Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних наук
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(назва освітнього ступеня)

на тему: Аналіз методів тестування програмних інтерфейсів для RESTFul API

Виконав(ла): студент(ка) 6 курсу, групи СНм-61
спеціальності 122 Комп'ютерні науки
(шифр і назва спеціальності)

(підпис) Яцишин В.А. (прізвище та ініціали)

Керівник (підпис) Дмитроца Л.П. (прізвище та ініціали)

Нормоконтроль (підпис) Никитюк В.В. (прізвище та ініціали)

Завідувач кафедри (підпис) Боднарчук І.О. (прізвище та ініціали)

Рецензент (підпис) (прізвище та ініціали)

Тернопіль
2025

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра комп'ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Боднарчук І.О.
(підпис) (прізвище та ініціали)
"17" листопада 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Магістр
(назва освітнього ступеня)

за спеціальністю 122 Комп'ютерні науки
(шифр і назва спеціальності)

студенту Яцишин Владислав Андрійович
(прізвище, ім'я, по батькові)

1. Тема роботи **Аналіз методів тестування програмних інтерфейсів для RESTFul API**

Керівник роботи **к.т.н., доц. Дмитроца Л.П.**
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від "27" листопада 2025 року № 4/7-1042

2. Термін подання студентом завершеної роботи 22 грудня 2025 р.

3. Вихідні дані до роботи Літературні джерела з тематики роботи

4. Зміст роботи (перелік питань, які потрібно розробити)
ВСТУП;1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ;1.1 Огляд інструментів для тестування програмних інтерфейсів (API);1.2 Аналіз предметної області;1.3 Опис методології дослідження;2 ПІДХОДИ ДО РОЗРОБКИ ТЕСТІВ ДЛЯ RESTFULL API;2.1 Розробка модульних тестів RESTful API;2.2 Покриття коду тестами;2.3 Пропозиції щодо генерації модульних тестів;2.4 Проблема автентифікації при модульному тестуванні RESTful API;3 ПІДСУМОК ОГЛЯДУ ЛІТЕРАТУРИ;3.1 Основні проблеми тестування RESTful API;3.2 Напрямки подальших досліджень;3.3 Приклад тестування RESTful API;4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ;4.1 Питання щодо охорони праці і галузі інформаційних технологій;4.2 Питання щодо безпеки в надзвичайних ситуаціях;4.3 Висновок до четвертого розділу;ВИСНОВКИ;СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ;ДОДАТКИ

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

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці	Сенчишин В.С., к.т.н., доц. каф. МТ		
Безпека в надзвичайних ситуаціях	Теслюк В.М., проректор з адміністративно-господарської роботи та будівництва		

7. Дата видачі завдання 17 листопада 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	17.11.25-18.11.25	Виконано
2.	Підбір наукових джерел за темою роботи	19.11.25-20.11.25	Виконано
3.	Переклад та опрацювання наукових джерел	20.11.25-23.11.25	Виконано
	за темою кваліфікаційної роботи		
4.	Виконання дослідження щодо теми	24.11.25-10.12.25	Виконано
	кваліфікаційної роботи		
5.	Оформлення першого розділу	11.12.25-12.10.25	Виконано
6.	Оформлення другого розділу	12.12.25-13.12.25	Виконано
7.	Оформлення третього розділу	13.12.25-14.12.25	Виконано
8.	Виконання завдання до підрозділу "Охорона праці"	08.12.25-09.11.25	Виконано
9.	Виконання завдання до підрозділу "Безпека в	10.12.25-12.12.25	Виконано
	надзвичайних ситуаціях"		
10.	Оформлення кваліфікаційної роботи	12.12.25-31.12.25	Виконано
11.	Нормоконтроль	14.12.25-15.12.25	Виконано
12.	Перевірка на плагіат	15.12.25	Виконано
13.	Попередній захист кваліфікаційної роботи	18.12.25	Виконано
14.	Захист кваліфікаційної роботи	23.12.2025	

Студент _____ Яцишин В.А.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Дмитроца Л.П.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

"Аналіз методів тестування програмних інтерфейсів для RESTFul API" // Кваліфікаційна робота освітнього рівня "Магістр" // Яцишин Владислав Андрійович // Тернопільський національний технічний університет ім. І. Пулюя, факультет комп’ютерно-інформаційних систем і програмної інженерії, кафедра комп'ютерних наук, група СНм-61 // Тернопіль, 2025 // с. – 49, рис. – 12, табл. – 0, джерел – 21.

Ключові слова: RESTful API; фреймворки для тестування; мікросервіси; хмарні сервіси; автоматична генерація тестових випадків; базові сервіси JSON

Сервісно-орієнтована архітектура стала основою інтеграції між програмами та платформами, що призвело до появи хмарних сервісів. Великі бізнес-платформи надають послуги кінцевим користувачам та іншим компаніям. REST став стандартним протоколом для впровадження RESTful API. Оскільки внутрішні деталі RESTful API недоступні під час споживання, тестування стало серйозною проблемою. Будь-яка зміна в API може призвести до збою сервісів, фінансових втрат та втрати довіри. Дослідники створили різні фреймворки для автоматичного створення модульних тестів. Однак бракує огляду сучасного стану тестування RESTful API. Метою роботи є виявлення, аналіз та синтез досліджень щодо методологій тестування RESTful API.

ANNOTATION

"Analysis of Programming Interface Testing Methods for RESTFul API" // Master's degree qualification paper // Yatsyshyn Vladyslav // Ternopil Ivan Puluj National Technical University, Faculty of Computer Information Systems and Software Engineering, Computer Science Department, group СНм-61 // Ternopil, 2025 // p. – 49, fig. – 12, tables – 0, references – 21.

Service-oriented architecture has become the foundation for integration between applications and platforms, leading to the emergence of cloud services. Large business platforms provide services to end users and other companies. REST has become the standard protocol for implementing RESTful APIs. Since the internal details of RESTful APIs are not accessible during consumption, testing has become a serious challenge. Any change in the API can lead to service failures, financial losses, and loss of trust. Researchers have created various frameworks for automatic generation of unit tests. However, there is a lack of review of the current state of RESTful API testing. The aim of this work is to identify, analyze, and synthesize research on RESTful API testing methodologies.

ЗМІСТ

ВСТУП

Актуальність задачі.

REST (REpresentational State Transfer – передача стану репрезентації) – це архітектура, що використовується для проєктування сервісів, що споживаються на різних платформах та в різних середовищах, для підтримки сумісності та WWW (Всесвітньої павутини) [5]. Бездержавність та готовність до кросплатформного використання – два основні атрибути цієї архітектури. Вона стала широко поширеним стандартизованим способом публікації сервісів через Інтернет [6]. Інтерфейси прикладного програмування (API) REST є широкою частиною проєктування мікросервісів [7]. Були докладені дослідницькі зусилля для розширення архітектури REST для підтримки розподілених систем [8].

Слабо зв'язний та адаптивний доступ до RESTful-сервісів відкриває широку можливість для надсилання неправильних вхідних даних із запитом. Це потенційно може призвести до помилок сервісу, які могли не бути виявлені під час статичного аналізу та модульного тестування. Додатковий рівень сторонніх бібліотек, що надаються іншими постачальниками, що використовуються в розробці RESTful API, додає складності тестуванню. У цьому сценарії стає ще важливішим виявляти та виправляти можливі помилки в сервісі для забезпечення стабільності. Це особливо серйозно, коли кінцевими клієнтами цих сервісів є критично важливі види діяльності або бізнес. Незалежно від того, що надсилається від клієнта як запит, сервіс повинен мати можливість реагувати коректно, вирішуючи будь-які можливі помилки під час виконання.

Проводились дослідження для полегшення проблем тестування RESTful API шляхом впровадження різних фреймворків та підходів до автоматичної генерації модульних тестів. На жаль в ТНТУ при пошуку літературних джерел було знайдено роботи, що містять опис RESTful. Як інструменту побудови програмних систем, хоча наукові публікації на тему кості ПЗ загалом та програмної архітектури зокрема є [1 – 4].

Однак досі бракує огляду сучасного стану тестування RESTful API. Тому метою цієї статті є виявлення, аналіз та синтез досліджень, проведених стосовно методологій тестування RESTful API та генерації модульних тестів.

Мета роботи.

Таким чином метою роботи є проведення систематизованого літературного огляду на тему методів і засобів проведення процесу тестування програмного інтерфейсу RESTful.

Для досягнення цієї мети варто дати відповіді на запитання:

1. Які основні виклики виникають при генерації модульних тестів для RESTful API.
2. Яким є покриття коду, коли мова йде про тестування RESTful API.
3. Які рішення наразі доступні для вирішення викликів тестування та генерації модульних тестів.
4. Яку підтримку надають рішення для тестування та генерації модульних тестів для RESTful API з увімкненою автентифікацією.

Об’єкт дослідження: процеси тестування програмних інтерфейсів RESTful.

Предмет дослідження: програмні продукти та компоненти, які використовують RESTful API для комунікації.

Наукова новизна отриманих результатів.

В результаті виконання цієї кваліфікаційної роботи було встановлено наступне:

1. На сьогоднішній час є досить небагато наукових досліджень на тему тестування RESTful API.

2. В цій роботі представлено актуальний огляд методологій тестування для RESTful API.

3. Доступні методології класифікуються на основі інструментів, підходів та фреймворків.

4. Визначено та обговорено основну перешкоду в досягненні високого рівня покриття коду.

Практичне значення отриманих результатів.

Результати дослідження мають значну практичну цінність для розробників програмного забезпечення та інженерів з тестування, які працюють з RESTful API. Наданий огляд методологій тестування та їх класифікація за інструментами, підходами та фреймворками дозволить фахівцям обирати ефективні рішення для конкретних проєктів, що сприятиме підвищенню якості програмного забезпечення та скороченню часу на розробку тестів. Виявлення та аналіз основних перешкод у досягненні високого рівня покриття коду надає розробникам практичні рекомендації щодо оптимізації процесу тестування та забезпечення надійності RESTful API. Також можна вважати практичним результатом заповнення існуючого дефіциту наукових досліджень у цій галузі, що робить роботу важливим джерелом інформації на тему тестування RESTful API.

Апробація результатів та особистий внесок здобувача.

Основні положення роботи доповідались, розглядались та обговорювались на науковій конференції Тернопільського національного технічного університету імені Івана Пулюя. Результати кваліфікаційної роботи опубліковані у тезах студентської наукової конференції "Актуальні задачі сучасних технологій – 2025", яка проводилась у ТНТУ.

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд інструментів для тестування програмних інтерфейсів (API)

API є основою сучасного програмного забезпечення, забезпечуючи зв'язок між додатками, сервісами та системами. Оскільки бізнес покладається на них у всьому, від платежів до автентифікації, забезпечення надійної, безпечної та ефективної роботи API є невід'ємним. Саме тут на допомогу приходять інструменти тестування API, які допомагають командам контролю якості перевіряти функціональність, продуктивність та інтеграцію перед випуском. У цьому розділі ми розглянемо декілька інструментів для тестування API, котрі найчастіше використовуються на даний час, їхні функції, переваги та недоліки, а також ключові фактори, які слід враховувати перед вибором.

API (Application Program Interface – інтерфейс прикладного програмування) – це набір правил, що дозволяє двом програмним застосункам взаємодіяти. Наприклад, API платіжного шлюзу обробляє транзакції для застосунку електронної комерції, тоді як API погоди надає прогнози в режимі реального часу для мобільного застосунку. API дозволяють застосункам безперешкодно обмінюватися даними та послугами, формуючи основу сучасного програмного забезпечення, котре в переважній більшості випадків не функціонує автономно, а вимагає підключення до інших програмних продуктів та компонентів.

Інструменти для тестування API – це спеціалізоване програмне забезпечення, яке допомагає розробникам та командам контролю якості перевіряти, чи працюють API правильно, безпечно та ефективно. На відміну від тестування інтерфейсу користувача, яке зосереджується на тому, що бачать користувачі, тестування API перевіряє логіку, надійність та комунікацію за лаштунками. Ці інструменти можуть виконувати різні типи тестування, зокрема:

– Функціональне тестування – перевірка того, що кінцеві точки API повертають очікувані результати.

– Тестування навантаження – забезпечення можливості обробки великих обсягів запитів API.

– Інтеграційне тестування – перевірка того, чи API безперебійно працюють з іншими сервісами або програмами.

– Тестування продуктивності – вимірювання часу відгуку, пропускної здатності та масштабованості за різних умов.

– Тестування безпеки – забезпечення захисту API від таких загроз, як SQL-ін'єкції, XSS або несанкціонований доступ.

– Регресійне тестування – підтвердження того, що оновлення або виправлення помилок не порушують існуючу функціональність API.

Перш ніж перейти до аналізу інструментів тестування, слід звернути увагу, що кожен інструмент має унікальні сильні сторони. Деякі ідеально підходять для автоматизації без кодування, тоді як інші найкраще підходять для середовищ, орієнтованих на розробників.

1. Testsigma.

Testsigma – це платформа автоматизації на базі штучного інтелекту без коду, створена для спрощення тестування API для команд контролю якості будь-якого рівня кваліфікації. Інтерфейс користувача цього інструменту показано на орисунку 1.1. Вона дозволяє створювати тести API простою англійською мовою, усуваючи необхідність написання скриптів. Завдяки таким функціям, як самовідновлювальні тести, управління середовищем та безшовна інтеграція CI/CD, Testsigma допомагає командам об'єднати тестування API, веб-, мобільних та настільних версій в одній платформі.

Особливості Testsigma

– Створення тестів без коду простою англійською мовою з використанням мовної моделі на основі NLP.

– Підтримує REST, GraphQL та всі основні методи HTTP.

– Тестування на основі даних зі змінними в різних середовищах.

– Підтримує GET, POST, PUT, PATCH, DELETE із заголовками та корисними навантаженнями.

– Безперебійна інтеграція з Jenkins, GitHub, GitLab та Azure DevOps.

– Підтверджуйте коди стану, заголовки, час відповіді та перевірки схеми.

– Виконайте той самий тест на кількох наборах даних для ширшого охоплення.

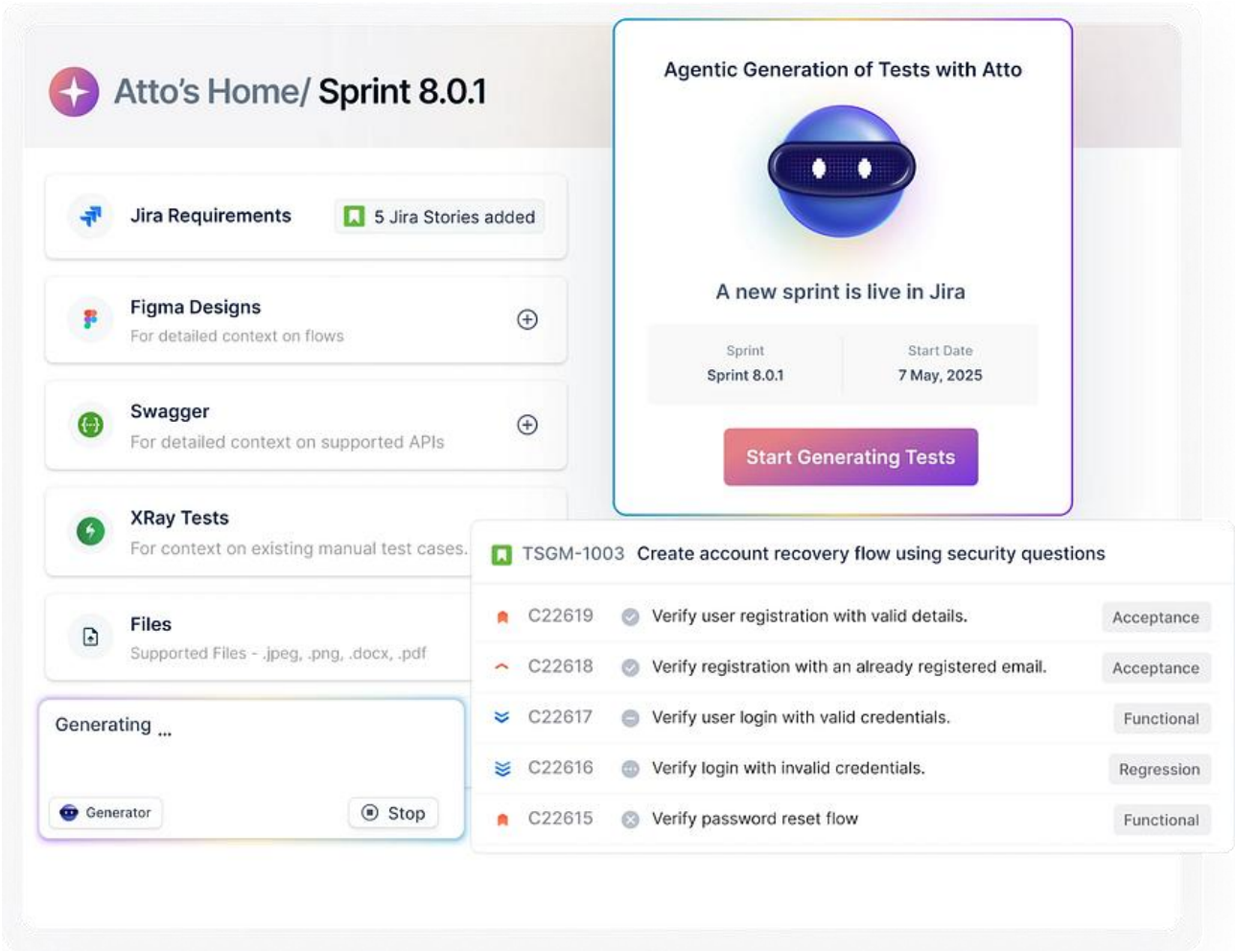

Рисунок 1.1 – Інтерфейс користувача Testsigma

Плюси.

– Автоматизація на базі штучного інтелекту без необхідності кодування пришвидшує створення тестів.

– Підтримує кросплатформне тестування (веб, мобільні пристрої, настільні комп'ютери, API).

– Вбудовані функції управління тестуванням.

Мінуси.

– Розширене налаштування може бути обмежене для команд, які багато працюють з кодом.

2. Postman.

Postman – одна з найпопулярніших платформ для тестування та співпраці API, якій довіряють розробники та команди контролю якості по всьому світу (див. рис. 1.2). Вона пропонує надійне середовище для створення, налагодження, документування та автоматизації запитів API. Завдяки своїй величезній екосистемі, сильній спільноті та функціям співпраці, Postman ідеально підходить для команд, які працюють над усім життєвим циклом API.

Рисунок 1.2 – Логотип засобу тестування Postman

Особливості Postman.

– Надійний конструктор запитів для REST, SOAP та GraphQL.

– Підтримує керування середовищем та змінними.

– Автоматизація за допомогою Collection Runner та Newman CLI.

– Створення макетних серверів для імітації API під час розробки.

– Планування та запуск тести через певні проміжки часу, щоб контролювати час безвідмовної роботи та продуктивність.

– Автоматична генерація та публікація інтерактивної документації API.

– Керування версіями API та відстеження зміни в різних командах.

Плюси.

– Масивна екосистема та активна спільнота.

– Чудово підходить для співпраці та документування.

– Включено імітацію та моніторинг API.

– Мінуси.

– Значно споживає системні ресурси.

– Багато розширених функцій вимагають платного плану.

3. SoapUI.

SoapUI – це давній інструмент для тестування SOAP та REST API, який широко використовується в корпоративних умовах (див. рис. 1.3). Він надає розширені функції для функціонального, безпекового та навантажувального тестування, що робить його особливо потужним для складних або застарілих систем. Завдяки безкоштовній версії з відкритим кодом та комерційній версії (ReadyAPI), SoapUI пропонує гнучкість для різних організаційних потреб.

Рисунок 1.3 – Логотип інструменту тестування SoapUI

Особливості SoapUI.

– Потужна підтримка SOAP та REST сервісів.

– Функціональне, безпекове та навантажувальне тестування в одному інструменті.

– Тестування на основі даних з використанням CSV, Excel та вхідних даних бази даних.

– Підтримка WSDL для легкого імпорту та тестування SOAP-сервісів.

– Розширене тестування безпеки на наявність вразливостей, таких як SQL-ін'єкції, XML-бомби та фаззинг (fuzzing).

– Бібліотека для перевірки кодів відповідей, заголовків, схеми та вмісту.

– Моделювання сервісів для тестування API до того, як бекенд-сервіси будуть готові.

– Комплексна звітність з детальними журналами та показниками виконання тестів.

Плюси.

- Тестування API корпоративного рівня.
- Найкращий у своєму класі для протоколу SOAP.
- Підтримується інтеграція CI/CD.

Мінуси.

- Застарілий інтерфейс та крута крива навчання.
- Безкоштовна версія обмежена порівняно з ReadyAPI.

4. REST Assured.

REST Assured – це бібліотека на базі Java, розроблена для розробників, які надають перевагу написанню автоматизованих API-тестів у коді (див. рис. 1.4). Вона використовує вільну мову Java DSL (Domain Specific Language) для спрощення створення експресивних тестів для REST-сервісів. Завдяки високій сумісності з такими фреймворками, як JUnit та TestNG, REST Assured є ідеальним вибором для команд розробників, що зосереджені на автоматизації.

Рисунок 1.4 – Логотип інструменту тестування REST Assured

Особливості REST Assured.

- Бібліотека на базі Java з вільним DSL для написання експресивних REST API-тестів.
- Підтримує перевірку схеми для відповідей JSON та XML.
- Безшовна інтеграція з JUnit та TestNG для виконання тестів.
- Підтримує тестування в стилі BDD з синтаксисом, подібним до Gherkin, для зручності читання.
- Вбудовані методи для обробки запитів, відповідей, заголовків, параметрів та файлів cookie.

– Підтримка OAuth, OAuth 2.0 та базової автентифікації одразу після встановлення.

– Підтримка XMLPath та JSONPath для легкого розбору відповідей.

Плюси.

– Чудово підходить для середовищ, орієнтованих на розробників.

– Потужний для автоматизованого регресійного тестування.

– Плавна інтеграція конвеєра CI/CD.

Мінуси.

– Потрібні знання кодування на Java.

– Немає графічного інтерфейсу для нетехнічних тестувальників.

5. Insomnia.

Insomnia – це легкий та зручний для розробників API-клієнт, відомий своїм простим інтерфейсом та швидкістю. Він підтримує протоколи REST, GraphQL, gRPC та WebSocket, що робить його універсальним для сучасних архітектур API. Завдяки потужним опціям автентифікації, управлінню середовищем та функціям співпраці, Insomnia є улюбленим серед команд, які шукають швидкий та простий досвід тестування API (див. рис. 1.5).

Рисунок 1.5 – Логотип інструменту тестування Insomnia

Особливості Insomnia.

– Підтримує протоколи REST, GraphQL, gRPC та WebSocket.

– Кілька варіантів автентифікації, таких як OAuth 2.0, JWT, ключі API, базова автентифікація та дайджест-автентифікація.

– Керування середовищем та змінними для гнучких конфігурацій тестування.

– Зрозумілий, інтуїтивно зрозумілий інтерфейс, який робить створення та тестування запитів швидким та простим.

– Генерація коду для кількох мов (cURL, JavaScript, Python тощо).

– Перевірка відповідей за допомогою JSONPath та візуального попереднього перегляду.

– Робочі простори та функції співпраці для обміну API та запитами з командами.

– Підтримка розробки та специфікацій API для імпорту схем OpenAPI, Swagger та GraphQL.

– Плагіни та розширення для налаштування робочих процесів і доповнень спільноти.

Плюси.

– Легкий, швидкий та зручний для розробників.

– Підтримує кілька сучасних протоколів.

– Проста співпраця з колекціями.

Мінуси.

– Обмежені функції автоматизації.

– Немає розширеного тестування навантаження/продуктивності.

Фактори, які слід враховувати під час вибору найкращого інструменту для тестування API.

Вибір правильного інструменту для тестування API – це не просто встановлення прапорців у відповідних функціях, а й забезпечення відповідності інструменту робочим процесам вашої команди, потребам масштабованості та довгостроковим цілям. Ключові фактори, які слід оцінити, включають:

– Підтримка протоколів – слід впевнитись, що інструмент підтримує протоколи, з якими працює команда на даному проєкті, такі як REST, SOAP, GraphQL, gRPC або черги повідомлень.

– Простота використання – баланс між інструментами без коду для швидкого впровадження та фреймворками, орієнтованими на код, для гнучкості та глибини.

– Інтеграція CI/CD – потрібна для безперебійної сумісності з конвеєрами DevOps та серверами автоматизації, такими як Jenkins, GitLab або GitHub Actions.

– Масштабованість – інструмент повинен обробляти зростаючі обсяги тестування, паралельне виконання та середовища корпоративного масштабу.

– Керування даними та середовищем – підтримка змінних, параметризації та кількох середовищ спрощує складні сценарії тестування.

– Співпраця та звітність – такі функції, як спільні робочі простори, панелі інструментів та детальні звіти, підвищують продуктивність команди та прозорість.

– Вартість та ліцензування – слід розглядати інструменти з відкритим кодом та комерційні інструменти, а також враховувати ліцензування з урахуванням довгострокової рентабельності інвестицій.

Переваги використання інструментів тестування API.

Інструменти тестування API не лише економлять час, вони допомагають командам створювати високоякісне та надійніше програмне забезпечення, роблячи перевірку API швидшою, узгодженішою та легшою для масштабування.

Переваги.

– Раннє виявлення помилок – виявлення дефектів на етапі інтеграції, перш ніж вони вплинуть на кінцевих користувачів.

– Підвищена надійність і безпека – забезпечення відповідності API вимогам продуктивності, безпеки та масштабованості.

– Ефективність автоматизації – мінімізація повторюваної ручної роботи за допомогою автоматизованих наборів тестів.

– Покращена співпраця – спільні середовища, звіти та тестові ресурси покращують узгодженість між розробниками, командами QA та DevOps.

– Краще тестове покриття – перевірка широкого спектру сценаріїв, таких як функціональність, навантаження, безпека та інтеграція, з меншими зусиллями.

Недоліки.

– Крива навчання – деякі просунуті інструменти вимагають технічних знань та навчання.

– Фактор вартості – рішення корпоративного рівня можуть бути дорогими для малих або середніх команд.

– Накладні витрати на налаштування та обслуговування – початкове налаштування та інтеграція з конвеєрами можуть потребувати часу та ресурсів.

Тестування API є важливим для забезпечення безперебійної, безпечної та масштабованої роботи сучасних додатків. Незалежно від того, чи надається перевага платформі без кодування, такій як Testsigma, чи Postman, чи інструменту, орієнтованому на розробників, такому як REST Assured, правильний вибір залежить від навичок, робочих процесів та бюджету команди. Оцінюючи підтримку протоколів, функції автоматизації та можливості інтеграції, можна вибрати найкращий інструмент для тестування API та швидше створювати високоякісне програмне забезпечення.

1.2 Аналіз предметної області

Виконаний вище огляд інструментів тестування виявив, що не всі вони надають можливість тестувати RESTful API, а також мають різний рівень підтримки тестування програмних інтерфейсів інших типів. Розглянемо детальніше, що представляють собою інтерфейси RESTful.

RESTful API, широко відомі як веб-API, складаються з кінцевих точок. Кожна кінцева точка – це конкретна реалізована функціональність бізнес-процесу. Ці API, як правило, доступні через HTTP (протокол передачі гіпертексту) шляхом включення визначених стандартних дієслів, таких як GET, POST, PUT та DELETE. RESTful API викликаються за допомогою адреси, відомої як URI (уніфікований ідентифікатор ресурсу).

Одним із завдань при проєктуванні цього інтерфейсу було визначення стандартного формату обміну повідомленнями (тобто запиту та відповіді).

Спочатку для опису REST API використовувався неформальний текст [9]. Пізніше як стандарт розвинулися документи JSON (JavaScript Object Notation). Формат є чистим текстом, його легко ідентифікувати та обробляти машинами в різних мережах та на різних платформах. Однак, питання стандартизованого способу опису REST-сервісів все ще залишається відкритим. Специфікація OpenAPI [10] стає одним із рішень цієї проблеми.

Як було сказано у вступі, метою цього дослідження та огляд наукових публікацій та досліджень, пов'язаних з тестуванням RESTful API.

Для виконання більш комплексного аналізу ставилось за мету також розглянути проблеми, пов'язані з тестуванням, особливо вплив на покриття коду. Крім того, ця робота має на меті зарахувати та обговорити доступні рішення для тестування RESTful API та їхню здатність тестувати API на основі автентифікації.

Для відповіді на дослідницькі питання, визначені в вище, було використано добре відомий протокол SLR, тобто Since Literature Review, і було проведено пошук у різних базах даних для відбору відповідних статей на нашу тематику. Були визначені критерії відбору досліджень для включення та виключення знайдених статей, а також проведена оцінка якості відібраних статей. Статті, які не відповідали критеріям якості, були виключені з аналізу, а вибірка була проведена на основі високоякісних статей.

HTTP має власні методи, також відомі як команди, що сприяють зв'язку клієнт-сервер. Спочатку в HTTP/1.0 були визначені лише команди GET, HEAD та POST. Однак, у HTTP/1.1 дієслова були розширені, включивши PUT, DELETE, CONNECT, OPTIONS та TRACE, а у 2010 році, в RFC 5789 було запропоновано PATCH [11].

Репрезентативну передачу стану (REST) запропоновано у роботі [12]. Метою цього стилю було покращення продуктивності, масштабованості, простоти, можливості модифікації, видимості комунікації, портативності та надійності. Для досягнення таких властивостей автор визначив шість елементів керування наступним чином: артефакти архітектури, пов'язані з клієнтом і

сервером, відсутність станів, можливість кешування даних, багаторівнева система, стандартний інтерфейс та, за бажанням, код на вимогу. Таким чином, з HTTP-команд для RESTful API використовуються лише такі: DELETE, GET, POST та PUT [13].

Щоразу, коли викликається кінцева точка REST API, це можна зробити за допомогою однієї з описаних вище команд. Наприклад, якщо є кінцева точка, яка повертає список продуктів з веб-сайту онлайн-магазину, її можна викликати за допомогою команди GET, що означає, що є деякі дані, які ми хочемо отримати у вигляді списку екземплярів або окремого екземпляра. Запит називається запитом GET і може мати деякі параметри.

Аналогічно, дані можна вставляти за допомогою запиту POST, видаляти за допомогою запиту DELETE та редагувати за допомогою запиту PUT. Доступ до будь-якої кінцевої точки REST API передбачає підготовку URL-адреси з адресою, параметрами, якщо такі є, та відповідне позначення її однією із команд. Команда POST використовується для надсилання даних із запитом як частини тіла запиту, а не для вбудовування їх у URL-адресу. Це робить її невидимим, що краще для конфіденційних даних, таких як облікові дані. Наприклад, коли користувач входить у систему, ім'я користувача та пароль надсилаються до кінцевої точки REST API як запит POST.

1.3 Опис методології дослідження

Оскільки тема роботи є важливою та актуальною для сучасних методологій розробки програмного забезпечення, будемо вважаємо, що подібні дослідження будуть проведені найближчим часом в більшій кількості, і поточне дослідження буде розширено за рахунок опублікованих статей. Ця стаття корисна не лише для дослідників, але й для практиків у галузі програмного забезпечення.

Під час нашого пошуку ми натрапили на дослідження, яке зосереджено на методах тестування веб-сервісів [14]. В ній автори виконали огляд декількох

статей як на тематику SOAP для сервіс-орієнтованих систем, так і на тему RESTful.

У роботі [15] автори представили проблеми тестування веб-сервісів у SOA-реалізаціях та зосередилися головним чином на аспектах безпеки. Вони перерахували такі поширені загрози веб-сервісів: підміна повідомлень, конфіденційність, фальсифіковані повідомлення, людина посередині, підміна принципала, підроблені заяви, повторне відтворення частин повідомлення, повторне відтворення та відмова в обслуговуванні. Вони також перерахували такі вимоги до наскрізної безпеки веб-сервісів: взаємна автентифікація, авторизація доступу до ресурсів, цілісність та конфіденційність даних, наскрізна цілісність та конфіденційність повідомлень, цілісність транзакцій та комунікацій, аудиторські записи та механізми, а також розподілене забезпечення політики безпеки. Однак їхня увага не була зосереджена на тестуванні RESTful веб-сервісів.

На рисунку 1.6 показано основні етапи методології дослідження.

У нашому дослідженні ми використовували такі бази даних:

- IEEE Xplore.
- Science Direct.
- Scopus.
- Web of Science.
- Springer Link.
- Wily.
- Google Scholar.

Унікальність поточного дослідження полягає в тому, що це актуальна тема. Тому досі було проведено небагато досліджень. Бази даних, що використовувалися, видали аналогічні дослідницькі статті. Ми включили до нашого огляду лише релевантні темі роботи дослідження.

Пошук відповідних досліджень був зосереджений на методологіях тестування та генерації модульних тестів для RESTful API. Опубліковано кілька

досліджень, пов'язаних з RESTful API загалом. Ці статті описують інші аспекти, такі як ефективна розробка RESTful API, інтеграція між програмами та RESTful API тощо. Однак, обмежена кількість досліджень була опублікована щодо тестової частини RESTful API, оскільки це одна з найактуальніших тем на сьогодні. Пошук проводився з використанням ключових слів ((“RESTful APIs Testing” OR “RESTful APIs Unit Testing”) AND “RESTful APIs” AND (“Web API Testing”) AND (“RESTful Web Services” OR “REST APIs Blackbox Testing”)).

У результаті пошуку було знайдено 16 статей: 7 в IEEE Xplore, 6 в Scopus та 3 в Google Scholar. Як бачимо більшість статей було отримано з баз даних IEEE та Scopus. Оскільки Scopus – це метадана база даних, яка індексує кілька баз даних, було охоплено різні статті від кількох видавництв.

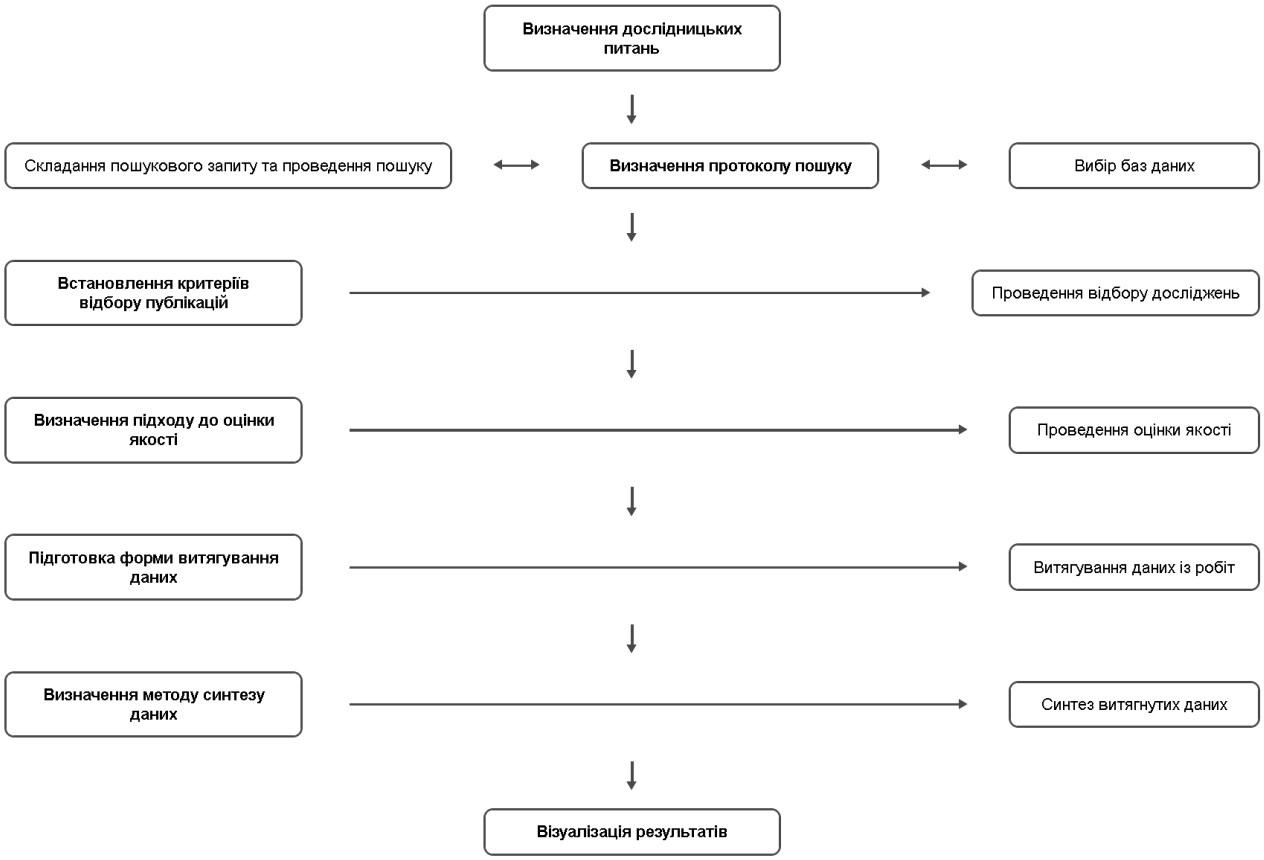

Рисунок 1.6 – Схема виконання огляду літератури за темою роботи

Критерії виключення були застосовані до робіт, які не відповідали тематиці тестування RESTful API, але за ключовими словами відповідали критерієві пошуку.

Ці критерії представлені наступним чином:

1. Публікації, що не пов'язані безпосередньо з тестуванням RESTful API.
2. Дублікати публікацій.
3. Публікації, що містять лише анотацію.
4. Вторинні дослідження.

Синтез даних виконує агрегацію зібраних даних для ефективного пошуку відповідей на питання дослідження, сформульовані у вступі. Деякі питання в цій роботі вимагають від нас отримання категоріальних результатів шляхом кількісного аналізу даних, таких як доступні рішення. Однак деякі питання, такі як основні задачі, що виникають під час тестування RESTful API, вимагають використання якісного синтезу даних. Для визначення основних задач були потрібні текстові описи та аналіз контенту. Також біли розглянути методології, інструменти та фреймворки, що використовуються для вирішення задач, що виникають.

2 ПІДХОДИ ДО РОЗРОБКИ ТЕСТІВ ДЛЯ RESTFULL API

2.1 Розробка модульних тестів RESTful API

Перегляд усіх первинних досліджень та аналізів показав, що існує кілька проблем у створенні модульних тестів для RESTful API.

Безпека відіграє життєво важливу роль у забезпеченні захисту та стабільності API. Однак, водночас, це створює великі труднощі у генерації модульних тестів та тестуванні загалом. Звичайний підхід до генерації модульних тестів залежить від опису API, який типово виконується за допомогою OpenAPI або будь-якої іншої сумісної описової мови. Ця методологія використовується для тестування "чорної скриньки", оскільки доступ до вихідного коду недоступний. Описовий документ не містить жодної інформації про безпеку. Через такий сценарій процес генерації модульних тестів або обмежується меншою кількістю модульних тестів, або призводить до недійсних модульних тестів, оскільки необхідна інформація про безпеку не була включена до процесу генерації.

Недотримання стандартів описових документів є ще однією проблемою у генерації модульних тестів. Найважливішим каталізатором для автоматичної генерації модульних тестів для RESTful API є описовий документ. Різні організації підтримують цей документ по-різному. XML, звичайний JSON та OpenAPI – це деякі зі стандартів документів. RESTful API є досить гнучкими в цьому відношенні, що є ще однією причиною виникнення цієї проблеми. Оскільки немає жодної обов'язкової підтримки стандартного описового документа для роботи RESTful API, цьому аспекту не надається великої уваги. В результаті стандартні інструменти та фреймворки іноді не можуть навіть продовжити роботу або закінчуються помилками модульних тестів.

Невідповідний описовий документ також додає до переліку проблем. Дослідження показали, що організації часто не оновлюють описовий документ для RESTful API. Як результат, API, що знаходяться у експлуатації,

відрізняються від описового документа. Швидкі зміни та тиск на те, щоб модифіковані API були доступні якомога раніше у робочому продукті, призводять до невідповідності описового документа. Отже, генерація модульних тестів призводить до неповних або невідповідних тестів.

Складні типи вхідних даних – це ще одна проблема, з якою доводиться мати справу. Сучасні RESTful API дозволяють взаємодіяти між гібридними мережами, серверними частинами та всіма можливими типами пристроїв. Дані, що передаються, можуть бути будь-якого часу та варіюватися від простих цілих чисел та рядків до цілого файлу або групи файлів. Однією з ключових частин будь-якої системи генерації модульних тестів є виведення вхідних значень тесту, що дуже складно, коли задіяні складні типи.

2.2 Покриття коду тестами

Покриття коду є однією з головних цілей модульного тестування. Це важливий показник для вимірювання ефективності модульних тестів. Завжди бажано мати максимальну кількість коду, що охоплена модульними тестами. Для досягнення високого покриття коду має бути достатня кількість модульних тестів із ширшим масштабом, щоб охопити всі можливі частини коду.

Недосягнення рекомендованого рівня покриття коду є ознакою або меншої кількості модульних тестів, або обмеженого масштабу модульних тестів з точки зору виконання коду. У випадку RESTful API досягнення високого покриття коду може бути складним завданням. Однією з основних причин є звернення до захищених API. Як пояснюється в попередньому пункті, через природу опису документів RESTful API, стає дуже важко генерувати модульні тести для захищеної кінцевої точки.

Тому більша частина RESTful API обходиться без автоматичної генерації модульних тестів. Для цього потрібне ручне тестування та генерація модульних тестів, що завжди створює ймовірність пропустити деякий код, що призводить до нижчого покриття коду.

2.3 Пропозиції щодо генерації модульних тестів

У вибраних первинних дослідженнях було запропоновано та використано різні категоріальні рішення. Вони належать до таких категорій:

- Інструменти.
- Методика/ метод.
- Структура/модель.

На рисунку 2.1 показано класифікацію пропонованих в літературних джерелах шляхів для побудови модульних тестів. Згідно з рисунком 1.6 більшість робіт пропонують підхід на рівні методології, а найменша кількість пропозицій пов'язана з пропозицією фреймворку та моделі програмного продукту.

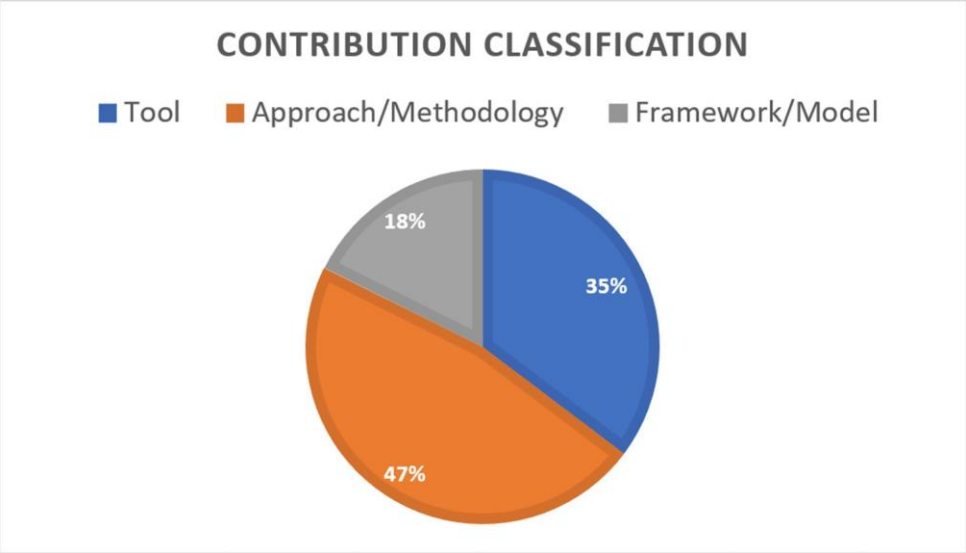

Рисунок 2.1 – Класифікація рішень щодо шляхів вирішення проблеми генерації модульних тестів для RESTful API

Однак розробка фреймворків/моделей є такою ж важливою, як і підходів та методології, і для розробки нових моделей/фреймворків потрібні додаткові дослідження.

2.4 Проблема автентифікації при модульному тестуванні RESTful API

Розробка автентифікації для RESTful API передбачає кілька методологій, яких слід дотримуватися.

Існують програмні інтерфейси RESTful API, відомі, як захищені. До них анонімний користувач не може отримати доступ. Спочатку необхідно успішно завершити процес входу, щоб отримати ідентифікацію, яку потім можна використовувати для доступу до захищеної частини колекції кінцевих точок RESTful API. Різниця досить суттєва, коли справа доходить до фактичної реалізації. Виходячи з аналізу первинних досліджень, можна зробити висновок, що надзвичайно важко знайти рішення, яке б враховувало цей аспект. Кілька досліджень виключали генерацію модульних тестів для захищених (тобто з увімкненою автентифікацією) кінцевих точок RESTful API. Деякі поширені методи реалізації автентифікації для RESTful API описані нижче.

Аутентифікація на основі токенів. Після успішної автентифікації (тобто входу за допомогою імені користувача та пароля) сеансу користувача призначається довгий ідентифікаційний рядок, який називається токеном. Зазвичай він шифрується та надсилається туди-сюди вручну між сеансами запиту-відповіді. Механізми створення та перевірки токенів повністю залежать від розробника або організації; тому це можна зробити багатьма різними способами, і тому не існує суворого стандарту. Це створює величезну проблему під час спроби інтегрувати захищені RESTful API у генерацію тестів. Кожен токен зазвичай має обмеження часу дії, після якого він закінчується, і потрібен новий токен.

Аутентифікація на основі файлів cookie. Цей механізм залежить від створення фрагмента інформації для автентифікації, який називається файлом cookie, і який надсилається туди-сюди між сеансами запиту-відповіді, подібно до підходу на основі токенів. Однак створення файлу cookie залежить від фреймворку, що використовується для розробки RESTful API. Файл cookie

створюється після успішного входу за допомогою облікових даних і має певний термін дії . Існує багато варіантів вибору фреймворку.

Автентифікація на основі ключів API. Автентифікація на основі ключів API не вимагає використання облікових даних (тобто імені користувача та пароля) для автентифікації. Натомість нам потрібно використовувати вже виданий секретний ключ API від хоста RESTful API, щоб мати змогу користуватися його послугами. Ключ видається або безкоштовно, або за допомогою платної підписки, наприклад, за допомогою картографічних сервісів або сервісів соціальних мереж. Оскільки це секретний ключ, який зазвичай не розголошується, неможливо створити модульні тести для таких сервісів та належним чином їх протестувати.

Під час нашого аналізу ми не знайшли жодного дослідження, яке пропонує рішення, що враховує захищені RESTful API під час генерації модульних тестів, а це означає, що значну частину RESTful API доведеться тестувати вручну за допомогою згенерованих вручну модульних тестів. Це також допомагає зробити висновок, що розробка платформи тестування з підтримкою автентифікації для RESTful API є галуззю з величезним обсягом досліджень.

3 ПІДСУИОК ОГЛЯДУ ЛІТЕРАТУРИ

3.1 Основні проблеми тестування RESTful API

Це дослідження призвело до надання опису труднощів, проблем, а також доступних рішень і підходів, пов'язаних з областю тестування RESTful API. Представлені результати мають на меті класифікувати та категоризувати доступні рішення. Це визначило шлях для подальшої роботи над цими підходами та впровадження подальших інструментів на основі представлених моделей та фреймворків.

Одним з ключових аспектів цього огляду є пошук рішень для автентифікації під час тестування RESTful API. Це пояснюється тим, що більшість кінцевих точок RESTful API захищені, і якщо вони не охоплені автоматичною генерацією модульних тестів, це може вплинути на покриття коду. Це призведе до недостатнього та ненадійного тестування та покриття коду. На разі не знайдено жодного повного рішення, яке б підпадало під категорію тестування на основі автоматично згенерованих модульних тестів, що охоплює автентифікацію, а це означає, що існує велика залежність від ручного тестування, яке завжди має ймовірність низького покриття коду.

Під час аналізу літературних джерел ми виявили, що вибрані дослідження використовують спільний підхід до вивчення документа опису REST API для генерації модульних тестів. Формат Open-API (або Swagger) є одним з найпоширеніших форматів для опису REST API. Через це, а також через те, що автентифіковані кінцеві точки REST API не охоплюються жодним із підходів, вважається, що дослідження мають однаковий рівень продуктивності для покриття коду. З іншого боку, виявлення помилок найкраще досягається за допомогою RESTTESTGEN – fвтоматизоване тестування RESTful API методом чорної скриньки.

У дослідженні було протестовано деякі з відомих загальнодоступних REST API. Наведена статистика чітко підтверджує висновки.

Існує багато ситуацій, коли REST API розміщуються локально та в хмарі, але вони недоступні публічно, оскільки використовуються для внутрішніх систем. З іншого боку, кілька екземплярів REST API є загальнодоступними з точки зору використання. В обох випадках доступ до вихідного коду зазвичай недоступний.

Це одна з поширених причин, чому тестування "чорної скриньки" стає життєздатним варіантом. Перевага тестування "чорної скриньки" полягає в тому, що запропоноване рішення може легко генерувати та запускати модульні тести, просто аналізуючи опис REST API. Недоліком є те, що точне місцезнаходження або причину помилки буде нелегко знайти, особливо коли реалізація приховує фактичну причину помилки у виводі тестованої функції.

Тестування "білої скриньки" має явну перевагу з точки зору пошуку фактичної причини та місця розташування помилки або багу. Однак доступ до вихідного коду є обов'язковим для такого виду тестування. Аспект покриття коду залишається однаковим в обох методологіях тестування. Однією з головних перешкод для досягнення високого рівня покриття коду є тестування автентифікованих REST API. Запропоноване рішення може досягти високого рівня покриття коду, якщо воно здатне викликати всі або максимум кінцеві точки REST API, використовуючи тестування методом чорного або білого ящика.

Існують також додаткові методи, які можуть покращити генерацію тестових випадків на основі пошуку для тестування RESTful веб-сервісів та спрямовані на повну автоматизацію оцінки підходів до тестування. В літературі запропоновано каталог із 10 критеріїв покриття тестами для RESTful API та структуру для оцінки підходу до тестування. Вони продемонстрували, що рівні покриття корелюють з вимірюваннями виявлення помилок.

3.2 Напрямки подальших досліджень

REST API дуже різноманітні, як з точки зору розробки, так і тестування. Різні організації застосовують на практиці різні методології, фреймворки та

підходи. Через цю різноманітність та гнучкість організаціям не потрібно дотримуватися певної фреймворку чи методології. Це призвело до проблем, пов'язаних саме з тестуванням.

Однією з основних проблем, яку ми виявили, була підтримка автентифікації під час створення модульних тестів для REST API. Оскільки сьогодні використовується багато різних методів автентифікації, це ускладнює автоматизацію створення модульних тестів. Більше того, організації зазвичай не мають спільного способу реалізації автентифікації, і вона постійно розвивається; тому стало ще важче розробити загальну платформу для створення модульних тестів, яка охоплює автентифіковані REST API.

У деяких недавніх статтях це запропоновано для майбутніх досліджень. Оскільки кінцеві точки автентифікованих або захищених REST API не включаються до генерації тестових випадків, досягнуте покриття коду є низьким – це друга за важливістю проблема. Це визначило напрямок майбутніх досліджень, зосереджений на вивченні можливого спільного та базового стандарту, якого слід дотримуватися під час реалізації автентифікації. Виходячи з цього дослідження, бажано, щоб підтримка автентифікації для тестування REST API була розроблена як підключаємий модуль у запропонованому рішенні.

3.3 Приклад тестування RESTful API

Наведена схема ресурсів для тестування функцій RESTful API забезпечує надзвичайну гнучкість, дозволяючи створювати власні об'єкти з різними атрибутами різних типів. Ці атрибути зберігаються як частина поля "даних", утворюючи налаштовуваний JSON-об'єкт. Ця унікальна функція дозволяє імітувати широкий спектр реальних сценаріїв застосування, від зберігання та отримання цін, дат та URL-адрес зображень до простих текстових полів тощо.

Щоб отримати доступ до даних, не завжди потрібно створювати їх з нуля. REST API дозволяє вам легко отримувати зарезервовані макетні дані за допомогою простого GET-запиту. Крім того, є можливість отримувати власні

дані, які користувач створив самостійно. Можна отримати один об'єкт або запросити кілька об'єктів разом, надаючи їхні відповідні ідентифікатори об'єктів як частину одного GET-запиту. Таким чином, наявний повний контроль та легкий доступ до даних, необхідних для програми.

RESTful API формують основу сучасних веб-додатків, забезпечуючи безперебійний зв'язок та обмін даними між різними системами. Щоб забезпечити надійну, стабільну та безпечну роботу цих API за різних умов, автоматизоване тестування є незамінним. Представляємо PyTest, потужний та гнучкий фреймворк для тестування Python, який спрощує написання та виконання тестів.

У цьому прикладі будуть розглянуті основи використання PyTest для оптимізації процесу тестування RESTful API. Буде описано, як перевіряти відповіді API, тестувати різні методи HTTP та обробляти потенційні сценарії помилок.

Щоби розпочати тестування RESTful API за допомогою PyTest, ось що знадобиться:

Мова Python.

Знайомство з REST.

PyTest та запити. Слід переконатися, що ці бібліотеки встановлені у віртуальному середовищі проєкту. Їх інсталяція показана на рисунку 3.1.

```
pip install pytest requests
```

Рисунок 3.1 – Інсталяція pytest

Перейдемо до опису установки та налаштування програмного середовища для тестування.

В першу чергу потрібно створити папку для проєкту. А в цій папці створимо файл з іменем test_rest_api.py.

Оскільки не рекомендується використовувати напряму функції REST під час розробки, то осиь часто використовують сервіс, що дозволяє створити копії цих функцій та тестувати саме ці копії. Цими сервісами є, наприклад, swager чи mocky.io.

Перш ніж ми заглибимося в написання тестів, давайте закріпимо кілька базових концепцій, необхідних для ефективного тестування RESTful API.

Методи HTTP: RESTful API використовують стандартні методи HTTP для позначення дій з ресурсами. Ось розбивка найпоширеніших:

- GET: Отримує дані з сервера (наприклад, отримує список книг, вибирає одну книгу за ідентифікатором).
- POST: Створює новий ресурс на сервері (наприклад, додає нову книгу до каталогу).
- PUT: Оновлює існуючий ресурс на сервері (наприклад, редагує інформацію про книгу).
- DELETE: Видаляє ресурс із сервера (наприклад, видаляє книгу).

Сервер повертає коди стану HTTP, щоб передати результат запиту API. Розуміння цих кодів допоможе вам успішно провести тестування:

- 2xx Успіх: Вказує на успішні дії (наприклад, 200 OK для успішного GET, 201 Created після успішного POST).
- 4xx Помилка клієнта: Сигналізує про проблему, ймовірно пов'язану з самим запитом (наприклад, 400 Bad Request для недійсних даних, 404 Not Found, якщо ресурс не існує).
- 5xx Помилка сервера: Вказує на проблеми на стороні сервера (наприклад, 500 Internal Server Error).

Важливим аспектом тестування API є забезпечення валідності відповідей від сервера та їх відповідності очікуванням. Це включає:

- Структуру: перевірка відповідності відповіді очікуваному формату, зазвичай JSON для RESTful API.
- Тип вмісту: перевірка відповідності заголовка Content-Type формату відповіді (наприклад, application/json).

– Цілісність даних: перевірка відповідності даних у відповіді вашим очікуванням, як за значеннями, так і за типами.

Ось короткий приклад, що демонструє, як перевірити коди стану та структуру відповіді в тесті PyTest (див. рис. 3.2).

```
def test_get_book_by_id():
    url = "https://run.mocky.io/v3/9b2fc100-4c56-473d-b488-323dfd26396c/books/1"  # Replace with
    your mock API URL
    response = requests.get(url)

    # Verify status code
    assert response.status_code == 200

    # Verify content-type
    assert response.headers["Content-Type"] == "application/json; charset=UTF-8"

    # Verify response structure (Assuming a book object with 'id', 'title', and 'author')
    data = response.json()
    assert isinstance(data, dict)
    assert "id" in data
    assert "title" in data
    assert "author" in data
```

Рисунок 3.2 – Означення функції REST для тестування

Коротко пояснимо, що зображено на рисунку.

Ми використовуємо бібліотеку requests для надсилання GET-запиту до фіктивної кінцевої точки API.

Ми стверджуємо, що код стану відповіді – 200 OK, що вказує на успіх.

Ми перевіряємо, чи для заголовка Content-Type встановлено значення application/json.

Нарешті, ми конвертуємо відповідь у JSON та стверджуємо наявність очікуваних ключів (id, title, author) для перевірки структури.

Звичайно замість фіктивної URL-адреси API має бути фактична, створена у вже згаданому mocky.io.

Тепер розширимо набір тестів, щоб охопити весь спектр операцій RESTful API.

Перевіримо можливість створення нових ресурсів на сервері. Ось приклад, що стосується додавання нової книги (див. рис. 3.3).

```python
def test_create_book():
    url = "https://run.mocky.io/v3/628dca34-286a-4850-902b-b5fdd89e0ce3/books"
    # Replace with your mock API URL
    new_book = {"title": "The Hobbit", "author": "J.R.R. Tolkien"}
    response = requests.post(url, json=new_book)

    assert response.status_code == 201
    assert response.headers["Content-Type"] == "application/json; charset=UTF-8"

    # Check if the book was created (specifics depend on your mock API's response)
    data = response.json()
    assert "id" in data
    assert data["title"] == "The Hobbit"
    assert data["author"] == "J.R.R. Tolkien"
```

Рисунок 3.3 – Тестування методу POST

На рисунку 3.4 показано тестування методу PUT для редагування існуючих даних.

```python
def test_update_book():
    url = "https://run.mocky.io/v3/..."  # Replace with mock API URL for updating a book
    book_id = 1  # Existing book ID to update
    updated_book = {"title": "The Lord of the Rings: The Fellowship of the Ring", "author":
    "J.R.R. Tolkien"}
    response = requests.put(url, json=updated_book)

    assert response.status_code == 200  # Or the appropriate code for your mock API
    assert response.headers["Content-Type"] == "application/json; charset=UTF-8"

    # Verify update was successful (depends on mock API's response)
    data = response.json()
    assert data["id"] == book_id
    assert data["title"] == updated_book["title"]
    assert data["author"] == updated_book["author"]

```

Рисунок 3.4 – Редагування даних методом PUT

Під час виконання всіх методів має проводитись обробка помилок. Варто включати тести, які навмисно запускають сценарії помилок (наприклад, недійсні вхідні дані, відсутні ресурси) та переконатися, що API відповідає відповідними кодами помилок та повідомленнями.

Параметризація: варто використовати функцію параметризації PyTest для ефективного тестування кількох варіантів вхідних даних в одній тестовій функції. Декоратор parametrize у PyTest дозволяє запускати одну тестову функцію з кількома сценаріями входу та виходу, оптимізуючи тести та зменшуючи кількість повторень. Приклад його використання показано на рисунку 3.5.

```python
@pytest.mark.parametrize("book_id, status_code", [
    (1, 200),    # Valid book ID
    (999, 404),  # Non-existent book ID
])
def test_get_book_various_ids(book_id, status_code):
    url = f"https://run.mocky.io/v3/9b2fc100-4c56-473d-b488-323dfd26396c/books/{book_id}"
    response = requests.get(url)
    assert response.status_code == status_code
```

Рисунок 3.5 – Параметризація тестів

Опанувавши RESTful API-тестування за допомогою PyTest, можна отримали потужний набір інструментів для захисту якості та надійності веб-сервісів. У міру розвитку та ускладнення API, методи та найкращі практики, які ми дослідили, допоможуть забезпечити їхню постійну відповідність очікуванням. Безперервне тестування є невід’ємною частиною створення надійних та зручних у підтримці додатків.

4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1 Питання щодо охорони праці і галузі інформаційних технологій

Інформаційні технології – це галузь, яка швидко розвивається і змінюється, а також одна з найбільших за масштабами впливу на сучасне суспільство. Проте, незважаючи на те, що більшість процесів у цій галузі відбуваються віртуально, питання охорони праці та безпеки співробітників залишаються важливими. Оскільки розробка програмного забезпечення, зокрема чат-систем у реальному часі, вимагає високої концентрації уваги, роботи з комп'ютерними технологіями та участі в командній роботі, охорона праці в IT-сфері має багато аспектів.

Незважаючи на відсутність фізичних ризиків, типових для деяких інших галузей, робота в IT-секторі не є безпечною для здоров’я працівників. Основні фізичні проблеми, з якими стикаються розробники програмного забезпечення, включають: Охорона зору, робота за комп'ютером протягом тривалого часу може викликати різноманітні проблеми із зором, такі як втома очей, сухість очей або навіть серйозніші порушення зору, зокрема синдром комп’ютерного зору (CVS).

Для запобігання цим проблемам важливо дотримуватись режиму відпочинку (правило 20-20-20, тобто кожні 20 хвилин робити перерву на 20 секунд і дивитися на об'єкт, що знаходиться на відстані 20 футів). Також рекомендується використовувати екрани з антибліковим покриттям і налаштовувати контрастність та яскравість екрану відповідно до умов освітлення в робочому просторі.

Проблеми з поставою: Постійне сидіння перед комп'ютером може спричинити проблеми з поставою, болі в спині та шиї. Це особливо актуально для розробників програмного забезпечення, які працюють у сидячому положенні понад 6 годин на день.

Важливо використовувати ергономічні стільці, налаштовувати робоче місце так, щоб екран перебував на рівні очей, а клавіатура — на зручній висоті для запобігання напруженню в руках і спині.

У розробці чат-систем в реальному часі програмістам потрібно активно працювати з кодом, вирішувати складні задачі, співпрацювати з іншими членами команди та підтримувати високий рівень комунікації. Все це може призводити до психологічного навантаження. Часто у процесі розробки програмного забезпечення можуть виникати ситуації, коли терміни здачі проекту стають все більш обмеженими. Це може викликати стрес та вигорання у працівників.

Для зменшення стресу рекомендується використовувати методи тайм-менеджменту, організовувати робочі процеси в межах реалістичних термінів і створювати комфортні умови для командної роботи. Важливу роль відіграє підтримка і допомога з боку керівництва та колег. Психоемоційне вигорання: Робота в IT-сфері може бути виснажливою через постійну необхідність адаптуватися до нових технологій і змінюваних вимог. Вигорання може статися, коли програмісти працюють у надмірно інтенсивному режимі без належного відпочинку.

Важливо забезпечити гнучкий графік роботи, давати можливість для професійного розвитку та підтримувати здоровий баланс між роботою та особистим життям. Регулярні перерви і фізична активність допомагають зменшити напругу.

Розробка чат-систем в реальному часі зазвичай передбачає активну командну роботу. Це включає програмістів, тестувальників, дизайнерів інтерфейсів, а також інших фахівців. Така діяльність пов’язана з постійною комунікацією та координацією.

Взаємодія з іншими членами команди може бути як джерелом мотивації, так і стресу. Правильне управління проектами, чітке визначення завдань і вміння ефективно вирішувати конфлікти є важливими аспектами забезпечення здорового клімату в команді. Часто IT-фахівці повинні брати участь у численних нарадах, як онлайн, так і офлайн. Перевантаження наради можуть спричиняти

втому та дратівливість. Оптимізація комунікаційних процесів, використання відповідних інструментів для спілкування (наприклад, Slack, Microsoft Teams) дозволяють знизити цей стрес.

Розробка чат-систем, особливо тих, що функціонують в реальному часі, має свої особливості, які безпосередньо впливають на умови праці. Програмісти повинні працювати з великими обсягами даних, взаємодіяти з різними протоколами передачі інформації і тестувати взаємодію між користувачами в реальному часі.

Важливим аспектом є забезпечення безпеки даних, захист від хакерських атак, безпечне зберігання даних користувачів. Це вимагає постійного оновлення знань, уважності до змін у галузі безпеки та застосування найкращих практик програмування.

Також важливим є оптимізація програмного забезпечення для забезпечення високої продуктивності при одночасній роботі великої кількості користувачів. Для цього розробники повинні мати доступ до сучасних інструментів і технологій, що дозволяють зменшити ризики програмних помилок, збоїв і перегрузок серверів.

Охорона праці в ІТ-сфері є важливим аспектом, який стосується не тільки фізичного, а й психологічного стану працівників. Профілактика стресу, підтримка здоров'я співробітників та створення комфортних умов для ефективної роботи є основними чинниками успіху в розробці програмного забезпечення. У випадку розробки чат-систем у реальному часі особлива увага повинна приділятися як технічній безпеці, так і здоров’ю працівників, що дозволяє забезпечити не лише ефективну, але й безпечну розробку програмного продукту.

4.2 Питання щодо безпеки в надзвичайних ситуаціях

Надзвичайні ситуації – це події, що призводять до порушення нормальної життєдіяльності людей, забруднення навколишнього середовища, значних економічних і соціальних втрат. У світі існує велика кількість різноманітних надзвичайних ситуацій, що можуть виникнути внаслідок техногенних катастроф, природних катастроф, радіаційних, хімічних або біологічних інцидентів, а також військових конфліктів.

Надзвичайні ситуації можна класифікувати за різними ознаками:

– За характером: природні (землетруси, повені, урагани) та техногенні (аварії на виробництві, техногенні катастрофи).

– За масштабами: локальні, регіональні, національні та глобальні.

– За типами загрози: хімічні, радіаційні, біологічні, екологічні.

Кожен з типів НС має специфічні методи захисту, що залежать від виду загрози. Найбільш поширеними НС є техногенні катастрофи, що включають аварії на атомних електростанціях, хімічні вибухи, аварії на транспорті, які спричиняють значні загрози для здоров’я та життя людей.

Захист населення від наслідків надзвичайних ситуацій вимагає комплексного підходу, що включає інформаційне забезпечення. Важливим аспектом є своєчасне інформування населення про загрозу, використовуючи засоби масової інформації, спеціальні попереджувальні сигнали, сирени, телевізійні та радіопрограми.

В разі виникнення загрози на об'єкті необхідно організувати евакуацію людей в безпечні місця, забезпечити укриття в укриттях, що мають відповідну захист від радіаційних, хімічних або біологічних загроз.

У випадку НС необхідно організувати роботу медичних пунктів, забезпечити постраждалих швидкою допомогою, а також вжити заходів для запобігання епідеміям і хворобам. Важливо організувати роботу психологів та волонтерів, які допомагають населенню подолати стресові ситуації, що виникають під час катастроф.

Для ефективного захисту населення необхідно проводити постійну підготовку як для населення, так і для спеціалізованих служб: Проводяться тренування, інструктажі та курси для населення, щоб люди знали, як діяти в разі НС (як евакуюватися, куди йти, як захистити себе від певних загроз).

Спеціалізовані служби повинні мати навички надання першої медичної допомоги, організації евакуації, управління панікою та збереження безпеки громадян. Окремі структури повинні мати спеціальні засоби для надання першої допомоги, санітарної обробки постраждалих та лікування.

Радіаційний захист є важливою складовою охорони праці на підприємствах, що мають справу з радіоактивними матеріалами. Працівники, що виконують роботи в умовах радіаційного забруднення або потенційно небезпечних для здоров’я умовах, підлягають особливому захисту від радіаційного випромінювання.

Захист від радіації включає низку технологічних та організаційних заходів, що спрямовані на запобігання впливу радіації на організм людини. До основних принципів радіаційного захисту відносяться:

– Зменшення часу перебування на забруднених територіях.
– Збільшення відстані від джерела радіації.
– Захист за допомогою бар'єрів.

Працівники, які виконують роботи в умовах можливого радіаційного впливу, повинні проходити регулярне медичне обстеження та інструктажі щодо дій у разі надзвичайних ситуацій. Окрім того, необхідно дотримуватись усіх вимог. Робітники повинні проходити медичні огляди, щоб визначити наявність або відсутність впливу радіації на їхній організм. Окремо проводяться обстеження на наявність радіаційних захворювань. На виробничих об’єктах має проводитись моніторинг рівня радіації, а також оцінка потенційних ризиків для працівників.

Для роботи в зонах з підвищеною радіаційною небезпекою працівники повинні мати відповідний одяг, який забезпечує бар'єр від радіації, а також прилади для контролю за рівнем радіації. На виробничих об’єктах, що працюють

з радіоактивними матеріалами, повинні бути організовані спеціальні режими захисту.

Визначення максимально дозволених доз для працівників, контроль за їх виконанням. Використання технологічного обладнання, яке обмежує або усуває викиди радіації, а також має системи моніторингу.

Доступ на об'єкти, де присутні радіоактивні матеріали, повинний бути обмежений для сторонніх осіб. На таких об’єктах повинна бути розроблена чітка система допуску для працівників та проведення навчань.

4.3 Висновок до четвертого розділу

У четвертому розділі кваліфікаційнох роботи розглянуто основні аспекти охорони праці в ІТ-індустрії, зокрема ризики, пов'язані з роботою в офісах та при розробці програмного забезпечення, таких як чат-системи в реальному часі.

Особлива увага приділена фізичним та психологічним факторам, що впливають на здоров'я працівників: тривала робота за комп’ютером, стресові навантаження та вигорання.

Проаналізовано заходи для запобігання цим проблемам, такі як організація комфортних умов праці, використання сучасних технологій для захисту даних та підтримка психологічного здоров’я співробітників.

Подано рекомендації щодо створення безпечних умов для працівників в ІТ-сфері, акцентуючи на важливості регулярних медичних оглядів, психосоціальної підтримки та технічних заходів для забезпечення кібербезпеки.

Розглянуто питання захисту населення в разі виникнення надзвичайних ситуацій (НС), зокрема організацію евакуації, укриттів та медичної допомоги, а також заходи щодо психологічної підтримки постраждалих.

Проаналізовано основні типи НС, такі як природні, техногенні, хімічні, радіаційні та біологічні загрози, та визначено ключові елементи захисту, що включають своєчасне інформування громадян і належну підготовку спеціалізованих служб.

Подано рекомендації щодо радіаційного захисту працівників та службовців, що працюють у зонах з підвищеною радіаційною небезпекою. Це включає використання засобів індивідуального захисту, проведення медичних оглядів та дотримання стандартів безпеки на виробничих об'єктах.

ВИСНОВКИ

У цій роботі виконано дослідження літературних джерел, які охоплюють різні аспекти тестування RESTful API, від генерації тестів до тестування "чорної скриньки", де немає доступу до вихідного коду; до генерації вхідних даних, наприклад, за допомогою фаззингу; та до прогнозування виводу за допомогою машинного навчання для автоматизації тестування та зворотного зв'язку.

Як зазначено в розділі результатів, дослідники зіткнулися з кількома труднощами, намагаючись створити ефективну структуру для генерації модульних тестів та тестування загалом. З обговорення можна зробити висновок, що дотримання універсальних стандартів під час розробки та опису RESTful API зробить навіть існуючі структури та методології ефективнішими. Це пов'язано з тим, що тестування та генерація модульних тестів сильно залежать від стандартного опису документа RESTful API (наприклад, XML, звичайний JSON або OpenAPI (також відомий як swagger)) .

Як перший крок, ця стандартизація полегшить перший рівень проблеми. По-друге, цей огляд також показує, що підтримка узгодженості описового документа є не менш важливою. По-третє, відсутність комплексної підтримки тестування RESTful API з підтримкою автентифікації все ще залишається відкритою областю для проведення досліджень.

Таким чином можна зробити висновок, що ця робота допоможе прокласти шлях для подальших досліджень на основі тестування RESTful API. Машинне навчання використовується для прогнозування результатів заданого тестового випадку. Використання машинного навчання для прогнозування покриття коду заданого набору тестів було б цікавою та важливою темою дослідження. Воно може допомогти визначити, які набори тестів залишити, а які оновити або відкинути через покриття коду, що має першорядне значення для підтримки набору тестів, а також для тестування, орієнтованого на безпеку, як уже було зазначено вище. Існує серйозна нестача безпечних наборів тестів RESTful API, які мають додаткову чутливість, оскільки такі виклики API керували б

конфіденційними або захищеними частинами сервісу, і, як така, неочікувана поведінка такого API сама по собі є загрозою безпеці .

Подальша робота в майбутньому полягатиме в покращенні результатів цього дослідження та зосередженні на ролі машинного навчання, що застосовується для покращення рішень для тестування, а також на забезпеченні автентифікації для RESTful API.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Kharchenko, A., Bodnarchuk, I., & Yatcyshyn, V. (2014). The method for comparative evaluation of software architecture with accounting of trade-offs. American Journal of Information Systems, 2(1), 20-25.

2. Bodnarchuk, I., Lisovyi, V., Kharchenko, O., & Galai, I. (2018, September). Adaptive method for assessment and selection of software architecture in flexible techniques of design. In 2018 IEEE 13th International Scientific and Technical Conference on Computer Sciences and Information Technologies (CSIT) (Vol. 1, pp. 292-297). IEEE.

3. Ihor, B., Oleksii, D., Aleksandr, K., Nataliia, K., Oleksandr, M., & Volodymyr, P. (2019, January). Multicriteria choice of software architecture using dynamic correction of quality attributes. In International Conference on Computer Science, Engineering and Education Applications (pp. 419-427). Cham: Springer International Publishing.

4. Яцишин В. Технологія оцінювання якості web-застосувань/ В.Яцишин // Вісник ТДТУ. – 2009. – Том 14. – № 4. – С. 132-140. – (приладобудування та інформаційно-вимірювальні технології).

5. Li, L., Chou, W., Zhou, W., & Luo, M. (2016). Design patterns and extensibility of REST API for networking applications. IEEE Transactions on Network and Service Management, 13, 154–167.

6. Neumann, A., Laranjeiro, N., & Bernardino, J. (2018). An analysis of public REST web service APIs. IEEE Transactions on Services Computing, 14, 957–970.

7. Pahl, C., & Jamshidi, P. (2016). Microservices: A systematic mapping study. In Proceedings of the 6th International Conference on Cloud Computing and Services Science (Vol. 1–2, pp. 137–146).

8. Khare, R., & Taylor, R. (2004). Extending the Representational State Transfer (REST) architectural style for decentralized systems. In Proceedings of the 26th International Conference on Software Engineering (pp. 428–437).

9. Pautasso, C., Zimmermann, O., & Leymann, F. (2008). RESTful web services vs. "big" web services: Making the right architectural decision. In Proceedings of the 17th International Conference on World Wide Web (pp. 805–814).

10. Ed-Douibi, H., Izquierdo, J. L. C., & Cabot, J. (2018). OpenAPItoUML: A tool to generate UML models from OpenAPI definitions. In Proceedings of the International Conference on Web Engineering (pp. 487–491). Springer.

11. Dusseault, L., & Snell, J. (2010). PATCH method for HTTP. RFC 5789. Internet Engineering Task Force. https://www.rfc-editor.org/rfc/rfc5789

12. Fielding, R. T., & Taylor, R. N. (2008). Architectural styles and the design of network-based software architectures. University of California.

13. Mogul, J. C., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (1999). Hypertext Transfer Protocol–HTTP/1.1. https://www.w3.org/Protocols/HTTP/1.1/rfc2616.pdf

14. Ghani, I., Wan-Kadir, W. M. N., & Mustafa, A. (2019). Web service testing techniques: A systematic literature review. International Journal of Advanced Computer Science and Applications, 10, 443–458.

15. Barbir, A., Hobbs, C., Bertino, E., Hirsch, F., & Martino, L. (2007). Challenges of testing web services and security in SOA implementations. In Test and analysis of web services (pp. 395–440). Springer.

16. Kitchenham, B., & Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering. Technical Report, 5, 1–57.

17. Голінько В. І. Охорона праці в галузі інформаційних технологій: навч. посіб. / В. І. Голінько, М. Ю. Іконніков, Я. Я. Лебедєв; М-во освіти і науки України, Держ. вищий навч. закл. "Нац. гірн. ун-т". - Дніпропетровськ: НГУ, 2015. - 246 с.

18. Гандзюк М.П. Основи охорони праці: Підручник. 4-е вид./Гандзюк М.П., Желібо Є.П., Халімовський М.О. - Київ: Каревела, 2008. – 384с.

19. Техноекологія та цивільна безпека. Частина «Цивільна безпека»: Навчальний посібник; укл.: Стручок В. С. Тернопіль: ФОП Паляниця В.А., 2022. 150 с.

20. Безпека в надзвичайних ситуаціях. Методичний посібник для здобувачів освітнього ступеня «магістр» всіх спеціальностей денної та заочної (дистанційної) форм навчання / укл.: Стручок В. С. Тернопіль: ФОП Паляниця В. А., 2022. 156 с.

21. Умови праці працівників, які використовують у роботі персональні комп'ютери. Zolochiv.Net. URL: https://zolochiv.net/umovy-pratsi-pratsivnykiv-iaki-vykorystovuiut-u-roboti-personal-ni-komp-iutery/ (дата звернення: 25.10.2024).

ДОДАТКИ

Додаток А

Тези доповіді

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Тернопільський національний технічний університет імені Івана Пулюя
Маріборский університет (Словенія)
Технічний університет у Кошице (Словаччина)
Вільнюський технічний університет ім. Гедимінаса (Литва)
Краківський економічний університет (Польща)
Вроцлавський економічний університет (Польща)
Університет «Опольська Політехніка» (Польща)
Національний університет «Полтавська політехніка імені Юрія Кондратюка»
Вінницький національний аграрний університет
Львівський національний університет ім. І. Франка
Головне управління Пенсійного фонду в Тернопільській області
Наукове товариство ім. Шевченка
Тернопільський обласний комунальний інститут післядипломної педагогічної освіти
Сумський державний педагогічний університет
Запорізький національний університет

АКТУАЛЬНІ ЗАДАЧІ СУЧАСНИХ ТЕХНОЛОГІЙ

Збірник

тез доповідей

XIV Міжнародної науково-технічної конференції молодих учених та студентів

11-12 грудня 2025 року

УКРАЇНА
ТЕРНОПІЛЬ – 2025

УДК 004.451:004.738.5:004.75
В.А. Яцишин, П.В. Налутка, О.В. Доберчак, І.О. Боднарчук
(Тернопільський національний технічний університет імені Івана Пулюя, Україна)

СУЧАСНА АРХІТЕКТУРА СТРІМІНГОВИХ ПЛАТФОРМ В РЕАЛЬНОМУ ЧАСІ

V.A. Yatsyshyn, P.V. Nalutka, O.V. Doberchak, I.O. Bodnarchuk

MODERN ARCHITECTURE OF REAL-TIME STREAMING PLATFORMS

На відміну від систем минулих часів, які спиралися на асинхронну та пакетну обробку, архітектура програмного забезпечення реального часу зараз є важливою в сучасному цифровому світі, де миттєва обробка інформації є нормою. Швидкість, надійність та передбачуваність є ключовими в застосунках реального часу, від платформ електронної комерції, що реагують на дії клієнтів, до фінансових систем, що виконують транзакції за мікросекунди. Саме тут на допомогу приходить архітектура програмного забезпечення реального часу, яка робить рівень продуктивності цих застосунків прийнятним для роботи в режимі реального часу.

Термін «реальний час» означає для системи, що вона повинна реагувати в чітко встановлені терміни або дедлайни. Для архітекторів та розробників розуміння того, як проектувати ці системи, є критично важливим для створення рішень у відповідності од сформульованих вимог. Навіть правильний результат може бути марним, якщо він надходить занадто пізно.

Системи реального часу поділяються на три категорії:

- Системи жорсткого реального часу. Недотримання термінів є системним збоєм. Прикладами є промислові системи управління та торговельні платформи, де час транзакцій є критично важливим.
- Системи роботи в режимі реального часу надійних систем. Пропуск термінів погіршує якість обслуговування, але не призводить до збою системи. Прикладами є відеоконференції, де випадкові переривання кадрів дратують, але не завершують дзвінок.
- Системи м’якого реального часу. Корисність результату знижується після закінчення терміну, але система продовжує функціонувати. Прикладами є механізми рекомендацій контенту, де персоналізація з невеликою затримкою все ще цінна.

Щоб врахувати ці різні очікування, цільовий додаток має бути спроектований та розроблений з урахуванням продуктивності в режимі реального часу. Незалежно від очікуваних термінів отримання результатів, добре побудована архітектура реального часу керує ресурсами, плануванням завдань, зв'язком та обробкою помилок, щоб забезпечити дотримання часових обмежень для своєї конкретної категорії.

Також важливо розуміти, що підпадає під архітектуру програмного забезпечення реального часу, а що ні (таблиця 1). Пакетна обробка, яка опрацьовує дані масово через заплановані проміжки часу, є класичним прикладом системи, що не працює в реальному часі. Хоча архітектура реального часу є поширеним шляхом, не всі варіанти використання та сценарії вимагають таких можливостей.

Майже всі програми містять компоненти реального часу поряд з аналізом історичних даних. Цей зсув підкреслює важливу роль програмного забезпечення та архітектур даних реального часу в сучасних програмах, що зумовлено ключовими факторами, серед яких варто згадати наступні.

Операції, критично важливі для бізнесу.

Для багатьох систем час впливає на бізнес-результати. Численні програми та галузі покладаються на справжні системи реального часу, щоб забезпечити дохід. Ось деякі приклади цього:

- Платформи електронної комерції: оновлення товарних запасів у режимі реального часу, персоналізація та обробка транзакцій безпосередньо впливають на коефіцієнти конверсії та задоволеність клієнтів.
- Фінансові послуги: торговельні платформи, системи обробки платежів та виявлення шахрайства потребують для роботи часу реагування на рівні мілісекунд.

Таблиця 1. Порівняння програмної архітектури реального часу та решти

Аспект	**Архітектура програмного забезпечення реального часу**	**Архітектура програмного забезпечення не для роботи в реальному часі**
випадки використання	Виявлення шахрайства, персоналізація в режимі реального часу, моніторинг у реальному часі, торгові додатки	Нарахування заробітної плати, створення звітів, пакетний імпорт, публікація контенту
Критерії успіху	Залежить як від точності , *так і* від своєчасності результатів	Залежить виключно від точності, незалежно від того, коли буде отримано результат
Дизайн застосунку	Компоненти, що реагують на події, неблокуючі та враховують час	Синхронні, запит/відповідь, блокувальні потоки прийнятні
Структура коду	Пріоритетність передбачуваних шляхів виконання, мінімальний вплив на збирання сміття (GC), асинхронний ввід/вивід	Надає пріоритет ремонтопридатності або пропускній здатності над точністю часу
Вплив технічного боргу	Архітектурно-технічний борг створює затримки, непередбачуваність та пропущені терміни — катастрофічні наслідки для систем реального часу. Навіть незначні недоліки, такі як блокування викликів або необмежені черги, можуть порушувати угоди про рівень обслуговування (SLA) або спричиняти каскадні збої.	Борг уповільнює доставку, збільшує витрати на обслуговування та може погіршити продуктивність, але рідко призводить до негайного збою. Терміни є гнучкими.

Кращий користувацький досвід.

Як ми вже обговорювали, очікування «миттєвого» обслуговування є складним завданням. Справжнього миттєвого зворотного зв'язку можна досягти лише завдяки інтеграції можливостей реального часу в базові сервіси. Наприклад, користувачі очікують миттєвого зворотного зв'язку, використовуючи:

- Веб- та мобільні додатки: адаптивні інтерфейси із часом завантаження менше секунди та миттєвими оновленнями (наприклад, стрічки соціальних мереж, спільне редагування) зараз є нормою [3].
- Потокові сервіси: доставка контенту з мінімальною буферизацією та адаптивною якістю вимагає прийняття рішень у режимі реального часу.

- Резервні механізми зі зниженою, але прийнятною продуктивністю.
- Моніторинг справності з швидким виявленням збоїв.

5. Моделі узгодженості даних.

Залежно від типу даних та рішень, що отримуються на їх основі, багато систем реального часу послаблюють сувору узгодженість для підвищення продуктивності. У таких випадках зазвичай застосовують:

- Узгоджені моделі для некритичних даних.
- Стратегії вирішення конфліктів для одночасних оновлень для підтримки цілісності даних.
- Шаблони CQRS (Command Query Responsibility Segregation) для розділення операцій читання та запису.

6. Подієво-орієнтований дизайн

Асинхронні, подієво-орієнтовані архітектури часто формують основу систем реального часу. Це означає, що кодові та архітектурні компоненти системи включатимуть:

- Брокери повідомлень, такі як Kafka або RabbitMQ, для надійної та впорядкованої доставки подій [2].
- Шаблони джерел подій для змін стану, що підлягає аудиту.
- Потокова обробка для безперервного аналізу даних.

Дотримуючись цих принципів, розробники можуть створювати системи, які відповідають потребам реального часу їхніх випадків використання. Ці шість принципів складають основні вимоги під час проектування та впровадження програм і сервісів реального часу. Крім того, розуміння різних аспектів продуктивності має вирішальне значення для системи реального часу. Це буде предметом нашого наступного обговорення.

Архітектура програмного забезпечення реального часу еволюціонувала від нішевої потреби у вбудованих системах до загальноприйнятого підходу для сучасних додатків. Оскільки компанії прагнуть надавати досвід, керований даними, здатність обробляти дані та реагувати в режимі реального часу стала ключовою конкурентною перевагою.

У цій доповіді досліджувалися основи систем реального часу: їх класифікація (жорсткі, стійкі, м'які), керівні принципи та ключові показники продуктивності.

Сучасні архітектури реального часу спираються на такі технології, як платформи потокової передачі подій (Kafka), сховища даних в пам'яті, моделі реактивного програмування та хмарні шаблони. При продуманому поєднанні ці компоненти дозволяють створювати масштабовані системи, які відповідають суворим гарантіям часу. Але чудове програмне забезпечення – це не лише технологія, а й те, як воно архітектурно спроектовано.

Щоб задовольнити вимоги систем реального часу, архітекторам потрібна постійна видимість того, як створюються та працюють додатки.

Література

1. Microservices architecture and design: A complete overview - vFunction. vFunction. URL: https://vfunction.com/blog/microservices-architecture-guide/ (date of access: 01.10.2025).

2. Ланевич Т. Apache Kafka та Rabbitmq: порівняння архітектур та можливостей // Тези XIII МНПК „Актуальні задачі сучасних технологій“, 11-12 грудня 2024 року. Тернопіль: ФОП Паляниця В. А., 2024. С. 402–403.

3. Хом’як А. С. Підвищення ефективності обробки в реальному часі у службах чат-ботів через інтеграцію черги запитів для розподілення навантаження //