Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет Комп'ютерні технології та системи
Кафедра Комп'ютерні інформаційні технології

Пояснювальна записка

до кваліфікаційної роботи
магістра

на тему: «Дослідження і розробка засобів автоматизованного тестування програмного забезпечення»
за освітньою програмою **12 Інженерія програмного забезпечення**
зі спеціальності: **121 Інженерія програмного забезпечення**

Виконав: студент групи ПЗ2226: / Юлія ВОДЯНІК /

Керівник: / Олена КУРОП'ЯТНИК /

Нормоконтролер: /Світлана ВОЛКОВА /

Засвідчую, що у цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент

Дніпро – 2024 рік

Ministry of Education and Science of Ukraine

Ukrainian State University of Science and Technologies

Faculty Computer technologies and systems
Department Computer information technology

Explanatory Note

to Master's Thesis
master

on the topic: « Research and development of automated software testing tools»

according to educational curriculum **12 software engineering**
in the Speciality: **121 software engineering**

Done by the student of the group PZ2226: /Yuliia VODIANIK/

Scientific Supervisor: / Olena KUROPIATNYK /

Normative controller: / Svitlana VOLKOVA/

Dnipro – 2024

РЕФЕРАТ

Об’єктом дослідження є процес автоматизованого тестування API. Предметом дослідження є засоби підвищення швидкості та ефективності автоматизованого тестування API.

Метою дослідження є підвищення швидкості та ефективності тестування програмного забезпечення за допомогою розробленого інструменту для автоматичного підрахунку покриття API автотестами.

Методи дослідження: статистичний аналіз - для обробки та аналізу результатів тестування інструменту, що дозволяє отримати об'єктивну оцінку його ефективності; спостереження та аналіз - для вивчення даних, зібраних програмним забезпеченням під час його виконання, для аналізу його роботи, проблем, користувацької поведінки тощо.

Результати та їх новизна: розроблено ефективний інструмент на мові Python для автоматичного підрахунку покриття API автотестами.

Пояснювальна записка складається зі вступу, чотирьох розділів, висновків, бібліографічного списку та трьох додатків.

Вступ описує суть, мету та актуальність роботи (3 сторінки).

Перший розділ: аналіз сучасних практик та тенденції в тестуванні програмного забезпечення (14 сторінок).

Другий розділ: обґрунтування необхідності і методу вирішення проблеми розрахунку покриття арі ендпоінтів автотестами (11 сторінок).

Третій розділ: впровадження автоматизованого тестування і розробка інструменту для підрахунку покриття API автотестами (21 сторінка).

Четвертий розділ: дослідження доцільності автоматизації тестування арі та оцінювання його результатів (27 сторінок).

Додатки: технічне завдання, робочий проект, тези доповіді.

Таблиць – 8 , рисунків – 31, бібліографія – 11 джерел.

Ключові слова: автоматицація; тестування API; обчислення покриття автотестами; алгоритм

алгоритму

ЗМІСТ

ВСТУП

Актуальність роботи. Сучасний розвиток програмної інженерії акцентує увагу на постійному покращенні якості програмного забезпечення. Автоматизоване тестування, як один із етапів розробки, відіграє ключову роль у забезпеченні високої якості.

На сьогоднішній день процес автоматизованого тестування є обов'язковою частиною розробки ПЗ, і часто потребує значних ресурсних витрат. Зокрема, перевірка покриття API автотестами вимагає чіткого підходу та інструментарію, який би дозволяв швидко визначити і контролювати рівень тестового покриття.

Оптимізація цього процесу може стати вирішальним фактором в розробці програмного забезпечення. Розвиток методів та інструментів для автоматичного підрахунку покриття дозволить зменшити час тестування, підвищити його якість та, як результат, прискорити вивід продукту на ринок.

Більшість існуючих інструментів зосереджена на виконанні конкретних тестових сценаріїв, проте контроль покриття API автотестами залишається відкритим питанням.

У даній роботі пропонується новий підхід та інструментарій, що дозволить легко автоматизувати цей процес, зменшивши ручний вклад спеціалістів.

Це дослідження є частиною програми, спрямованої на покращення якості продуктів, а также також вдосконалення методів та технік тестування програмного забезпечення. Результати дослідження плануються до впровадження автоматичного тестування в компанії Apriorit.

Об’єкт та предмет дослідження.

Об'єктом дослідження є процес автоматизованого тестування API.

Предметом дослідження є засоби підвищення швидкості та ефективності автоматизованого тестування API.

Мета та задачі дослідження. Метою цього дослідження є підвищення швидкості та ефективності тестування програмного забезпечення за допомогою розробленого інструменту для автоматичного підрахунку покриття API автотестами.

Відповідно до поставленої мети передбачено розв'язання наступного комплексу завдань:

1. Проаналізувати існуючі методи та інструменти для визначення покриття коду автотестами.
2. Розробити алгоритм для автоматичного підрахунку покриття API автотестами, що враховує специфіку роботи з різними API.
3. Створити програмний інструмент на мові Python, що реалізує розроблений алгоритм.
4. Провести випробування розробленого інструменту на реальних проектах з метою визначення його ефективності та точності.
5. Зіставити результати роботи розробленого інструменту з мануальним тестуванням для оцінки його переваг та недоліків.

Методи дослідження. Для перевірки досягнення мети дослідження та вирішення поставлених завдань у даній магістерській роботі необхідно використовувати такі емпіричні методи як:

Статистичний аналіз - для обробки та аналізу результатів тестування інструменту, що дозволяє отримати об'єктивну оцінку його ефективності.

Спостереження та аналіз - для вивчення даних, зібраних програмним забезпеченням під час його виконання, для аналізу його роботи, проблем, користувацької поведінки тощо.

Наукова новизна. Наукова новизна роботи полягає у розробці інструменту для автоматизованого тестування програмного забезпечення з метою підвищення ефективності процесу тестування за критеріями швидкості та покриття тестами. Нова методика перевірки покриття API автотестами може бути використана тестувальниками програмного забезпечення для контролю якості автотестів та

дотримання вимог до тестування. Впровадження автоматичного підрахунку покриття дозволить значно зекономити час і людські ресурси.

Практичне значення. Практичне значення роботи проявляється у застосуванні існуючих рішень для організації автоматичного тестування , а також створенні нового інструменту, що може бути застосований в реальних проектах для покращення якості автоматизованого тестування, збільшення продуктивності команди та оптимізації робочого процесу. Для цього було досліджено і впроваджено:

-CI/CD - Jenkins -Інструмент для неперервної інтеграції та доставки. -Тестування API - JMeter - Інструмент для виконання запитів,

а також розроблено нову програму на Python, яка є доповненням до існуючих рішень та спрямована на автоматичне тестування покриття.

В цілому, практична значущість отриманих результатів полягає в можливості покращення процесу тестування та збільшення його ефективності.

Апробація результатів дослідження та публікації. Результати дослідження були представлені та обговорені на науковому семінарі кафедри комп'ютерних інформаційних технологій 06.10.23, на XVII Міжнародної науково-практичної конференції «СУЧАСНІ ІНФОРМАЦІЙНІ ТА КОМУНІКАЦІЙНІ ТЕХНОЛОГІЇ НА ТРАНСПОРТІ, В ПРОМИСЛОВОСТІ ТА ОСВІТІ» (13-14 грудня 2023, м. Дніпро).

1 АНАЛІЗ СУЧАСНИХ ПРАКТИК ТА ТЕНДЕНЦІЇ В ТЕСТУВАННІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Сучасні практики та тенденції в тестуванні програмного забезпечення

До сучасних практик та тенденції в тестуванні програмного забезпечення належать:

1. Неперервна інтеграція (CI) і неперервна доставка (CD)

Ці підходи передбачають постійне злиття коду та його автоматичне тестування, а також автоматичне розгортання виробничої версії ПЗ.

Проблеми: Потреба в постійному моніторингу та обслуговуванні пайплайн; можливі проблеми зі стабільністю при швидких релізах.

2. Shift-left тестування

Тестування проводиться на ранніх етапах розробки, щоб виявити та виправити дефекти швидше.

Проблеми: Вимагає більше ресурсів на ранніх етапах розробки; може впливати на швидкість розробки.

3. Тестування на основі штучного інтелекту (AI) та машинного навчання (ML). Використання AI і ML для автоматизації написання тестів, аналізу результатів тестування та оптимізації процесу тестування.

Проблеми: Потенційна ненадійність AI-генерованих тестів; висока вартість та складність інтеграції.

4. Тестування в облаках

Використання хмарних платформ для проведення тестування, що дає гнучкість та масштабованість.

Проблеми: Залежність від зовнішніх постачальників; можливі питання безпеки даних.

5. Тестування в контейнерах (Docker, Kubernetes)

Використання контейнеризації для створення ізольованого середовища для тестування.

Проблеми: Складність налаштування та управління; потреба в додаткових знаннях.

6. Тестування продуктивності в режимі реального часу

Моніторинг продуктивності ПЗ в режимі реального часу для виявлення та усунення "вузьких місць".

Проблеми: Може вимагати додаткових інвестицій в інструменти та ресурси.

7. Тестування безпеки "зсередини"

Тестування безпеки ПЗ з точки зору внутрішнього користувача або атаки "зсередини".

Проблеми: Важкість виявлення внутрішніх загроз; потреба в глибокому розумінні і знанні існуючих загроз і вразливостей програмного продукту. [1]

1.2 Головні принципи при організації процеса автоматизованого тестування на проектах

Процес автоматизованого тестування повинен враховувати ряд важливих вимог та аспектів, щоб бути ефективним і допомагати команді розробників у підтримці високої якості продукту. Ось деякі ключові вимоги і аспекти:

1. Планування

Стратегія автоматизації:

Визначення областей для автоматизації, засобів та технологій.

Вибір інструментів:

Обрання інструментів, які відповідають технологічному стеку і потребам проекту.

Розробка плану тестування:

Встановлення цілей, обсягу, ресурсів і часових рамок тестування.

2. Розробка тестів

Скрипти для тестування:

Розробка та оптимізація скриптів тестування з урахуванням можливості повторного використання.

Тестові дані:

Підготовка та управління даними, необхідними для виконання тестів.

Організація тестів:

Розробка структури тестових сценаріїв, групування та параметризація тестів.

3. Виконання тестів

Налаштування середовища:

Гарантування стабільності та консистентності тестового середовища.

Запуск тестів:

Автоматизація процесу запуску тестових скриптів.

Моніторинг:

Нагляд за ходом виконання автоматизованих тестів.

4. Аналіз результатів

Логування:

Збір, зберігання та аналіз логів тестових сесій.

Звітність:

Автоматизація генерації та аналізу тестових звітів.

Баг-трекінг:

Інтеграція з системами відстеження помилок і ефективне управління багами.

5. Підтримка та оптимізація

Підтримка тестів:

Адаптація тестів до змін у продукті та їх актуалізація.

Оптимізація тестів:

Перегляд та оптимізація тестових сценаріїв з метою підвищення ефективності тестування.

Подальше розвиток:

Постійний розвиток навичок команди і удосконалення процесу автоматизованого тестування.

Ці вимоги допомагають організувати процес автоматизованого тестування так, щоб він був консистентний, прозорий і сприяв ефективному забезпеченню якості продукту. Слід пам’ятати, що кожен проект унікальний, тому вимоги і підходи можуть адаптуватися з урахуванням специфіки конкретного виробничого середовища. [2-3]

1.3 Рівні автоматизованого тестування

Автоматизоване тестування є ключовим елементом в сучасній розробці програмного забезпечення. Воно може бути поділене на різні рівні, кожен із яких фокусується на конкретних аспектах процесу тестування.

1. Юніт-тестування:

Мета: перевірка коректності роботи окремих модулів чи функцій програмного забезпечення.

Інструменти: JUnit, NUnit, TestNG тощо.

2. Інтеграційне тестування:

Мета: перевірка взаємодії між різними модулями або системами.

Інструменти: JUnit, TestNG, JMeter, Postman, SoapUI тощо.

3. Тестування системи:

Мета: перевірка повної системи щоб пересвідчитися, що вся аплікація працює відповідно до вимог.

Інструменти: Selenium, Appium, JMeter тощо.

4. Тестування прийняття (UAT – User Acceptance Testing):

Мета: переконатися, що система готова до продакшну і відповідає вимогам бізнесу.

Інструменти: Cucumber, JBehave, SpecFlow тощо.

Більше того, автоматизоване тестування може бути поділене згідно з іншими основними принципами, наприклад:

1. Тестування інтерфейсу користувача:

Мета: перевірка UI на відповідність вимогам і коректність взаємодії з користувачем.

Інструменти: Selenium, Cypress, Appium тощо.

2. Тестування продуктивності:

Мета: визначення продуктивності, стабільності, та масштабуємості системи під навантаженням.

Інструменти: JMeter, LoadRunner, Apache Bench тощо.

3. Тестування безпеки:

Мета: виявлення потенційних слабких місць та вразливостей у системі.

Інструменти: OWASP ZAP, Burp Suite, Nessus тощо.

4. Тестування сумісності:

Мета: забезпечення коректної роботи програми на різних пристроях, браузерах, ОС тощо.

Інструменти: BrowserStack, Sauce Labs тощо.

Важливо відзначити, що кожен рівень та вид тестування має свою унікальну цінність у процесі розробки ПЗ і допомагає забезпечити створення високоякісного продукту. При цьому, інструменти, обрані для автоматизації тестування, можуть суттєво залежити від конкретних потреб та контексту проекту. [4-5]

1.4 Вимоги до розробки автотестів

Для забезпечення швидкого та надійного тестування, що важливо для постійної перевірки якості продукту в процесі розробки, є кілька вимог, які часто пред'являються до автотестів:

1. Надійність

Стабільність: Тести повинні бути стабільними і визначати однаковий результат за однакових умов.

Точність: Автотест повинен чітко відповідати визначеним сценарієм і перевіряти конкретні вимоги.

2. Масштабованість

Оптимізація: Тести повинні бути оптимізованими так, щоб вони працювали швидко і не використовували зайві ресурси.

Паралелізація: Здатність до паралельного виконання, щоб підтримувати швидкі ітерації розробки.

3. Підтримуваність

Легкість підтримки: Тести повинні бути структурованими і документованними, таким чином, щоб їх було легко підтримувати та адаптувати до змін.

Повторне використання: Елементи тестів, такі як функції, бібліотеки та дані, повинні бути організовані так, щоб їх можна було легко повторно використовувати.

4. Доступність

Версіонування: Тести повинні бути під контролем версий, щоб забезпечити консистентність і відслідковуваність змін.

Інтеграція: Легка інтеграція з системою контролю версій, CI/CD пайплайнами та іншими інструментами.

5. Звітність

Логування: Автотести повинні вести детальні логи, щоб спростити аналіз проблем.

Звіти: Автоматична генерація зрозумілих і інформативних тестових звітів.

6. Покриття

Декомпозиція: Тестові сценарії повинні бути добре декомпозовані, кожен тест повинен перевіряти щось одне.

Повнота: Автотести повинні надавати якомога більше покриття функціоналу за рахунок різноманітних сценаріїв.

7. Взаємодія

Незалежність: Тести повинні бути незалежними і не впливати один на одного під час виконання.

Автоматизована перевірка: Всі перевірки повинні бути повністю автоматизовані без потреби в ручній перевірці результатів.

8. Універсальність

Крос-платформеність: Здатність тестів працювати на різних платформах та конфігураціях.

Локалізація: Здатність тесту адаптуватися до різних локалізацій та конфігурацій.

Вимоги до автотестів можуть варіюватися в залежності від проекту та організації, але згадані вище аспекти є загальноприйнятими стандартами якості для автоматизованих тестів у більшості випадків.[6]

1.5 Аналіз існуючих проблем мануального і автоматичного тестування

Проблеми мануального тестування:

- Часоємність: Мануальне тестування, особливо для великих систем, може бути дуже трудомістким і потребувати багато часу.
- Помилки через людський фактор: Тестувальники можуть пропустити деякі дефекти або не виконати тестовий сценарій правильно.
- Втома: Повторювані тести можуть стати рутинними і монотонними, що збільшує шанси на помилки.

- Вартість: На довготривалі проекти мануальне тестування може виявитися дорожчим, оскільки потребує більше ресурсів у порівнянні з автоматизованим.
- Обмежена можливість відтворення: Може бути складно відтворити певні дефекти, особливо якщо тестувальник не зберіг докладну інформацію про умови тесту.

Проблеми автоматизованого тестування:

- Вартість на початковому етапі: Введення автоматизованого тестування вимагає початкових інвестицій у засоби та навчання персоналу.
- Потреба в кваліфікованих спеціалістах: Створення ефективних автоматичних тестів вимагає спеціальних знань і досвіду.
- Обслуговування тестів: Автоматичні тести можуть "ламатися" змінами в ПЗ, і їх потрібно регулярно оновлювати.
- Обмеженість: Не всі типи тестування можуть бути повністю автоматизовані, наприклад, тестування з точки зору користувача або дизайну.
- Помилки в автоматичних тестах: Якщо автоматичний тест написаний з помилками, він може не виявляти дефекти або виявляти неіснуючі.
- Відсутність контексту: Автоматичні тести виконуються за строго визначеними сценаріями і можуть пропустити проблеми, які людина помітила б.

Обидва підходи мають свої переваги та недоліки. Найкращий підхід, як правило, полягає в комбінації обох методів: автоматизація регресійних тестів і мануальне тестування для комплексних або суб'єктивних аспектів ПЗ. [7]

1.6 Огляд основних проблем при організації та підтримки процесса автоматизованого тестування програмного забезпечення

Організація та підтримка процесу автоматизованого тестування програмного забезпечення може стикатися з рядом проблем. Ось деякі з них:

- Високі початкові витрати: Запуск автоматизованого тестування вимагає інвестицій у засоби, налаштування середовища та навчання команди. Ці витрати можуть бути високими, особливо для маленьких компаній або стартапів.

- Потреба в кваліфікованих спеціалістах: Автоматизоване тестування вимагає знань не тільки у тестуванні, але й у програмуванні. Знайти грамотних інженерів з автоматизованого тестування може бути викликом.
- Підтримка автотестів: Код тестів також потребує підтримки. Зі змінами в програмному забезпеченні тести можуть ставати застарілими або непрацездатними.
- Фальшиві позитивні та фальшиві негативні результати: Автоматизовані тести можуть іноді видавати (надавати) неточні результати, що може призвести до пропущених дефектів або непотрібних спроб їх виправлення.
- Обмеження інструментів: Деякі інструменти автоматизації можуть мати обмеження у підтримці платформ, браузерів або мов програмування.
- Неправильне сприйняття автоматизації: Існує думка, що автоматизація може замінити мануальне тестування повністю. Це може призвести до недостатнього покриття тестами або пропуску важливих аспектів тестування.
- Складність розгортання: Встановлення, налаштування та інтеграція інструментів автоматизації у поточне середовище може бути складною задачею.
- Проблеми з переносимістю: Тести, створені для конкретного середовища або версії ПЗ, можуть не працювати коректно в інших умовах.
- Відсутність стратегії: Без чіткої стратегії та планування автоматизація може стати хаотичною та менш ефективною.
- Вартість інструментів: Деякі комерційні інструменти для автоматизованого тестування можуть бути дорогими, що ставить під загрозу їх придбання та використання в бюджетних проектах.

Для ефективної автоматизації важливо враховувати ці проблеми та шукати шляхи їх вирішення, розробляючи стратегію тестування і вибираючи інструменти. [8]

1.7 Аналіз існуючих засобів та інструменти перевірки програмного забезпечення

Автоматизоване тестування програмного забезпечення включає в себе велику кількість інструментів, що покривають різні аспекти тестування. У табл. 1.1 наведено аналіз найбільш поширених інструментів.

Таблиця 1.1 - Аналіз інструментів для автоматизованого тестування програмного забезпечення.

Назва інструменту/Джерело	Тип	Мова	Опис
1	2	3	4
JUnit https://junit.org/junit5/	Фреймворк для юніт-тестування	Java	Є стандартом де-факто для юніт-тестування в Java.
Selenium https://www.selenium.dev/	Інструмент для автоматизації веб-браузерів.	Java, C#, Python	Дозволяє автоматизувати дії користувача в браузері для тестування веб-додатків. Має багатий API і підтримує більшість сучасних браузерів.
TestNG https://testng.org/doc/	Фреймворк для тестування	Java	Альтернатива JUnit з додатковими можливостями, такими як параметризоване тестування, групування тестів.

Продовження таблиці 1.1 - Аналіз інструментів для автоматизованого тестування програмного забезпечення.

1	2	3	4
QUnit https://qunitjs.com/	Фреймворк для юніт-тестування JavaScript.	JavaScript	Часто використовується для тестування JavaScript-коду в браузері або у середовищі Node.js.
Appium https://appium.io/docs/en/2.1/	Інструмент для автоматизації мобільних додатків.	Багато мов	На основі Selenium та підтримує автоматизацію для Android і IOS додатків.
Jenkins https://www.jenkins.io/	Інструмент для неперервної інтеграції та доставки.	Java	Дозволяє автоматизувати ряд завдань, включаючи збірку, тестування та розгортання додатків.
Cucumber https://cucumber.io/	Інструмент для BDD	Ruby Java, .NET …	Дозволяє описувати тестові сценарії на природній мові.
Postman https://www.postman.com/	Інструмент для тестування API	JavaScript	Зручний для виконання запитів
JMeter https://jmeter.apache.org/	Інструмент для тестування API	Java	Інструмент для тестування навантаження і виконання запитів.

Всі ці інструменти мають свої унікальні переваги і можуть бути використані в залежності від конкретних потреб проекту. Наприклад, Selenium та Appium відмінно підходять для автоматизації функціональних тестів веб і мобільних додатків відповідно, в той час як JMeter спрямований на тестування продуктивності та навантаження.

1.8 Огляд найбільш поширених мов програмування для написання автотестів

Для написання автотестів існує чимало мов програмування, які мають різні призначення. У таблиці 1.2 зроблено огляд мов, популярних серед фахівців у автоматизації.

Таблиця 1.2 - Дослідження мов програмування для автоматичного тестування програмного забезпечення.

Мова/Джерело	Інструменти	Переваги	Недоліки
1	2	3	4
Java https://www.java.com/	Selenium, JMeter, TestNG, JUnit.	Багато ресурсів для навчання, підтримка багатьох бібліотек і фреймворків для тестування.	Відносно велика вимога до ресурсів системи; може бути складніше для вивчення порівняно з іншими мовами.
Python https://www.python.org/	PyTest, Robot Framework, Unittest, Selenium з Python API.	Синтаксис є інтуїтивно зрозумілим, має багатий набір бібліотек.	Може бути менш продуктивним в деяких випадках порівняно з компільованими мовами, такими як Java або C#.

Продовження таблиці 1.2 - Дослідження мов програмування для автоматичного тестування програмного забезпечення.

1	2	3	4
C# https://learn.microsoft.com/en-us/dotnet/csharp/	NUnit, MSTest, SpecFlow, Selenium з C# API.	Інтеграція з Visual Studio, підтримка Microsoft, зручно для команд, які вже використовують .NET стек.	В основному зосереджений на екосистемі Microsoft.
JavaScript https://www.javascript.com/	Mocha, Jasmine, Jest, Protractor, Cypress.	Дозволяє тестувати веб-додатки в тому ж оточенні, в якому вони працюють; широка підтримка фронтенд-технологій.	Може бути менш стабільним і повільним порівняно з серверними мовами програмування.

1.9 Постановка задачі

В рамках данної задачі необхідно:

- Проаналізувати існуючі інструменти і мови програмування.
- Визначитись зі стеком технологій для забезпечення ефективного процесу розробки програмного продукту.
- Проаналізувати існуючі методи та інструменти для визначення покриття коду автотестами.
- Розробити метод та алгоритм для автоматичного підрахунку покриття API автотестами, що враховує специфіку роботи з різними API.

- Створити програмний інструмент на мові Python, що реалізує розроблений алгоритм.
- Провести випробування розробленого інструменту на реальних проектах з метою визначення його ефективності та точності.
- Зіставити результати роботи розробленого інструменту з аналогічними інструментами, що існують на ринку, для об'єктивної оцінки його переваг та недоліків.

Висновки до першого розділу

Протягом дослідження було проаналізовано різні аспекти тестування, а саме стратегії автоматизації, методології розробки, вимоги до покриття коду тестами та практичні аспекти впровадження автоматизованих тестів. Виявлено, що наявні підходи до автоматизації вимагають значних ресурсів, особливо у контексті покриття ендпоінтів API. Тому доцільним є розробка методу та алгоритму для оптимізації процесу автоматизації тестування та відповідного інструменту, для ефективної перевірки API.

Для автоматизації тестування покриття ендпоінтів автотестами було обрано Jenkins, JMeter та Python. Вибір ґрунтується на потребі проекту та можливостях цих інструментів. Використання цього стеку технологій сприяє швидкій адаптації до змін у вимогах, оптимізації процесу тестування та зниженню ризиків, пов'язаних з автоматизацією на проекті.

Переваги обраних інструментів включають, але не обмежуються наступним:
Jenkins.

- Автоматизує процеси збірки, тестування та розгортання, що сприяє швидкому виявленню проблем та зменшує витрати часу на їх вирішення.
- Неперервна інтеграція та доставка дозволяють зменшити ризики, пов'язані з ручними втручаннями та забезпечують стабільність.
- Має плагіни для багатьох інструментів, включаючи JMeter і Python, що спрощує налаштування.
- Підтримка різноманітних сценаріїв автоматизації завдяки широкому спектру плагінів.

- Велика спільнота, що надає підтримку та розробляє нові функції, що сприяє постійному вдосконаленню інструменту.

JMeter.

- Дозволяє виконувати API запити та має велику кількість плагінів, що може виконувати запити у БД, конектитись по ssh і віддалено виконувати команди на інших ПК, писати і виконувати скрипти парсити відповіді різних форматів.
- Гнучкість у створенні тестових сценаріїв.
- Має відкритий код, що дає можливість модифікації та адаптації інструменту до специфічних вимог.

Python:

- Простий синтаксис, що полегшує навчання та розробку автоматизованих тестів.
- Поширен серед тестувальників.
- Має великий вибір бібліотек для автоматизованого тестування, що забезпечує гнучкість у виборі підходів та методик.
- Надає широкі можливості для розробки та інтеграції з іншими інструментами та платформами.

Завдяки вищезазначеним перевагам розробка автотестів на Python може бути швидкою та ефективною.

Завершуючи, важливо підкреслити, що успішна автоматизація тестування вимагає не лише вибору відповідних інструментів, але й стратегічного підходу, який враховує специфіку проекту, його архітектуру та бізнес-вимоги. Такий підхід забезпечить оптимальне використання ресурсів, підвищення якості продукту та ефективність процесів розробки.

2 ОБҐРУНТУВАННЯ НЕОБХІДНОСТІ І МЕТОДУ ВИРІШЕННЯ ПРОБЛЕМИ РОЗРАХУНКУ ПОКРИТТЯ API ЕНДПОІНТІВ АВТОТЕСТАМИ

2.1 Метрика покриття ендпоінтів API автотестами і її підрахунок

Серед існуючих метрик для якості процесу розробки програмного забезпечення є важливий показник - покриття ендпоінтів API автотестами.

Ця метрика вимірює, який відсоток ендпоінтів (або методів API) покрито автоматизованими тестами, порівняно з загальною кількістю доступних ендпоінтів. Показник покриття обраховується за наступним алгоритмом.

- Ідентифікація усіх ендпоінтів.

  Потрібно визначити перелік всіх ендпоінтів API. Це всі HTTP-запиті, такі як GET, POST, DELETE, PATCH, PUT тощо, які задокументовані в swagger специфікації.

- Ідентифікація покритих ендпоінтів.

  Потрібно визначити, які з цих ендпоінтів перевіряються автотестами. Для кожного ендпоінта слід з'ясувати, чи існує відповідний тест, який перевіряє його очікувану поведінку та відповідь.

- Обрахунок покриття.

  Покриття ендпоінтів може бути обчислено за формулою:

$$\frac{\text{Кількість ендпоінтів,покритих тестами}}{\text{Загальна кількість ендпоінтів}} * 100$$

Важливо також мати систему неперервної інтеграції, яка регулярно виконує ці тести, і інструменти покриття коду, які можуть автоматично звітувати про рівень покриття тестами ваших ендпоінтів.

2.2 Важливість метрики покриття ендпоінтів API автотестами

Ось декілька ключових причин, чому обчислення покриття ендпоінтів так важливо:

- Оцінка ризиків.

Знання відсотка покриття тестами дозволяє команді розробників оцінити рівень ризику. Непокриті частини коду можуть містити не виявлені помилки, що підвищує ризик випуску продукту з дефектами. Якщо відсоток покриття низький, це сигнал до дії для команди.

- Фокус на проблемні зони.

Метрики покриття можуть допомогти ідентифікувати області, які потребують покращення. Це дає можливість команді приділити більше уваги слабким місцям, покращуючи загальну якість та надійність продукту.

- Обґрунтування ресурсів.

Ці дані можуть бути використані для обґрунтування необхідності додаткових ресурсів або часу на розробку. Наприклад, якщо покриття тестами є недостатнім, може виникнути потреба в додатковому фінансуванні для розширення команди QA або для придбання додаткових інструментів для автоматизації тестів.

- Підтримка високих стандартів.

Встановлення цілей для відсотка покриття може служити якісним маркером для команди та стимулювати підтримку високих стандартів кодування і тестування.

- Впевненість у змінах та рефакторингу.

З високим рівнем покриття тестами, розробники можуть бути впевненішими при внесенні змін або рефакторингу коду. Це зменшує ймовірність негативного впливу на існуючу функціональність.

- Довіра з боку зацікавлених сторін.

Метрики покриття часто служать важливими показниками для стейкхолдерів, що свідчить про якість продукту та професіоналізм команди, що може сприяти підтримці проекту на вищому рівні управління.

Враховуючи ці фактори, важливо не тільки знати відсоток покриття ендпоінтів автотестами, але й розуміти контекст цих метрик. Повне покриття не завжди є практичним або можливим через різні обмеження, тому важливо встановити оптимальний рівень покриття для проекту, враховуючи специфіку та вимоги бізнесу.

2.3 Методи і техніки впровадження процесу автоматизації перевірки API

Визначимо до якого рівня і типу належить автоматизоване тестування API на проекті. Для цього будемо використовувати типізацію яку пропонує ISTQB.

ISTQB (International Software Testing Qualifications Board) - це організація, що встановлює стандарти для сертифікації професійних навичок у сфері тестування

програмного забезпечення. Вона визначає різні рівні тестування, щоб структурувати процеси перевірки якості ПЗ.

2.3.1 Рівні тестування

Існує декілька рівнів тестування. З метою визначення рівня, до якого належить автоматизоване тестування API, розглянемо їх детально.

Компонентне тестування (Component Testing, також відоме як модульне тестування або Unit Testing). Його мета - перевірка функціональності окремих компонентів програми (найчастіше це єдині класи або методи). Воно полягає у проведенні тестування розробниками. Програмисти переконуються, що конкретні частини програми працюють як очікувалося на найбільш базовому рівні. Це може включати мок-тестування, що означає використання фальшивих об'єктів для імітації частин системи.

Розглянемо Інтеграційне тестування (Integration Testing):

Його мета - перевірити, як різні компоненти системи працюють спільно. На цьому етапі тестери перевіряють, чи правильно різні модулі або служби спілкуються один з одним, і чи правильно вони обмінюються даними. Це може виявити проблеми в інтерфейсах та взаємодіях.

Розглянемо Системне тестування (System Testing).

Його мета - перевірка поведінки цілої системи в умовах, що максимально наближені до реальних. Тестери оцінюють, чи відповідає система проектним вимогам та очікуванням користувачів, а також чи правильно вона функціонує в цілому. Тести можуть включати стрес-тестування, тестування безпеки, тестування продуктивності та інші.

Розглянемо Приймальне тестування (Acceptance Testing):

Його мета - визначити, чи готовий продукт до використання кінцевими користувачами. На цьому етапі реальні користувачі (або замовники) перевіряють програму, щоб переконатися, що вона відповідає їхнім потребам і вимогам. Якщо продукт не проходить приймальне тестування, він може бути відправлений назад на доопрацювання.

Кожен рівень тестування має свої особливості, цілі та завдання. Процес тестування зазвичай передбачає проведення декількох циклів тестів на різних рівнях, щоб максимально забезпечити якість продукту.

З опису рівнів тестування можемо визначити що автоматизоване тестування API переважно відноситься до рівня інтеграційного тестування. Це пов'язано з тим, що основна мета такого тестування - перевірка взаємодії та правильності обміну даними між різними компонентами системи. На відміну від компонентного тестування, яке зосереджене на перевірці окремих модулів або класів, автоматизоване тестування API займається перевіркою взаємодії цих модулів. Воно відрізняється від системного тестування, оскільки не оцінює поведінку цілої системи в комплексі, а фокусується на конкретних інтерфейсах і взаємодіях між компонентами. Але незважаючи на свою специфіку, автоматизоване тестування API може інтегруватися з іншими рівнями тестування, такими як компонентне, системне та приймальне тестування, для забезпечення всебічної якості та надійності програмного продукту.

2.3.2 Типи тестування

Тестування поділяється на наступні види:

- Функціональне тестування.

Мета: переконатися, що система відповідає визначеним функціональним вимогам.

Опис: Перевірка конкретних функцій програмного забезпечення шляхом їх запуску з визначеними вихідними даними та аналізу результатів відповідно до очікуваних.

- Нефункціональне тестування.

Мета: перевірка аспектів програмного забезпечення, що не пов'язані з конкретними поведінками або функціями, наприклад, продуктивність, масштабованість, безпека.

Опис: Тестування, спрямоване на визначення, чи відповідає система певним критеріям продуктивності, безпеки, доступності, локалізації тощо.

- Тестування на зміни (Regression Testing).

Мета: переконатися, що новий код, що був доданий у систему, не порушив існуючу функціональність.

Опис: Систематичне тестування для виявлення дефектів в уже тестованому коді після внесення змін.

- Тестування сумісності (Compatibility Testing).

Мета: переконатися, що програмне забезпечення сумісне з іншими системами, додатками або середовищами.

Опис: Тестування виконується в різних середовищах, включаючи різні версії операційних систем, браузерів, мережевих умов тощо.

- Тестування користувацького інтерфейсу (Usability Testing).

Мета: визначення зручності та ефективності системи в контексті кінцевого користувача.

Опис: Проводиться для перевірки, наскільки інтуїтивним, зручним та легким у використанні є програмний продукт.

- Тестування безпеки (Security Testing).

Мета: виявлення потенційних вразливостей системи, що можуть призвести до втрати даних або інших атак.

Опис: Цей процес включає симуляцію атак, аналіз правил доступу, пошук вразливостей, які можуть бути використані зловмисниками.

- Тестування продуктивності (Performance Testing).

Мета: визначення швидкодії, стабільності та ефективності системи під навантаженням.

Опис: Зазвичай включає стрес-тестування, тестування навантаження, тестування стабільності тощо, щоб переконатися, що програма працює стабільно під навантаженням та працює ефективно.

- Тестування встановлення (Installation Testing):

Мета: забезпечення того, що система встановлюється та працює належним чином в різних середовищах.

Опис: Перевіряє, чи встановлюється програмне забезпечення без помилок, і чи воно готове до використання після встановлення.

Ці типи тестування можуть перекриватися або комбінуватися в залежності від специфіки проекту та вимог до якості програмного забезпечення. Також вони можуть використовуватися на різних етапах розробки продукту.

На проекті де впроваджується процес автоматизації тестування, перевірку API можна класифікувати:

– За типом як - Функціональне тестування, оскільки воно перевіряє функції та функціональність, які надаються API. Це включає перевірку правильності відповідей на запити, обробку помилок, інтеграцію з іншими компонентами та виконання очікуваних результатів згідно з вимогами.

– За рівенями тестування-Компонентне (Component Testing), оскільки воно проводиться на рівні окремої частин програмного забезпечення. API на цьому проекті є окремим компонентом, що інтерфейсує з іншими компонентами системи.

2.4 Техніки тестування

Згідно з ISTQB, техніки тестування можна розділити на кілька категорій, кожна з яких слугує різним цілям у процесі забезпечення якості.

Далі розглянемо статичні техніки тестування.

2.4.1 Статичне тестування

Данна техніка тестування не вимагає виконання програмного коду, замість цього воно фокусується на аналізі документації, включаючи артефакти, такі як вимоги, специфікації, дизайн, і, звісно, сам код.

Основні методи:

– Перегляди (формальні, неформальні, код-ревью).

– Статичний аналіз коду (зазвичай за допомогою спеціалізованого програмного забезпечення).

2.4.2 Специфікаційно-орієнтовані техніки (чорний ящик)

Ці техніки базуються на вимогах та функціональності, не залежачи від внутрішньої структури системи.

Серед них:

– Тестування на основі еквівалентності (поділ вхідних даних та умов на класи еквівалентності, для яких стан системи має бути однаковим).

– Тестування межових значень (фокус на значеннях на краях допустимого діапазону).

– Тестування використання (тести засновані на сценаріях використання системи).

– Тестування державних переходів (перевірка правильності переходів між різними станами системи).

2.4.3 Структурно-орієнтовані техніки (білий ящик)

Використовуються для тестування внутрішньої структури продукту. Основні методи:

– Тестування покриття коду (інструкції, гілки/умови, шляхи тощо).
– Тестування потоку керування (перевірка логічних умов і шляхів керування у програмному забезпеченні).

2.4.4 Техніки на основі досвіду

Вони засновані на досвіді та інтуїції тестувальників і часто використовуються, коли існує недостатньо документації або для доповнення інших типів тестування.

Включають:

– Експлораторне тестування (неформальний, гнучкий процес, який залучає одночасне навчання, дизайн тесту та виконання тестів).
– Тестування за атакуючими сценаріями (фокусується на "зламі" або виявленні слабких місць системи).
– Тестування помилок (симуляція відомих проблем для перевірки, чи не відтворюється подібна поведінка).

2.4.5 Техніки тестування на основі моделей

Тестування, яке використовує абстрактні моделі системи для генерації тестових сценаріїв та випадків.

Це може включати:

– Тестування на основі станів (моделювання системи у вигляді станів та переходів, що дозволяє генерувати тести для перевірки).
– Тестування на основі усього покриття (аспектів моделі, наприклад, поведінки системи).

Всі ці техніки можуть перекриватися та доповнювати одна одну в залежності від специфіки проекту, доступних ресурсів, часових рамок, а також від рівня

критичності програмного забезпечення. Вони слугують для підтримки різних аспектів якості, таких як надійність, продуктивність, безпека та інші.

2.5 Метрики для оцінки якості і швидкісті процеса тестування

Визначимо метрики, які будемо використовувати для оцінки покращення якості і прискорення швидкісті процеса тестування.

Метрики тестування використовуються для кількісної оцінки процесів та результатів тестування. Вони допомагають командам вимірювати ефективність тестування, якість продукту, продуктивність роботи та інші важливі аспекти. Згідно з ISTQB [8], деякі ключові метрики тестування:

– Rate of Testing Coverage (Ступінь покриття тестуванням):

Опис: Вимірює відсоток частин програмного забезпечення (наприклад, модулів, функцій, рядків коду), які були протестовані. Це допомагає зрозуміти, наскільки повно команда перевірила продукт.

Як це використовується: Для планування додаткових тестів, якщо покриття занизьке, та для оцінки загального ризику.

– Defect Density (Щільність дефектів):

Опис: Вимірює кількість дефектів відносно певної міри (наприклад, кількість дефектів на 1,000 рядків коду). Це дозволяє оцінити якість продукту та ефективність тестування.

Як це використовується: Для ідентифікації ділянок коду з високою концентрацією дефектів та потенційного вдосконалення процесів розробки та тестування.

– Defect Discovery Rate (Темп виявлення дефектів):

Опис: Вимірює швидкість, з якою команда виявляє дефекти протягом певного періоду часу. Це може допомогти визначити, коли процес тестування стає менш продуктивним.

Як це використовується: Для оцінки потреби в додаткових ресурсах або часі для тестування та як індикатор готовності продукту до випуску.

– Defect Resolution Time (Час вирішення дефектів):

Опис: Середній час між виявленням дефекту та його вирішенням. Ця метрика вказує на ефективність процесу управління дефектами.

Як це використовується: Для планування циклів тестування та випуску, а також для виявлення можливих затримок в процесі розробки.

– Percentage of Defects Removed (Відсоток усунутих дефектів):

Опис: Відсоток виявлених дефектів, які були успішно усунені перед випуском продукту. Це допомагає оцінити ефективність процесів усунення дефектів.

Як це використовується: Для оцінки готовності продукту до випуску та якості роботи команди управління дефектами.

– Testing Effectiveness (Ефективність тестування):

Опис: Відношення кількості виявлених дефектів до загальної трудомісткості процесу тестування.

Як це використовується: Для оцінки якості стратегії тестування та її здатності виявляти дефекти.

Кожна з цих метрик відіграє важливу роль у забезпеченні якості програмного забезпечення. Вони допомагають командам розуміти поточний стан проекту, приймати обґрунтовані рішення щодо подальших дій та покращувати процеси розробки та тестування.

2.6 Опис методу вирішення задачі розрахунку покриття API ендпоінтів автотестами

Для вирішення задачі розрахунку покриття API ендпоінтів автотестами пропонується метод, що грунтується на використанні технологій (вписати назви) та таких інструментів як (вписати назви).

Метод включає у себе такі складові:

1 Створення опису архітектури API за допомогою Swagger:

Використання Swagger для документування архітектури API, включаючи детальний опис ендпоінтів, параметрів запиту, методів HTTP і відповідей.

2 Автоматизація тестування за допомогою JMeter.

Налаштування JMeter для автоматичного виклику HTTP-запитів, визначених у Swagger документації, та перевірки відповідей.

3 Розробка Python-скрипту для аналізу даних, який має такий функціонал:

– Отримання даних з Swagger специфікації та JMeter .jmx файлів.

- Очищення, конвертація та порівняння даних між JMeter колекціями та Swagger документацією.
- Розрахунок відсотку покриття ендпоінтів, базуючись на отриманих даних.
- Автоматичне створення звітів, які деталізовано відображають рівень покриття, вказуючи на покриті та непокриті ендпоінти.

4 Неперервна інтеграція з Jenkins:

- Конфігурація Jenkins для автоматичного запуску тестів на регулярній основі.
- Інтеграція системи звітів з Jenkins для автоматичного сповіщення команди про статус покриття та потенційні проблеми у програмному забезпеченні.

Практичне значення:

Цей метод представляє собою нове рішення в галузі автоматизованого тестування API. Ось деякі аспекти, які це підтверджують:

- Унікальність.

На даний час ринок не пропонує прямих аналогів даного рішення, що робить розроблений метод новаторським у цій області.

- Адаптивність.

Python-скрипт, розроблений для аналізу, конвертації, та порівняння даних, можна легко адаптувати (під легкістю адаптації будемо розуміти легкість внесення змін) під конкретні вимоги або умови, що забезпечує гнучкість загального підходу.

- Доступність.

Використання Python сприяє легкості розробки та підтримки, оскільки ця мова програмування є широко поширеною серед тестувальників програмного забезпечення та має велику спільноту, що сприяє швидкому вирішенню потенційних проблем.

- Автоматизація та інтеграція.

Комбінація автоматизованих тестів з JMeter, скриптів Python для аналізу даних, та системи неперервної інтеграції Jenkins створює комплексне рішення, яке значне прискорює процеси тестування та підвищує надійність програмного забезпечення.

Висновки до другого розділу

У цьому розділі були досліджені та оцінені методи та техніки автоматизації процесу перевірки API і були зроблені висновки що:

1. Автоматизоване тестування API на проекті є специфікаційно-орієнтовані (чорний ящик) технікою тестування.Тобто зосереджено на перевірці відповідності поведінки API очікуваним результатам, визначеним у специфікації API.

2. Автоматичний підрахунок покриття API автотестами відноситься до структурно-орієнтованих технік тестування (білий ящик). Специфічно, це частина техніки аналізу покриття коду, яка фокусується на вимірюванні того, яка частина коду програми виконується під час автоматизованого тестування і включає покриття операторів (Statement coverage). Міряє відсоток виконаних операторів коду. Це найбільш базовий рівень покриття коду, який дозволяє зрозуміти, які рядки коду були виконані під час тесту. Такий рівень покриття на проекті, для якого впроваджується автоматичне тестування API, був визнаний, з боку менеджмента і споживачів, як достатній.

Також були вибрані метрики для аналізу якості та ефективності тестування:

1. Темп виявлення дефектів Ця метрика допомагає визначити, наскільки швидко виявляються помилки під час розробки продукту. Знаходження багів і помилок на ранніх етапах, дозволяє зменшуючи загальні затрати на їх виправлення і час на випуск продукту.

2. Ефективність тестування Ця метрика враховує обсяг витрачених зусиль на проведення тестування відносно кількість виявлених дефектів. Допомагаючи зрозуміти, наскільки ефективними є обрана стратегія тестування. Ця метрика допоможе нам оцінити як вплинула автоматізація процесу тестування API на процес розробки, як змінилась якість продукту та скоротився час його виходу на ринок. На підставі проведеного аналізу було запропоновано метод, що відповідає поточним вимогам тестування API і створює основу для майбутніх наукових розробок у цьому напрямку.

На підставі проведеного аналізу було запропоновано метод, що відповідає поточним вимогам тестування API і створює основу для майбутніх наукових розробок у цьому напрямку.

3 ВПРОВАДЖЕННЯ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ І РОЗРОБКА ІНСТРУМЕНТУ ДЛЯ ПІДРАХУНКУ ПОКРИТТЯ API АВТОТЕСТАМИ

3.1 Очікувані результати від впровадження автоматизованого тестування

Впровадження автоматизованого тестування може вимагати певних витрат на початкових етапах для розробки тестів та налаштування інфраструктури, але очікувані довгострокові переваги зазвичай перевищують ці витрати. До таких переваг можна віднести:

- Зниження кількості помилок.

  Оскільки тести виконуються автоматично, знижується ризик пропустити помилки через людський фактор.
- Скорочення часу на ринок.

  Швидкість тестування та можливість частішого випуску оновлень скорочує час, необхідний для виведення продукту на ринок.
- Економія витрат.

  На довгостроковій основі, автоматизація може зменшити витрати на тестування, оскільки знижуються витрати на трудові ресурси та час.
- Підвищення якості продукту.

  Регулярне та консистентне тестування забезпечує вищий рівень якості продукту та зменшує ймовірність випуску продукту з багами.

3.2 Візуалізація компонентів та зв'язків у процесі автоматизованого тестування

Процес автоматизованого тестування охоплює численні інструменти, платформи та практики. На рисунку 3.1 представимо схематичне зображення, яке допоможе уявити структуру та зв'язки між різними компонентами системи.

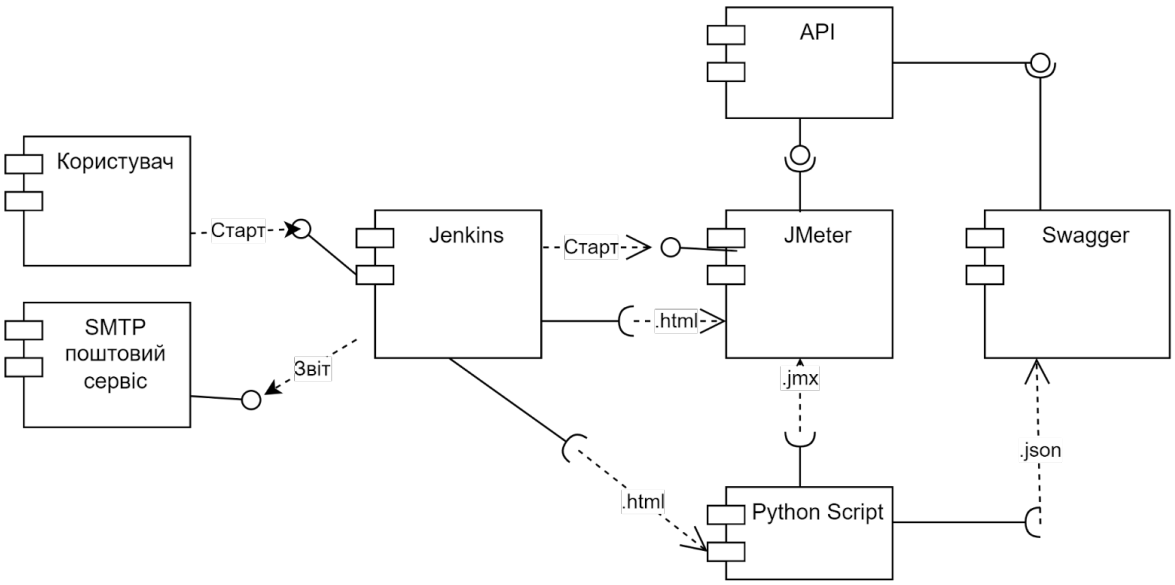

Рисунок 3.1- Схематичне зображення компонентів та зв'язків у процесі автоматизованого тестування.

Розглянемо кожен компонент докладніше, а також визначимо важливість та функції, які вони виконують у загальній схемі автоматизованого тестування.

Користувач.

Ініціалізація автоматизованого процесу тестування, інтерпретація результатів та формулювання висновків.

SMTP поштовий сервіс.

Забезпечення зв'язку з користувачем-інформування користувача про результати тестування або про будь-які проблеми, що виникли.

Jenkins.

- Деплоймент та конфігурація середовища;
- Ініціація JMeter;
- Агрегація результатів від JMeter та Python Script;
- Відправка звіту користувачу.

JMeter.

- Збереження набору запитів до API у форматі .jmx;
- Реалізація тестових сценаріїв шляхом відправки запитів до API та обробки відповідей;

- Формування детальних звітів.

API.

Представлення інтерфейсу для автоматизованого тестування функціональних можливостей.

Swagger:

- Документування API, формування специфікації у форматі .json;
- Надання інтерфейсу для експериментальних запитів.

Python Script:

- Розробленний аналітичний інструмент для порівняння та аналізу;
- Аналіз співставлення між специфікацією Swagger та набором запитів.

JMeter

- Обчислення ступеню покриття ендпоінтів;
- Формування комплексних звітів.

Більш детальній обзор огляд кожного компоненту зробимо далі.

3.2.1 Налаштування та робота з Jenkins

Розглянемо процес встановлення і конфігурації Jenkins для автоматичного запуску тестів. Він включає наступні етапи:

1. Встановлення Jenkins.

Скачуємо з офіційного джерела https://www.jenkins.io/

Встановлюємо згідно документації https://www.jenkins.io/doc/book/installing/windows/

2. Встановлення необхідних плагінів

Через "Manage Jenkins" > "Manage Plugins" встановлюємо:

- Ant - Додає підтримку Apache Ant до Jenkins;
- bouncycastle API -Забезпечує стабільний API для завдань, пов'язаних з Bouncy Castle;
- Build Timeout-Дозволяє автоматично завершувати збірки після вказаного часу;
- Build With Parameters-Дозволяє користувачу вказувати параметри для збірки в URL (аналогічно до /job/JOBNAME/buildWithParameters), запитуючи підтвердження перед тригерингом завдання;

- Command Agent Launcher-Дозволяє запускати агентів за допомогою вказаної команди;
- EnvInject API - Зберігає спільну логіку для управління введенням змінних середовища в Jenkins. Використовується в плагінах, як наприклад EnvInject Plugin;
- HTML Publisher- Цей плагін публікує HTML звіти;
- Managed Scripts- Цей плагін дозволяє централізовано управляти shell-скриптами та посилатися на них як на кроки збірки у збірках;
- Matrix Authorization Strategy - Пропонує стратегії авторизації на основі матриці (глобальні та по проекту);
- Parameterized Trigger - Цей плагін дозволяє тригерити нові збірки, коли ваша збірка завершилася, з різними способами вказівки параметрів для нової збірки;
- SSH Build Agents-Дозволяє запускати агентів через SSH, використовуючи Java реалізацію протоколу SSH;
- SSH Credentials- Дозволяє зберігати дані для автентифікації SSH у Jenkins;
- WMI Windows Agents-Дозволяє налаштувати агентів на Windows машинах через Windows Management Instrumentation (WMI) ;
- Workspace Cleanup- Цей плагін видаляє робочий простір проекту, коли викликається.

3. Підготовка середовища

Для роботи автотестів знадобляться чотири віртуальні машини.

За допомогою VMware створюємо віртуальні машини і встановлюємо Jenkins агентів на наступні системи:

-CentOS7 - для сервера;

-Windows -для Windows агента;

-MacOS- для Mac агента;

-CentOS7 - для Linux агента.

Jenkins агентів встановлюємо за інструкцією з офіціального джерела: https://www.jenkins.io/doc/book/using/using-agents/

4. Створення Jenkins задач.

Для організації нашого процессу автотестування нам потрібно створити чотири задачі:

– Перша - деплоймент та конфігурація середовища

Підіймає в VMware чотири віртуальні машини (одна для сервера і три для встановлення агентів);

– Друга - розгортає серверну частину і встановлює агентів;

– Третя - Запускає тестування ендпоінтів за допомогою JMeter;

– Четверта - Стартує тестування покриття за допомогою Python Script.

Задачі створюємо наступним чином (рисунок 3.2):

-Вибираємо "New Item",

-вводимо назву задачі,

- тиснемо "Freestyle project".

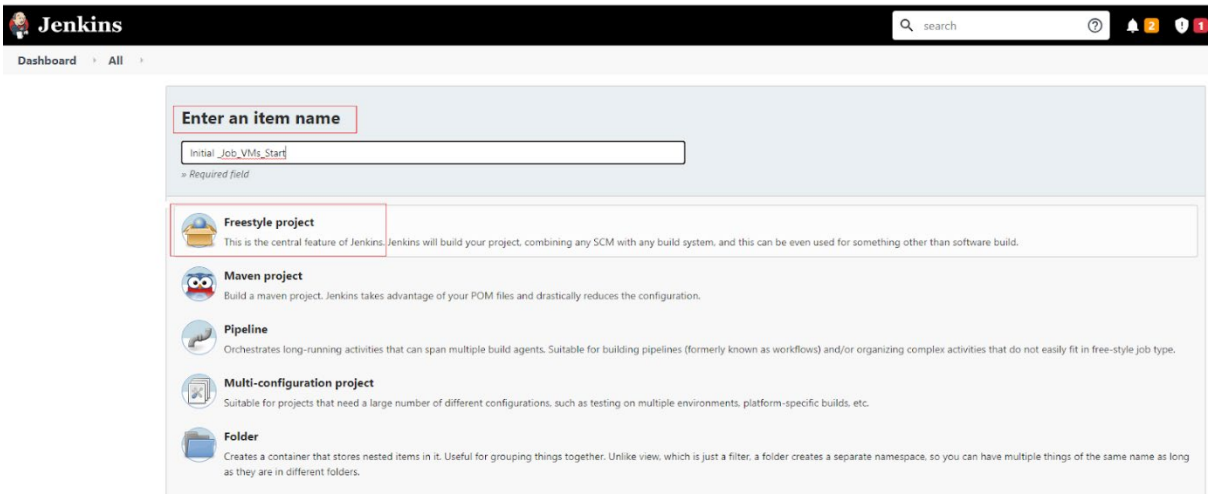

Рисунок 3.2- Створення задачі в Jenkins.

Задача перша.

Розглянемо деплоймент та конфігурація середовища.

Деплоймент та конфігурація середовища це перша задача яку стартує Jenkins. Ця задача за допомогою VMware запускає чотири віртуальні машини з наступними системи:

-CentOS7 - для сервера;

-Windows -для Windows агента;

-MacOS- для Mac агента;

-CentOS7 - для Linux агента і стартує Jenkins агентів на них.

Розглянемо конфігурацію збірки Jenkins, що скадається з кількох секцій:

- General;
- Source Code Management секція - за замовчуванням;
- Build Triggers секція -за замовчуванням;
- Build Environment секція- за замовчуванням;
- Build;
- Post-build Actions.

Параметри General секції:

-BUILD_URL - посилання на сховище де зберігається файли для встановлення серверної частини і агентів.

Запуск проекту на master (рисунок 3.3)

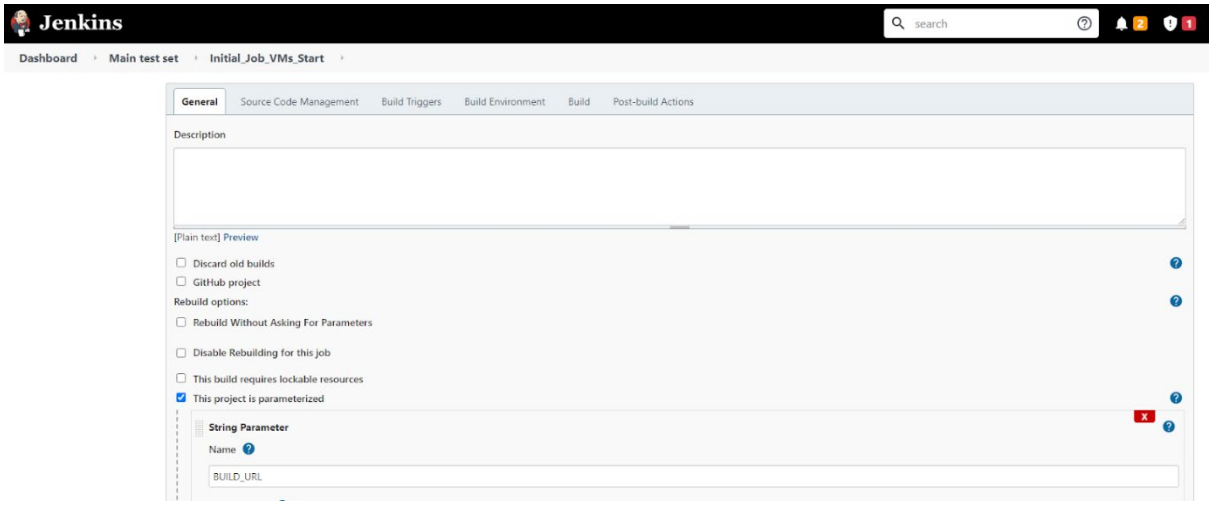

Рисунок 3.3- Запуск проекту на Jenkins.

Параметри Source Code Management секції - за замовчуванням.

Параметри Build Triggers секції -за замовчуванням.

Параметри Build Environment секції- за замовчуванням.

Build секція.

В секції "Build" додамо кроки збірки, щоб виконувались batch команди, необхідні для розгортання середовища (рисунок 3.4):

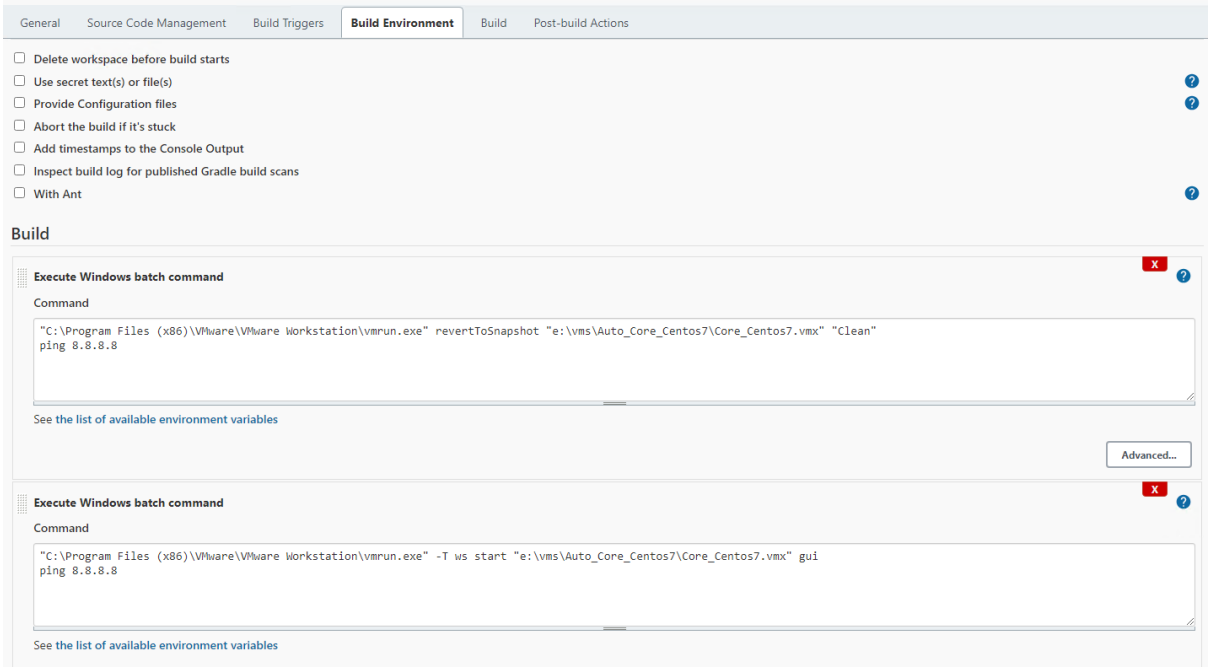

Рисунок 3.4- Build секція, розгортання середовища.

Post-build Actions секція.

В секції Post-build Actions увімкнем опцію запуску параметризованої збірки другої задачі по розгортнню серверної частини і встановлення агентів (рисунок 3.5).

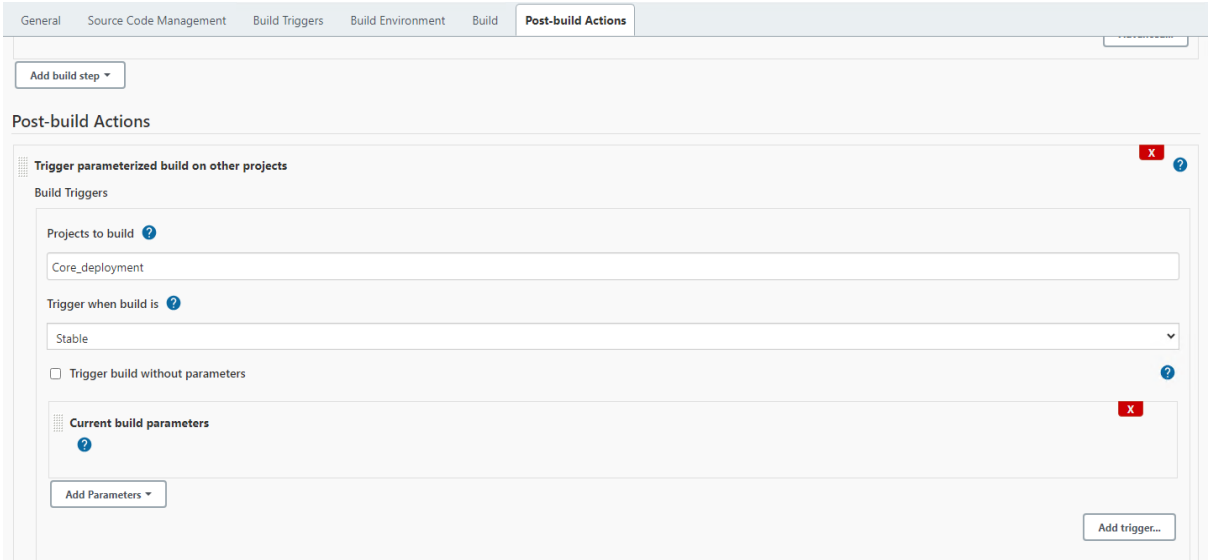

Рисунок 3.5- Post-build Actions секція, серверної частини і встановлення агентів.

Задача друга.

Розгортання серверної частини і встановлення агентів.

Розгортання серверної частини і встановлення агентів це друга задача, яку стартує Jenkins. Ця задача приєднується до віртуальних машин що були запущені попередньою задачею, і за допомогою -ssh з'єднання встановлює агентів і серверну частину програми.

Розглянемо конфігурацію збірки Jenkins, для другої задачі:

- General;
- Source Code Management секція - за замовчуванням;
- Build Triggers секція -за замовчуванням;
- Build Environment секція- за замовчуванням;
- Build;
- Post-build Actions.

Параметри General секції:

-BUILD_URL - посилання на сховище де зберігається файли для встановлення серверної частини і агентів.

-Запуск проекту на віртуальних машинах для серверної частини і агентів.

Параметри Source Code Management секції - за замовчуванням.

Параметри Build Triggers секції -за замовчуванням.

Параметри Build Environment секції- за замовчуванням.

Параметри Build секції.

В секції "Build" запускаємо скрипти для встановлення серверної частини агентів з параметрами:

- BUILD_URL - посилання на сховище де зберігається файли для встановлення серверної частини і агентів;
- USER_LOGIN - логін до сховища де зберігається файли для встановлення серверної частини і агентів;
- USER_PASS - пароль до сховища де зберігається файли для встановлення серверної частини і агентів;

Параметри Post-build Actions секції.

В секції Post-build Actions увімкнем опцію запуску параметризованої збірки задачі по тестуванню покриття ендпоінтів за допомогою JMeter (рисунок 3.6).

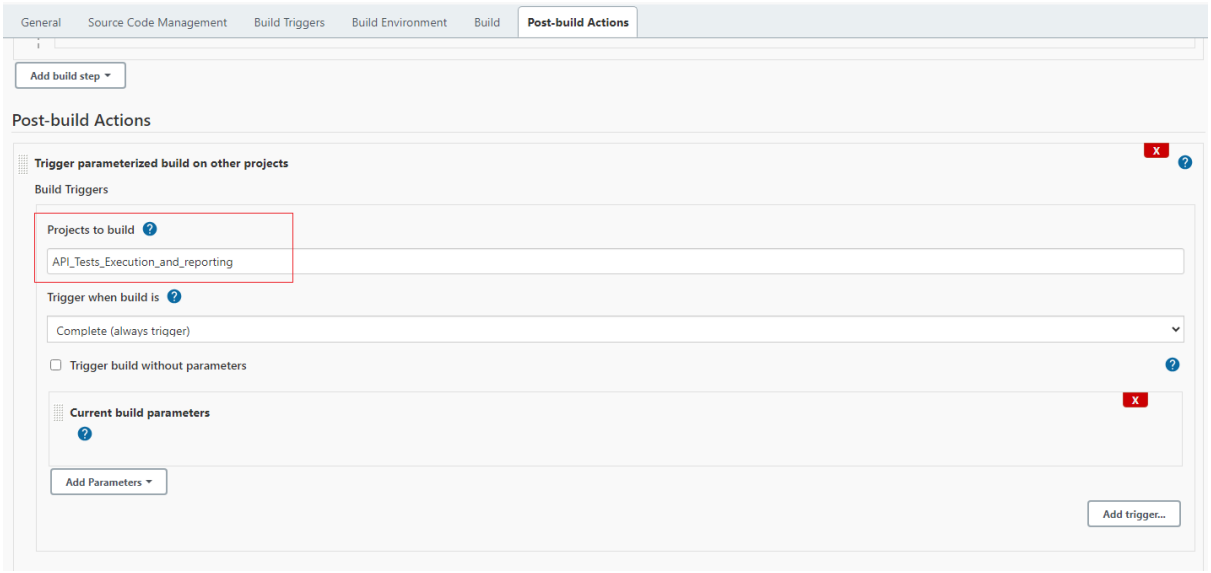

Рисунок 3.6- Post-build Actions секція, запуск параметризованої збірки задачі по тестуванню покриття ендпоінтів за допомогою JMeter.

Задача третя.

Тестування ендпоінтів за допомогою JMeter.

Тестування ендпоінтів за допомогою JMeter це третя задача яку стартує Jenkins. Ця задача запускає JMeter з параметрами для виконання автотестування і генерації звіту.

Розглянемо конфігурацію збірки Jenkins, для третьої задачі:

- General;
- Source Code Management секція - за замовчуванням;
- Build Triggers секція -за замовчуванням;
- Build Environment секція- за замовчуванням;
- Build;
- Post-build Actions.

Параметри General секції:

-BUILD_URL - посилання на сховище де зберігається файли. Ця інформація у даному завданні буде використовуватися для формування назви звіту.

Запуск проекту на master

Параметри Source Code Management секції - за замовчуванням.

Параметри Build Triggers секції -за замовчуванням.

Параметри Build Environment секції- за замовчуванням.

Параметри Build секції.

В секції "Build" запускаємо скрипти для:

- створення директорій для звіту;
- запуску JMeter і сбереження результатів https://www.jenkins.io/doc/book/using/using-jmeter-with-jenkins/;
- копіювання результатів у папку з назвою білда (рисунок 3.7).

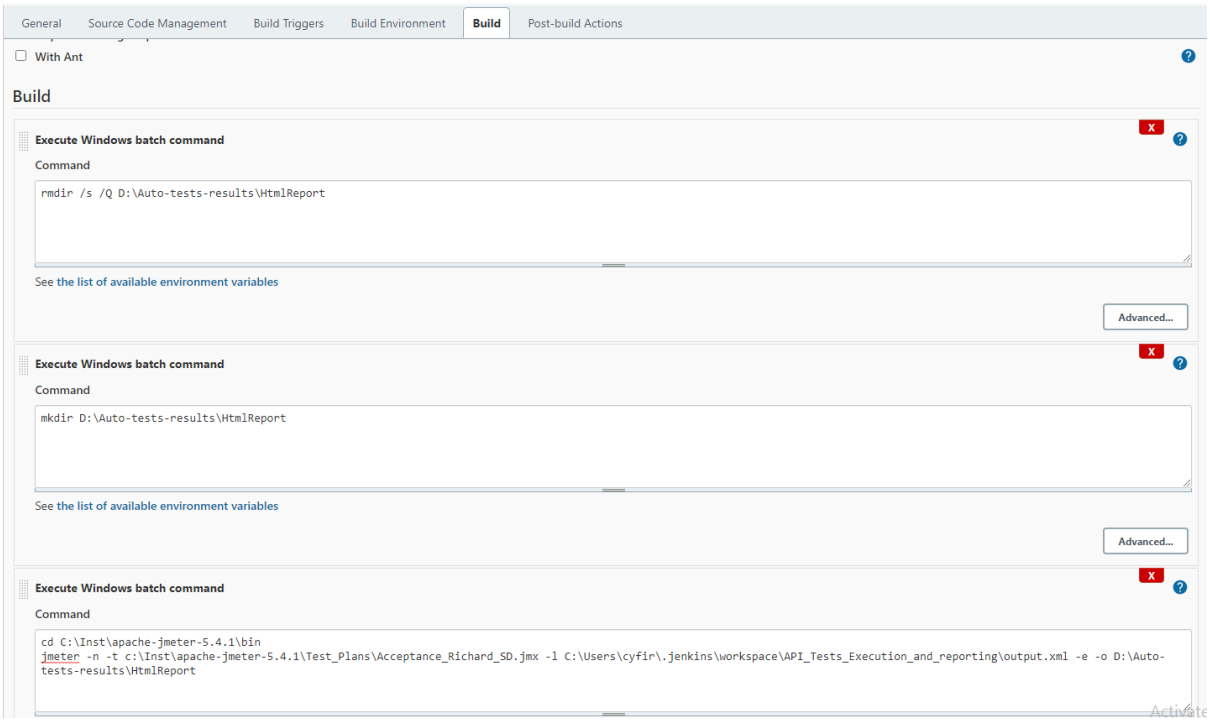

Рисунок 3.7- Build секція, запуску JMeter і сбереження результатів.

Параметри Post-build Actions секції

В секції Post-build Actions увімкнем опцію запуску параметризованої збірки задачі по тестуванню ендпоінтів за допомогою Python Script

Задача четверта.

Тестування покриття за допомогою Python Script.

Тестування ендпокриття за допомогою Python Script - це четверта і остання задача, яку стартує Jenkins. Ця задача запускає скрипт з параметрами і генерує звіт.

Розглянемо конфігурацію збірки Jenkins, для четвертої задачі:

- General;
- Source Code Management секція - за замовчуванням;
- Build Triggers секція -за замовчуванням;
- Build Environment секція- за замовчуванням;
- Build;
- Post-build Actions.

Параметри General секції:

-BUILD_URL - посилання на сховище де зберігається файли. Ця інформація у даному завданні буде використовуватися для формування назви звіту.

Запуск проекту на master.

Параметри Source Code Management секції - за замовчуванням.

Параметри Build Triggers секції -за замовчуванням.

Параметри Build Environment секції- за замовчуванням.

Параметри Build секції.

В секції "Build" запускаємо скрипти для (рисунок 3.8):

- запуску Python Script з параметрами;
- збереження результатів;
- копіювання результатів у папку з назвою білда;

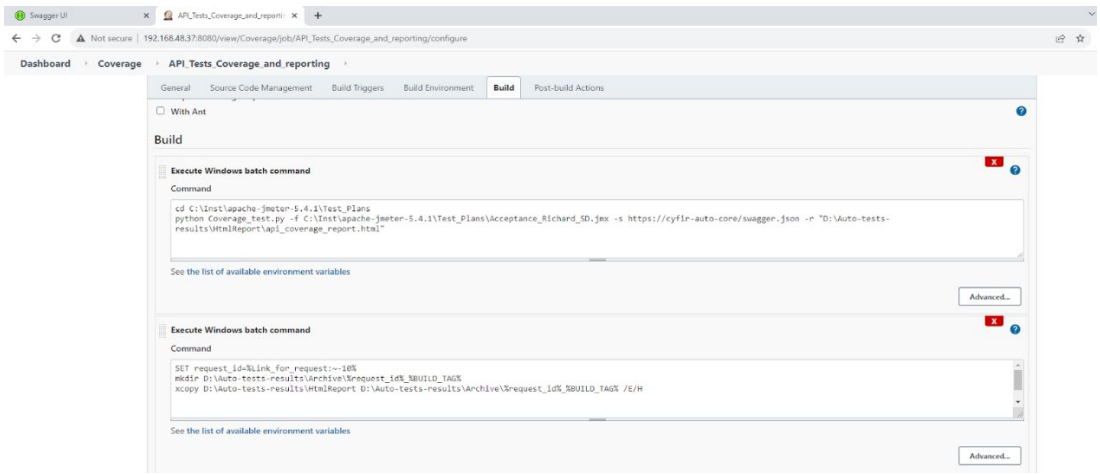

Рисунок 3.8- Build секція, запуск Python Script і збереження результатів.

Параметри Post-build Actions секції.

В секції Post-build Actions збираємо результати від JMeter та Python Script, формуємо та відправляємо звіт (рисунок 3.9).

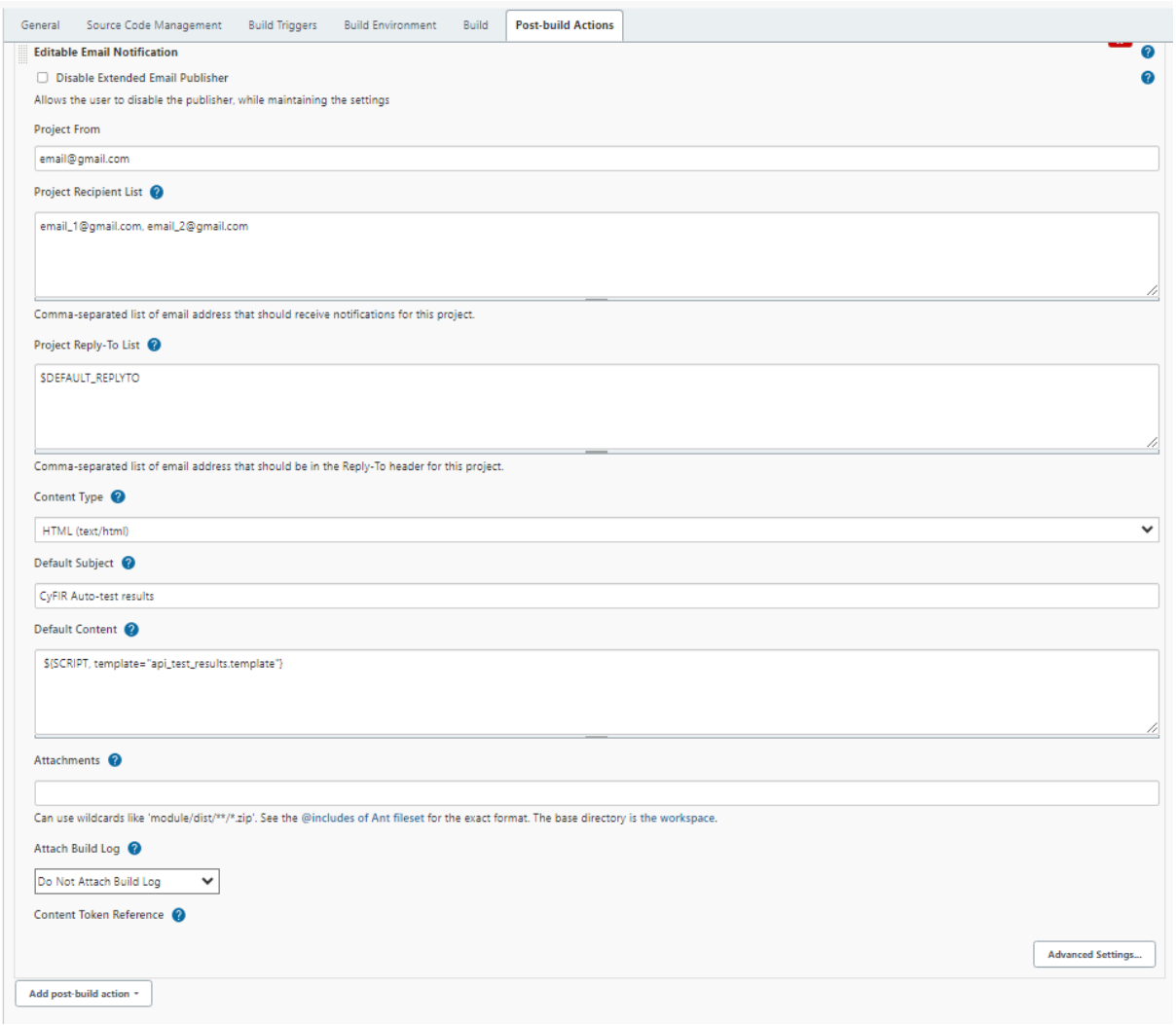

Рисунок 3.9- Post-build Actions, збір результатів від JMeter та Python Script, та формування звіту.

3.2.2 Робота з JMeter для створення автотестів

Детально розглянемо процес встановлення, конфігурації JMeter, створення колекції автотестів їх запуск і а також формування репорту. Він включає слідуючі етапи:

1) Встановлення і конфігурації JMeter.

Покроково опишемо процесс встановлення і конфігурації JMeter.

1) Завантаження JMeter.

Apache JMeter завантажується з офіційного веб-сайту: https://jmeter.apache.org/.

2) Конфігурація JMeter.

Для конфігурації JMeter використовується файл jmeter.properties. Він знаходиться у каталогу JMeter і може бути відредактований у текстовому редакторі. Додатково до налаштувань за замовчуванням встановимо слідуючі параметри:

- jmeter.save.saveservice.output_format=csv
- jmeter.save.saveservice.assertion_results_failure_message=true
- jmeter.save.saveservice.assertion_results=all
- jmeter.save.saveservice.response_code=true
- jmeter.save.saveservice.response_data=true
- jmeter.save.saveservice.response_message=true
- jmeter.save.saveservice.assertions=true
- jmeter.save.saveservice.responseHeaders=true
- jmeter.save.saveservice.requestHeaders=true
- sampleresult.timestamp.start=true
- HTTPResponse.parsers=htmlParser wmlParser cssParser
- cssParser.className=org.apache.jmeter.protocol.http.parser.CssParser
- cssParser.types=text/css
- remote_hosts=<host IP>

3) Встановлення бібліотек.

Встановлення додаткових бібліотек в Apache JMeter є необхідним для розширення його функціональності.

Для встановлення бібліотек з офіційного джерела https://jmeter.apache.org завантажують:

- ApacheJMeter_components.jar - містить компоненти та розширення для JMeter, які можуть бути використані для створення різних типів тестів, таких як HTTP, JDBC, FTP, тощо;
- ApacheJMeter_core.jar: - містить основну функціональність Apache JMeter, включаючи основні класи та інструменти для створення, налаштування та виконання навантажувальних тестів;

- ApacheJMeter_ftp.jar - містить класи та інструменти для роботи з протоколом FTP, що дозволяє виконувати тести FTP-серверів та передачу файлів;
- ApacheJMeter_functions.jar - містить функції та змінні, які можна використовувати у скриптах JMeter для генерації даних та виконання різних операцій;
- ApacheJMeter_http.jar - містить класи та інструменти для роботи з протоколом HTTP, що дозволяє виконувати тести веб-додатків та веб-сервісів;
- ApacheJMeter_java.jar - включає можливості для виконання Java-коду в скриптах JMeter;
- ApacheJMeter_jdbc.jar - містить класи та інструменти для роботи з JDBC (Java Database Connectivity), що дозволяє виконувати тести баз даних;
- ApacheJMeter_native.ja - містить низку класів, що використовуються для нативних компонентів на певних платформах, таких як Windows.
- ApacheJMeter_ssh-1.2.0.jar - дозволяє виконувати тести SSH-з'єднань із віддаленими серверами;
- ApacheJMeter_tcp.jar -містить класи та інструменти для роботи з протоколом TCP, що дозволяє виконувати тести мережевого з'єднання.

Всі завантаженні бібліотеки сберігаємо у каталог, \apache-jmeter-5.4.1\lib\ext

4) Запуск JMeter.

Для Widows за допомогою командного рядка переходимо до каталогу з JMeter і виконуємо jmeter.bat

5) Початок роботи з JMeter

Після запуску JMeter починаємо створювати, редагувати та запускати автотести. Створюємо колекцію запитів в JMeter як https://training.qatestlab.com/blog/technical-articles/using-jmeter-in-testing/

Для створення колекції запитів для JMeter використовуємо Swagger специфікацію.

Swagger (зараз відомий як OpenAPI) – це специфікація для опису RESTful API, яка дозволяє розробникам визначати структуру API, доступні ендпойнти, параметри запитів та відповідей тощо.

Далі розглянемо етапи процесу створення колекції запитів.

a) Ознайомлення з Swagger специфікацією (рисунок 3.10).

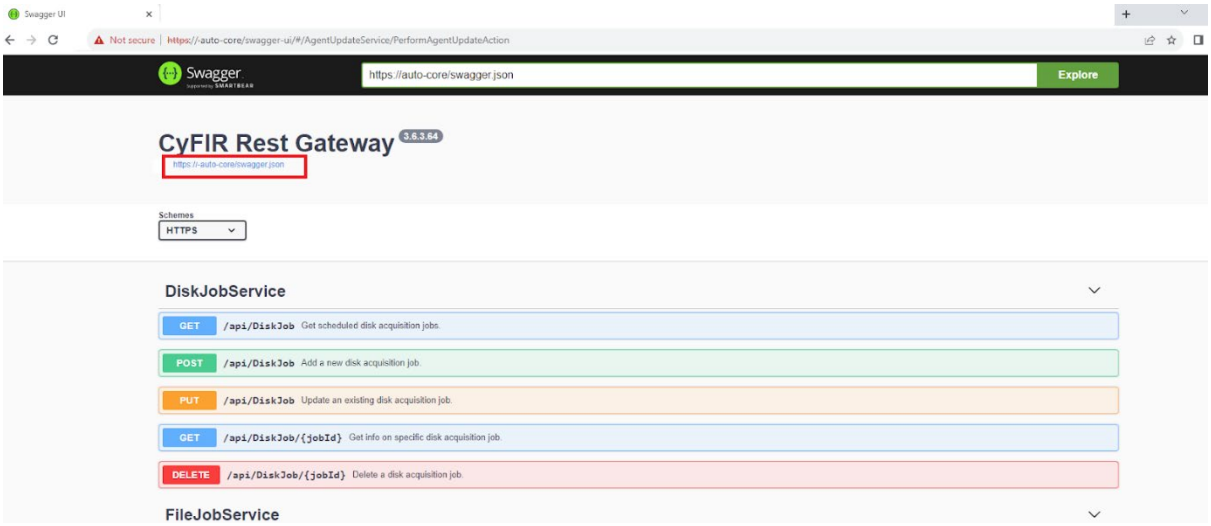

Рисунок 3.10- Swagger специфікація.

Завантажуємо та відкриваємо Swagger специфікацію (форматі JSON) у Swagger UI або будь-якому редакторі для перегляду файлів цих форматів.

Детально опрацьовуємо ендпойнти, методи запитів, параметри тощо.

b) Створення тестового плану.

– В JMeter обираємо опцію "File" > "New" для створення нового тестового плану (рисунок 3.11).

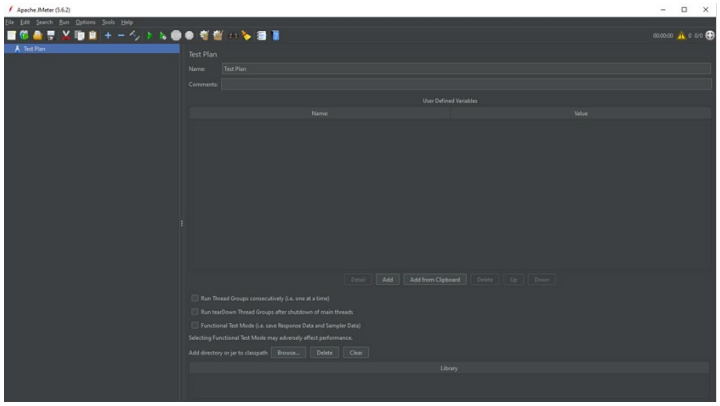

Рисунок 3.11- Створення нового тестового плану.

– Дамо назву плану Test_plan.jmx

– Додаємо Thread Group до вашого тестового плану (рисунок 3.12).

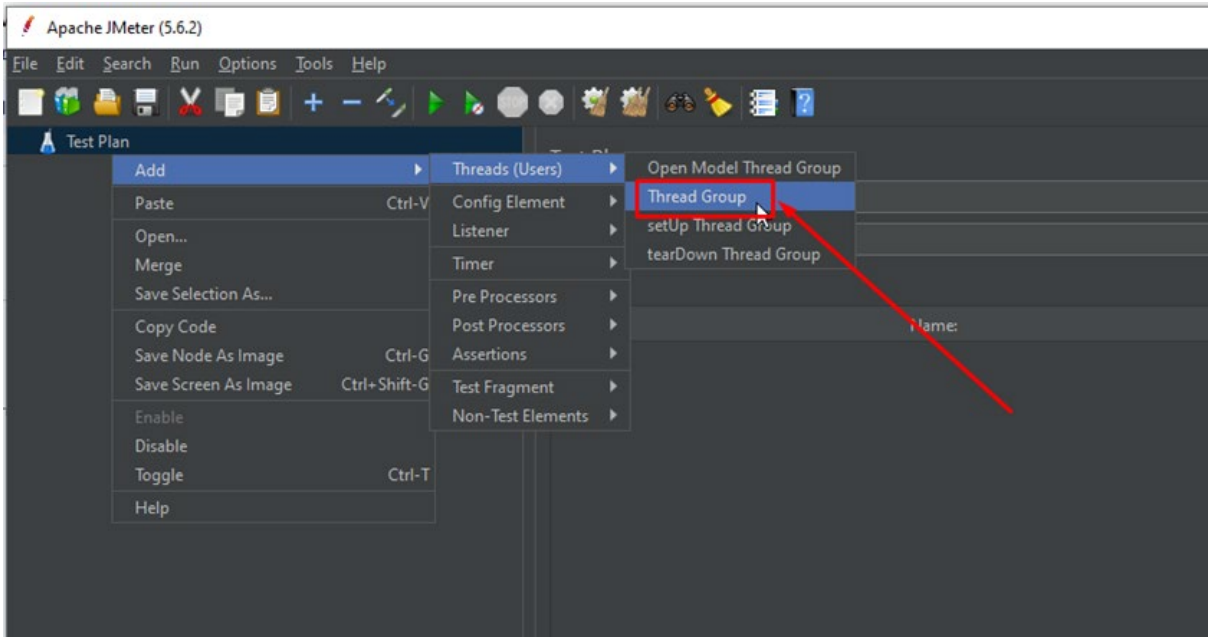

Рисунок 3.12- Створення Thread Group у тестовому плану.

– Додаємо запити до тестового плану (рисунок 3.13):

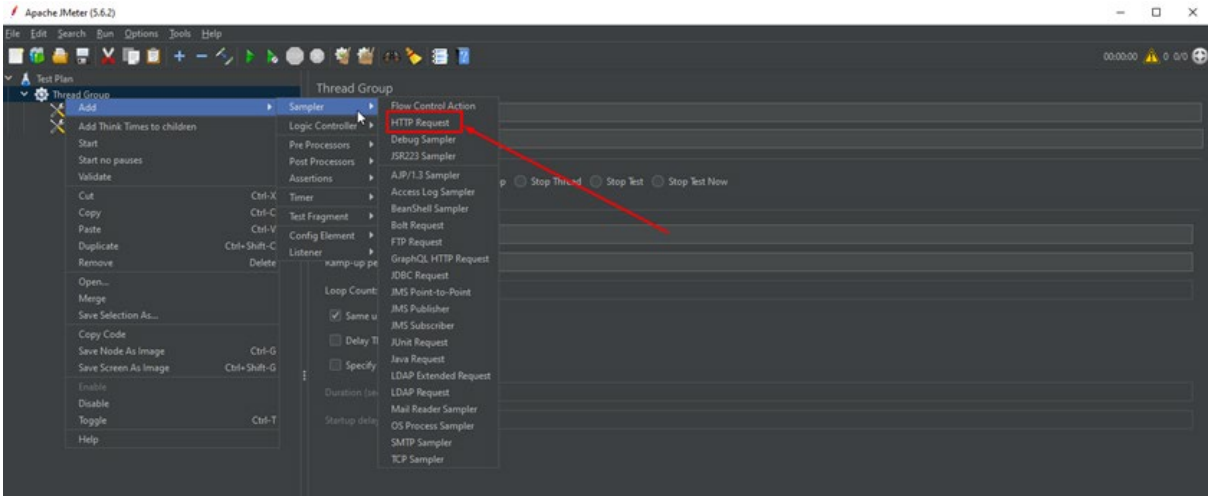

Рисунок 3.13- Додавання запитів до тестового плану.

– На основі Swagger специфікації додайте HTTP Request Sampler для кожного ендпойнту, який вам потрібен.

– Задайте необхідні параметри для кожного запиту (URL, метод, заголовки, тіло запиту тощо) на основі відомостей з Swagger.

– Налаштовуємо аутентифікацію.

– Додаємо необхідні заголовки та параметри для аутентифікації запитів у JMeter.

– Додаємо лісенери (рисунок 3.14).

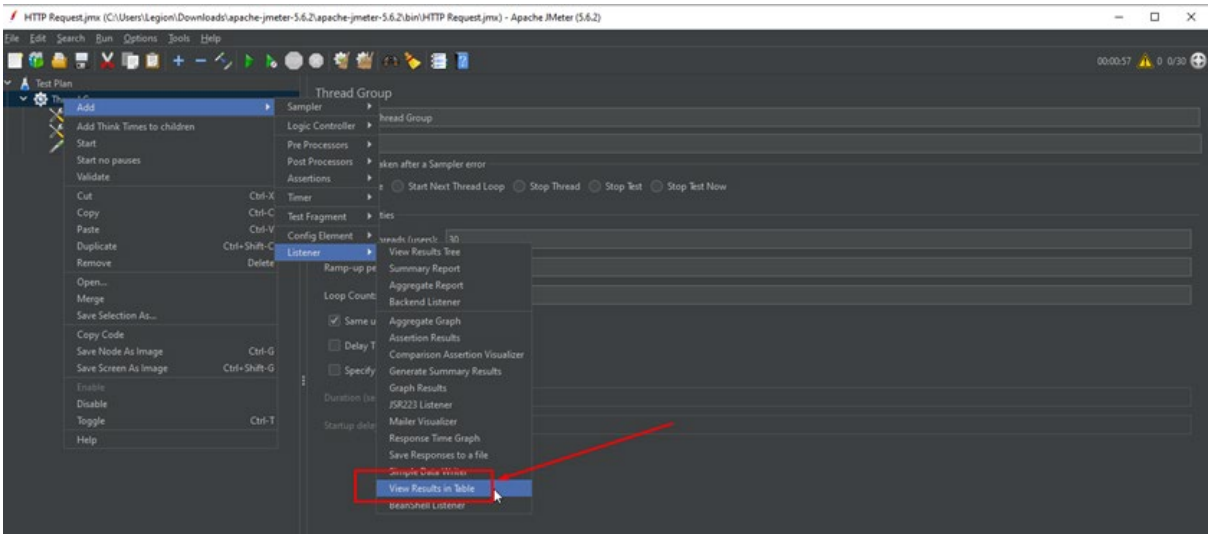

Рисунок 3.14- Додавання лісенерів до тестового плану.

– Додаємо лісенери "View Results Tree" до тестового плану, щоб переглядати результати виконання запитів (рисунок 3.15).

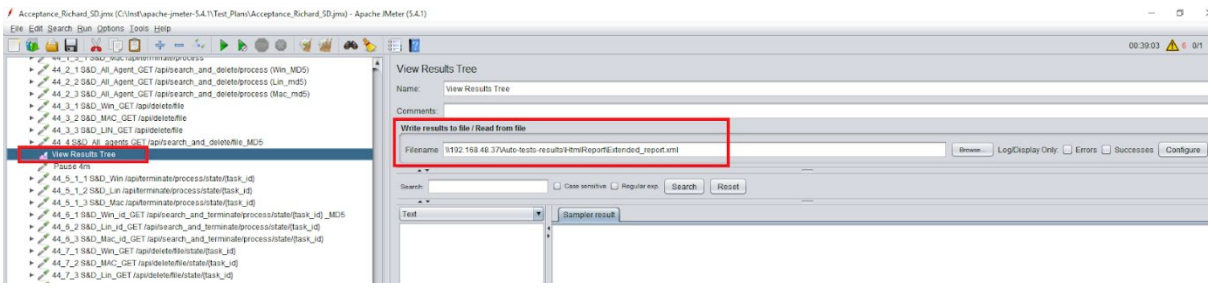

Рисунок 3.15- Додавання "View Results Tree" до тестового плану.

c) Запуск плану і збереження звіту.

Фінальним етапом створення тестового плану є його запуск і збереження звіту для подальшого аналізу.

Процесс запуску тест-плану (рисунок 3.16).

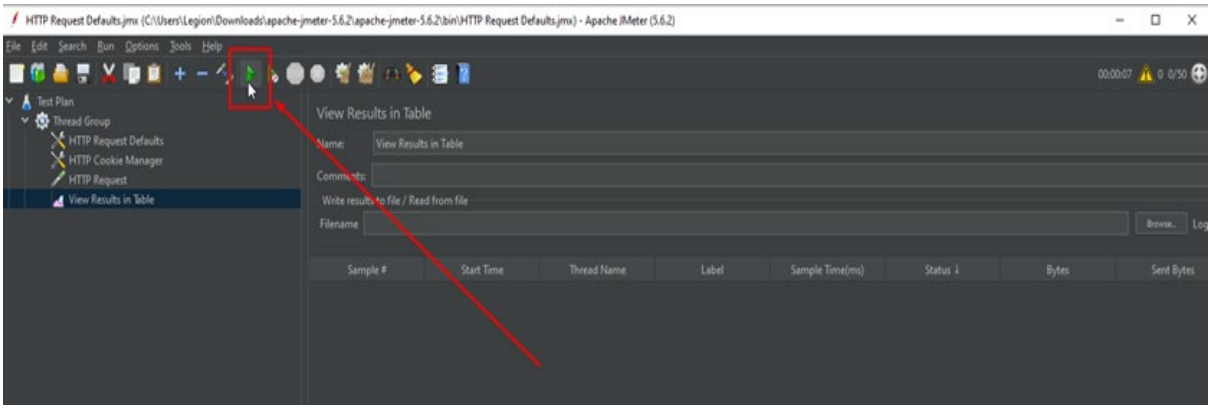

Рисунок 3.16- Запуск тест-плану.

Запуск тестування робимо за шагами:

- Натискаєм “Run”;
- Спостерігаємо за виконанням запитів у лісенері "View Results Tree" (рисунок 3.17);
- Детально переглядаємо тести, які пройшли невдало (червоного кольору)
- Переробляємо невдалі тести;
- Запускаємо тестування знов;

Всі ці кроки повторюємо доки усі тести пройдуть вдало без помилок (колір усіх тестів зелений)

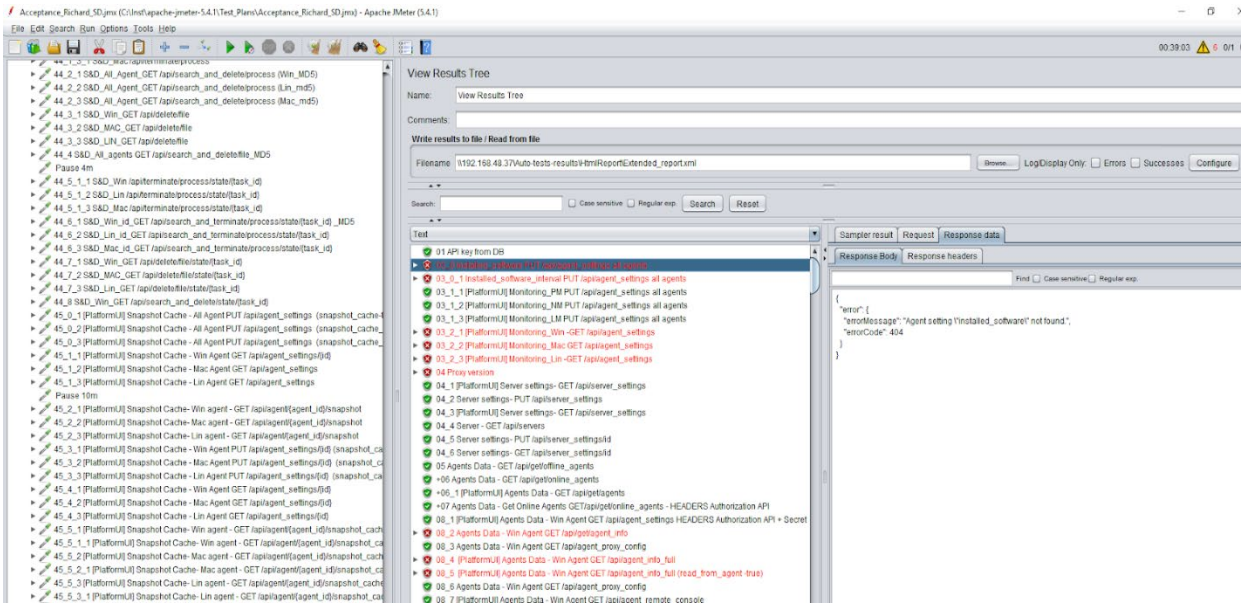

Рисунок 3.17- Виконання запитів у лісенері.

Формування звіту

Під час тестування, JMeter, за умовчанням, зберігає результати у файл у форматі CSV або XML. Це можна налаштовується безпосередньо у лісенері "View Results Tree"

Для того щоб мати можливість переглянути результати тестування у будь-якому веб-браузері потрібно згенерувати звіт у HTML форматі (рисунок 3.18).

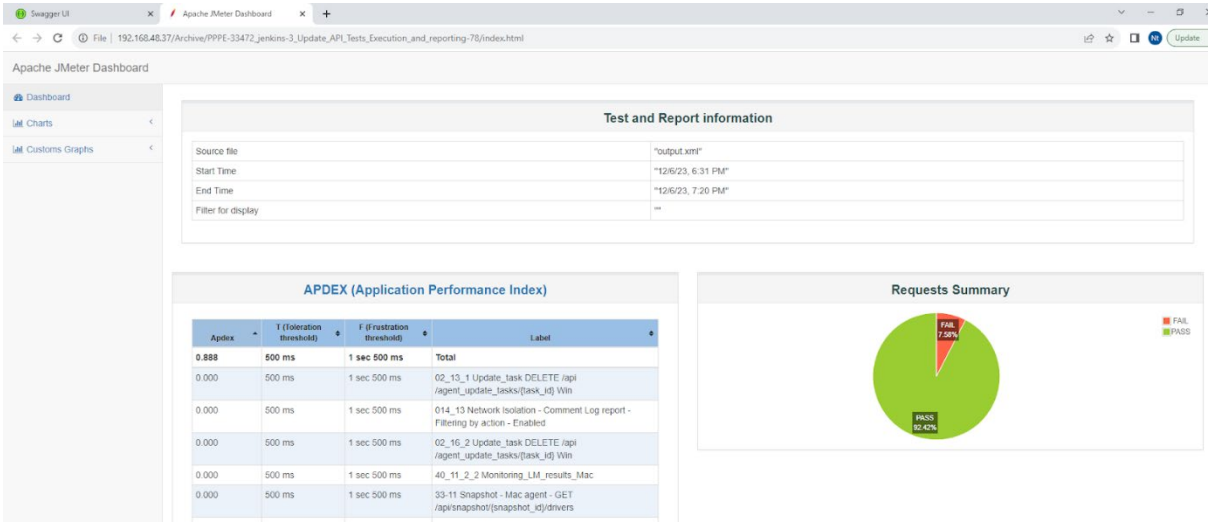

Рисунок 3.18- Звіт у HTML форматі.

Для цього додамо у скрипт, який запускає JMeter, інформацію про HTML формат і місце де його зберігати.

У наведеному процесі тестування JMeter запускається за допомогою Jenkins. Тож додамо до третьої задачі “API_Tests_Execution_and_reporting” в Jenkins наступний код:

```
cd C:\Inst\apache-jmeter-5.4.1\bin
jmeter -n -t c:\Inst\apache-jmeter-5.4.1\Test_Plans\Test_plan.jmx -l
C:\Users\cyfir\.jenkins\workspace\API_Tests_Execution_and_reporting\output.xml
-e -o D:\Auto-tests-results\HtmlReport
```

На рисунок 3.19 показано як віглядає блок запуску тестплану в JMeter.

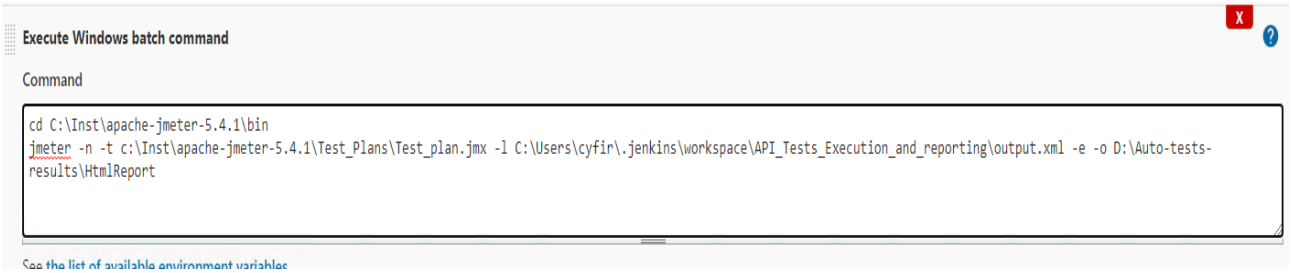

Рисунок 3.19 Блок запуску тестплану в JMeter

Після запуску автотестів у вказаній директорії з'явиться звіт у форматі HTML. Цей звіт надасть детальну візуалізацію результатів тестування, включаючи графіки, таблиці та інші метрики.

d) Оптимізація та налаштування.

На основі отриманих результатів оптимизуємо та налаштовуваємо у JMeter колекції, запити та послідовність їх виконання для забезпечення:

- стабільності роботи,
- максимальної швидкості
- точності отриманих результатів.

3.2.3 Python Script для аналізу покриття API

Розглянемо процес створення Python-скрипта для аналізу покриття API на основі файлів JMeter (.jmx) та Swagger специфікації.

Цей скрипт дозволяє вам тестувальнику автоматично аналізувати покриття API на основі даних із JMeter та Swagger і створювати звіт, який вказує на те, які ендпоінти покриті і які не покриті тестами.

Для реалізації задачі по обчисленню покриття ендпоінтів автотестами Python-скрипт порівняннює Swagger і JMX дані за наступним алгоритмом:

- Завантаження специфікації Swagger з вказаного URL за допомогою бібліотеки requests.
- Завантаження специфікації з JMX файлу.
- Очищення обох специфікації (з Swagger URL і згенеровану з JMX) від зайвих даних за допомогою функції clean_swagger_paths.
- Порівняння ендпоінти між специфікацією Swagger і згенерованою специфікацією з JMX файлу.

- Визначення покритих і непокритих ендпоінтів.
- Сегментація ендпоінтів.
- Обчислення покриття API. У скрипті обчислюється процентне співвідношення покритих ендпоінтів до всіх ендпоінтів (за винятком спеціальних). Значення виводиться як результат аналізу покриття API.
- Генерація HTML звіту. Скрипт використовує бібліотеку jinja2 для створення HTML звіту, який включає процент покриття та списки покритих, спеціальних покритих і непокритих ендпоінтів. Ендпоінти виділяються кольорами в звіті для легшого сприйняття (рисунок 3.19).
- Збереження звіту.

Звіт зберігається у файл за шляхом, який був переданий у аргументі командного рядка при запуску скрипту, для подальшого використання його у Jenkins.

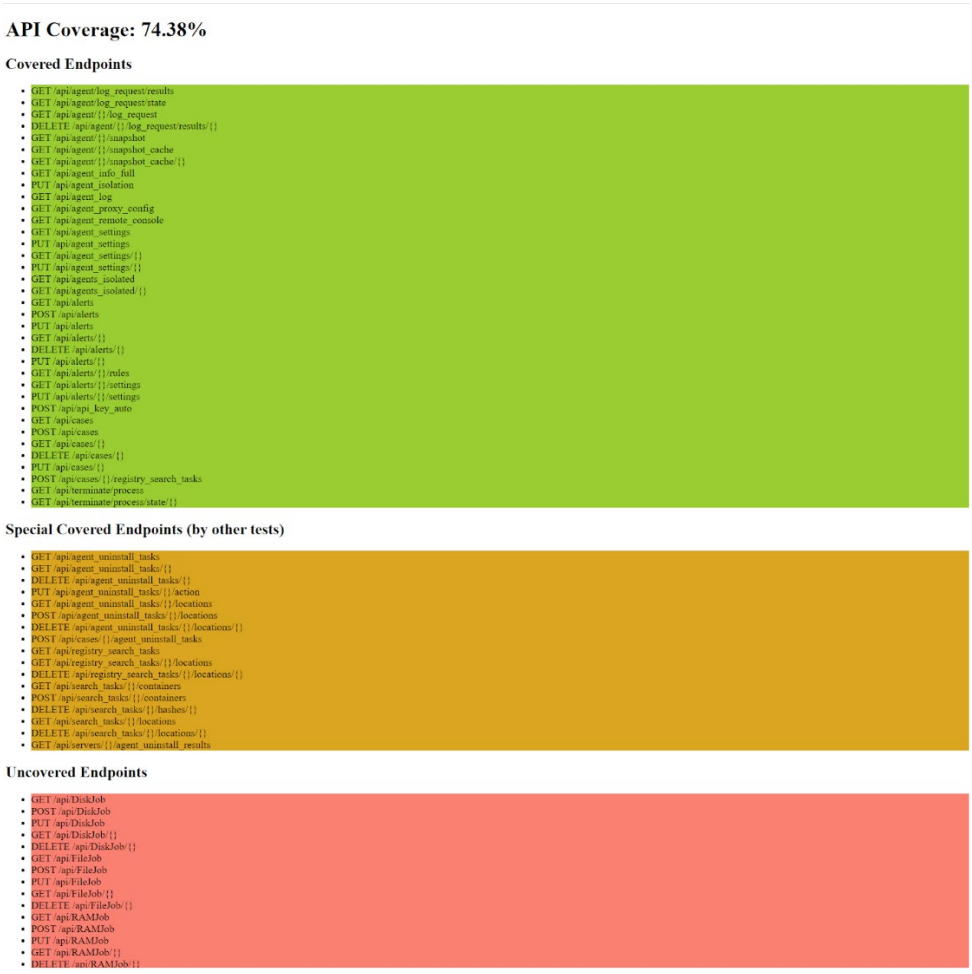

Рисунок 3.19-HTML звіт з результатами аналізу покриття API астотестами.

У звіті ендпоінти поділяються на групи:

- Покриті ендпоінти - ті, які присутні як в Swagger, так і в JMX.
- Спеціальні покриті ендпоінти - ендпоінти, які покриті іншими тестами.
- Непокриті ендпоінти - ендпоінти, які не знайдені ні в Swagger, ні в JMX.

Для реалізації функционалу автоматичного аналізу API покриття на основі даних із JMeter та Swagger і створення звіту, з інформацією про те, які ендпоінти покриті і які не покриті тестами, скрипт має наступні функції:

- clean_url - Функція очищає URL від зайвих параметрів і підготовлює його для порівняння;
- clean_swagger_paths - Функція видаляє з Swagger специфікації ендпоінти, які містять /v1/ і підготовлює специфікацію до порівняння;
- extract_api_calls_from_jmx - Функція читає JMX файл, витягує з нього дані про API виклики (HTTPSamplerProxy), очищає та форматує їх для подальшого порівняння.
- generate_swagger_from_api_calls - Функція створює Swagger специфікацію на основі API викликів, отриманих з JMX файлу.

Для запуску Python-скрипта необхідно встановити інтерпрітатор Python 3.9.5 з офіційного сайту https://www.python.org/downloads/.

У скрипті також використовуються додаткові бібліотеки: requests і jinja2:

- argparse - бібліотека для обробки аргументів командного рядка. Користувач передає: шлях до JMX файлу, URL Swagger специфікації та шлях для збереження звіту при запуску скрипта.
- Jinja2 - бібліотека яка створює HTML звіт.

Бібліотеки встановлюємо через pip install:

-python -m pip install requests;

-python -m pip install.

Висновки до третього розділу

У даному розділі дипломної роботи представлено результати конкретизації методу, описаного у підрозділі 2.6. Зокрема, було здійснено деталізацію кожної

складової методу, приведено відповідні приклади, а також описано процес встановлення, конфігурації та інтеграції всіх компонентів.

Основну увагу приділено розробці Python-скрипту, який використовується для аналізу покриття коду тестами. Цей інструмент є ключовим елементом в процесі автоматизованого тестування, демонструючи покриття функціоналу автотестами. Загальні результати розробки представлено у [11]. Розроблений скрипт має такі переваги:

- незалежність та можливість легкого запуску з командної строки, що робить його окремим продуктом;
- гнучкість інтеграції у різноманітні архітектури автоматизації, як це продемонстровано на прикладі;
- легкість внесення змін;
- простота використання та зрозумілість звітів.

Огляд конкретних прикладів застосування розглянутих інструментів та методів ілюструє, яким чином можна ефективно впроваджувати автоматизоване тестування у реальні проекти. Це включає аналіз типових сценаріїв використання та визначення найкращих практик, вибір інструментів, налаштування тестового середовища та розробку стратегій тестування, що враховують особливості API.

В цілому, правильне використання автоматизованих інструментів та методів підвищує ефективність тестування, скорочує час на виконання тестів та сприяє підвищенню якості програмного продукту.

4 ДОСЛІДЖЕННЯ ДОЦІЛЬНОСТІ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ API ТА ОЦІНЮВАННЯ ЙОГО РЕЗУЛЬТАТІВ

4.1 Цілі і задачі дослідження

Основною метою цього дослідження є виявлення та аналіз переваг впровадження методологій автоматизованого тестування та інструментарію для оцінювання покриття API у контексті реального програмного продукту, розробляємого компанією Apriorit. Це дослідження спрямоване на досягнення наступних ключових цілей:

1) демонстрація прискорення темпу виявлення дефектів через оптимізацію часових та ресурсних витрат, з використанням підходів, таких як Shift-left, для ефективного виявлення та усунення помилок на ранніх етапах розробки;
2) підвищення ефективності процесу тестування за рахунок зниження трудомісткості виявлення дефектів, що призводить до зменшення загальних витрат на розробку продукту.

В рамках цього дослідження визначено такі завдання:

1) Проведення аналізу, порівняння та оцінку мануального та автоматизованого процесу:
   - тестування API;
   - обчислення покриття API автотестами,

   зосередившись на:
   - трудомісткості - скільки людино-годин витрачається на тестування;
   - швидкості виконання - скільки часу витрачається і тестування.
2) Оцінка:

   - ефективністі тестування - кількість дефектів відносно кількості людино-годин витраченних на їх знайдення;

   - темпу виявлення дефектів -час визраченний на знайдення дефекту після початку розробки, в рамках підходу Shift-left, з урахуванням

використання моделі Trunk Based Development [10] для повного циклу тестування кожного бранча.

4.2 Підготовка експериментів

Всі експерименти будуть проводитись на базі реального проекту, розробленого компанією Apriorit.

Сфера діяльності продукту - кібер безпека.

Модель розробки - Trunk Based Development.

Продукт - мережевий інструмент для дослідження та реагування на інциденти.

Функціонал - моніторингу активності користувачів.

Цей інструмент, здатний виявляти та запобігати внутрішнім загрозам, проводити розслідування інцидентів, відстежувати підозрілі дії та перевіряти доступ користувачів, і включає інтелектуальну систему оповіщення із набором додаткових інструментів для автоматичного реагування на всі типи інцидентів.

Основні можливості системи керування безпекою включають:

- виявлення, запобігання та аналіз внутрішніх загроз;
- Розслідування інцидентів;
- Моніторинг діяльності користувачів.

4.2.1 Розробка сценаріїв експериментів

Експеримент 1. Тестування API.

Мета експерименту полягає у визначенні часу та трудомісткості, необхідних для ручного та автоматичного тестування API в рамках реального проекту (див. розділ 4.2). Зокрема, має бути підтверджено, що автоматичне тестування дозволяє заощадити 53 години і 53.3 людино-години на кожен цикл тестування API порівняно з ручним тестуванням.

Експеримент буде проводитись за такими кроками:

- розгортання середовища;
- встановлення програмного продукту;
- виконання тестування API;
- формування звіту;
- вимірювання швидкості і трудомісткість кожного етапу;

– занесення результатів вимірювання в таблицю 4.1.

Експеримент 2. Обчислення покриття API автотестами.

Мета експерименту полягає у визначенні часу та трудомісткісті, необхідних для ручного і за допомогою Python-скрипта тестування покриття API автотестами в рамках реального проекту (див. розділ 4.2). Зокрема, виходячи з аналізу попередніх випадків використання автоматизації, має бути підтверджено, що тестування за допомогою Python-скрипта дозволяє скоротити час на на кожен цикл тестування з 2.6 годин при мануальній перевірки до 20 хвилин, а також повністю виключити участь людини з цього процессу і тим самим заощадити 2.6 людино-години на кожен цикл тестування покриття API автотестами.

Експеримент буде проводитись за слідуючими кроками:

1) разгортання середовища;
2) встановлення програмного продукту;
3) визначення покритих і не покритих ендпоінтів;
4) підрахунок покриття;
5) формування звіту;
6) вимірювання швидкості і трудомісткість кожного етапу;
7) занесення результатів вимірювання в таблицю 4.2.

4.2.2 Джерела та інструменти для отримання даних

Для отримання даних про трудомісткість операцій і часу їх виконання будемо використовувати такі інструменти і джерела:

1) Jira - система для управления проектами в компанії Apriorit містить час який витрачається на мануальне тестування API
2) Jenkins - фіксує результат виконання та тривалість кожної операції, яка була запущена.
3) Годинник для вимірювання часу витраченого для виконання мануального аналізу покриття API автотестами.

4.2.3 Розгортання середовища і встановлення і налаштування необхідних інструментів

Для проведення експериментів нам знадобляться:

- персональний комп'ютер, на якому буде розгорнуте середовище для проведення експериментів. Мінімальні параметри: Windows Server 2016, RAM 64GB;
- VMware 2016 з чотирма віртуальними машинами (одна для сервера і три для встановлення агентів) і чисті снепшоти для кожної з них, як у п.3.2.1.
- Jenkins налаштований як у п.3.2.1;
- JMeter, налаштований як у п. з колекцією автотестів для тестування API, як показано у п.3.2.2;
- інтерпретатор Python 3.9.5 встановлений разом з додатковими бібліотеками, як в п.3.2.3.

Підготовка середовища є однаковою для обох видів тестування, тож ми його не будемо враховувати під час розрахунків ергономічності способів тестування.

4.3 Проведення експерименту

4.3.1 Мануальне тестування

Опишемо послідовність дій для вимірювання швидкості і трудомісткості операцій при мануальному тестуванні.

Експеримент 1. Тестування API.

Всі задачі у цьому експеременті віконуються користувачем і включає наступні операції:

1) запуск віртуальних машин з чистих снепшотів на VMware для налаштування сервера і трьох видів агентів.Час вимірюємо годинником;
2) встановлення серверної частини і агентів:
   - на CentOS7 за допомогою термінала запускаєм bash скрипти для розгортання сервера API і Swagger;
   - на Windows 10 копіюємо .exe файл і встановлюємо Windows агент;
   - на MacOS 12 за допомогою термінала запускаєм bash скрипти для розгортання Mac агента;
   - на другому CentOS7 за допомогою термінала запускаєм bash скрипти для розгортання Linux агента, час вимірюємо годинником;
3) виконання тестування.

Для виконання тестування переходимо за посиланням, за яким разом з серверною частиною було розгорнуто API і Swagger, далі:

- виконуємо виклик і аналізуємо відповідь для кожного ендпоінту. Едпоінти, які перевіряють роботу агентів, запускаються окремо для кожного з них;
- обчислюємо час. Час разраховується емпірично згідно формули на базі данних у Jira (Apriorit).

Розрахунок загального часу на мануальне тестування всіх ендпоінтів обчислюється за формулою:

$$T=Es*t+Ea*t*n=10*0.1+170*0.1*3=52 \text{ год.}$$

де:

Tt- загальний час тестування всіх ендпоінтів;

Es-кількість серверних ендпоінтів;

Ea-кількість агентських ендпоінтів;

t-середній час мануального тестування одного ендпоінту;

n -кількісь агентів для яких треба провести тестування;

4) формування звіту.

На базі отриманих результатів у текстовому редакторі складаємо перелік тестів, які були успішно пройдені і ті які повернули помилку або не очікуваний результат. Час обчислюємо емпірично на базі данних у Jira (Apriorit) як середні витрати на звіт;

5) занесення результатів вимірювання в таблицю 4.1.

Експеримент 2. Обчислення покриття API автотестами.

1) запуск віртуальної машини з чистого снепшоту для налаштування сервера. Час вимірюємо годинником;
2) встановлення сервера. На CentOS7 за допомогою термінала запускаєм bash скрипти для розгортання сервера API і Swagger. Час вимірюємо годинником;
3) виконання порівняння даних Swagger з даними у .jmx файлі робимо наступним чином:

- в браузері відкриваєм посилання за яким разом з серверною частиною було розгорнуто API і Swagger;
- знаходимо у директорії JMeter .jmx файл з колекцією автотестів для API і відкриваємо за допомогою Notepad;
- у Swagger копіюємо шлях для кожного ендпоінту і за допомогою Search перевіряємо його наявність у .jmx файлі відкритому у Notepad;
- обчислюємо час. Час разраховується емпірично згідно формули на базі данних у Jira (Apriorit).

Розрахунок загального часу на мануальне обчислення покриття API автотестами обчислюється за формулою.

$$Te=e*t= 180*0.01=1.8$$

де:

Te- загальний час тестування всіх ендпоінтів;

e-кількість існуючих ендпоінтів;

t-середній час пошуку одного ендпоінту.

4) формування звіту.

На базі отриманних результатів складаємо перелік ендпоінтів які були знайдені у .jmx файлі і тих які в ньому відсутні. Час обчислюємо емпірично на базі данних у Jira (Apriorit), як середні витрати на звіт.

Обчислюємо покриття ендпоінтів автотестами за формулою:

$$\frac{\text{Кількість ендпоінтів,покритих тестами}}{\text{Загальна кількість ендпоінтів}} * 100$$

5) занесення результатів вимірювання в таблицю 4.2.

4.3.2 Експериментальні дослідження автоматизованого тестування

Опишемо послідовність дій для вимірювання швидкості і трудомісткості операцій при автоматичному тестуванні.

Експеримент 1. Тестування API.

Користувач вводить шлях до нової версії продукту у Jenkins інтерфейс і запускає задачу. Далі всі задачі виконуються Jenkins автоматично, без участі користувача. Перелік задач наступний:

1) Запуск віртуальних машин з чистих снепшотів для налаштування сервера і трьох видів агентів.
2) Встановлення серверної частини і агентів.
3) Виконання тестування.
4) Формування звіту.

Час на виконання кожної задачі відображається у Jenkins інтерфейс.

Результати експериментів наведено у таблиці 4.1.

Експеримент 2. Обчислення покриття API автотестами за допомогою Python-скрипта.

Користувач вводить шлях до нової версії продукту у Jenkins інтерфейс і запускає задачу.

Далі всі задачі виконуються Jenkins автоматично, без участі користувача. Перелік задач наступний:

1) Запуск віртуальної машини з чистого снепшоту для налаштування сервера.
2) Встановлення сервера
3) Виконання порівняння данних Swagger з даними у .jmx файлі.
4) Формування звіту.

Час на виконання кожної задачі відображається у Jenkins інтерфейс.

Результати експериментів наведено у таблиці 4.2.

4.3.3 Обчислення трудомісткості операцій

За трудомісткість операцій будемо вважити час витрачений тестувальником на її виконання і вимірювати у людино-годинах. Для мануального тестування трудомісткість операцій дорівнює часу її виконання так як всі задачі виконуються беспосередньо тестувальником. Для автоматичного тестування трудомісткість операцій дорівнює нулю так як всі задачі виконуються автоматично без витрачання часу тестувальника. Результати експериментів наведено у таблиці 4.2.

Таблиця 4.1 - Результати експерименту 1 по тестуванню API

Експеримент 1 Тестування API								
Операція	Мануальне тестування				Автоматичне тестування			
	Засіб	Інструмент для вимірювання часу	Час (год)	Трудомісткість (люд. год)	Засіб	Інструмент для вимірювання часу	Час (год)	Трудомісткість (люд. год)
1	2	3	4	5	6	7	8	9
Запуск віртуальних машин з чистих снепшотів	VMware	Годинник	0.25	0.25	Jenkins - задача перша см.п. 3.2.1.4.1	Jenkins інтерфейс	0.02	0
Встановлення серверної частини і агентів	Win - consol Mac/Linux - Terminal	Годинник	0.75	0.75	Jenkins - задача друга см.п. 3.2.1.4.2	Jenkins інтерфейс	0.3	0

Продовження таблиці 4.1 - Результати експерименту 1 по тестуванню API

1	2	3	4	5	6	7	8	9
Виконання тестування	Swagger	Jira (Apriorit) середні витрати на один тест	52	52	Jenkins+ JMeter - задача третя см.п. 3.2.1.4.3	Jenkins інтерфейс	1.3	0
Формування звіту	Текстовий редактор	Jira (Apriorit) середні витрати на звіт	0.3	0.3	Jenkins+ JMeter - задача третя см.п. 3.2.1.4.3	Jenkins інтерфейс	0	0

Таблиця 4.2 - Результати експерименту 2 по обчисленню покриття API автотестами

Експеримент 2 Обчислення покриття API автотестами.								
Операція	Мануальне тестування				Тестування з використанням Python-скрипту			
	Засіб	Інструмент для вимірювання часу	Час (год)	Трудомісткість (люд. год)	Засіб	Інструмент для вимірювання часу	Час (год)	Трудомісткість (люд. год)
1	2	3	4	5	6	7	8	9

Продовження таблиці 4.2 - Результати експерименту 2 по обчисленню покриття API автотестами

1	2	3	4	5	6	7	8	9
Запуск одної віртуальної машини	VMware	Годинник	0.1	0.1	Jenkins - задача перша см.п. 3.2.1.4.1	Jenkins інтерфейс	0.02	0
Встановлення серверної частини продукту	Linux - Terminal	Годинник	0.5	0.5	Jenkins - задача друга см.п. 3.2.1.4.2	Jenkins інтерфейс	0.3	0
Порівняння Swagger з даними у .jmx файлі.	Swagger + .jmx +Search	Jira (Apriorit) середні витрати на один ендпоінт	1.8	1.8	Jenkins+ Python Script - задача четверта см.п. 3.2.1.4.4	Jenkins інтерфейс	0.003	0
Формування звіту	Калькулятор +Текстовий редактор	Jira (Apriorit) середні витрати на звіт	0.2	0.2	Jenkins+ Python Script - задача четверта см.п. 3.2.1.4.4	Jenkins інтерфейс	0	0

Для наглядності побудуємо пайчарм і гистограми по результатам експериментів.

Трудомісткість мануального тестування API відобразимо на рисунку 4.1

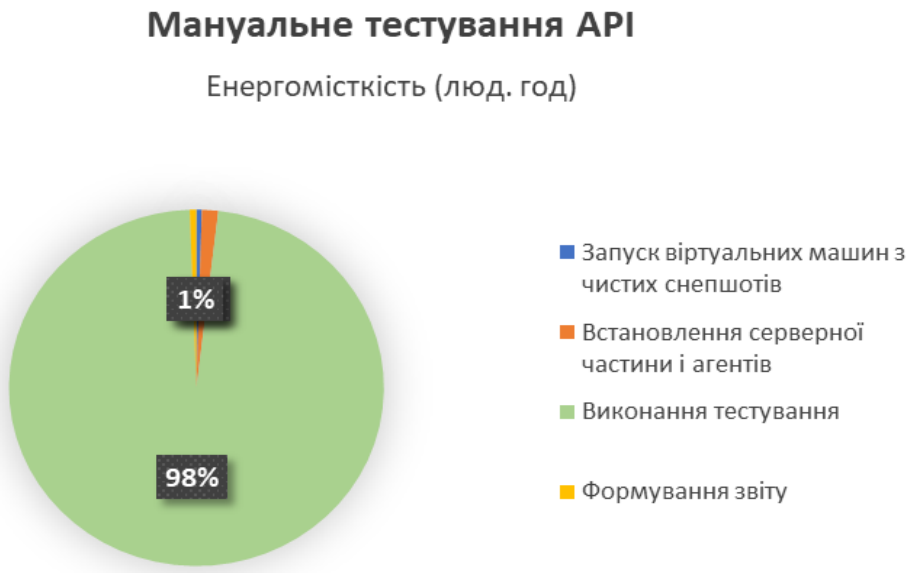

Рисунок 4.1- Трудомісткість мануального тестування API.

Трудомісткість мануального обчислення покриття API автотестами відобразимо на рисунку 4.2

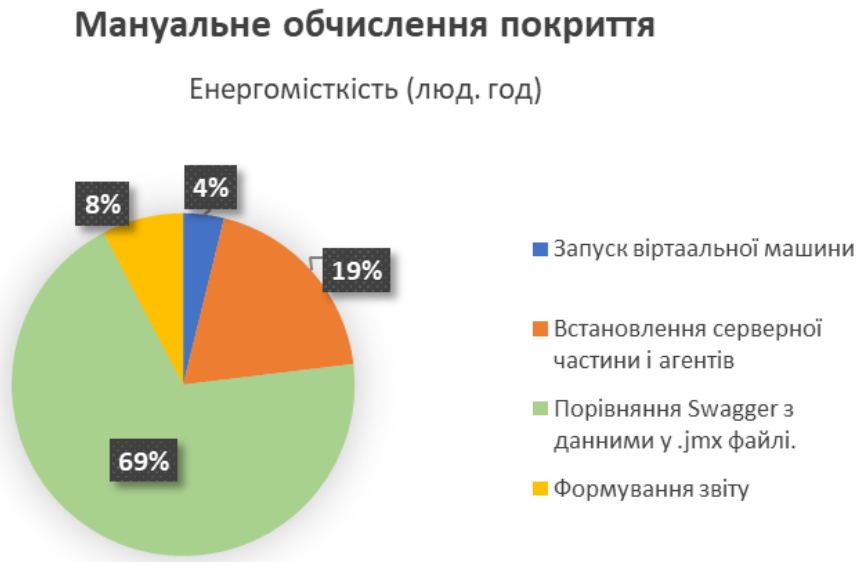

Рисунок 4.2-Трудомісткість мануального обчислення покриття API автотестами.

Порівняння витрат часу на тестування API мануально і автоматично відобразимо на рисунку 4.3.

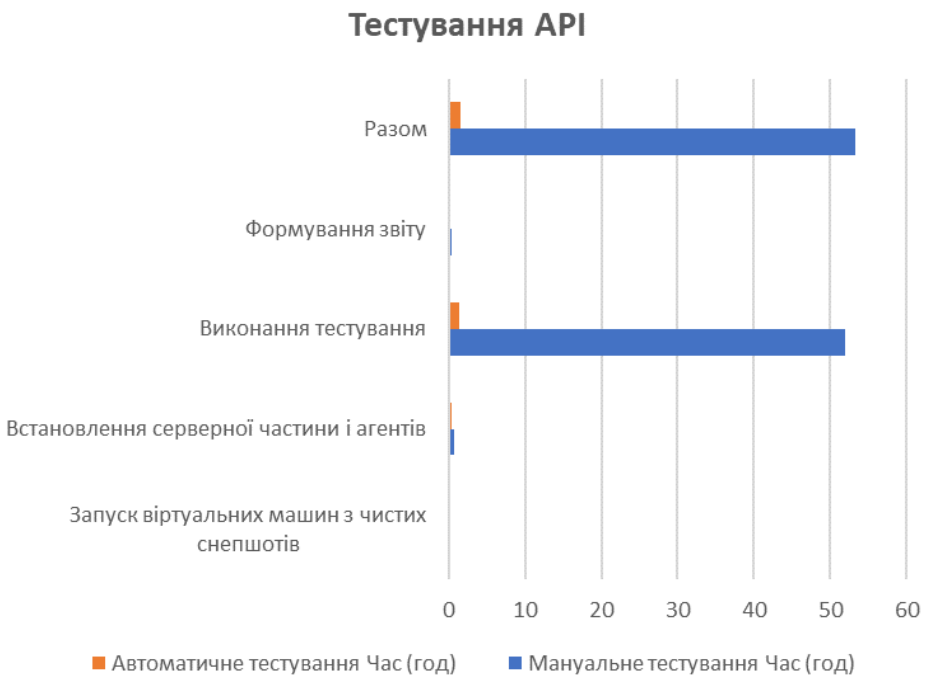

Рисунок 4.3- Порівняння витрат часу на тестування API мануально і автоматично.

Порівняння витрат часу на обчислення покриття API автотестами мануально і за допомогою Python-скрипту відобразимо на рисунку 4.4.

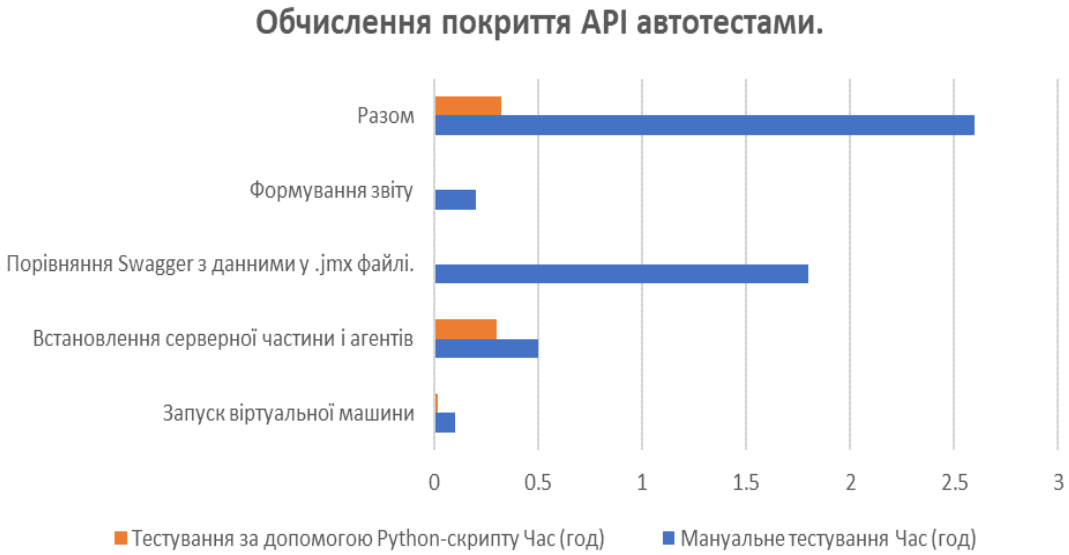

Рисунок 4.4 - Порівняння витрат часу на обчислення покриття API автотестами мануально і за допомогою Python-скрипту.

Результати експерименту явно демонструють нам перевагу автоматичного тестування над мад мануальним як за трудомісткістю так і за витратами часу.

4.4 Аналіз результатів експериментів

З метою обчислення економії часу і ресурсів більш детально проаналізуємо і порівняємо результати експериментів.

4.4.1 Обчислення економії ресурсів для одного циклу тестування

На базі отриманих результатів (див. Таблиця 4.1) по трудомісткості кожної операції можно підрахувати економію ресурсів для одного циклу тестування.

Розрахунок економії ресурсів для одного циклу тестування.

Обчислимо трудомісткість мануального тестування API і його покриття.

Ем=Ем1+Ем2= 53.3+2.6=55.9 (люд.год)

де:

Ем- загальна трудомісткість циклу тестування API мануально;

Ем1- загальна трудомісткість тестування API мануально;

Ем2- загальна трудомісткість тестування покриття API мануально.

Обчислимо трудомісткість автоматичного тестування API і його покриття мануально (без Python-скрипту).

Еам=Еа1+Ем2=0+2.6=2.6 (люд.год)

де:

Еам- загальна трудомісткість циклу тестування API автоматично з тестуванням покриття мануально;

Еа1- загальна трудомісткість тестування API автоматично;

Ем2- загальна трудомісткість тестування покриття API мануально.

Обчислимо трудомісткість автоматичного тестування API і його покриття.

Еа=Еа1+Еа2=0+0=0 (люд.год)

де:

Еа- загальна трудомісткість циклу тестування API автоматично;

Еа1- загальна трудомісткість тестування API автоматично Python-скриптом;

Еа2- загальна трудомісткість тестування покриття API автоматично, за допомогою Python-скрипту.

Обчислимо економію ресурсів для одного циклу тестування тільки функционалу API автоматично (без тестування покриття).

Еф=Ем1-Еа1=53.3-0=53.3 (люд.год)

де:

Еф-економія ресурсів для одного циклу тестування функционалу API автоматично.

Ем1-загальна трудомісткість тестування API мануально;

Еа1-загальна трудомісткість тестування API автоматично.

Обчислимо економію ресурсів для одного циклу тестування тільки покриття API скриптом (без функционалу API).

Ес=Ем2-Еа2=2,6-0=2,6 (люд.год)

де:

Ес-економія ресурсівдля одного циклу тестування функционалу API;

Ем2-загальна трудомісткість тестування покриття API мануально;

Еа2-загальна трудомісткість тестування покриття API автоматично, за допомогою Python-скрипту.

Обчислимо економію ресурсів для одного циклу тестування API автоматично і його покриття мануально.

Е1=Ем-Еам=55.9-2.6=53.3 (люд.год)

де:

Е1-економія ресурсів для одного циклу тестування API автоматично і його покриття мануально (без скрипта).

Ем-загальна трудомісткість циклу тестування API мануально.

Еам-загальна трудомісткість циклу тестування API автоматично з тестуванням покриття мануально.

Обчислимо економію ресурсів для одного циклу тестування API автоматично і його покриття за допомогою Python-скрипта.

Е2=Ем-Еа=55.9-0=55.9 (люд.год)

де:

Е2-економія ресурсів для одного циклу тестування API і його покриття.

Ем-загальна трудомісткість циклу тестування API мануально.

Еа-загальна трудомісткість циклу тестування API автоматично.

Економія людського ресурсу завдяки використовування автоматизації тестування API дозволяє зменшити бюджет на тестування і скоротити час випуску продукту.

4.4.2 Обчислення економії часу для одного циклу тестування

На базі отриманих результатів (див. Таблиця 4.1) по часу, за який виконується операція, можно підрахувати економію часу для одного циклу тестування.

Розрахунок економії часу для одного циклу тестування.

Обчислимо час мануального тестування API і його покриття.

Тм=Тм1+Тм2= 53.3+2.6=55.9 (год)

де:

Тм- загальний час циклу тестування API мануально;

Тм1- загальний час тестування API мануально;

Тм2- загальний час тестування покриття API мануально.

Обчислимо час автоматичного тестування API і його покриття мануально (без Python-скрипту).

Там=Та1+Тм2= 1.62+2.6=3.22 (год)

де:

Там- загальний час циклу тестування API автоматичноо з тестуванням покриття мануально;

Та1- загальний час тестування API автоматично;

Тм2- загальний час тестування покриття API мануально.

Обчислимо час автоматичного тестування API і його покриття.

Та=Та1+Та2= 1.62+0.323=1.943 (год)

де:

Та- загальний час циклу тестування API автоматично;

Та1- загальний час тестування API автоматично;

Та2- загальний час тестування покриття API автоматично, за допомогою Python-скрипту.

Обчислимо економію часу для одного циклу тестування тільки API автоматично (без тестування покриття).

Тф=Тм1-Та1= 53.3-1.62=51.68 (год)

де:

Тф- економія часу для одного циклу тестування функционалу API;

Тм1- загальний час тестування API мануально;

Та1- загальний час тестування API автоматично.

Обчислимо економію часу для одного циклу тестування тільки покриття API скриптом (без функционалу API).

Тс=Тм2-Та2= 2.6-0.323=2.277 (год)

де:

Тс- економія часу для одного циклу тестування функционалу API;

Тм2- загальний час тестування покриття API мануально;

Та2- загальний час тестування покриття API мануально.

Обчислимо економію часу для одного циклу тестування API автоматично і його покриття мануально (без Python-скрипту).

Т1=Тм-Там= 55.9-3.22=52.68 (год)

де:

Т1- економія часу для одного циклу тестування API і його покриття.

Тм- загальний час циклу тестування API мануально;

Там- загальний час циклу тестування API автоматичноо з тестуванням покриття мануально.

Обчислимо економію часу для одного циклу тестування API автоматично і його покриття за допомогою Python-скрипта.

Т2=Тм-Та= 55.9-1.943=53.957 (год)

де:

Т- економія часу для одного циклу тестування API і його покриття.

Тм- загальний час циклу тестування API мануально;

Та- загальний час циклу тестування API автоматично.

Економія часу завдяки автоматизації тестування API дозволяє:

- зменшити час на виявлення дефектів в програмному продукті;
- покращити якість продукту через можливість регулярного запуску тестування на різних етапах його розробки;
- скоротити час на випуск продукту.

Результати обчислення економії часу і трудомісткості для одного циклу тестування занесемо у таблицю 4.3

Таблиця 4.3 - Результати обчислення економії часу і трудомісткості для одного циклу тестування.

Тип тестування	Функционал	Час (год)		Трудомісткість (люд. год)	
		Позначення	Значення	Позначення	Значення
1	2	3	4	5	6
Мануальне	Тестування API	Тм1	53.3	Ем1	53.3
	Тестування покриття API	Тм2	2.6	Ем2	2.6
Мануальне тестування API і його покриття	Тестування API і його покриття	Тм=Тм1+ Тм2	55.9	Ем=Ем1+ Ем2	55.9
Автоматичне тестування	Тестування API	Та1	1.62	Еа1	0
	Тестування покриття API скриптом	Та2	0.323	Еа2	0

Продовження таблиця 4.3 - Результати обчислення економії часу і трудомісткості для одного циклу тестування.

1	2	3	4	5	6
Автоматичне тестування API і мануально покриття API (без скрипта)	Тестування API і його покриття	Там=Та1+Тм2	3.22	Еам=Еа1+Ем2	2.6
Автоматичне тестування API і покриття API (з скриптом)	Тестування API і його покриття	Та=Та1+Та2	1.943	Еа=Еа1+Еа2	0
Економія	Тестування тільки API автоматично (без тестування покриття)	Тф=Тм1-Та1	51.68	Еф=Ем1-Еа1	53.3
	Тестування тільки покриття API скриптом	Тс=Тм2-Та2	2.227	Ес=Ем2-Еа2	2.6
	Тестування API (автоматично) і покриття (мануально)	Т1=Тм-Там	52.68	Е1=Ем-Еам	53.3
	Тестування API і його покриття скриптом (автоматично)	Т2=Тм-Та	53.96	Е2=Ем-Еа	55.9

4.4.3 Оцінка доцільності впровадження автоматизованого тестування на проекті

Завдяки економії часу і зниження трудомісткості процесу тестування, і задля покращення темпу виявлення дефектів і підвищення його ефективності, впровадимо регулярний запуск автотестів для бранчів.

Розглянемо як змінився об'єм тестування до його автоматизації і після.

Спрощена схема Trunk Based Development моделі відображена на рисунку 4.5.

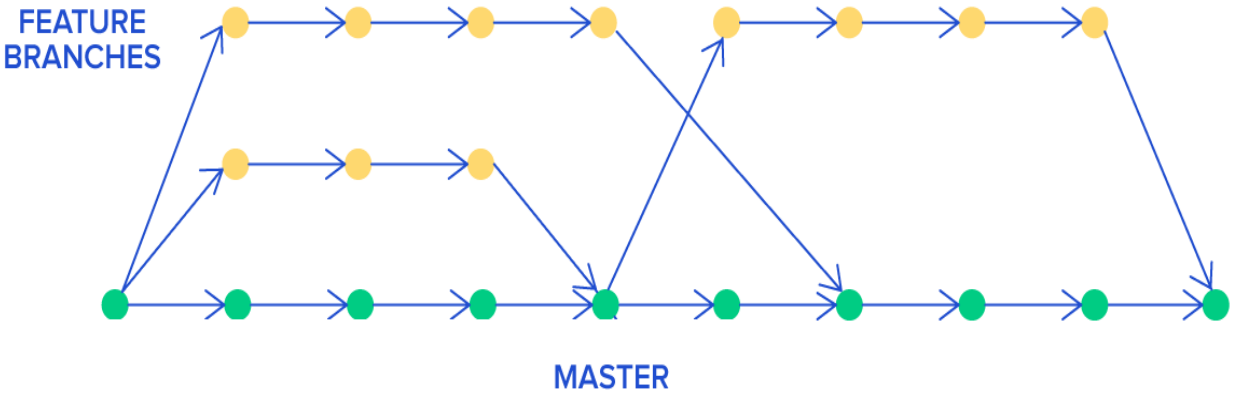

Рисунок 4.5- Спрощена схема Trunk Based Development моделі.

Схема тестування до автоматизації процесу відображена на рисунку 4.5.

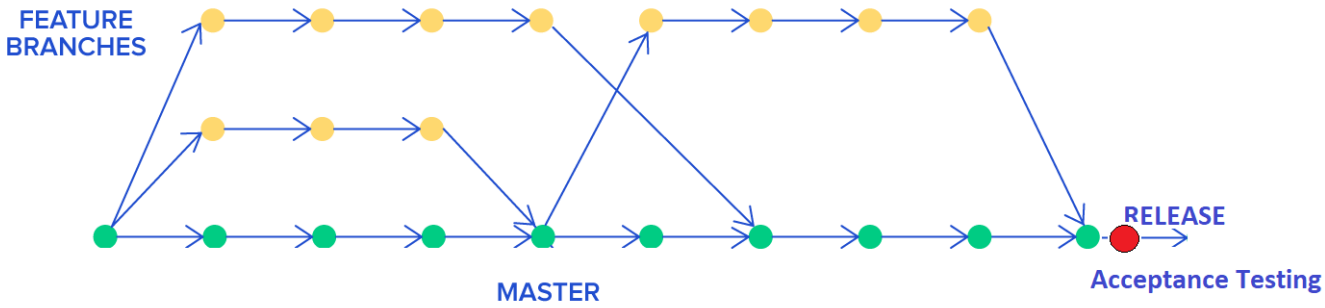

Рисунок 4.5- Схема тестування до автоматизації процесу.

Shift-left схема тестування після автоматизації процесу відображена на рисунку 4.6.

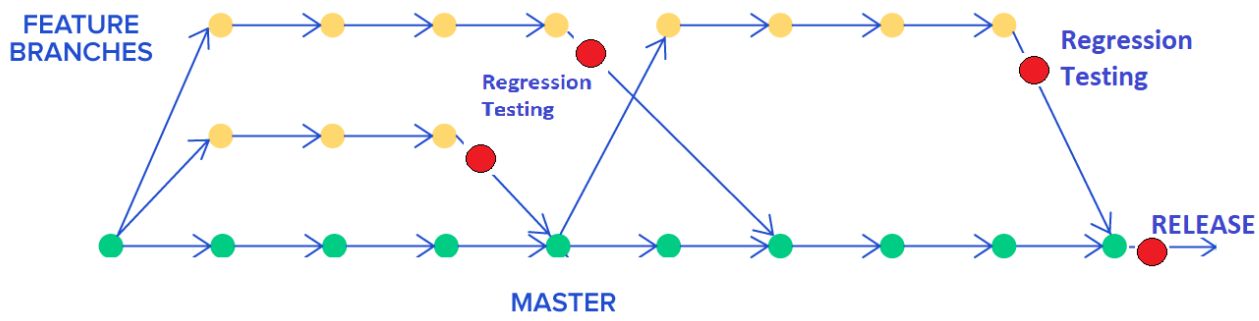

Рисунок 4.6- Shift-left схема тестування після автоматизації процесу

Для оцінки доцільність впровадження автоматизованого тестування на проекті за новою схемою обчислимо і порівняємо мануальне і автоматичне тестування за метриками:

1) темп виявлення дефектів.

Швидкість виявлення дефектів під час тестування продукту. Знаходження багів і помилок на ранніх етапах, дозволяє зменшуючи загальні витрати на їх виправлення і час на випуск продукту;

2) ефективність тестування.

Кількість виявлених дефектів відносно трудомісткості проведеного тестування. Допомагає зрозуміти, наскільки ефективними є зусилля, витрачені на перевірку програмного продукту.

Ці метрики допоможу оцінити як вплинула автоматизація процесу тестування API на процес розробки, і як змінились:

- трудомісткість процесу тестування;
- забезпечення якості продукту;
- час його виходу продукту ринок.

Розробимо формули для обчислення часу виявлення дефектів при мануальному і автоматичному тестуванні і їх порівняння.

Початкова точка виявлення дефектів при мануальном тестуванні відображено на рисунку 4.7.

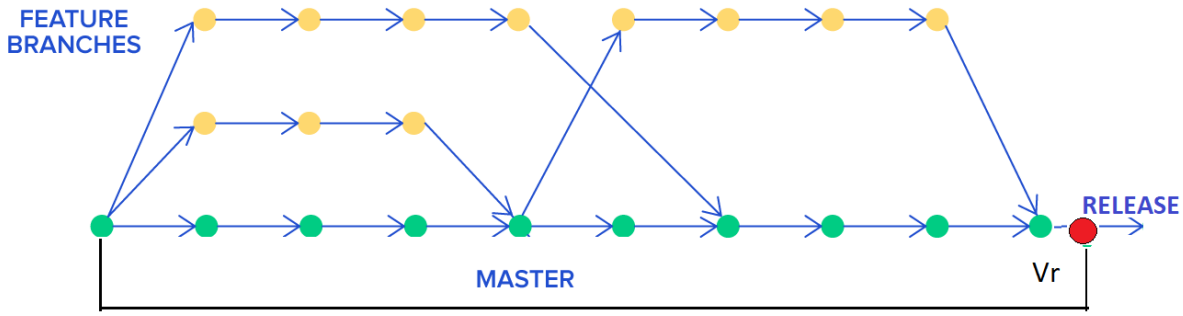

Рисунок 4.7-Початкова точка виявлення дефектів при мануальном тестуванні.

Опишемо формулу для обчислення часу на виявлення дефектів при мануальному тестуванні.

$$Vм=Vr=Vn+Vb+Tм \text{ (год)}$$

де:

Vм- час виявлення дефектів мануально;

Vr- час від початку розробки до випуску продукту;

Vn - час мержу останнього бранчу;

Vb - час на збирання версії продукту;

Тм- загальний час тестування API мануально.

Точки виявлення дефектів при автоматичном тестуванні відображено на рисунку 4.8.

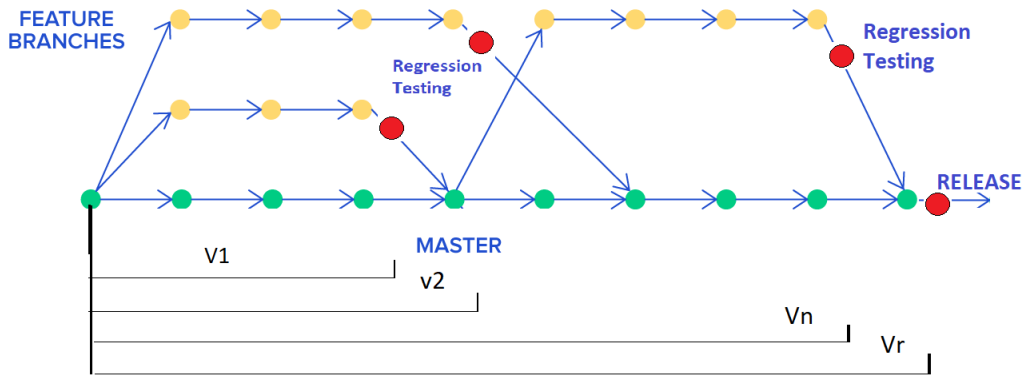

Рисунок 4.8-Точки виявлення дефектів при при автоматичном тестуванні.

Опишемо формули для обчислення часу виявлення дефектів при автоматичному тестуванні.

При автоматичному тестуванні час виявлення дефекту буде залежати від місця й часу його виникнення.

Va=Vr=Vn+Vb+Ta1 (год) (год)- під час прийомочного тестування;

Va=V(x)+Ta1 (год)- під час тестування бранчу;

де:

Va- час виявлення дефектів автоматично;

V(x) - час бранчу;

Ta- загальний час тестування API автоматично.

Порівняємо час на виявлення дефектів за їх формулами.

Для порівняння розглянемо випадок де дефект знайдено під час прийомочного тестування.

Vм=Vn+Vb+Tм1 (год)

Va=Vn+Vb+Ta1 (год)

=>

Vм-Va=(Vn+Vb+Tм1)-(Vn+Vb+Ta1)

=>

Vм-Va=Tм-Ta =53.957 (год)

Темп виявлення дефектів є обернено пропорційним до часу виявлення дефектів, і може бути обчислений так:

- при мануальном темп виявлення дефектів дорівнює $\frac{1}{\text{Тм1}} => \frac{1}{53.3}$;
- при автоматичному тестуванні темп виявлення дефектів дорівнює $\frac{1}{\text{Ta1}} => \frac{1}{1.62}$;

Підрахуємо на скількі відсотків підвищилась ефективність виявлення дефектів

100%-(153.3:11.62*100%)=100%-3,04%=96,96%

В результаті отримали пришвидшення темпу виявлення дефектів на 96,96%

Можемо зробити висновки, що темп виявлення дефектів при автоматичному тестуванні завжди буде значно швидше темпу виявлення дефектів при мануальному

тестуванні, що призведе до зменшення загальних витрат на їх виправлення і часу на випуск продукту. А також дозволить починати процесс тестування на ранніх етапах розробки без негативного впливу на його швидкість.

Розробка формул для обчислення трудомісткісті знайдення дефектів під час мануального і автоматичного тестування і їх порівняння.

Початкова точка виявлення дефектів при мануальном тестуванні відображено на рисунку 4.9.

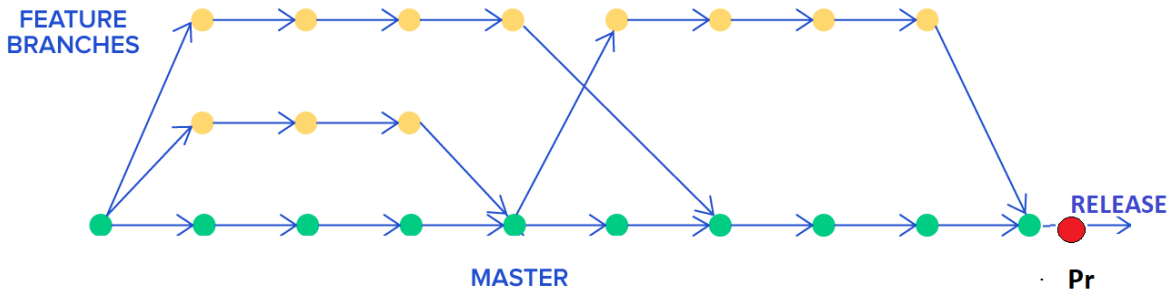

Рисунок 4.9-Початкова точка виявлення дефектів при мануальном тестуванні.

Опишимо формулу підрахунку трудомісткості виявлення дефектів при мануальном тестуванні.

$$Рм=Pr=\frac{Ем}{k}\ (люд.год/деф.)$$

де:

Рм- трудомісткість виявлення дефектів мануально;

Pr- трудомісткість виявлення дефектів під час прийомочного тестування;

Ем- загальна трудомісткість тестування API мануально;

k - кількість знайдених дефектів.

Точки виявлення дефектів при автоматичном тестуванні відображено на рисунку 4.10.

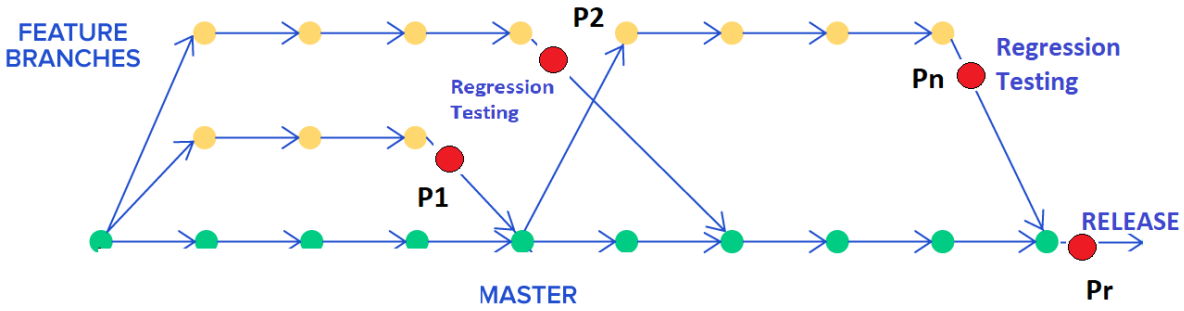

Рисунок 4.10-Точки виявлення дефектів при при автоматичном тестуванні.

Розробимо формулу підрахунку трудомісткості виявлення дефектів при автоматичному тестуванні.

При автоматичному тестуванні трудомісткість виявлення дефектів буде залежати від кількості бранчів, які були протестованні .

$$Pa=Pr=\frac{Ea}{k}$$ (люд.год/деф.) -дефект знайдено під час прийомочного тестування;

$$Pa=P1+P2..+Pn+Pr=\frac{Ea}{k}*n$$ (люд.год/деф.) -дефекти знайденно під час тестування бранчів;

де:

Pa- трудомісткість виявлення дефектів автоматично;

Pr- трудомісткість виявлення дефектів під час прийомочного тестування;

Ea- загальна трудомісткість тестування API автоматично (~0.001 годин);

k - кількість знайдених дефектів.

Порівняємо трудомісткості виявлення дефектів для мануального і автоматичного тестування за їх формулами.

Для порівняння розглянемо випадок де було протестовано n кількість бранчів і було знайдено k дефектів під час прийомочного тестування.

$$Pм=Pr=\frac{Eм}{k}$$ (люд.год/деф.)

$$Pa=P1+P2..+Pn+Pr=\frac{Ea}{k}*n$$ (люд.год/деф.)

=>

$$Рм\text{-}Ра=\frac{Ем}{k}-\frac{Еа}{k}*n$$

=>

$$Рм\text{-}Ра=\frac{55.9}{k}-\frac{0}{k}*n$$

=>

$$Рм\text{-}Ра=\frac{55.9}{k}$$ (люд.год/деф.)

В результаті отримаємо що трудомісткість виявлення дефектів при мануальном тестуванні буде $\frac{55.9}{k}$люд.год/деф і буде завжди перевищувати трудомісткість виявлення дефектів при автоматичному тестуванні що дорівнює $\frac{0.001}{k}*n$ люд.год/деф. і буде обернено пропорційною до трудомісткість виявлення дефектів.

Так як ефективність виявлення дефектів є обернено пропорційною до трудомісткості цього процесу, то її можна обчислити так:

- при мануальном тестуванні ефективність виявлення дефектів дорівнює $\frac{k}{Ем1}=>\frac{k}{53,3}$;

- при автоматичному тестуванні ефективність виявлення дефектів дорівнює $\frac{k}{Еа1}=>\frac{k}{0,001}$.

Підрахуємо на скількі відсотків підвищилась ефективність виявлення дефектів

$$(100\%-(\frac{k}{0,001}:\frac{k}{53,3}*100\%)=100\%-0,002\%=99,998$$

В результаті отримали покращення еффективності на 99,998%

За цими результатами можно зробити висновок, що автоматизоване тестування, порівняно з мануальним, характеризується значно вищою ефективністю у виявленні помилок. Це, у свою чергу, сприяє зниженню трудових витрат та вартості кінцевого продукту.

Дуже низька трудомісткість у автоматичному тестуванні дозволяє розпочати процес перевірки якості програмного продукту на більш ранніх стадіях розробки, не впливаючи при цьому негативно на його ефективність. Це важливо, оскільки раннє виявлення та усунення помилок може значно знизити загальні витрати на розробку та підтримку продукту.

Результати порівняння темпу і ефективності виявлення дефектів для одного циклу тестування відобразимо у таблиці 4.4

Таблиця 4.4 - Результати порівняння темпу і ефективністі виявлення дефектів для одного циклу тестування.

Тип тестування	Темп виявлення дефектів (1/год)		Ефективність виявлення дефектів (дефект/люд. год)	
	Позначення	Значення	Позначення	Значення
Мануальне	$\frac{1}{\text{Тм1}}$	$\frac{1}{53.3}$	$\frac{k}{\text{Ем1}}$	$\frac{k}{53.3}$
Автоматичне тестування	$\frac{1}{\text{Та1}}$	$\frac{1}{1.62}$	$\frac{k}{\text{Еа1}}$	$\frac{k}{0.001}$
Порівняння	$\frac{1}{\text{Тм1}} < \frac{1}{\text{Та1}}$	$\frac{1}{53.3} < \frac{1}{1.62}$	$\frac{k}{\text{Ем1}} < \frac{k}{\text{Еа1}}$	$\frac{k}{53.3} < \frac{k}{0.001}$
Результат	Підвищился на 96,96%		Підвищилась на 99.9%	

Висновки до четвертого розділу

На основі проведеного дослідження було отримано наступні результати:

1) Було успішно розроблено і впроваджено Python-скрипт для оцінки покриття API автотестами. Цей скрипт скоротив час тестування з 2.6 години до 18 хвилин, і також ефективно інтегрується в загальний процес автоматичного

тестування програмного забезпечення і може бути адаптований та модифікований для подальшого удосконалення.

2) Встановлено, що впровадження автоматизації дозволило значно скоротити час та зусилля, необхідні для тестування функціоналу API. Процес тестування став швидшим на 51.68 годин і вимагає на 53.3 людино-години менше на кожен цикл тестування.
3) Доказано, що використання Python-скрипта сприяло значному скороченню часу та зусиль, необхідних для оцінки покриття API автотестами. Цей процес займає на 2.3 години та на 2.6 людино-годин менше на кожен цикл тестування.
4) Встановлено, що спільне використання автоматизації тестування і Python-скрипта є ефективним для прискорення темпу виявлення дефектів та підвищення загальної ефективності тестування функционалу API та його покриття автотестами. Це може призвести до зниження загального часу тестування на приблизно 54 години та зменшення трудомісткості на 55.9 людино-годин за кожен цикл тестування.
5) Підтверджено, що впровадження автоматизованого тестування на проекті є доцільним завдяки прискоренню темпу виявлення дефектів на 96,96% та підвищенню ефективності тестування функціоналу API на 99.9%, особливо при регулярному запуску автотестів на ранніх етапах розробки програмного продукту.

ВИСНОВКИ

В ході виконання дипломної роботи було досягнуто ряд важливих результатів у галузі автоматизованого тестування програмного забезпечення.

Основним досягненням є розробка ефективного інструменту на мові Python для автоматичного підрахунку покриття API автотестами. Цей інструмент спрощує процес оцінки покриття тестами, і значно скорочує його час.

Аналіз існуючих методів та інструментів для тестування показав, що на сьогоднішній день більшість рішень зосереджена на виконанні конкретних тестових сценаріїв, в той час як контроль покриття API автотестами залишається менш розробленим аспектом. В результаті роботи над скриптом було досягнено цілей що до забезпечення:

- аналізу і контролю покриття API автотестами. Скрипт автоматизує процес визначення покриття коду API автотестами, значно скорочуючи час та зусилля, необхідні для цієї задачі;
- -інтеграції скрипта у загальну систему автоматизованого тестування програмного забезпечення. Розроблений інструмент є сумісним з іншими інструментами автоматизації, і інтегрован у процеси CI/CD, що дозволяє автоматично виконувати тестування і аналіз у реальних проектах.

Для перевірки ефективності розробленого інструменту було проведено випробування на реальному проекті компанії Apriorit. Результати показали значне зниження часових та ресурсних витрат на тестування. Зокрема, час тестування скоротився з 2.6 години до 18 хвилин, а процес оцінки покриття API автотестами став на 2.3 години та на 2.6 людино-годин менше за кожен цикл.

Отже, мета дипломної роботи була повністю досягнута. Розроблений інструмент демонструє високу працездатність та може бути адаптований та модифікований для різних потреб у галузі програмної інженерії.

Напрямки подальших досліджень включають:

- розвиток методів для автоматичного розпізнавання особливостей різних API, що дозволить подальше вдосконалення точності оцінки покриття;

- адаптація інструменту для використання з іншими інстументами призначеними для тестуваня API, такими як: Postman, REST-assured , розширюючи його

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Курс лекцій "TM Course_2023" компанії Apriorit для внутрішнього користування
2. Lisa Crispin, Janet Gregory. "Agile Testing: A Practical Guide for Testers and Agile Teams;" Addison-Wesley Professional, 2009.
3. Elfriede Dustin. "Effective Software Test Automation: Developing an Automated Software Testing Tool;" Sybex, 2004.
4. Cem Kaner, James Bach, Bret Pettichord. "Lessons Learned in Software Testing: A Context-Driven Approach;" Wiley, 2001.
5. Rex Black. "Critical Testing Processes: Plan, Prepare, Perform, Perfect;" Addison-Wesley Professional, 2003.
6. Mark Fewster, Dorothy Graham. "Software Test Automation: Effective Use of Test Execution Tools;" Addison-Wesley, 1999.
7. James A. Whittaker. "Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design;" Addison-Wesley Professional, 2009.
8. Dustin, Elfriede, et al. "Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality;" Addison-Wesley Professional, 2009.
9. ISTQB Certified Tester - Foundation Level Syllabus v4.0
10. Сайт з описом Trunk Based Development https://trunkbaseddevelopment.com/
11. ТЕЗИ XVII Міжнародної науково-практичної конференції «СУЧАСНІ ІНФОРМАЦІЙНІ ТА КОМУНІКАЦІЙНІ ТЕХНОЛОГІЇ НА ТРАНСПОРТІ, В ПРОМИСЛОВОСТІ ТА ОСВІТІ»

ДОДАТКИ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖУЮ

Перший проректор Дніпровського національного університету залізничного транспорту імені академіка В. Лазаряна

Борис БОДНАР

СИСТЕМА ВИЗНАЧЕННЯ ВІДПОВІДНОСТІ ТЕКСТУ ПРОГРАМИ ГРАФІЧНОМУ ПРЕДСТАВЛЕННЮ АЛГОРИТМУ

Технічне завдання

ЛИСТ ЗАТВЕРДЖЕННЯ

1116130.01222-01-ЛЗ

Завідувач кафедри КІТ

Вадим ГОРЯЧКІН

Керівник розробки

Олена КУРОП'ЯТНИК

Виконавець

Богдан ЯКОВЕНКО

Нормоконтролер

Олена КУРОП'ЯТНИК

2021

ДОДАТОК А
Технічне завдання

ЗАТВЕРДЖУЮ
Проректор
Українського державного
університету науки і технології
Анатолій РАДКЕВИЧ

ІНСТРУМЕНТ ДЛЯ ОЦІНКИ І МОНІТОРИНГУ РІВНЯ ТЕСТОВОГО ПОКРИТТЯ API АВТОТЕСТАМИ

Технічне завдання
ЛИСТ ЗАТВЕРДЖЕННЯ
1116130.01259-01-ЛЗ

Завідувач кафедри КІТ
_____Вадим ГОРЯЧКІН
Керівник розробки
_____Олена КУРОП’ЯТНИК
Виконавець
_____Юлія ВОДЯНІК
Нормоконтролер
_____Світлана ВОЛКОВА

2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖЕНО

1116130.01259-01

ІНСТРУМЕНТ ДЛЯ ОЦІНКИ І МОНІТОРИНГУ РІВНЯ ТЕСТОВОГО ПОКРИТТЯ API АВТОТЕСТАМИ

Технічне завдання

Аркушів 12

2024

АНОТАЦІЯ

Документ 1116130.01259-01 ««Інструмент для оцінки і моніторингу рівня тестового покриття API автотестами». Технічне завдання» є частиною програмної документації для Python-скрипта. Цей скрипт дозволяє швидко оцінити і моніторити рівень тестового покриття API автотестами. В документі описано цілі та сферу використання інструменту, основні вимоги до нього, етапи розробки проекту та строки, а також технічні характеристики, які пред'являються до програмного продукту.

ЗМІСТ

ВСТУП

Сучасний розвиток програмної інженерії акцентує увагу на постійному покращенні якості програмного забезпечення. Автоматизоване тестування, як один із етапів розробки, відіграє ключову роль у забезпеченні високої якості.

На сьогоднішній день процес автоматизованого тестування є обов'язковою частиною розробки ПЗ, і часто потребує значних ресурсних витрат. Зокрема, перевірка покриття API автотестами вимагає чіткого підходу та інструментарію, який би дозволяв швидко визначити і контролювати рівень тестового покриття.

Оптимізація цього процесу може стати вирішальним фактором в розробці програмного забезпечення. Розвиток методів та інструментів для автоматичного підрахунку покриття дозволить зменшити час тестування, підвищити його якість та, як результат, прискорити вивід продукту на ринок.

Більшість існуючих інструментів зосереджена на виконанні конкретних тестових сценаріїв, проте контроль покриття API автотестами залишається відкритим питанням.

У даній роботі пропонується новий підхід та інструментарій, що дозволить легко автоматизувати цей процес, зменшивши ручний вклад спеціалістів.

Це дослідження є частиною програми, спрямованої на покращення якості продуктів, а также також вдосконалення методів та технік тестування програмного забезпечення. Результати дослідження плануються до впровадження автоматичного тестування в компанії Apriorit.

1 ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є наказ № 274 ст. від 23.03.2023 в. о. ректора Українського державного університету науки і технологій «Про затвердження керівників та затвердження тем магістерських робіт».

Тема дипломної роботи: Дослідження і розробка засобів автоматизованого тестування програмного забезпечення.

Керівник дипломної роботи – Куроп’ятник О. С.

2 ПРИЗНАЧЕННЯ РОЗРОБКИ

2.1 Функціональне призначення

Метою даного інструменту є обчислення і моніторинг рівня покриття API автотестами.

Функціональні вимоги до додатку:

- завантаження специфікації Swagger з вказаного URL у форматі JSON;
- завантаження з JMeter .jmx файлу переліком автотестів у специфічному форматі XML;
- розбір та очищення обох специфікації від зайвих даних;
- порівняння переліків ендпоінтів між специфікацією Swagger і специфікацією і .jmx файлу;
- визначення ендпоінтів які покрити а які ні;
- складання переліків покритих і непокритих ендпоінтів;
- обчислення, у відсотках, покриття API автотестами;
- генерація звіту з рівнем покриття та переліками покритих і непокритих ендпоінтів.

Відповідно до функціональних вимог додатку було визначено наступні вимоги до даних програми:

- Swagger специфікація у форматі JSON;
- .jmx файл переліком автотестів у специфічному форматі XML.

2.2 Експлуатаційне призначення

Основне призначення програмного додатку полягає в автоматизації процесу обчислення показника рівня покриття API автотестами.

3 ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

3.1 Вимоги до функціональних характеристик

Вимоги до функціональних характеристик наступні:

- завантаження специфікації Swagger з вказаного URL;
- завантаження специфікації з .jmx файлу;
- обчислення покриття API автотестами шляхом порівняння специфікації Swagger і .jmx файлу;
- генерація HTML звіту для користувача у вигляді переліку покритих і непокритих ендпоінтів та процент покриття API автотестами.

Вхідні дані:

- файл .json зі Swagger специфікацією;
- файл .jmx з переліком автотестів.

Вихідні дані:

- перелік покритих і непокритих ендпоінтів;
- процент покриття API автотестами.

3.2 Вимоги до надійності

Вимоги до надійності програмного додатку наступні:

- програма має запобігати будь-яким пошкодженням даних під час роботи;
- програма має забезпечувати захист від непередбачених втрат пам'яті;
- програма має демонструвати стабільну роботу, при цьому частота збоїв системи не має перевищувати один раз на кожні 2000 запусків системи.

3.3 Умови експлуатації

Скрипт призначене для використання в умовах, які відповідають тим, що описані в документі [1].

Для забезпечення надійної роботи цього програмного продукту необхідно врахувати наступні вимоги:

- базові навички використання комп'ютера;;
- користувачі мають бути ознайомлені з інструкцією користувача перед використанням продукту;
- електронно-обчислювальні машини (ЕОМ), що використовуються з цим продуктом, мають відповідати діючим стандартам та нормативам охорони праці в Україні, згідно з документом [2];
- програмний комплекс має бути використаний в приміщеннях, призначених для ЕОМ, з кліматичними умовами: температура 21-25°С і відносна вологість повітря 40-60%.

3.4 Вимоги до складу і параметрів технічних засобів

Для використання скрипту потрібен комп'ютер із такими мінімальними характеристики:

- процесор: двоядерний процесор 64-бітний із швидкістю 2.5 ГГц;
- оперативна пам’ять: 16ГБ;
- місце на жорсткому диску: 250 МБ;
- додаткові пристрої: монітор, клавіатура, миша.

3.5 Вимоги до інформаційної і програмної сумісності

Для роботи скрипта має бути:

- встановленний Python 3.9.5 разом з додатковими бібліотеками argparse і jinja2;
- 10 Кб дискового простору для розміщення скрипту.

3.6 Вимоги до маркування і упаковки

Якщо виникає потреба у розповсюдженні цього інструменту у фізичній формі, то як скрипт, так і електронна інструкція для користувачів мають бути розміщені на USB-накопичувачі.

Приклад маркування можна знайти на рисунку 3.1:

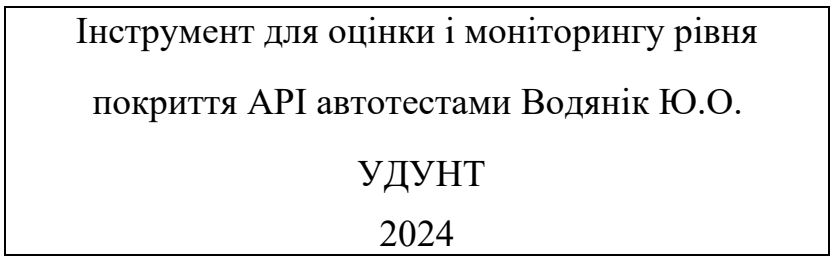

Рисунок 3.1 – Приклад маркування

3.7 Вимоги до транспортування і зберігання

Транспортування програмного продукту має забезпечити його збереження, недоторканність та захист від неавторизованого доступу. Переміщення фізичної копії продукту має відбуватися під наглядом довіреної особи та в комплектах у спеціальній упаковці, яка складається з термоплівки та захисної піни, щоб уникнути пошкоджень. Місце зберігання має бути сухим і без пилу, з температурою 21-25°C та вологістю 40-60%.

4 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

Програмна документація представляє собою перелік наступних документів:

- специфікація;
- текст програми;
- опис програми;
- керівництво користувача. Керівництво для оцінки і моніторингу рівня тестового покриття API автотестами

Усі документи, пов'язані з програмою, повинні відповідати стандартам державного ГОСТ 19.101-77 «Єдина система програмної документації. Види програм та програмних документів» [3].

5 СТАДІЇ ТА ЕТАПИ РОЗРОБКИ

Стадії розробки наведені в табл. 5.1.

Таблиця 5.1 – Стадії та етапи розробки

Стадії	Зміст робіт	Терміни виконання
1	2	3
1. Створення технічного завдання	Постановка задачи Попередні дослідження Встановлення та обґрунтування необхідності розробки інструменту Розробка структури вхідних та вихідних даних Вибір методів для вирішення задачі Аргументація використання існуючих програм Встановлення технічних вимог Розробка вимог до інструменту Планування стадій, етапів і термінів розробки інструменту та документації Вибір мови програмування Узгодження і затвердження технічного завдання.	27.11.2023 – 01.12.2023
2. Розробка і випробування інструменту	Написання та налаштування скрипту Створення документації відповідно до стандартів ГОСТ 19.101-77 Розробка і узгодження інструменту методів його роботи та тестування Проведення випробувань інструменту	01.12.2023 - 24.12.2023
3. Впровадження інструменту	Підготовка і передача програми і програмної документації для супроводу і (або) виготовлення	24.12.2023 - 14.01.2024

6 ПОРЯДОК КОНТРОЛЮ І ПРИЙМАННЯ

Контроль за виконанням роботи виконує керівник розробки.

Прийом програмного продукту здійснюється прийомною комісією і керівником розробки.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. ДСанПіН 3.3.2-007-98. Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин [Текст] / Постанова Головного державного санітарного лікаря України від 10 грудня 1998 р. № 7 – К.,1998.
2. ДСН 3.3.6.042-99. Санітарні норми мікроклімату виробничих приміщень [Текст]/ Постанова Головного Державного санітарного лікаря України від 01.12.1999 № 42 – К., 1999.
3. ГОСТ 19.101-77. Виды програм и программных документов [Текст]/ Постановление Государственного комитета стандартов Совета Министров СССР от 20 мая 1977 г. – М., 1977.

ДОДАТОК Б
Рабочий проект

ЗАТВЕРДЖУЮ
Проректор
Українського державного
університету науки і технології
Анатолій РАДКЕВИЧ

ІНСТРУМЕНТ ДЛЯ ОЦІНКИ І МОНІТОРИНГУ РІВНЯ ТЕСТОВОГО ПОКРИТТЯ API АВТОТЕСТАМИ

Рабочий проект
ЛИСТ ЗАТВЕРДЖЕННЯ
1116130.01259-01-ЛЗ

Завідувач кафедри КІТ
_____Вадим ГОРЯЧКІН
Керівник розробки
_____Олена КУРОП'ЯТНИК
Виконавець
_____Юлія ВОДЯНІК
Нормоконтролер
_____Світлана ВОЛКОВА

2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖЕНО

1116130.01259-01

ІНСТРУМЕНТ ДЛЯ ОЦІНКИ І МОНІТОРИНГУ РІВНЯ ТЕСТОВОГО ПОКРИТТЯ API АВТОТЕСТАМИ

Рабочий проект

Аркушів 12

2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖЕНО

1116130.01259-01

ІНСТРУМЕНТ ДЛЯ ОЦІНКИ І МОНІТОРИНГУ РІВНЯ ТЕСТОВОГО ПОКРИТТЯ API АВТОТЕСТАМИ

Специфікація

Аркушів 2

2024

Позначення	Найменування	Примітка
1116130.01259-01-ЛЗ	Лист затвердження	
1116130.01259-01 12 01-ЛЗ	Лист затвердження	
1116130.01259-01 12 01	Текст програми	
1116130.01259-01 13 01-ЛЗ	Лист затвердження	
1116130.01259-01 13 01	Опис програми	
1116130.01259-01 І3 01-ЛЗ	Лист затвердження	
1116130.011259-01 І3 01	Керівництво користувача. Керівництво для оцінки і моніторингу рівня тестового покриття API автотестами	

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖЕНО

1116130.01259-01 13 01

ІНСТРУМЕНТ ДЛЯ ОЦІНКИ І МОНІТОРИНГУ РІВНЯ ТЕСТОВОГО ПОКРИТТЯ API АВТОТЕСТАМИ

Опис програми

Аркушів 15

2024

АНОТАЦІЯ

Документ 1116130.01259-01 13 01 ««Інструмент для оцінки і моніторингу рівня тестового покриття API автотестами». Опис програми» є частиною програмної документації для Python-скрипта. Даний скрипт дозволяє швидко оцінити і моніторити рівень тестового покриття API автотестами.

Цей документ містить детальний опис клієнтської частини системи, включаючи її функціональне призначення, логічну структуру, використовуване технічне обладнання, процеси виклику та завантаження, вхідні та вихідні параметри та інструкції щодо користування програмою.

ЗМІСТ

1 ЗАГАЛЬНІ ВІДОМОСТІ

Програмний додаток «Інструмент для оцінки і моніторингу рівня тестового покриття API автотестами» являє собою скрипт для порівняння між собою даних Swagger специфікації та .jmx файлу з переліком автотестів із JMeter, та обчислює рівень покриття і створює звіт, в якому відображається процент покриття з переліком покритих та не покритих ендпоінтів тестами.

Для коректного функціонування програми необхідно щоб на персональному комп’ютері було встановлено:

- операційну систему Windows 8 або вище;

- Python 3.9.5 разом з додатковими бібліотеками argparse і jinja2.

Програма реалізована на мові програмування Python. Swagger специфікація має формат даних у JSON. Специфічний XML формат має .jmx файл з переліком автотестів. Звіт у форматі HTML.

2 ФУНКЦІОНАЛЬНЕ ПРИЗНАЧЕННЯ

Метою даного інструменту є обчислення і моніторинг рівня покриття API автотестами.

Функціональні вимоги до додатку:

- завантаження специфікації Swagger з вказаного URL у форматі JSON;
- завантаження з JMeter .jmx файлу переліком автотестів у специфічному форматі XML;
- розбір та очищення обох специфікації від зайвих даних;
- порівняння переліків ендпоінтів між специфікацією Swagger і специфікацією і .jmx файлу;
- визначення ендпоінтів які покрити а які ні;
- складання переліків покритих і непокритих ендпоінтів;
- обчислення, у відсотках, покриття API автотестами;
- генерація звіту з рівнем покриття та переліками покритих і непокритих ендпоінтів.

Відповідно до функціональних вимог додатку було визначено наступні вимоги до даних програми:

- Swagger специфікація у форматі JSON;
- .jmx файл переліком автотестів у специфічному форматі XML..

3 ОПИС ЛОГІЧНОЇ СТРУКТУРИ

3.1 Алгоритм та методи скрипту

Python-скрипт складається з наступних ключових логічних блоків:

- парсинг аргументів командного рядка;
- вичитка специфікації Swagger і переліку автотестів з файлу .jmx;
- очищення даних із специфікації Swagger та файлу .jmx;
- аналіз та порівняння отриманих даних за допомогою базового ітеративного проходження структури даних, та створення списків ендпоінтів, покритих та не покритих автотестами та розрахунок рівня покриття API автотестами;
- генерація звіту про рівень покриття API автотестами, і переліком ендпоінтів, що покриваються та не покриваються автотестами.

Парсинг аргументів командного рядка приймає шлях до JMX-файлу, URL специфікації Swagger та шлях для збереження звіту про покриття API.

Вичитка та аналіз специфікації Swagger і переліку автотестів з файлу .jmx складається з наступних етапів:

- очищення URL-адреси. Функція clean_url видаляє версійність і параметри з URL-адреси swagger, перетворюючи їх на уніфікований формат;
- завантаження та очищення специфікації Swagger. Скрипт завантажує та очищає оригінальну специфікацію Swagger, залишаючи лише необхідні дані;
- завантаження та екстракція API викликів з .jmx файлу. Скрипт читає .jmx файл і виділяє тільки унікальні виклики API, очищаючи їх від зайвих параметрів та дублікатів.

Очищення даних із специфікації Swagger та файлу .jmx включає наступні етапи:

– очищення і уніфікація шляхів ендпоінтів у Swagger документі. Цей етап видаляє зайві версії та параметри з шляхів ендпоінтів у Swagger документі, залишаючи тільки шляхи ендпоінтів;

– очищення API викликів з .jmx файлу і генерація документу у swagger форматі. Скрипт створює спрощену уніфіковану версію Swagger-документа, що базується на викликах API, визначених у .jmx файлі.

Аналіз та порівняння отриманих даних за допомогою базового ітеративного проходження структури даних, та створення списків ендпоінтів, покритих та не покритих автотестами та розрахунок рівня покриття API автотестами включає наступні етапи:

– порівняння ендпоінтів. Скрипт порівнює ендпоінти зі специфікації Swagger з даними уніфікованої версії Swagger-документа, що містить виклики API з .jmx файлу, визначаючи, які з них покриті тестами;

– обчислення рівня покриття у відсотках. Розрахунок відсотка покриття API на основі загальної кількості ендпоінтів і кількості покритих.

Генерація звіту про рівень покриття API автотестами, і переліком ендпоінтів, що покриваються та не покриваються автотестами, створює звіт у форматі HTML, що відображає у відсотках покриття API автотестами, і переліки покритих та непокритих ендпоінтів що виділені зеленим та червоним кольором відповідно.

3.2 Опис функцій скрипту.

На рис. 3.1 відображена схема скрипта та його функції.

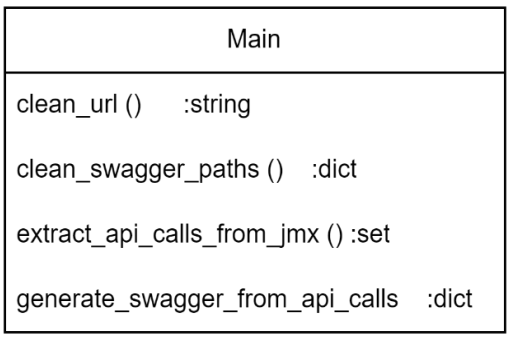

Рисунок 3.1 – Схема скрипта та його функції

Нижче представлений опис функцій скрипту.

Головний модуль Main - Представляє точку входу в скрипт, де обробляються аргументи командного рядка і викликаються функції:

- clean_url, якиа обробляє URL-адреси для їх стандартизації.

- clean_swagger_paths, яка очищує та стандартизує шляхи у специфікації Swagger.

-extract_api_calls_from_jmx, що вилучає інформацію про виклики API з JMX файлів.

- generate_swagger_from_api_calls, яка використовується для генерації специфікації Swagger на основі викликів API, отриманих з JMX файлів.

3.3 Зв’язки інструменту з іншими програмами

Для коректної роботи програмного додатку необхідні такі програми:

− операційна система Windows 8 або більше;

− інтерпретатор Python 3.9.5 або більше зі встановленими бібліотеками:

- xml.etree.ElementTree для роботи з XML,
- json для обробки даних JSON,
- re для регулярних виразів,
- argparse для обробки аргументів командного рядка,
- requests для здійснення HTTP-запитів,
- jinja2 для рендерингу HTML-шаблонів.

4 ВИКОРИСТАНІ ТЕХНІЧНІ ЗАСОБИ

Для коректного функціонування програмного забезпечення достатньо мати робочий персональний комп’ютер, що має наступні технічні характеристики:

− процесор з тактовою частотою 800 Мгц;

− не менше 256 Мб ОЗУ;

− 100 Мб вільного місця на жорсткому диску;

− архітектура x86 або x64;

− периферійні пристрої – VGA-монітор з мінімальним розширенням екрану 1024x768, клавіатура, миша.

5 ВИКЛИК ТА ЗАВАНТАЖЕННЯ

Для запуску коду скрипта необхідно:

1) створити текстовий файл, скопіювати в нього код скрипта;
2) зберегти файл з розширенням .py, наприклад, api_analysis.py;
3) запустити скрипт через командний рядок:
   - відкриємо командний рядок;
   - переходимо до директорії, де збережений файл скрипта. Це можна зробити за допомогою команди cd, наприклад:

     cd <path/to/directory>
   - запустіть скрипт з аргументами:

     -- f - шлях до файлу JMX;

     -- s - URL Swagger специфікації;

     -- r - шлях для збереження звіту про покриття API.

     Приклад команди запуску скрипта:

python api_analysis.py -f path_to_file.jmx -s https://example.com/swagger.json -r api_coverage_report.html

За результатом виконання цього скрипта, звіт з результатами обчислення покриття API автотестами має бути збережений зберігтись за вказаним у аргументі шляхом.

6 ВХІДНІ ДАНІ

Вхідні дані програмного продукту:

- мова скрипту Python;
- файл .py з програмним кодом на мові програмування Python;
- файл .jmx з переліком автотестів для API;
- URL до Swagger специфікації з описом ендпоінтів API.

7 ВИХІДНІ ДАНІ

Вихідними даними програмного продукту є звіт у форматі HTML, що відображає у відсотках покриття API автотестами, і переліки покритих та непокритих ендпоінтів що виділені зеленим та червоним кольором відповідно.

8 ІНТЕРФЕЙС КОРИСТУВАЧА

Інтерфейс користувача для Python скрипта відсутній так як скрипт виконує автономну задачу, яка не вимагають інтерактивності з користувачем після запуску. Параметри, які потрібно вказати, передаються одноразово при запуску через командний рядок, і далі скрипт працює автономно. Відсутність UI також обумовлена необхідністю легко інтегрувати скрипт в існуючий процес автоматизації без необхідності взаємодії з графічним інтерфейсом.

9 ПОРЯДОК РОБОТИ ЗІ СКРИПТОМ

Порядок роботи зі скриптом передбачає наступну послідовність операцій:

- завантаження або створення файлу з текстом скрипта мові Python та розширенням .py;
- завантаження .jmx файлу з переліком автотестів для API;
- запуск скрипта з параметрами;
- перевірка даних у HTML звіті.

10 ПОВІДОМЛЕННЯ

У табл. 10.1 наведені повідомлення користувачу, що можуть з’явитися під час роботи з програмою.

Таблиця 10.1 – Повідомлення користувачу

Текст повідомлення	Опис ситуації	Рекомендовані дії
Помилка під час витягування API викликів з .jmx з програмним кодом	.jmx файл з переліком автотестів кодом не було відкрито	Перевірити формат .jmx файла за допомогою JMeter.
Помилка під час завантаження Swagger специфікації з програмним кодом	.json файл з переліком ендпоінтів API не було завантажено	Перевірити: - наявність інтеренету; - URL; - доступ за URL.
Помилка під час очищення шляхів Swagger з програмним кодом	.json файл з переліком ендпоінтів API має специфічні данні	Знайти та видалити специфічні дані та перезапустити скрипт
Помилка під час генерації звіту з програмним кодом	Відсутність доступу до директорії де має зберегтись звіт	Змінити шлях до для збереження звіту та перезапустити скрипт

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖЕНО

1116130.01259-01 І3 01

ІНСТРУМЕНТ ДЛЯ ОЦІНКИ І МОНІТОРИНГУ РІВНЯ ТЕСТОВОГО ПОКРИТТЯ API АВТОТЕСТАМИ

Керівництво користувача. Керівництво використання скрипту для оцінки і моніторингу рівня тестового покриття API автотестами

Аркушів 9

2024

АНОТАЦІЯ

Документ 1116130.01259-01 І3 01 ««Інструмент для оцінки і моніторингу рівня тестового покриття API автотестами». Керівництво користувача. Керівництво використання скрипту для оцінки і моніторингу рівня тестового покриття API автотестами» є частиною програмної документації для Python-скрипта. Даний скрипт дозволяє швидко оцінити і моніторити рівень тестового покриття API автотестами.

Цей документ включає опис цілей та умов використання інструменту, рекомендації щодо підготовки до запуску, а також детальний огляд ключових функцій та можливих проблемних ситуацій, які можуть виникнути під час використання скрипту.

ЗМІСТ

ВСТУП

Програмний додаток «Інструмент для оцінки і моніторингу рівня тестового покриття API автотестами» являє собою скрипт для порівняння між собою даних Swagger специфікації та .jmx файлу з переліком автотестів із JMeter, та обчислює рівень покриття і створює звіт, в якому відображається процент покриття з переліком покритих та не покритих ендпоінтів тестами.

Для коректного функціонування програми необхідно щоб на персональному комп'ютері було встановлено:

- операційну систему Windows 8 або вище;

- Python 3.9.5 разом з додатковими бібліотеками argparse і jinja2.

Програма реалізована на мові програмування Python. Swagger специфікація має формат даних у JSON. Специфічний XML формат має .jmx файл з переліком автотестів. Звіт у форматі HTML.

Для коректної роботи зі скриптом необхідно ознайомитися з описом програми та керівництвом користувача.

1 ПРИЗНАЧЕННЯ ТА УМОВИ ЗАСТОСУВАННЯ

1.1 Функціональне призначення інструменту

Метою даного інструменту є обчислення і моніторинг рівня покриття API автотестами.

Функціональні вимоги до додатку:

- завантаження специфікації Swagger з вказаного URL у форматі JSON;
- завантаження з JMeter .jmx файлу переліком автотестів у специфічному форматі XML;
- розбір та очищення обох специфікації від зайвих данних;
- порівняння переліків ендпоінтів між специфікацією Swagger і специфікацією і .jmx файлу;
- визначення ендпоінтів які покрити а які ні;
- складання переліків покритих і непокритих ендпоінтів;
- обчислення, у відсотках, покриття API автотестами;
- генерація звіту з рівнем покриття та переліками покритих і непокритих ендпоінтів.

Відповідно до функціональних вимог додатку було визначено наступні вимоги до даних програми:

- Swagger специфікація у форматі JSON;

.jmx файл переліком автотестів у специфічному форматі XML.

1.2 Вимоги до складу і параметрів технічних засобів

Для коректного функціонування інструменту необхідно мати робочий персональний комп’ютер, що має наступні технічні характеристики:

- процесор з тактовою частотою 800 Мгц;
- не менше 256 Мб ОЗУ;
- 100 Мб вільного місця на жорсткому диску;
- архітектура x86 або x64;

− периферійні пристрої – VGA-монітор з мінімальним розширенням екрану 1024x768, клавіатура, миша.

1.3 Вимоги до інформаційної і програмної сумісності

Для коректного функціонування програми необхідно щоб на персональному комп’ютері було встановлено:

- операційну систему Windows 8 або вище;

- Python 3.9.5 разом з додатковими бібліотеками argparse і jinja2.

2 ПІДГОТОВКА ДО РОБОТИ

Для запуску коду скрипта:

1) створіть текстовий файл скопіювати в нього код скрипта;
2) збережіть файл з розширенням .py, наприклад, api_analysis.py;
3) запустіть скрипт через командний рядок:

- відкрийте командний рядок;
- переходьте до директорії, де збережений файл скрипта. Це можна зробити за допомогою команди cd, наприклад:

  cd <path/to/directory>

- запустіть скрипт з аргументами:

  -- f - шлях до файлу JMX;

  -- s - URL Swagger специфікації;

  -- r - шлях для збереження звіту про покриття API.

  Приклад команди запуску скрипта:

python api_analysis.py -f path_to_file.jmx -s https://example.com/swagger.json -r api_coverage_report.html.

3 ОПИС ОПЕРАЦІЙ

Після запуску, скрипт автоматично виконує всі задані операції. Користувач має наступні можливості взаємодії з інструментом:

- моніторинг процесу виконання скрипту, зокрема перевірка відсутності повідомлень про помилки;
- огляд і аналіз згенерованого скриптом звіту, який створюється за шляхом вказаним при запуску скрипту, і містить інформацію про рівень покриття, а також та переліками покритих і непокритих ендпоінтів.

4 АВАРІЙНІ СИТУАЦІЇ

У випадку виникнення помилок при виконанні скрипта, рекомендується перевірити правильність введених даних і перезапустити скрипт.

Якщо виникають інші нештатні ситуації, слід також закрити програму та звернутися до технічної підтримки або відповідального персоналу за обслуговування інструменту..

5 РЕКОМЕНДАЦІЇ ЩОДО ЗАСВОЄННЯ

1. Параметри для запуску інструменту та приклад команди зображено на рис. 5.1.

Рисунок 5.1 – Параметри для запуску інструменту

2. Приклад успішного завершення скрипту зображено на рис. 5.2.

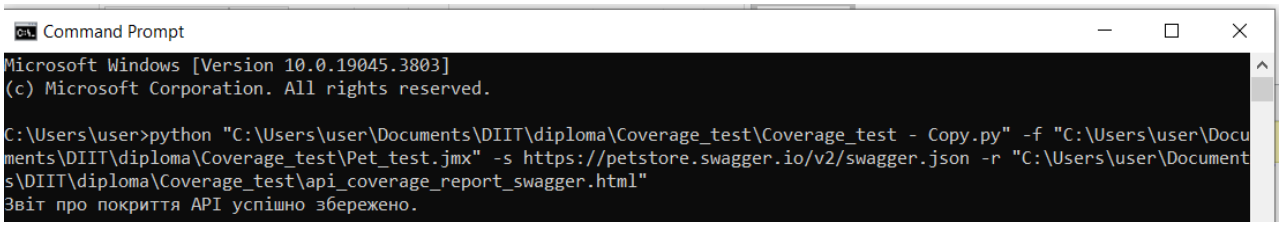

Рисунок 5.2 – Приклад успішного завершення скрипту

3. Приклад типових помилок, які можуть виникнути у разі невдалого завершення виконання скрипту, представлені на рис. 5.3.

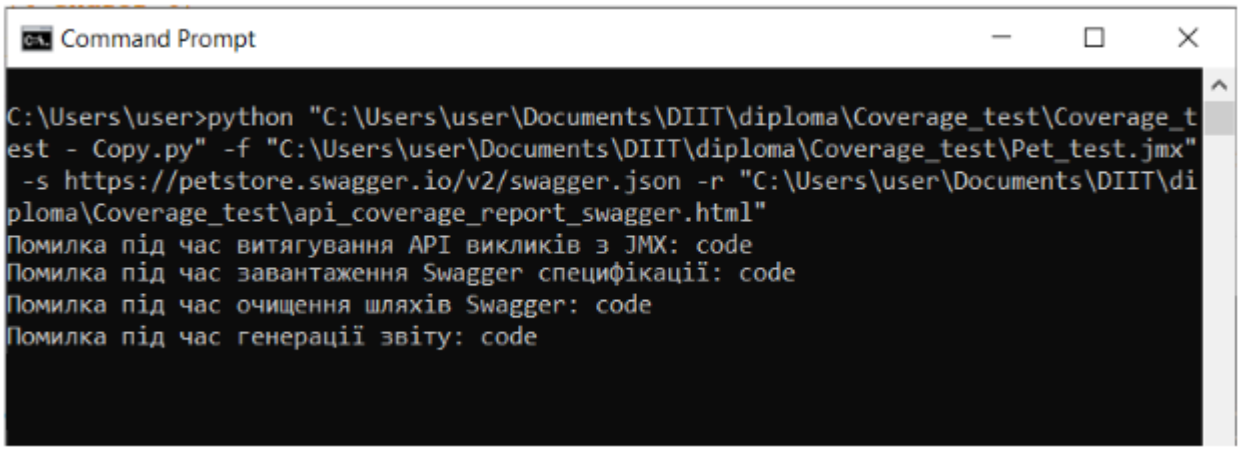

Рисунок 5.3 – Приклад типових помилок

4. Приклад звіту з результатами обчислення у відсотках покриття API автотестами, і переліками покритих та непокритих ендпоінтів, що виділені зеленим та червоним кольором відповідно зображено на рис. 5.4.

API Coverage: 5.0%

Covered Endpoints

- GET /pet/findByStatus

Uncovered Endpoints

- POST /pet/{}/uploadImage
- POST /pet
- PUT /pet
- GET /pet/findByTags
- GET /pet/{}
- POST /pet/{}
- DELETE /pet/{}
- POST /store/order
- GET /store/order/{}
- DELETE /store/order/{}
- GET /store/inventory
- POST /user/createWithArray
- POST /user/createWithList
- GET /user/{}
- PUT /user/{}
- DELETE /user/{}
- GET /user/login
- GET /user/logout
- POST /user

Рисунок 5.4 – Приклад звіту

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖЕНО

1116130.01259-01 12 01

ІНСТРУМЕНТ ДЛЯ ОЦІНКИ І МОНІТОРИНГУ РІВНЯ ТЕСТОВОГО ПОКРИТТЯ API АВТОТЕСТАМИ

Текст програми

Аркушів 9

2024

1116130.01259-01 12 01

АНОТАЦІЯ

Документ 1116130.01259-01 12 01 ««Інструмент для оцінки і моніторингу рівня тестового покриття API автотестами». Текст програми» є частиною програмної документації для Python-скрипта. Даний скрипт дозволяє швидко оцінити і моніторити рівень тестового покриття API автотестами.

Документ містить текст скрипта. Інструмент реалізований на мові Python.

1116130.01259-01 12 01

ЗМІСТ

1116130.01259-01 12 01

1 СТРУКТУРА ПРОГРАМИ

Структура програми обмежується виключно логічним рівнем, оскільки він є компонентом загальної системи автоматизованого тестування програмного забезпечення і, зазвичай, використовується іншими інструментами автоматизації, тому рівень представлення в ньому відсутній. Рівень логіки (рис. 1.1) відповідає за завантаження .jmx файлу з переліком автотестів для API, завантаження Swagger специфікації з описом ендпоінтів API у форматі JSON, очищення .jmx файлу і Swagger специфікації, визначення відповідності між специфікацією Swagger і з даними .jmx файлом, обчислення покриття API, генерація HTML звіту.

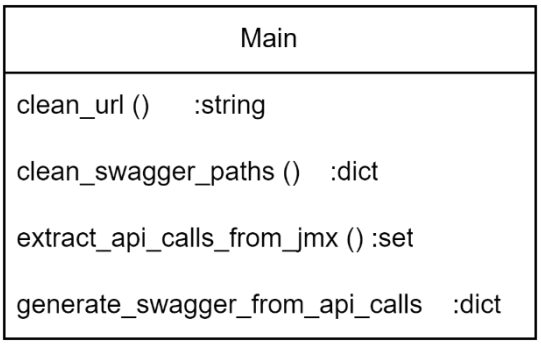

Рисунок 1.1 – Діаграма компонентів для рівня логіки

Інструмент складається тільки з одного файлу скрипта з розширенням .py, наприклад, api_analysis.py. Main є вхідною точкою програми, а інші функції відповідають за виконання всіх інших задач рівня логики.

3 ТЕКСТ ПРОГРАМИ

api_analysis.py

```
import xml.etree.ElementTree as ET
import json
import re
import argparse
import requests
from jinja2 import Template
from requests.packages.urllib3.exceptions import InsecureRequestWarning

requests.packages.urllib3.disable_warnings(InsecureRequestWarning)

def main():
    parser = argparse.ArgumentParser(description="Скрипт для аналізу покриття API на основі файлів JMX та Swagger.")
    parser.add_argument("-f", "--file", type=str, required=True, help="Шлях до файлу JMX. Наприклад: path_to_file.jmx")
    parser.add_argument("-s", "--swagger", type=str, required=True, help="URL Swagger специфікації. Наприклад: https://example.com/swagger.json")
    parser.add_argument("-r", "--report", type=str, required=True, help="Шлях для збереження звіту про покриття API. Наприклад: api_coverage_report.html")

    args = parser.parse_args()

    file_path = args.file
    swagger_url = args.swagger
    report_path = args.report

    def clean_url(url):
```

```
    cleaned_url = re.sub(r"/v[0-9]+", "", url)  # Видалення /vN (де N - число)
    cleaned_url = re.sub(r"/{[^{}]*}", "/{}", cleaned_url)  # Заміна /{data} на /{}
    return cleaned_url

def clean_swagger_paths(swagger):
    clean_paths = {}
    for path, methods in swagger["paths"].items():
        if "/v1/" in path:
            continue

        methods_copy = methods.copy()
        if "parameters" in methods_copy:
            del methods_copy["parameters"]

        if not methods_copy:
            continue

        cleaned_path = re.sub(r'\{.*?\}', '{}', path)
        cleaned_path = re.sub(r"\?.*", "", cleaned_path)
        cleaned_path = re.sub(r'\(single-mode\)', '{}', cleaned_path)
        cleaned_path = re.sub(r'/v[0-9]+/', '/', cleaned_path)

        clean_paths[cleaned_path] = methods_copy

    swagger["paths"] = clean_paths
    return swagger

def extract_api_calls_from_jmx(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
```

```
        content = file.read()

    root = ET.fromstring(content)
    api_calls = set()

    for http_sample in root.findall(".//HTTPSamplerProxy"):
        method = http_sample.find("stringProp[@name='HTTPSampler.method']").text
        path = http_sample.find("stringProp[@name='HTTPSampler.path']").text
        path = re.sub(r'\$\{.*?\}', '{}', path)
        path = re.sub(r'/\d+', '/{}', path)
        path = re.sub(r"\?.*", "", path)
        path = re.sub(r'\(single-mode\)', '{}', path)
        cleaned_path = clean_url(path)
        api_calls.add((method, cleaned_path))

    return api_calls

def generate_swagger_from_api_calls(api_calls):
    swagger = {
        "swagger": "2.0",
        "info": {
            "version": "1.0.0",
            "title": "API from JMX"
        },
        "paths": {}
    }

    for method, path in api_calls:
        if path not in swagger["paths"]:
```

```
            swagger["paths"][path] = {}
        swagger["paths"][path][method.lower()] = {
            "responses": {
                "200": {
                    "description": "Successful response"
                }
            }
        }

    return swagger

try:
    api_calls = extract_api_calls_from_jmx(file_path)
except Exception as e:
    print(f"Помилка під час витягування API викликів з JMX: {e}")
    return

try:
    swagger = generate_swagger_from_api_calls(api_calls)
except Exception as e:
    print(f"Помилка під час генерації Swagger специфікації: {e}")
    return

try:
    with open("swagger_output.json", "w") as outfile:
        json.dump(swagger, outfile, indent=4)
except Exception as e:
    print(f"Помилка під час збереження Swagger специфікації: {e}")
```

```
    try:
        response = requests.get(swagger_url, verify=False)
        swagger_spec = response.json()
    except Exception as e:
        print(f"Помилка під час завантаження Swagger специфікації: {e}")
        return

    try:
        swagger_spec = clean_swagger_paths(swagger_spec)
    except Exception as e:
        print(f"Помилка під час очищення шляхів Swagger: {e}")
        return

    with open("swagger_output.json", "r") as file:
        generated_swagger = json.load(file)

    covered_endpoints = []
    uncovered_endpoints = []

    for path, methods in swagger_spec["paths"].items():
        for method in methods:
            if path in generated_swagger["paths"] and method in
generated_swagger["paths"][path]:
                covered_endpoints.append((method, path))
            else:
                uncovered_endpoints.append((method, path))

    special_endpoints = []
    regular_uncovered_endpoints = []
```

```
    for method, path in uncovered_endpoints:
        if 'uninstall' in path or 'update' in path or 'search' in path:
            special_endpoints.append((method, path))
        else:
            regular_uncovered_endpoints.append((method, path))

    coverage = (len(covered_endpoints) / (len(covered_endpoints) +
len(regular_uncovered_endpoints))) * 100

    html_template = """
    <!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>API Coverage Report</title>
    </head>
    <body>
        <h1>API Coverage: {{ coverage|round(2) }}%</h1>
        <h2>Covered Endpoints</h2>
        <ul>
            {% for method, path in covered_endpoints %}
                <li style="background-color: YellowGreen;">{{ method.upper() }} {{ path
}}</li>
            {% endfor %}
        </ul>
        <h2>Special Covered Endpoints (by other tests)</h2>
        <ul>
```

```
        {% for method, path in special_endpoints %}
            <li style="background-color: Goldenrod;">{{ method.upper() }} {{ path
}}</li>
        {% endfor %}
    </ul>
    <h2>Uncovered Endpoints</h2>
    <ul>
        {% for method, path in regular_uncovered_endpoints %}
            <li style="background-color: Salmon;">{{ method.upper() }} {{ path }}</li>
        {% endfor %}
    </ul>
</body>
</html>
"""

try:
    template = Template(html_template)
    html_report = template.render(
        coverage=coverage,
        covered_endpoints=covered_endpoints,
        special_endpoints=special_endpoints,
        regular_uncovered_endpoints=regular_uncovered_endpoints
    )
except Exception as e:
    print(f"Помилка під час генерації звіту: {e}")
    return

try:
    with open(report_path, "w", encoding="utf-8") as file:
```

```
            file.write(html_report)
        print("Звіт про покриття API успішно збережено.")
    except Exception as e:
        print(f"Помилка під час збереження звіту у файл: {e}")
        return

if __name__ == "__main__":
    main()
```

Міністерство освіти і науки України
Міністерство інфраструктури України

Український державний університет науки та технологій

Східний науковий центр транспортної академії наук

ПКТБ ІТ

TEMPUS: CITISET & SEREIN & CRENG

ТЕЗИ
XVII Міжнародної науково-практичної конференції
«СУЧАСНІ ІНФОРМАЦІЙНІ ТА КОМУНІКАЦІЙНІ ТЕХНОЛОГІЇ НА ТРАНСПОРТІ, В ПРОМИСЛОВОСТІ ТА ОСВІТІ»

ABSTRACTS
of the XVII International Conference
«MODERN INFORMATION AND COMMUNICATION TECHNOLOGIES ON A TRANSPORT, IN INDUSTRY AND EDUCATION»

13.12.2023 – 14.12.2023

Дніпро
2023

УДК 658.512.2:681.3.06

Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості і освіті: Тези XVII Міжнародної науково-практичної конференції (Дніпро, 13-14 грудня 2023 р.). – Д.: УДУНТ, 2023. – 152 с.

У збірнику представлені тези доповідей XVII Міжнародної науково-практичної конференції «Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості та освіті», яка відбулася 13-14 грудня 2023 року в Українському державному університеті науки та технологій в онлайн форматі. Розглянуто результати теоретичних і експериментальних досліджень, а також проблемні питання функціонування та перспективи розвитку інформаційних технологій транспорту, промисловості й освіти.
Збірник призначений для науково-технічних працівників залізниць, підприємств транспорту, викладачів вищих навчальних закладів, докторантів, аспірантів і студентів.

Адреса редакційної колегії:
49010, м. Дніпро, вул. Лазаряна, 2, УДУНТ

Тези доповідей друкуються мовою оригіналу в редакції авторів

ЗМІСТ

Автоматизація оцінки повноти тестування API за допомогою python-скрипту

Водянік Ю. О., Apriorit,

Куроп'ятник О. С., Український державний університет науки та технологій

У сучасному світі програмної інженерії, що характеризується неперервним прогресом, важливими є вимоги до якості та надійності програмного забезпечення (ПЗ). Автоматизація тестування є ключовим аспектом, що впливає на якість, особливо у контексті розвитку хмарних технологій, мобільних додатків та систем великих даних. Автоматизоване тестування API відіграє критичну роль у забезпеченні надійності та безпеки програмних продуктів. Організація процесу та обґрунтований вибір інструментарію автоматизованого тестування є однією з важливих задач, що вирішується при розробці ПЗ.

Сьогодні існує широкий вибір інструментального ПЗ для автоматизації тестування, а саме: JUnit, Selenium, TestNG, QUnit, Appium, Postman і інші. Автоматизація дозволяє суттєво скоротити часові витрати на процес тестування в цілому, проте задача аналізу його повноти залишається такою, що вирішується у ручному режимі при тестуванні API.

Для спрощення та прискорення оцінки повноти тестування API пропонується метод на основі python-скрипту.

Основними етапами методу є:

1. створення опису архітектури API за допомогою Swagger, що передбачає використання Swagger для документування архітектури API, включаючи детальний опис ендпоінтів, параметрів запиту, методів HTTP і відповідей, а також формування специфікації у jsonформаті.
2. створення колекції тестів для API за допомогою JMeter у .jmx форматі на основі документації, представленої у Swagger (див. п. 1);
3. запуск Jenkins задач для:
   3.1. підготовки тестового середовища;
   3.2. встановлення актуальної версії API, який буде тестуватися;
   3.3. виконання python-скрипту;
4. аналіз результатів.

Для обчислення покриття ендпоінтів автотестами python-скрипт порівнює Swagger і JMX дані за таким алгоритмом:

1. завантаження специфікації Swagger з вказаного URL за допомогою бібліотеки requests;
2. очищення обох специфікації (з Swagger URL і згенеровану з JMX) від зайвих даних за допомогою функції clean_swagger_paths;
3. порівняння ендпоінтів: зіставлення специфікації Swagger і згенерованої специфікації з JMX-файлу;
4. визначення покритих і непокритих ендпоінтів;
5. сегментація ендпоінтів.

Вхідними даними для скрипту є: Swagger URL і колекція автотестів для API у .jmx форматі. Вихідними даними є: рівень покриття автотестами API у процентному відношенні і перелік покритих і не покритих ендпоінтів. Вихідні дані представляються у вигляді htmlзвіту.

Скрипт, що застосовується, є незалежним компонентом, який може бути запущений з командного рядка, що в подальшому дозволить інтегрувати його у різні проекти, що потребують тестування API.

Для перевірки ефективності та доцільності застосування запропонованого методу пропонується підхід на основі статистичних даних про витрачений час на тестування проектів без використання запропонованого методу та з ним. Передбачається зменшення витрат часу шляхом підвищення ергономіки операції оцінок повноти покриття за рахунок її автоматизації.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖУЮ

Перший проректор Дніпровського

національного університету

залізничного транспорту

імені академіка В. Лазаряна

Борис БОДНАР

СИСТЕМА ВИЗНАЧЕННЯ ВІДПОВІДНОСТІ ТЕКСТУ ПРОГРАМИ ГРАФІЧНОМУ ПРЕДСТАВЛЕННЮ АЛГОРИТМУ

Робочий проект

1116130.01222-01-ЛЗ

ЛИСТ ЗАТВЕРДЖЕННЯ

Завідувач кафедри КІТ

Вадим ГОРЯЧКІН

Керівник розробки

Олена КУРОП'ЯТНИК

Виконавець

Богдан ЯКОВЕНКО

Нормоконтролер

Олена КУРОП'ЯТНИК

2021