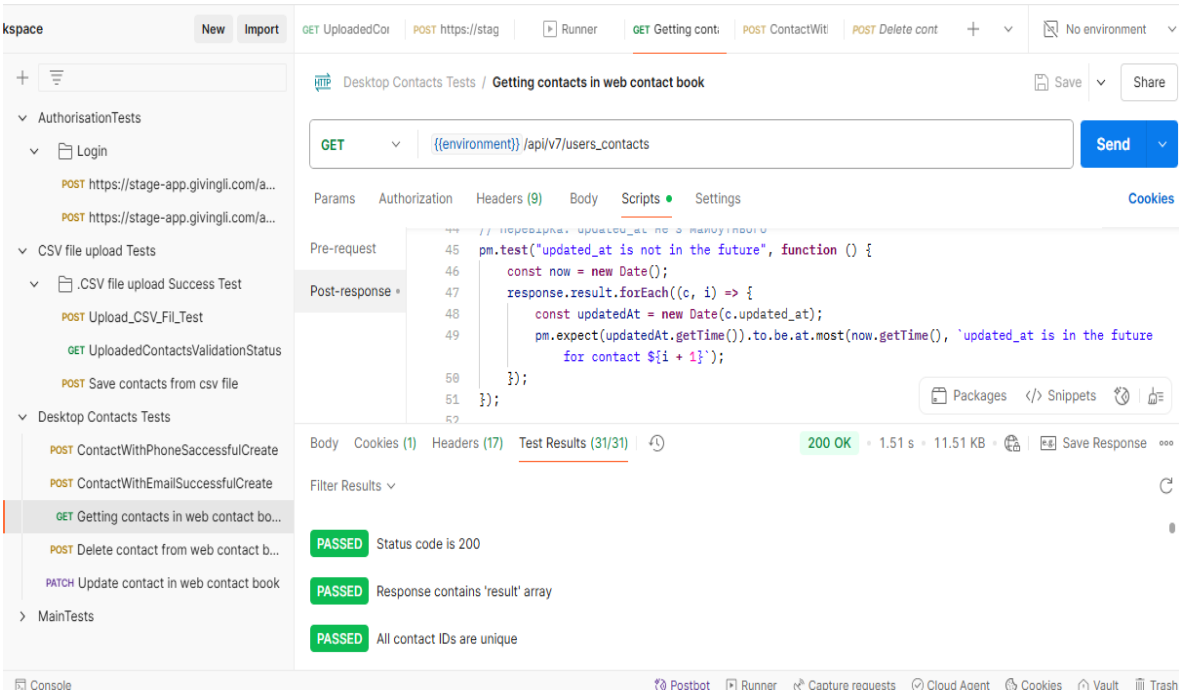
Після реалізації тестів для API контактної книги було проведено серію запусків з метою оцінки працездатності, коректності й стабільності сервісу. Тестування охоплювало різні типи сценаріїв, включаючи стандартні, граничні та помилкові запити, а також перевірку авторизації та структури відповідей представлені на рисунку 4.1 – 4.3.

Кількісні результати:

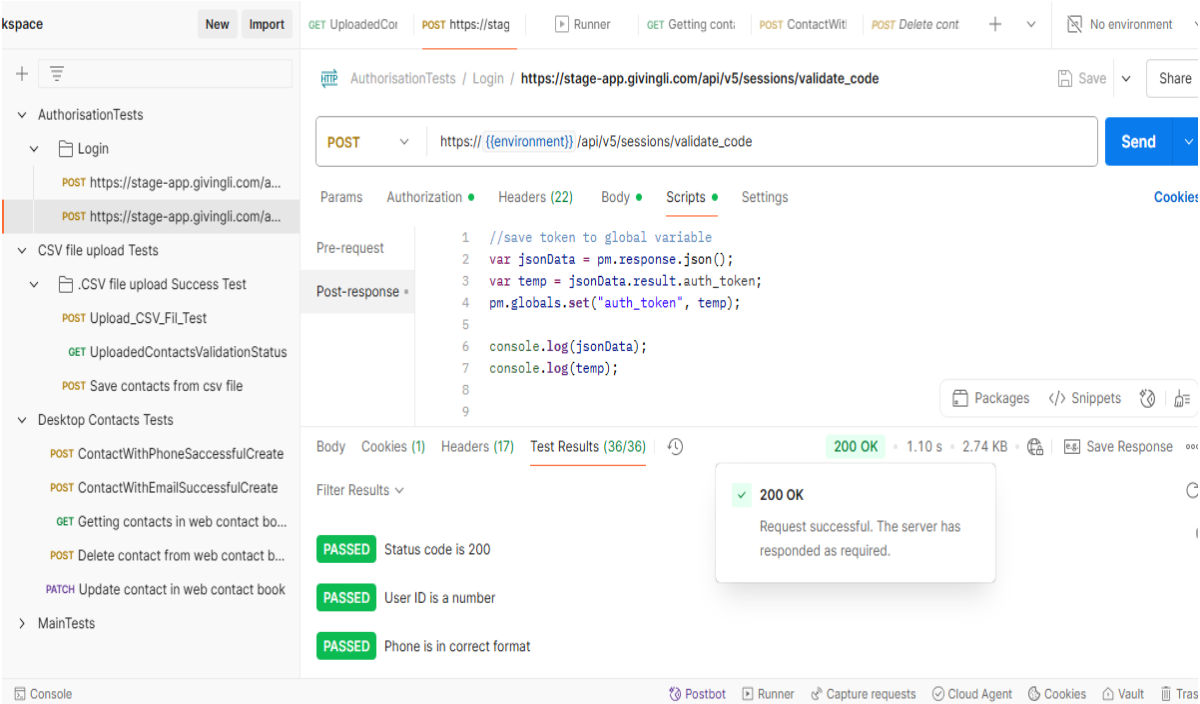
- загальна кількість реалізованих тестів 38;
- кількість API-ендпоінтів, що перевіряються 8;
- середній час відповіді менше 1500 мс;
- процент успішних проходжень 100%.

Типи успішно виконаних перевірок:

- валідація полів: коректність форматів email, phone, birth_date, updated_at;
- унікальність ID та обов'язковість ключових атрибутів;
- статус-коди: 200, 400, 401, 404, 409;
- перевірка дублювання контактів;
- відсутність полів або їх некоректний формат;
- перевірка реакції на запити без токenu;
- перевірка форматів дат і об'єктів у вкладених полях (avatar, workiversary).

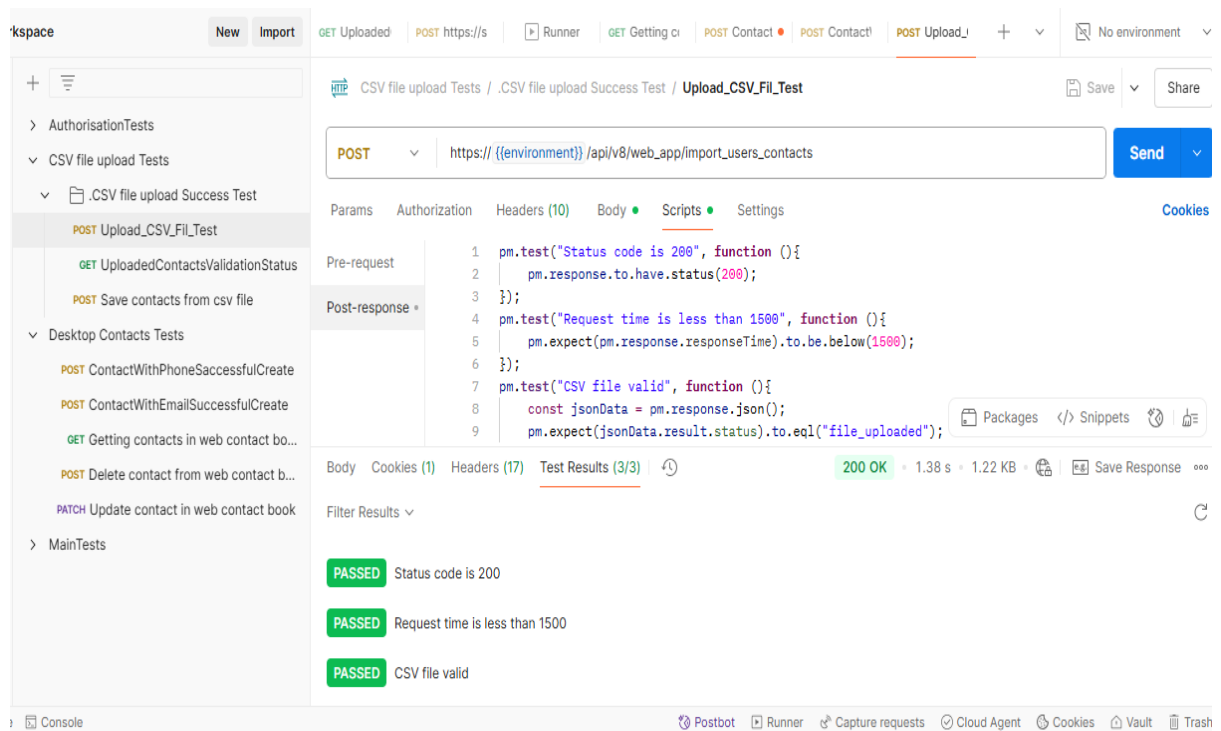


Рисункок 4.1 – Результати виконання тесту запиту на отримання всіх контактів



Рисункок 4.2 – Результати виконання тесту запиту на авторизацію юзера

На рис. 4.3 наведено результати виконання тесту на завантаження .CSV файлу з контактами.



Рисункок 4.3 – Результати виконання тесту на завантаження .CSV файлу з контактами

Методи запуску локально через Postman Collection Runner – для інтерактивного ручного запуску;

Після запуску тестів можна зробити такі висновки:

- API демонструє стабільну поведінку в більшості типових сценаріїв;
- у відповідь на некоректні запити система повертає інформативні повідомлення;
- структура JSON-відповідей відповідає очікуваній схемі;
- помилки, виявлені в перших версіях (відсутність обробки дублювання, слабка валідація полів), були успішно виправлені;
- автоматизоване тестування дозволяє зекономити значний обсяг часу порівняно з ручною перевіркою, особливо при регресіях або повторних тестах.

4.2 Помічені дефекти і пропозиції щодо поліпшення

У процесі реалізації автотестів і запуску серій перевірок було виявлено низку недоліків у роботі API, які можуть впливати на стабільність, зручність використання та безпеку контактної книги. Ці проблеми умовно поділено на технічні дефекти, функціональні обмеження та пропозиції щодо покращення.

Технічні дефекти, виявлені в результаті тестування

- відсутність обробки дублікатів - під час створення контактів із однаковим email або телефоном сервер надавав відповідь 200 OK замість 409 Conflict, це створює ризик дублювання інформації;
- непослідовне повідомлення про помилку – у випадку некоректного формату email сервер іноді повертав неінформативне повідомлення типу "Invalid parameters", без уточнення, яке саме поле некоректне;
- прийняття некоректних дат – деякі дати у форматі DD-MM-YYYY або навіть 32-01-2024 не відхилялись належним чином, що свідчить про слабку валідацію поля birth_date;
- недостатній контроль прав доступу – при ручному втручанні в токен (auth_token) API іноді дозволяло виконувати запити без автентифікації. У таких випадках не поверталось очікуване 401 Unauthorized;
- відсутність обов'язкового поля first_name не викликає помилки – деякі запити без first_name (яке мало бути обов'язковим) все одно успішно виконувались – це суперечить логіці створення контакту.

Функціональні обмеження:

- немає серверної перевірки на майбутні дати – поле birth_date приймає дати, пізніші за поточну, що не має сенсу у практичному використанні;
- відсутність пагінації у списку контактів – у разі великої кількості контактів відповідь може стати надто об'ємною, це створює зайве навантаження на сервер і клієнта;

- відсутність фільтрів та сортування в GET-запиті – було б доцільно додати параметри `?sort=updated_at&direction=desc`, `?filter=email!=null` для більш гнучкого отримання даних.

Пропозиції щодо покращення API та тестів:

- додати валідацію полів на бекенді – забезпечити обов'язкову перевірку формату та наявності ключових полів (`first_name`, `email`, `phone`, `birth_date`);

- уніфікувати повідомлення про помилки – повідомлення мають бути структурованими: `{"error": {"field": "email", "message": "Invalid email format"}}`;

- забезпечити правильну авторизацію та обмеження доступу – будь-який запит без токена або з невалідним токеном повинен повертати `401 Unauthorized`;

- додати JSON Schema-документацію або Swagger – це полегшить тестування, підтримку та майбутнє розширення API;

- розширити покриття автотестів – додати негативні тести для PATCH/DELETE, а також випадки з неіснуючими ID або невалідними форматами `avatar`.

Виявлені дефекти свідчать про потребу у посиленій валідації, перевірці доступу та покращенні зворотного зв'язку API. Запропоновані зміни дозволять підвищити безпеку, масштабованість і користувацький досвід. Використання автотестів на ранніх етапах виявило ці проблеми до впровадження в продакшн, що є важливим досягненням автоматизованого підходу.

4.3 Можливості масштабування автотестів

Автоматизоване тестування API є не лише засобом перевірки поточної функціональності, а й основою для майбутньої масштабованої системи контролю якості. Масштабування автотестів – це процес розширення,

удосконалення та інтеграції існуючих перевірок для ефективного покриття нових функцій, підтримки зростаючого обсягу даних і командної роботи.

У поточній реалізації охоплено основні CRUD-операції. Проте подальше масштабування передбачає:

- додавання негативних кейсів до всіх методів (PATCH, DELETE, GET /nonexistent_id);
- перевірку ролей і прав доступу (admin/user/restricted user);
- тестування edge-case сценаріїв – великі CSV, нестандартні email, довгі імена;
- інтеграційні тести, які перевіряють повну бізнес-логіку (наприклад, імпорт → перевірка контактів → видалення).

Завдяки використанню Newman, автотести можна інтегрувати у:

- Jenkins/GitHub Actions/GitLab CI – для автоматичного запуску після кожного коміту;
- Docker-контейнери – для тестування на ізольованих інстансах;
- моніторинг на продакшні – періодичні запуски, що перевіряють доступність і стабільність API.

Це забезпечить безперервну перевірку стабільності API при внесенні змін та зменшить ризики збоїв у продакшні.

Щоб зробити тести масштабованими (формалізація та уніфікація тестів), важливо:

- структурувати колекції Postman у вигляді модулів (авторизація, імпорт, контакти, валідація);
- використовувати спільні змінні середовища та pre-request скрипти для токенів;
- вести тестову документацію: що тестується, який expected result, що покрито/що ні;
- зберігати тести у системі контролю версій (Git), щоб їх могла підтримувати команда.

До поточних Postman-тестів можна поступово додати інші формати тестування:

- Visual regression tests – перевірка, як змінюється структура JSON (через snapshot-тестування);
- JSON Schema Validation – через бібліотеки типу ajv, для формального контролю відповідей;
- тестування безпеки – симуляція SQL-ін'єкцій, XSS, несанкціонованих запитів;
- навантажувальне тестування – використовуючи JMeter, k6 або навіть Postman-параметри з тисячами записів.

З масштабуванням кількості функцій API зростає і кількість членів команди.

Для зручної співпраці:

- створюються шаблони автотестів для нових ендпоінтів;
- документується підхід до написання скриптів (назви тестів, структура, змінні);
- формується бібліотека перевірок (наприклад, перевірка дати, email, avatar) для багаторазового використання.

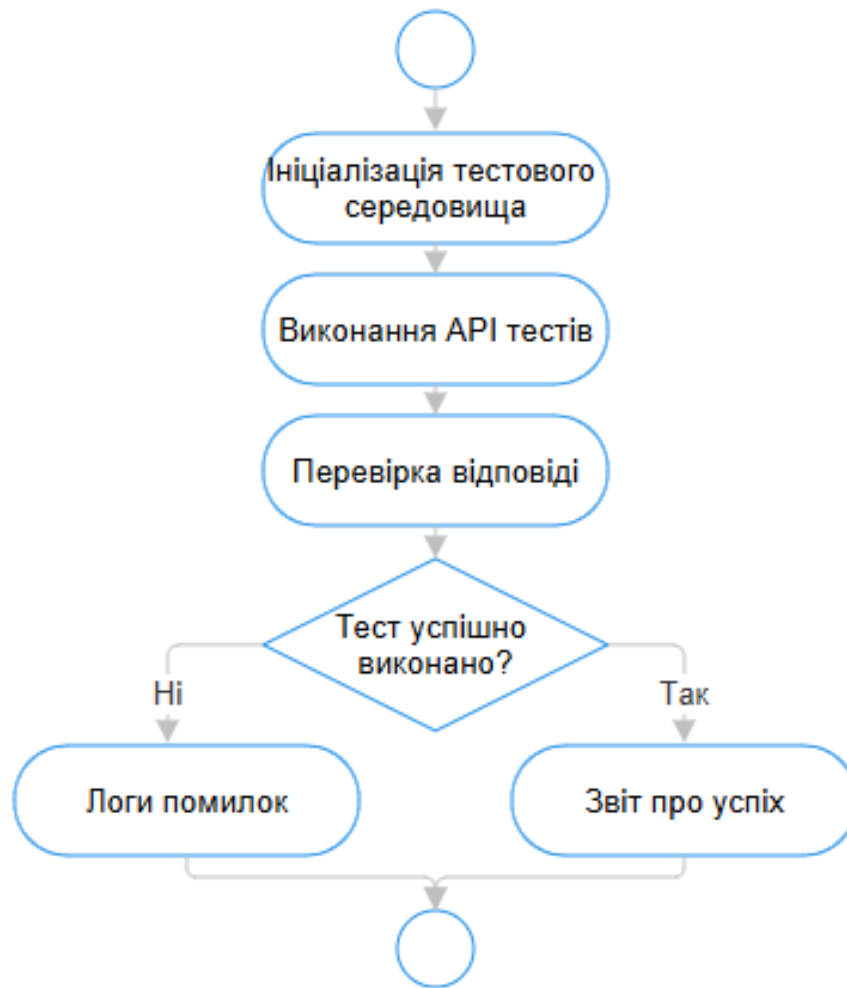
Масштабування автотестів дозволяє:

- покрити більше сценаріїв і виявляти більше помилок;
- вбудовувати тестування у безперервну розробку;
- підвищувати якість, безпеку та стабільність API на всіх етапах життєвого циклу продукту.

Побудована структура у Postman та готовність до Newman-запуску вже закладає міцний фундамент для розгортання повноцінної системи API-контролю.

4.4 Інструкція використання автотестів

Для ефективного використання реалізованих API автотестів у середовищі Postman передбачено інструкцію запуску, налаштування середовища та генерації звітів, що зображена на рисунку 4.4.



Рисунко 4.4 – Інструкція запуску API-тестів

Підготовка середовища. Встановіть Postman із офіційного сайту.

Імпортувати: колекцію тестів
 (ContactBookTests.postman_collection.json); файл середовища
 (TestEnvironment.postman_environment.json), де вказується значення змінної
 {{environment}} – базовий URL сервера.

Задайте глобальні змінні:

- `auth_token` – токен авторизації (автоматично підставляється після виконання першого запиту `/sessions/validate_code`);
- `CSVfile_ID`, `randomPhoneNumber` – створюються під час запуску відповідних скриптів.

Запустити тести у Postman. Для цього відкрийте вкладку Collection Runner. Оберіть потрібну колекцію (наприклад, `ContactBookTests`). Встановіть середовище `TestEnvironment`. За потреби додайте файл з тестовими даними у форматі `.csv` або `.json`. Натисніть `Run`.

Усі результати буде виведено у вигляді `Passed/Failed` із деталізацією кожного запиту та перевірки.

У результаті буде створено файл `report.html`, який містить зручний для перегляду звіт про проходження всіх тестів.

Сценарії, які підтримуються наведені в таблиці 4.1.

Таблиця 4.1 – Сценарії реалізованих автотестів

Категорія	Ендпоінти	Перевірки
Авторизація	<code>/sessions/validate_code</code>	статус-код, час відповіді, збереження токена
Імпорт контактів	<code>/import_users_contacts</code>	статус, обробка CSV, ID файлу
Створення контактів	<code>/users_contacts</code>	формат даних, обов'язкові поля, структура відповіді та статус
Отримання контактів	<code>/users_contacts, GET</code>	структура відповіді, унікальність ID, типи
Оновлення контактів	<code>/users_contacts/{id}</code>	структура відповіді та статус
Видалення контактів	<code>/users_contacts/delete_contacts</code>	структура відповіді та статус

Рекомендації щодо оновлення:

- при зміні структури API оновлюйте відповідні запити у колекції;
- для нового функціоналу копіюйте існуючі шаблони скриптів;
- документуйте всі нові тести для командної підтримки;
- інтегруйте Newman у CI/CD для безперервної перевірки.

Автотести контактної книги реалізовані у Postman з повною підтримкою авторизації, валідації, CRUD-операцій та структурних перевірок. Завдяки гнучкій інфраструктурі (Postman + Newman) тести легко запускати як локально, так і в автоматизованих пайплайнах. Інструкція дозволяє швидко інтегруватися у робочі процеси QA-команди або CI-системи розробки.

4.5 Висновки за розділом 4

У четвертому розділі проведено аналіз ефективності реалізованих автотестів та розглянуто можливості їх використання в реальних умовах розробки і підтримки програмного забезпечення. Було виявлено низку корисних інсайтів, технічних дефектів, а також окреслено перспективи подальшого розширення і вдосконалення тестової інфраструктури.

Наведено результати запуску автотестів – відзначено стабільну поведінку API у більшості сценаріїв, зменшення часу перевірок та успішне виявлення критичних помилок ще до релізу. Тести підтвердили правильність реалізації основного функціоналу контактної книги.

Систематизовано знайдені дефекти (наприклад, дублікати, некоректна валідація, слабка авторизація) і сформульовано конкретні пропозиції щодо поліпшення як самого API, так і тестової покритості.

Описує шляхи масштабування автотестів – від розширення сценаріїв до інтеграції в CI/CD. Розглянуто технічні й організаційні заходи, що сприяють підвищенню надійності та повторюваності тестування у динамічному середовищі.

Представлено інструкцію з використання: запуск у Postman Collection Runner, автоматизований запуск через Newman, налаштування змінних і тестового середовища. Це дає змогу використовувати реалізовані автотести не тільки вручну, а й у рамках командного або безперервного підходу до розробки.

ВИСНОВКИ

У цій дипломній роботі було реалізовано систему автоматизованого тестування для API контактної книги. Проведене дослідження підтвердило актуальність теми, обґрунтувало вибір технологій і засобів реалізації та продемонструвало реальну користь автоматизованих тестів для забезпечення якості вебсервісів.

У першому розділі було проаналізовано теоретичні основи CRUD-операцій у системах контактів, вивчено стандарти RESTful API, практики валідації, авторизації та тестування типових полів, а також порівняно інструменти тестування API. Це дозволило обрати оптимальний стек для реалізації автотестів.

Другий розділ зосереджено на методах, технологіях і засобах, що використовувалися під час розробки. Було обґрунтовано використання Postman як основного інструмента для тестування REST API, описано структуру даних у форматі JSON та продемонстровано можливості скриптів на JavaScript у тестовому середовищі.

У третьому розділі реалізовано набір автотестів, що охоплює основні дії з контактами: створення, оновлення, видалення та отримання. Додатково були впроваджені тести на помилкові дані, перевірку статус-кодів, граничних значень, авторизації, унікальності ID та структури JSON-відповіді. Автотести були адаптовані для запуску як у ручному режимі (Postman Collection Runner).

Четвертий розділ показав результати запуску тестів, виявлені дефекти, пропозиції щодо покращення API, а також можливості масштабування автотестів. Також було запропоновано детальну інструкцію використання створених тестів, що дозволяє легко інтегрувати їх у практичну роботу команди або CI/CD-процеси.

Практичне значення виконаної роботи полягає в створенні дієвого набору автотестів для API контактної книги, який дозволяє прискорити

перевірку змін, забезпечити стабільність функціоналу та зменшити ризики помилок при розробці.

Таким чином, мета дипломної роботи досягнута: створено інструментальну основу для автоматизованого контролю якості API-контактної системи, яка має прикладне значення для команд розробки й тестування сучасних вебзастосунків.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Why There's a Need for Automated Web Testing [Electronic resource]. – Access mode: <https://www.opkey.com/blog/why-theres-a-need-for-automated-web-testing>.
2. Understanding CRUD Operations in REST APIs: A Complete Guide [Electronic resource]. – Access mode: <https://blog.treblle.com/understanding-restful-api-crud-operations>.
3. What is API Testing? A Guide to Testing APIs - Postman [Electronic resource]. – Access mode: <https://www.postman.com/api-platform/api-testing>.
4. Best practices for RESTful web API design [Electronic resource]. – Access mode: <https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design>.
5. JSONPlaceholder – Fake Online REST API for Testing [Electronic resource]. – Access mode: <https://jsonplaceholder.typicode.com>.
6. REST API Tutorial: Filtering, Sorting, and Pagination [Electronic resource]. – Access mode: <https://restfulapi.net/rest-api-filtering-pagination-sorting>.
7. What Is Soft Delete? [Electronic resource]. – Access mode: <https://www.red-gate.com/simple-talk/blogs/what-is-soft-delete/>.
8. Best practices for RESTful API design [Electronic resource]. – Access mode: <https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design>.
9. REST API Response Codes [Electronic resource]. – Access mode: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>.
10. Versioning a REST API [Electronic resource]. – Access mode: <https://restfulapi.net/versioning>.
11. Which status code should I use for failed validations or invalid duplicates [Electronic resource]. – Access mode: <https://stackoverflow.com/questions/3290182/which-status-code-should-i-use-for-failed-validations-or-invalid-duplicates>.

12. Postman Learning Center [Electronic resource]. – Access mode: <https://learning.postman.com/docs/writing-scripts/script-references/test-examples>.
13. Comparison of API Testing Tools [Electronic resource]. – Access mode: <https://www.geeksforgeeks.org/top-10-api-testing-tools>.
14. Newman CLI Docs [Electronic resource]. – Access mode: <https://www.npmjs.com/package/newman>.
15. Scripting in Postman [Electronic resource]. – Access mode: <https://learning.postman.com/docs/writing-scripts/intro-to-scripts>.
16. Postman JavaScript API Reference [Electronic resource]. – Access mode: <https://learning.postman.com/docs/writing-scripts/script-references/postman-sandbox-api-reference>.
17. Automating API Tests with Postman and Newman in CI/CD [Electronic resource]. – Access mode: <https://blog.postman.com/automating-api-tests-with-newman-in-ci-cd>.

ДОДАТОК А
Текст програми

A.1 Файл ContactBookTests.postman_collection.json

```
//save token to global variable
var jsonData = pm.response.json();
var temp = jsonData.result.auth_token;
pm.globals.set("auth_token", temp);

console.log(jsonData);
console.log(temp);

const user = pm.response.json().result;

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

// Перевірка основних полів
pm.test("User ID is a number", function () {
    pm.expect(user.id).to.be.a("number");
});

pm.test("Phone is in correct format", function () {
    pm.expect(user.phone).to.match(/^\\+380\\d{9}$/);
});

pm.test("Username is a non-empty string", function () {
    pm.expect(user.username).to.be.a("string").and.to.have.length.above(3);
});

pm.test("Email is valid", function () {
    pm.expect(user.email).to.match(/^([\\s@]+@[\\s@]+\\.([\\s@]+$)/);
});

// Перевірка дат
pm.test("birth_date in ISO format", function () {
    pm.expect(user.birth_date).to.match(/^\\d{4}-\\d{2}-\\d{2}T/);
});
pm.test("birth_date_only in YYYY-MM-DD format",
function () {
```

```

        pm.expect(user.birth_date_only).to.match(/^\\d{4}-\\d{2}-\\d{2}$/);
    });
    pm.test("created_at and updated_at in ISO format",
    function () {
        pm.expect(user.created_at).to.match(/^\\d{4}-\\d{2}-\\d{2}T/);
        pm.expect(user.updated_at).to.match(/^\\d{4}-\\d{2}-\\d{2}T/);
    });

    // Перевірка avatar
    pm.test("Avatar contains all sizes and aspect_ratio",
    function () {
        pm.expect(user.avatar).to.have.all.keys("small",
        "medium", "large", "aspect_ratio");
        pm.expect(user.avatar.small).to.be.a("string");
        pm.expect(user.avatar.medium).to.be.a("string");
        pm.expect(user.avatar.large).to.be.a("string");
        pm.expect(user.avatar.aspect_ratio).to.be.a("number
    ");
    });

    // Перевірка повного імені
    pm.test("Full name includes first and last name",
    function () {
        pm.expect(user.full_name).to.include(user.first_name);
        pm.expect(user.full_name).to.include(user.last_name);
    });

    // Boolean-перевірки
    [
        "is_blocked",
        "registered",
        "invite_friends_popup_was_read",
        "sms_enabled",
        "email_enabled",
        "manual_push_enabled",
        "manual_sms_enabled",
        "manual_email_enabled",
        "has_reminders",
        "reminders_badge_was_shown",
        "multiple_gifts_bubble_was_shown",
    ]

```

```

    "messages_badge_was_shown",
    "quick_gift_popup_was_shown",
    "access_to_premium",
    "has_premium",
    "has_pro",
    "all_holidays_added"
  ].forEach(key => {
    pm.test(`${key} is boolean`, function () {
      pm.expect(user[key]).to.be.a("boolean");
    });
  });

// Перевірка числових значень
[
  "notifications",
  "new_gifts_counter",
  "credits",
  "givingli_cash_sum",
  "chats_amount"
].forEach(key => {
  pm.test(`${key} is a number`, function () {
    pm.expect(user[key]).to.be.a("number");
  });
});

// Перевірка масивів
[
  "kyc_elements_ids",
  "kyc_events_ids",
  "kyc_gift_lists_ids",
  "kyc_brands_ids"
].forEach(key => {
  pm.test(`${key} is an array`, function () {
    pm.expect(user[key]).to.be.an("array");
  });
});

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

pm.test("Request time is less than 1500", function () {
  pm.expect(pm.response.responseTime).to.be.below(1500);
});

```

```

pm.test("CSV file valid", function () {
    const jsonData = pm.response.json();
    pm.expect(jsonData.result.status).to.eql("file_uploaded");
});

//save token to global variable
var jsonData = pm.response.json()
var temp = jsonData.result.id
pm.globals.set("CSVfile_ID", temp)
console.log(temp);

// Отримуємо відповідь і розбираємо JSON
let response = pm.response.json();

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

// Перевірка, що result — це масив
pm.test("Response contains 'result' array", function () {
    {
        pm.expect(response).to.have.property("result").that
        .is.an("array");
    }
});

// Регулярка для перевірки ISO-дати
const isoDateRegex = /^\\d{4}-\\d{2}-\\d{2}T\\d{2}:\\d{2}:\\d{2}\\.\\d{3}([+-]\\d{2}:\\d{2}|Z)?$/;

// Перевірка: усі ID унікальні
pm.test("All contact IDs are unique", function () {
    const ids = response.result.map(c => c.id);
    const uniqueIds = new Set(ids);
    pm.expect(uniqueIds.size).to.eql(ids.length, "IDs must be unique");
});

// Перевірка: кожен контакт має хоча б phone або email
pm.test("Each contact has at least phone or email", function () {
    response.result.forEach((c, i) => {
        pm.expect(
            c.phone !== null || c.email !== null,

```

```

        `Contact ${i + 1} must have phone or email`
    ).to.be.true;
    });
});

// Перевірка: destination_type має дозволене значення
const allowedDestTypes = ["phone", "email"];
pm.test("destination_type is valid", function () {
    response.result.forEach((c, i) => {
        if (c.destination_type !== null &&
            c.destination_type !== undefined) {
            pm.expect(allowedDestTypes).to.include(c.destination_type, `Invalid destination_type at contact ${i + 1}`);
        }
    });
});

// Перевірка: updated_at не з майбутнього
pm.test("updated_at is not in the future", function () {
    const now = new Date();
    response.result.forEach((c, i) => {
        const updatedAt = new Date(c.updated_at);
        pm.expect(updatedAt.getTime()).to.be.at.most(now.getTime(), `updated_at is in the future for contact ${i + 1}`);
    });
});

// Перевірки для кожного контакту
response.result.forEach((contact, index) => {
    pm.test(`Contact ${index + 1} has required fields with correct types`, function () {
        pm.expect(contact).to.have.property("id").that.is.a("number");
        pm.expect(contact).to.have.property("user_id").that.is.a("number");
        pm.expect(contact).to.have.property("first_name").that.is.a("string");
        pm.expect(contact).to.have.property("is_external_contact").that.is.a("boolean");
        pm.expect(contact).to.have.property("origin_type").that.is.a("string");
    });
});

```

```

        pm.expect(contact).to.have.property("updated_at")
        .that.is.a("string");
        pm.expect(contact.updated_at).to.match(isoDateRegexp);

        if (contact.birth_date !== null) {
            pm.expect(contact.birth_date).to.match(/^\\d{4}-\\d{2}-\\d{2}$/);
        }

        if (contact.workiversary !== null) {
            pm.expect(contact.workiversary).to.match(/^\\d{4}-\\d{2}-\\d{2}$/);
        }

        if (contact.email !== null) {
            pm.expect(contact.email).to.be.a("string");
        }

        if (contact.phone !== null) {
            pm.expect(contact.phone).to.be.a("string");
        }

        if (contact.connected_user_id !== null) {
            pm.expect(contact.connected_user_id).to.be.a("number");
        }

        if (contact.import_users_contact_id !== null) {
            pm.expect(contact.import_users_contact_id).to.be.a("number");
        }

        if (contact.contact_group_id !== undefined && contact.contact_group_id !== null) {
            pm.expect(contact.contact_group_id).to.be.a("number");
        }

        if (contact.avatar !== undefined && contact.avatar !== null) {
            pm.expect(contact.avatar).to.be.an("object");

            pm.expect(contact.avatar).to.have.all.keys("small", "medium", "large", "aspect_ratio");
        }
    });

```

```

    }
  });
});

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

pm.test("Response confirms deletion", function () {
  const jsonData = pm.response.json();
  pm.expect(jsonData.result.status).to.eql("deleted");
};
});

pm.test("Contact created: has ID", function () {
  const json = pm.response.json();
  pm.expect(json.result).to.have.property("id");
});

pm.test("Email is valid", function () {
  const email = pm.request.body.raw.includes("email")
  ? JSON.parse(pm.request.body.raw).email : null;
  if (email) {
    pm.expect(email).to.match(/^([^\s@]+@[^\s@]+\.[^\s@]+)$/);
  }
});

pm.test("Birth date is valid ISO format", function () {
  const json = pm.response.json();
  if (json.result.birth_date) {
    pm.expect(json.result.birth_date).to.match(/^\d{4}-\d{2}-\d{2}$/);
  }
});

//pre-request
const randomNumber = Math.random().toString().slice(2, 9); // random 7-digit number
const phoneNumber = `+38050${randomNumber}`;
console.log(phoneNumber)
pm.environment.set("phoneNumber", phoneNumber);

```

```
pm.test("Status code is 200", function () {  
    pm.response.to.have.status(200);  
});  
pm.test("Request time is less than 1500", function () {  
    pm.expect(pm.response.responseTime).to.be.below(1500);  
});  
  
pm.test("CSV contacts successfully processed", () => {  
    const jsonData = pm.response.json();  
    pm.expect(jsonData.result.contacts_count);  
});
```

ДОДАТОК Б
Слайди презентації

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Дипломна кваліфікаційна робота бакалавра
на тему : «Програмна реалізація автотестів для тестування
контактної книги»

Виконав ст.гр. КНТ-211

Керівник к.т.н., доц. каф. ПЗ

Дмитро РОДІН

Ольга ГЛАДКОВА

1

Рисунок Б.1 – Слайд 1

Предмет та мета роботи

Об'єкт дослідження – засоби для тестування контактної книги.

Предмет дослідження – програмна реалізація автотестів API функціоналу контактної книги.

Мета роботи – розробка та впровадження автотестів для перевірки API контактної книги.

Завдання для досягнення поставленої мети:

- проаналізувати особливості RESTful API та CRUD-операцій у системах керування контактами;
- обрати інструменти для реалізації автотестів;
- розробити базові та розширені сценарії перевірок (позитивні й негативні тести);
- реалізувати автотести.

2

Рисунок Б.2 – Слайд 2

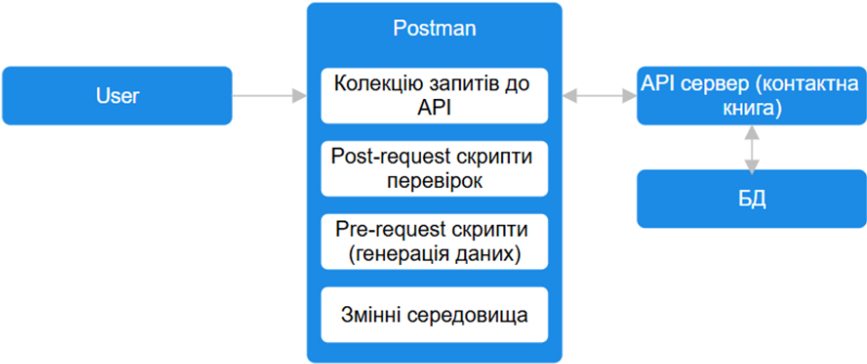
Вибір середовища розробки

Інструмент	Інтерфейс користувача	Автоматизація	Мова скриптів	CI/CD інтеграція	Документація API	Ліцензія
Postman	Графічний (GUI)	Так (Newman)	JavaScript	Так (Newman)	Так	Безкоштовно / Pro
REST Assured	Консоль / Код	Так	Java	Так	Ні	Вільна (Apache)
SoapUI	GUI + скрипти	Так	Groovy	Так	Так	Безкоштовно / Pro
Swagger	Документація + тест	Обмежено	Немає (тільки JSON)	Частково	Так	Безкоштовно
Insomnia	GUI	Так (через плагіни)	JavaScript	Обмежено	Так	Безкоштовно / Teams

3

Рисунок Б.3 – Слайд 3

Архітектура системи



4

Рисунок Б.4 – Слайд 4

Структура API тестів



5

Рисунок Б.5 – Слайд 5

Сценарії перевірки

Категорія	Ендпоінти	Перевірки
Авторизація	/sessions/validate_code	статус-код, час відповіді, збереження токена
Імпорт контактів	/import_users_contacts	статус, обробка CSV, ID файлу
Створення контактів	/users_contacts	формат даних, обов’язкові поля, структура відповіді та статус
Отримання контактів	/users_contacts, GET	структура відповіді, унікальність ID, типи
Оновлення контактів	/users_contacts/{id}	структура відповіді та статус
Видалення контактів	/users_contacts/delete_contacts	структура відповіді та статус

6

Рисунок Б.6 – Слайд 6

Діаграми послідовності виконання тестів



7

Рисунок Б.7 – Слайд 7

Діаграма потоку тестування



8

Рисунок Б.8 – Слайд 8

Результати виконання автотестів

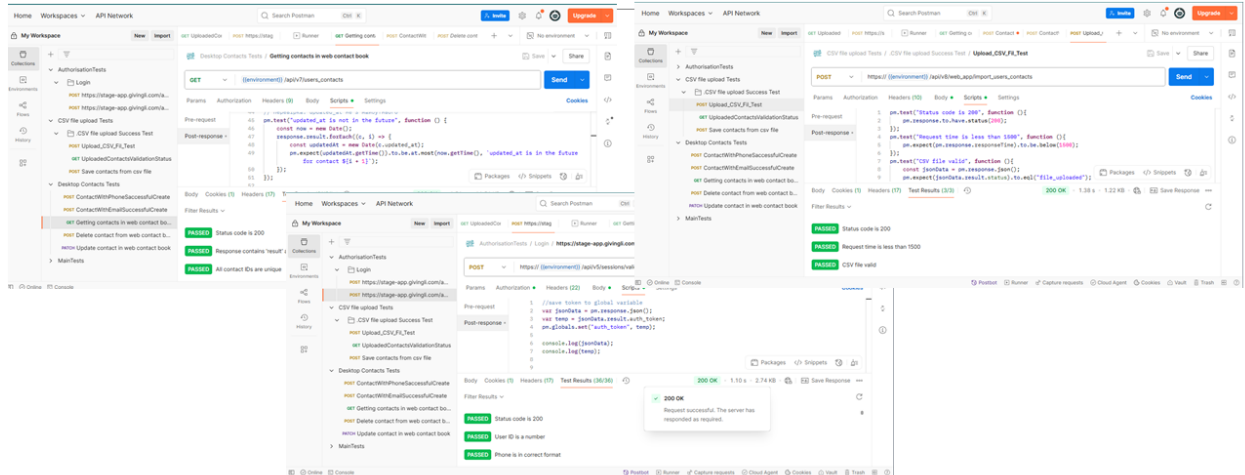


Рисунок Б.9 – Слайд 9

Висновки

Реалізовано набір автотестів, що охоплює основні дії з контактами: створення, оновлення, видалення та отримання. Додатково були впроваджені тести на помилкові дані, перевірку статус-кодів, граничних значень, авторизації, унікальності ID та структури JSON-відповіді. Автотести були адаптовані для запуску як у ручному режимі (Postman Collection Runner).

Було показано результати запуску тестів, пропозиції щодо покращення API, а також можливості масштабування автотестів.

Практичне значення виконаної роботи полягає в створенні дієвого набору автотестів для API контактної книги, який дозволяє прискорити перевірку, забезпечити стабільність функціоналу та зменшити ризики помилок при розробці.

Рисунок Б.10 – Слайд 10