Форма № Н-9.02

Національний університет «Полтавська політехніка імені Юрія Кондратюка»
(повне найменування закладу вищої освіти)
Навчально-науковий інститут інформаційних технологій і робототехніки
(повне найменування інституту, назва факультету (відділення))
Кафедра автоматики, електроніки та телекомунікацій
(повна назва кафедри (предметної, циклової комісії))

Пояснювальна записка

до кваліфікаційної роботи

магістра
(ступінь вищої освіти)

на тему «**УДОСКОНАЛЕННЯ МЕТОДУ МЕРЕЖЕВОГО ТЕСТУВАННЯ API ЧЕРЕЗ ІНТЕРФЕЙС СПОЖИВАЧА»**

Виконав: студент 6 курсу, групи 601ТТ
спеціальності 172 «Телекомунікації та радіотехніка»
(шифр і назва напряму підготовки, спеціальності

Штанько Д. В.
(прізвище та ініціали)
Керівник Сільвестров А.М.
(прізвище та ініціали)
Рецензент Шефер О.В.
(прізвище та ініціали)

Полтава - 2023 рік

Національний університет «Полтавська політехніка імені Юрія Кондратюка»
Інститут **Навчально-науковий інститут інформаційних технологій і робототехніки**
Кафедра Автоматики, електроніки та телекомунікацій
Освітній рівень магістр
Спеціальність 172 «Телекомунікації та радіотехніка»

ЗАТВЕРДЖУЮ
завідувач кафедри автоматики,
електроніки та телекомунікацій
_____ д.т.н., проф. О.В. Шефер
"04" 09 2023 р.

З А В Д А Н Н Я

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Штаньку Данилу Володимировичу

1. Тема проекту (роботи) «**УДОСКОНАЛЕННЯ МЕТОДУ МЕРЕЖЕВОГО ТЕСТУВАННЯ API ЧЕРЕЗ ІНТЕРФЕЙС СПОЖИВАЧА**»

керівник проекту (роботи) Сільвестров Антон Миколайович, д.т.н., професор
затверджена наказом вищого навчального закладу від 04.09. 2023 року № 986-фа

2. Строк подання студентом проекту (роботи) 13.12. 2023 р.

3. Вихідні дані до проекту (роботи). Вихдними даними до кваліфікаційної роботи слугують матеріали опрацьовані магістрантом під час переддипломної практики.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Вступ з обґрунтуванням теми кваліфікаційної роботи. API та його значення для сучасних інфокомунікаційних технологій. Принципи тестування та особливості тестування API. Аналіз бібліотек та мов прогамування для тестування інтерфейсу. Аспектно- орієнтовне програмування принципових засад тестування. Дослідження компонент та принципів Selenium. Удосконалення методу тестування API Веб-застосунку. Висновки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових плакатів):
Аналіз переваг середовища TestNG перед Junit. Компоненти активно-орієнтованого програмування. Алгоритм взаємодії компонентів Selenium. Архітектура WebDriver. Тестування API на прикладі кафедри автоматики, електроніки та телекомунікацій. Блок-схема фреймворку автоматизованого тестування. Стуктурні схеми проекту тестування. Кластерування тестів мовою Java. Принципи побудови авто-тесту. Функціонал сваггероподібних систем. Алгоритм створення тестів списками за допомогою раннера Insomnia. Принцип формування запитів на налаштування SOAP сервісів.

7. Дата видачі завдання 04.09.23 р.

КАЛЕНДАРНИЙ ПЛАН

Пор. №	Назва етапів магістерської роботи	Термін виконання етапів роботи			Прим (плак
1	Вступ з обґрунтуванням теми кваліфікаційної роботи.	11.10.23		15%	Пл.
2	API та його значення для сучасних інфокомунікаційних технологій.	18.10.23	I	30%	Пл.
3	Принципи тестування та особливості тестування API. Типи протоколів.	25.10.23		40%	Пл.
4	Аналіз бібліотек, переваг та мов прогамування для тестування інтерфейсу.	14.11.23		50 %	Пл.
5	Аспектно- орієнтовне програмування принципових засад тестування.	21.11.23	II	60%	Пл.
6	Дослідження компонент та принципів Selenium.	28.11.23		70%	Пл.
7	Удосконалення методу тестування API Веб-застосунку. Аналіз можливого негативного сценарію перевірки.	06.12.23		90%	Пл.
8	Висновки. Оформлення кваліфікаційної роботи та формування додатків.	13.12.23	III	100%	Пл. 9-

Студент _____ Штанько Д.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Сільвестров А.М.
(підпис) (прізвище та ініціали)

ЗМІСТ

ПЕРЕЛІК ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

QA – забезпечення якості (quality assurance), або тестувальник;

HTTP (HyperText Transfer Protocol) – широко поширений протокол передачі даних;

API (Application programming interface) – це інтерфейс програмування застосунка;

AOP (Aspect-oriented Programming) - аспектно-орієнтоване програмування;

UI (User interface) інтерфейс користувача;

SOAP (Simple Object Access Protocol) – простий протокол доступу до об’єктів;

REST (Representational State Transfer) – представницька державна передача;

XML (EXtensible Markup Language) – мова розмітки;

WEB – інтернет - простір;

JVM – Java Virtual Machine – віртуальна машина Java;

JDK – безкоштовний розповсюджуваний Oracle (раніше Sun) комплект розробника застосунків на мові Java;

JRE – Java Runtime Environment середовище виконання для Java;

AOP – Аспектно-орієнтоване програмування;

DDoS (Distributed Denial of Service) - атаки розподіленої відмови в обслуговуванні.

ВСТУП

Інформаційно-телекомунікаційні технології демонструють швидке зростання за останні 10 років. Створюється багато різних технологій та застосунків. В наші дні WEB сфера є однієї з найпопулярніших сфер технологій, бо зараз майже всі технології потребують інтернет. Зараз кожен бізнес використовує інтернет та WEB технології для своєї реалізації. Для кожного бізнесу час дорівнює гроші. Це є причиною швидкого розвитку та популяризації швидкої розробки ПО для телекомунікаційних та інформаційних мереж і систем.

В наші дні майже кожен бізнес використовує веб-застосунки. Кожен веб-застосунок має свій API, який потребує тестування. Якщо не тестувати API, то застосунок буде працювати не правильно, або взагалі не буде працювати. Це може бути причиною патері грошей для бізнесу. Також заради прибутку треба розробляти та тестувати API якомога швидше. Це надає можливість обійти конкурентів та достигати різних успіхів для бізнесу.

Є різні підходи до швидкого тестування API. Одним з таких є автоматизування API тестів для швидкого тестування старого функціонала при додаванні нового. При використовуванні даного підходу тестувальник повинен мати прямий доступ до API, якій тестується, бо без прямого доступу не можливо розробляти автоматизовані тести. Прямий доступ надає можливість виконувати запроси до API, та отримувати відповідь від серверу застосунка. Інколи тестувальник не має прямого доступу до API, або отримання доступу забирає багато часу, якого не має для тестування вчасно, яке бажає замовник зі сторони бізнесу. Останній варіант отримання доступу у цьому випадку який може бути – це доступ до API через інтерфейс користувача. Під час роботи з API через інтерфейс користувача тестувальник може використовувати метод автоматизації тестування API через інтерфейс користувача. Зазначений метод вирішує проблему з доступом та часом який може бути витрачено на отримання доступу та надає можливість почати тестування якомога швидше. Недолік

цього методу у тому, що тести, які працюють через інтерфейс користувача будуть повільніші ніж звичайні API тести, також можуть бути складніші для написання і відлагодження. Найбільша перевага даного методу це більш реальна імітація реального користувача, яка робить тестування більш точнішим, розширює тестове покриття, завдяки чому можна знайти критичні баги.

Отже, ми можемо зробити висновок, що в той час коли тестувальник не має прямого доступу до API та йому потрібно писати прості невеликі тест, він може використати метод автоматизації тестування API через інтерфейс користувача. Це вирішує проблему з витраченим часом на отримання доступу та надає можливість вчасно закінчити тестування.

1 АНАЛІТИЧНИЙ ОГЛЯД

1.1 API, його значення для сучасних інфокомунікаційних технологій

API – це набір програмного коду, який забезпечує передачу даних між одним програмним продуктом та іншим [1]. Він також містить умови обміну даними.

Інтерфейс програмування прикладних програм дозволяє компаніям відкривати дані та функціональні можливості своїх програм для зовнішніх сторонніх розробників, бізнес-партнерів і внутрішніх відділів у своїх компаніях. Це дозволяє службам і продуктам взаємодіяти один з одним і використовувати дані та функціональні можливості один одного через документований інтерфейс. Розробникам не потрібно знати, як реалізовано API; вони просто використовують інтерфейс для комунікування з іншими продуктами та послугами. За останнє десятиліття використання API зросло до такої міри, що багато з найпопулярніших веб-додатків сьогодні були б неможливими без API.

1.2 Принцип роботи API

API працює за таким принципом. Клієнтська програма ініціює виклик API для отримання інформації, також відомий як запит. Цей запит обробляється від програми до веб-сервера через уніфікований ідентифікатор ресурсу (URI) API і включає дієслово запиту, заголовки, а іноді й тіло запиту. Після отримання дійсного запиту API здійснює виклик до зовнішньої програми або веб-сервера. Сервер надсилає відповідь API із запитаною інформацією. API передає дані початковій програмі, яка запитує. Хоча передача даних буде відрізнятися залежно від веб-служби, що використовується, цей процес запитів і відповідей відбувається через API.

У той час, як інтерфейс користувача розроблений для використання людьми, API розроблено для використання комп’ютером або програмою. API

забезпечують безпеку за проектом, оскільки їхня позиція посередника полегшує абстракцію функціональних можливостей між двома системами — кінцева точка API відокремлює додаток-споживач від інфраструктури, яка надає послугу. Виклики API зазвичай включають облікові дані авторизації, щоб зменшити ризик атак на сервер, а шлюз API може обмежити доступ, щоб мінімізувати загрози безпеці. Крім того, під час обміну заголовки HTTP, файли cookie або параметри рядка запиту забезпечують додаткові рівні безпеки даних. Наприклад, розглянемо API, який пропонує служба обробки платежів. Клієнти можуть вводити дані своєї картки в інтерфейсі програми для магазину електронної комерції. Обробник платежів не вимагає доступу до банківського рахунку користувача; API створює унікальний маркер для цієї транзакції та включає його у виклик API до сервера. Це забезпечує більш високий рівень безпеки від потенційних загроз злому.

1.3 Призначення API

Одним з найголовніших призначень API є покращена спільна робот. Середнє підприємство використовує майже 1200 хмарних додатків, багато з яких відключені. API забезпечують інтеграцію, щоб ці платформи та програми могли безперешкодно взаємодіяти один з одним. Завдяки цій інтеграції компанії можуть автоматизувати робочі процеси та покращити співпрацю на робочому місці. Без API багатьом підприємствам не вистачало б підключення та страждало б від інформаційних розрізів, які ставлять під загрозу продуктивність і продуктивність.

Наступне призначення API – це простіші інновації. API пропонують гнучкість, дозволяючи компаніям налагоджувати зв’язки з новими бізнес-партнерами, пропонувати нові послуги на існуючому ринку і, зрештою, отримувати доступ до нових ринків, які можуть генерувати величезну віддачу та стимулювати цифрову трансформацію. Наприклад, компанія Stripe починала як API лише з семи рядків коду. З тих пір компанія співпрацює з багатьма

найбільшими підприємствами світу, диверсифікуючись, пропонуючи кредити та корпоративні картки, і нещодавно була оцінена в 36 мільярдів доларів США.

Іншим призначенням API є монетизація даних. Багато компаній вирішують пропонувати API безкоштовно, принаймні спочатку, щоб вони могли створити аудиторію розробників навколо свого бренду та налагодити відносини з потенційними діловими партнерами. Однак, якщо API надає доступ до цінних цифрових активів, ви можете монетизувати його, продаючи доступ (це називають економікою API). Коли AccuWeather запустив свій портал для розробників самообслуговування для продажу широкого спектру пакетів API, знадобилося всього 10 місяців, щоб залучити 24 000 розробників, продати 11 000 ключів API і створити процвітаючу спільноту в цьому процесі.

Останнє призначення API - додаткова безпека. Як зазначалося вище, API створюють додатковий рівень захисту між вашими даними та сервером. Розробники можуть додатково посилити безпеку API, використовуючи токени, підписи та шифрування безпеки транспортного рівня (TLS); шляхом впровадження шлюзів API для керування та аутентифікації трафіку, і практикуючи ефективне управління API.

1.4 Приклади API

Оскільки API дозволяють компаніям відкривати доступ до своїх ресурсів, зберігаючи безпеку та контроль, вони стали цінним аспектом сучасного бізнесу. Ось кілька популярних прикладів інтерфейсів прикладного програмування, з якими ви можете зіткнутися.

Перший - універсальні логіни. Популярним прикладом API є функція, яка дозволяє людям входити на веб-сайти, використовуючи дані для входу в свої профілі Facebook, Twitter або Google. Ця зручна функція дозволяє будь-якому веб-сайту використовувати API з однієї з найбільш популярних служб для швидкої автентифікації користувача, заощаджуючи час, створюючи новий профіль для кожної служби веб-сайту.

Наступна - Обробка платежів третьою стороною. Наприклад, повсюдно поширена функція "Оплатити за допомогою", яку ми бачимо на веб-сайтах електронної комерції, працює через API. Це дозволяє людям оплачувати продукти в Інтернеті, не розкриваючи будь-яких конфіденційних даних і не надавши доступ неавторизованим особам.

Порівняння бронювання подорожей: сайти бронювання подорожей об’єднують тисячі рейсів, демонструючи найдешевші варіанти для кожної дати та місця призначення. Ця послуга стала можливою завдяки API, які надають користувачам додатків доступ до останньої інформації про наявність місць у готелях та авіакомпаніях. Завдяки автономному обміну даними та запитами API різко скорочують час і зусилля, необхідні для перевірки наявності авіарейсів або розміщення.

Одним із найпоширеніших прикладів гарного API є служба Google Maps. На додаток до основних API, які відображають статичні або інтерактивні карти, програма використовує інші API та функції, щоб надавати користувачам маршрути або визначні місця. За допомогою геолокації та кількох шарів даних ми можемо спілкуватися з API Карт, прокладаючи маршрути подорожі або відстежуючи об’єкти в дорозі, наприклад, транспортний засіб.

Twitter: кожен твіт містить описові основні атрибути, включаючи автора, унікальний ідентифікатор, повідомлення, мітку часу, коли він був опублікований, і метадані геолокації. Twitter робить публічні твіти та відповіді доступними для розробників і дозволяє розробникам публікувати твіти через API компанії.

1.5 Типи API

Сьогодні більшість інтерфейсів програмування прикладних програм — це веб-API, які розкривають дані та функціональні можливості програми через Інтернет. Є чотири основних типи веб-API.

Відкриті API – це програмні інтерфейси програм з відкритим кодом, до яких можна отримати доступ за допомогою протоколу HTTP. Також відомі як публічні API, вони визначають кінцеві точки API та формати запитів і відповідей.

Партнерські API – це інтерфейси прикладного програмування, доступні для стратегічних бізнес-партнерів або від них. Зазвичай розробники можуть отримати доступ до цих API в режимі самообслуговування через публічний портал розробників API. Тим не менш, їм потрібно буде завершити процес адаптації та отримати облікові дані для доступу до партнерських API.

Внутрішні API – це інтерфейси прикладного програмування, які залишаються прихованими від зовнішніх користувачів. Ці приватні API недоступні для користувачів за межами компанії, а натомість призначені для підвищення продуктивності та спілкування між різними внутрішніми командами розробників.

Композитні API поєднують кілька API даних або послуг. Ці служби дозволяють розробникам отримати доступ до кількох кінцевих точок за один виклик. Композитні API корисні в архітектурі мікросервісів, де для виконання одного завдання може знадобитися інформація з кількох джерел.

1.6 Типи API протоколів

Оскільки використання веб-API розширилося, були розроблені певні протоколи, щоб надати користувачам набір визначених правил, які визначають прийняті типи даних і команди. Фактично, ці протоколи API полегшують стандартизований обмін інформацією.

SOAP (Simple Object Access Protocol) — це протокол API, створений із XML, що дозволяє користувачам надсилати та отримувати дані через SMTP та HTTP. Завдяки API SOAP легше обмінюватися інформацією між програмами або програмними компонентами, які працюють у різних середовищах або написані різними мовами.

XML-RPC — це протокол, який покладається на специфічний формат XML для передачі даних, тоді як SOAP використовує власний формат XML. XML-RPC старіший за SOAP, але набагато простіший і відносно легкий, оскільки використовує мінімальну пропускну здатність.

JSON-RPC — це протокол, подібний до XML-RPC, оскільки обидва вони є віддаленими викликами процедур (RPC), але цей протокол використовує JSON замість формату XML для передачі даних. Обидва протоколи прості. Хоча виклики можуть містити кілька параметрів, вони очікують лише одного результату.

REST (Representational State Transfer) — це набір принципів архітектури веб-API, що означає, що немає офіційних стандартів (на відміну від тих, що мають протокол). Щоб бути REST API (також відомим як RESTful API), інтерфейс повинен відповідати певним архітектурним обмеженням. Можна створювати RESTful API за допомогою протоколів SOAP, але ці два стандарти зазвичай розглядаються як конкуруючі специфікації. Саме про аналіз метод автоматизації тестування API через інтерфейс користувача для веб-застосунку з REST архітектурою ведеться мова в даному дипломному проекті. [1]

2 ТЕХНОЛОГІЧНА ЧАСТИНА

2.1 Принципи тестування API

Тестування API (інтерфейсу програмного програмування) — це тип тестування програмного забезпечення, яке має на меті визначити, чи відповідають розроблені API очікуванням щодо функціональності, продуктивності, надійності та безпеки програми.

Для розробників тестування API зводиться до перевірки того, що API повертає правильні відповіді. Це часто робиться за допомогою такого інструменту, як Postman, який дозволяє публікувати запити API та перевіряти відповіді. Часто тестування проводиться на так званий підроблений API. Це автономна реалізація нашого API, яка може генерувати правильну відповідь на основі вашої документації API. Однак він не пов’язаний з нашою справжньою базою даних.

Для тестувальників системи тестування API пропонує спосіб перевірити функціональність, яку може бути важко запустити безпосередньо через інтерфейс користувача. Наприклад, виклики API, пов’язані з обробкою помилок. Або відповіді API, які запускаються лише у виняткових випадках. Часто простіше безпосередньо перевірити ці виклики, ніж намагатися налаштувати інтерфейс користувача у належний стан. Також, можна тестувати простий API використовуючи інтерфейс користувача. Наприклад, API новинних веб-сайтів.

Фахівці DevOps можуть навіть неправильно використовувати тестування API, щоб посилатися на стрес-тестування нашого бекенда через API. Їм потрібен спосіб якомога більше навантаження на систему. Найпростіший спосіб зробити це – багаторазовий виклик API.

2.2 Складність API тестування

На перший погляд тестування API досить просте. Сподіваємося, у нас є чітко визначений API з хорошою документацією. Це означає, що ми знаємо кожен виклик і відповідні відповіді. Отже, тестування — це просто перевірка того, що ми отримуємо очікувану відповідь на кожен дзвінок. Все, що нам потрібно зробити, це відправити послідовність викликів до API і перевірити відповіді.

Реальність така, що тестування API, ймовірно, складніше, ніж є на словах. Потрібно піти набагато далі, ніж просто перевіряти кожну відповідь. Інші ключові аспекти включають тестування безпеки, перевірку умов помилок і перевірку того, як сервер справляється з викликами з неправильним форматом. Все це потребує ретельного продумування та планування. Повне тестування API також необхідно проводити з псевдореалістичними даними. Без цього ми не зможемо перевірити, наскільки добре сервер справляється з незвичайними даними.

2.3 Тестування UI

Все, що користувач, ймовірно, зробить, може зробити або може зробити, або, що ще важливіше, може зробити, але не повинен бути в змозі робити з нашою програмою, поки вона працює, є частиною інтерфейсу користувача, а отже, є кандидатом на тестування. Від кожного натискання кнопки, до введення даних. Ми хочемо перевірити не тільки те, що, як очікується, спрацює, але й те, що, як очікується, зазнає невдачі. Тестування автоматизації інтерфейсу користувача подібне до ручного тестування, але замість того, щоб користувач використовував нашу програму та візуально перевіряв дані. Ми створюємо сценарій для кожного тестового випадку, який бажаємо перевірити. Додаємо ряд кроків для виконання сценарію та перевіряємо дані.

2.4 Складність UI тестування

У тестуванні UI все може бути набагато складнішим. Зрештою, нам потрібно почати з визначення повних тестових випадків. Ми не можемо просто перевірити потрібну функцію. Замість цього ми повинні почати з переходу до правильної частини інтерфейсу користувача та переведення системи в правильний стан. Це означає, що нам потрібен повний інтерфейс користувача, перш ніж почати тестування. Більше того, масштаб сучасних інтерфейсів робить автоматизацію тестування необхідною. Справа в тому, що автоматичне тестування інтерфейсу користувача є важкою проблемою. Тести створюються повільно, їх важко аналізувати, і вони ламаються щоразу, коли змінюється наш інтерфейс користувача. [2]

2.5 API або UI, коли і який тест виконувати

Який тест має більше переваг перед іншими. Це постійні і нескінченні дебати для тестувальника - у будь-якому конкретному проекті, скільки відсотків тестувальник повинен писати тестів API, і скільки він повинен зосередитися на тесті інтерфейсу користувача. На це немає простої відповіді. Оскільки, більшість тестувальників розгублені, коли справа доходить до тесту UI та AIP.

Пошук помилок на початку циклу. Оскільки служби або API написані до розробки інтерфейсу користувача, тест API стане в нагоді, щоб знайти помилки бізнес-рівню в програмі. Тестер може зосередитися на написанні тесту API на початку життєвого циклу розробки та виявити критичні помилки бізнес-логіки. І ще один момент, багато бізнес-логіки важко повторити з передньої частини. Деякі функції продукту потребують дуже конкретних і детальних налаштувань середовища, щоб запустити цей рядок коду на бізнес-рівні. Тест UI для цього неефективі. У багатьох випадках це було б непрактично, особливо якщо задіяно інше обладнання, наприклад, апаратний пристрій для заправки автомобіля,

який надсилає інформацію на веб-сервер після завершення заправки. Замість цього ми можемо просто змоделювати повідомлення на рівні API/інтеграції. Тому в цьому аспекті API має більше переваг перед UI. Крихкість. Кожен, хто працює в автоматизації UI тестів, відчує, що тести UI дуже крихкі за своєю природою. Будь-яка зміна елемента інтерфейсу користувача або зміна середовища браузера порушить тест. Елементи інтерфейсу користувача можуть змінюватися досить швидко. Оновлення версії браузера можуть мати зміни в поведінці, реакція в різних браузерах неоднакова, і список можна продовжувати. У той час як API тести не є крихкими після того, як специфікація Swagger встановлена розробниками, API тести будуть виконуватися таким же чином без будь-яких збоїв.

Повільне виконання. Пройшли ті часи, коли ми думали про тестування програми за допомогою автоматизованих інструментів, ми думали лише про Win Runner/UFT/Selenium. Тепер кожен власник продукту хотів виявити критичні помилки в програмі на початку життєвого циклу за дуже мінімальний час. Тут API тести є дуже зручними для вирахування помилок бізнес-логіки завчасно і з мінімальним часом. Тести UI є громіздкими і займають багато часу. Наші UI тести можуть відтворювати поведінку користувача через інтерфейс програми, і вони можуть виконуватися годинами, перш ніж ми знайдемо повні помилки в програмі. Тоді як весь набір API буде запущено за кілька хвилин і охоплюватиме всі тести бізнес-логіки. Вони набагато ефективніші та ефективніші для виявлення будь-яких серйозних та критичних помилок у програмі.

Ремонтопридатність і трудомісткість розробки. UI тести зазвичай імітують дії кінцевого користувача. Таким чином, скрипти виконують усі завдання, які виконуватиме людина в додатку, як-от завантаження екрана/сторінок, написання та читання купи елементів, вставляючи багато часу очікування в тест. Скриптування всіх цих дій навіть з високоефективним і багаторазовим фреймворком є трудомістким завданням. Результатом цієї вправи зазвичай є обмін даними між клієнтом і сервером. Натомість ми можемо

досягти всього цього за менший час, створюючи дані, які клієнт надсилає на сервер (за допомогою відповіді POST), це буде порівняно менше рядка коду, ніж у UI тестів. Ця частина розробки, яка підтримується, стає величезними накладними витратами та громіздким завданням, коли тести UI починають давати збій, оскільки база коду в багато разів більша за тести API. Це стає набагато дорожчим і накладними в обслуговуванні. Чим більше рядків коду, тим більша складність означає більше витрат і часу тестувальника.

Тестове покриття. Дуже складне завдання повторити кожен бізнес-потік за допомогою інтерфейсу користувача. Деякі функції продукту потребують конкретних і детальних налаштувань середовища, щоб запустити рядок коду в бізнес-логіці. Тести UI дуже неефективні для цих сценаріїв. Для виконання невеликих завдань тестувальнику потрібно написати більше рядків коду, оскільки цей тестер не зможе охопити всі сценарії бізнес-логіки за допомогою тестів UI. Навіть якщо тестувальник має на меті охопити всю бізнес-логіку за допомогою тестів UI, це матиме величезний ефект, щоб охопити все. Набагато простіше і менше рядків коду та часу охоплювати складну логіку бізнес-рівню за допомогою API. У цьому випадку API тести стають зручними і ефективними. [3]

3 ДОСЛІДНО-ТЕХНОЛОГІЧНА ЧАСТИНА

3.1 Постановка завдання на тестування

Як зазначалося раніше, метою кваліфікаційної роботи магістра є аналіз методу тестування API через інтерфейс користувача. Тестування буде виконуватись на веб-застосунку Національного університету «Полтавська політехніка імені Юрія Кондратюка» кафедри «Автоматики, електроніки та телекомунікацій» за посиланням «https://nupp.edu.ua/page/kafedra-avtomatiki-yelektroniki-ta-telekomunikatsiy.html».

Іноді бувають ситуації коли не має можливості тестувати API на пряму. В таких випадках можна тестувати API через інтерфейс користувача, але зі своїми нюансами. Метод тестування API через інтерфейс користувача є гарною практикою, коли ми не маємо можливості запросити доступ до API та бізнес-логіка тестувального додатка є легкою. Було обрано саме цей веб-застосунок, тому що в нас не має можливості запросити доступ до його API та це новинний веб-застосунок. У новинних веб-застосунків майже завжди легка бізнес-логіка.

Веб-застосунок буде тестуватися в декілька етапів. Спочатку буде створено 50 ручних тест-кейсів на основний функціонал застосунку. Тест-кейси будуть рівня Smoke тестування. Потім буде створено фреймворк для автоматизації ручних тестів.

Автоматизаційний фреймворк буде надавати користувачу такий функціонал:

- створювати різні набори тестів;
- запускати конкретний тест;
- запускати декілька тестів;
- формування звіту з результатами тестування;
- зображення тривалості виконання тестів;
- запуск тестів на різних браузерах;

- зображення шагів тест-кейсів у консоль під час виконання тестів;
- створення нових тестів та редагування старих тестів;
- конфігурація даних для тестування;
- запуск одного тесту з різними даними.

Після розробки фреймворку буде виконано автоматизацію ручних тестів, та запуск автоматизованих тестів. Потім буде формуватися звіт про тестування на основі якого буде виконано аналіз методу тестування API через інтерфейс користувача [4].

3.2 Бібліотека TestNG

TestNG (Test Next Generation) — це платформа для тестування. TestNG заснований на JUnit 4 і NUnit, але він має нову функціональність, яка робить цю структуру потужнішою та простішою. TestNG розроблено таким чином, що він охоплює всі категорії тестів, що включають одиничні, функціональні та інтеграційні.

Фреймворк TestNG – це платформа для автоматизованого тестування з відкритим кодом, і тут NG означає наступне покоління. TestNG досить схожий на JUnit 4. TestNG натхненний фреймворком JUnit 4 і розроблений таким чином, що при тестуванні інтегрованих класів виходить краще, ніж JUnit 4. TestNG був створений Седриком Бюстом.

3.3 Переваги TestNG перед Junit 4

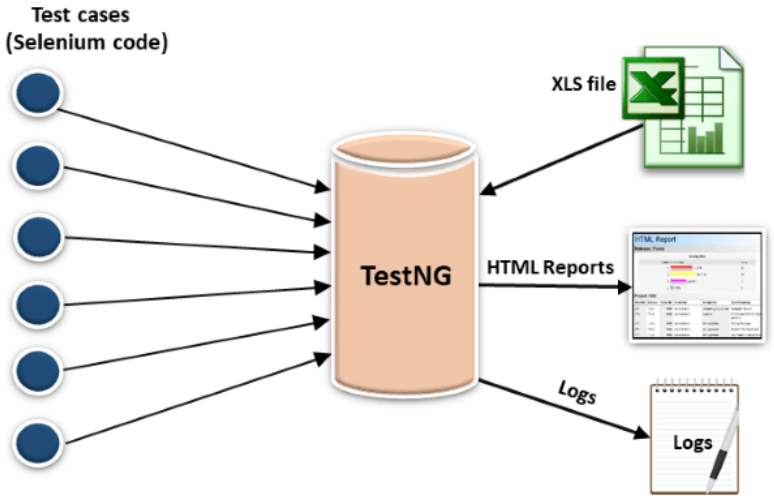

Рисунок 3.1 – Схема TestNG

У TestNG анотації легше зрозуміти, ніж у Junit. Він створює звіти HTML для впровадження.

Він також створює журнали.

У TestNG немає доступних обмежень, таких як @beforeclass і @afterclass,які присутні в Junit.

TestNG дозволяє легко групувати тестові випадки, що неможливо в JUnit.

TestNG підтримує три додаткові рівні, такі як пакет @Before/After, @Before/AfterTest і Before/AfterGroup.

TestNG не розширює жодного класу. Framework TestNG дозволяє визначити тестові випадки, у яких кожен тестовий приклад не залежить від інших тестових випадків.

Він дозволяє запускати тестові випадки певної групи. Давайте розглянемо сценарій, де ми створили дві групи, такі як «Дим» і «Регресія». Якщо ми бажаємо виконати тестові випадки в групі «Регресія», це можливо лише в рамках TestNG.

Паралельне виконання тестових випадків, тобто запуск кількох тестових випадків, можливе лише в платформі TestNG [5].

3.4 Мова програмування Java

JAVA була розроблена Джеймсом Гослінгом у Sun Microsystems Inc в 1991 році, пізніше придбана Oracle Corporation. Це проста мова програмування. Java спрощує написання, компіляцію та налагодження програмування. Це допомагає створювати багаторазовий код і модульні програми.

Java — це об'єктно-орієнтована мова програмування на основі класів і розроблена таким чином, щоб мати якомога менше залежностей реалізації. Мова програмування загального призначення, створена для розробників, які можуть писати код, який може бути запущений майже на будь-якому пристрої, бо скомпільований Java-код може працювати на всіх платформах, які підтримують Java. Програми Java компілюються в байтовий код, який може працювати на будь-якій віртуальній машині Java. Синтаксис Java подібний до c/c++.

Історія Java дуже цікава. Це мова програмування, створена в 1991 році. Джеймсом Гослінгом, Майком Шеріданом і Патріком Нотоном, команда інженерів Sun, відома як команда Green, започаткували мову Java в 1991 році. Sun Microsystems випустила свою першу публічну реалізацію в 1996 році як Java 1.0. Вона забезпечувала безкоштовний час виконання на популярних платформах. Компілятор Java1.0 був переписаний на Java Артуром Ван Хоффом, щоб суворо відповідати його специфікаціям. З приходом Java 2 нові версії мали кілька конфігурацій, створених для різних типів платформ.

У 1997 році Sun Microsystems звернулася до органу стандартів ISO і пізніше формалізувала Java, але незабаром вийшла з цього процесу. Свого часу Sun зробила більшість своїх реалізацій Java доступними безкоштовно, незважаючи на статус їх власного програмного забезпечення. Sun отримувала дохід від Java за рахунок продажу ліцензій на спеціалізовані продукти, такі як Java Enterprise System.13 листопада 2006 року Sun випустила більшу частину своєї віртуальної машини Java як безкоштовне програмне забезпечення з відкритим кодом. 8 травня 2007 року Sun завершила процес, зробивши весь

основний код своєї JVM доступним на умовах розповсюдження з відкритим кодом.Принципи створення Java були простими, надійними, захищеними, високопродуктивними, портативними, багатопоточними, інтерпретованими, динамічними тощо. Зараз Java використовується в мобільних пристроях, інтернет-програмуванні, іграх, електронному бізнесі тощо.

3.5 Складові частини мови програмування Java

Мова програмування Java складається за таких частин:

1. Віртуальна машина Java (JVM). Існує три етапи виконання програми – написання коду, компіляція та запуск. Компіляція виконується компілятором Javac, який є основним компілятором Java, що входить до комплекту розробки Java (JDK). Він приймає програму Java як вхідні дані і генерує байт-код як вихід. На етапі виконання програми JVM виконує байт-код, згенерований компілятором. На основі цього ми можемо зробити висновок, що функція віртуальної машини Java полягає у виконанні байт-коду, створеного компілятором. Кожна операційна система має різну JVM, але вихід, який вони виробляють після виконання байт-коду, однаковий для всіх операційних систем. Ось чому Java відома як платформно-незалежна мова.

2. Байт-код у процесі розробки: як обговорювалося, компілятор Javac JDK компілює вихідний код Java у байт-код, щоб його можна було виконати за допомогою JVM. Він зберігається компілятором як файл .class. Для перегляду байт-коду можна використовувати дизассемблер, наприклад javap.

3. Java Development Kit, як випливає з назви, це повний набір для розробки Java, який включає все, включаючи компілятор, середовище виконання Java (JRE), відладчики Java, документи Java тощо. Щоб програма могла виконуватися на Java, нам потрібно встановити JDK на нашому комп'ютера, щоб створити, компілювати та запустити програму Java.

4. Середовище виконання Java (JRE): JDK включає JRE. Установка JRE на наших комп’ютерах дозволяє запускати програму Java, однак ми не можемо її скомпілювати. JRE включає в себе браузер, JVM, підтримку аплетів і плагіни.

5. Збірник сміття. У Java програмісти не можуть видаляти об’єкти. Для видалення або відновлення пам’яті JVM має програму під назвою Garbage Collector. Збірники сміття видаляє з пам’яті об’єкти, на які немає посилання. Таким чином, Java полегшує життя програміста, обробляючи управління пам’яттю. Однак програмістам слід уважно ставитися до свого коду, чи використовують вони об’єкти, які використовуються протягом тривалого часу. Оскільки Збірники сміття не може відновити пам’ять об’єктів, на які посилається.

6. Шлях до класу – це шлях до файлу, за яким середовище виконання Java та компілятор Java шукають файли .class для завантаження. За замовчуванням JDK надає багато бібліотек. Якщо ми бажаємо включити зовнішні бібліотеки, їх слід додати до шляху до класу.

3.5.1 Основні можливості Java

Незалежність від платформи. компілятор перетворює вихідний код у байт-код, а потім JVM виконує байт-код, згенерований компілятором. Цей байт-код може працювати на будь-якій платформі, будь то Windows, Linux, macOS, що означає, якщо ми компілюємо програму на Windows, то ми можемо запустити її на Linux і навпаки. Кожна операційна система має різну JVM, але вихід, який виробляє всі ОС, однаковий після виконання байт-коду. Ось чому ми називаємо java незалежною від платформи мовою.

Організація програми в термінах колекції об'єктів є способом об'єктно-орієнтованого програмування, кожен з яких представляє екземпляр класу. Чотири основні концепції об'єктно-орієнтованого програмування: Абстракція, Інкапсуляція, Спадкування, Поліморфізм.

Java є однією з простих мов, оскільки вона не має складних функцій, таких як покажчики, перевантаження операторів, множинне спадкування, явне виділення пам'яті.

Мова Java є надійною. Вона розроблена таким чином, що докладає багато зусиль для перевірки помилок якомога раніше, тому компілятор java здатний виявити навіть ті помилки, які нелегко виявити іншою мовою програмування. Основними характеристиками Java, які роблять її надійною, є збір сміття, обробка винятків і виділення пам’яті.

У java ми не маємо покажчиків, тому ми не можемо отримати доступ до масивів поза межами, тобто вона показує виняток ArrayIndexOutOfBound, якщо ми спробуємо це зробити. Ось чому деякі недоліки безпеки, такі як пошкодження стека або переповнення буфера, неможливо використати в Java.

Ми можемо створювати розподілені програми, використовуючи мову програмування Java. Віддалений виклик методів і Enterprise Java Beans використовуються для створення розподілених додатків на java. Програми Java можна легко розповсюдити на одну або кілька систем, які підключені одна до одної через Інтернет.

Java підтримує багатопоточність. Це функція Java, яка дозволяє одночасно виконувати дві або більше частин програми для максимального використання ЦП у декількох потоків.

Як ми знаємо, код Java, написаний на одній машині, можна запускати на іншій машині. Незалежна від платформи функція Java, в якій її незалежний від платформи байт-код можна перенести на будь-яку платформу для виконання, робить java переносимою.

Архітектура Java визначена таким чином, що вона зменшує накладні витрати під час виконання, і в певний час Java використовує компілятор Just In Time (JIT), де компілятор компілює основи коду на вимогу, де він компілює лише ті методи, які називають прискоренням виконання програм.

Оскільки Java повністю об'єктно-орієнтована, ми можемо додавати класи, нові методи до існуючих класів і навіть створювати нові класи через підкласи.

Java навіть підтримує функції, написані іншими мовами, такими як C, C++, які називаються рідними методами.

Програми Java запускаються в окремому просторі, що дозволяє користувачеві виконувати свої програми, не впливаючи на базову систему за допомогою перевіряючого байт-коду. Перевірка байт-коду також забезпечує додаткову безпеку, оскільки його роль полягає в тому, щоб перевірити код на наявність будь-яких порушень доступу.

Як обговорювалося вище, програма Java генерує файл «.class», який відповідає нашим додаткам (програмі), але містить код у двійковому форматі. Він забезпечує легкість, нейтральну для архітектури, оскільки байт-код не залежить від архітектури будь-якої машини. Це основна причина, чому java використовується в підприємливій IT-індустрії в усьому світі.

Потужність компіляції та інтерпретації. Більшість мов розроблені з метою, або вони є мовою компіляції, або мовою інтерпретації. Але java інтегрує величезну потужність, яка виникає, оскільки компілятор Java компілює вихідний код у байт-код, а JVM виконує цей байт-код у машинний виконуваний код, залежний від ОС [6].

3.6 Аспектно-орієнтоване програмування

Аспектно-орієнтоване програмування (AOP), як випливає з назви, використовує аспекти в програмуванні. Його можна визначити як розбиття коду на різні модулі, також відоме як модульність, де аспект є ключовою одиницею модульності. Аспекти дозволяють реалізувати наскрізні проблеми, такі як транзакції, ведення журналів, що не є центральними для бізнес-логіки, не захаращуючи ядро коду його функціональністю. Це робиться шляхом додавання додаткової поведінки, яка є порадою до існуючого коду. Наприклад, безпека є наскрізною проблемою. До багатьох методів у програмі можуть застосовуватися правила безпеки, тому повторення коду для кожного методу,

визначення функціональності в загальному класі, а контроль повинен застосовувати цю функціональність у всьому додатку.

3.6.1 Принцип роботи аспектно-орієнтованого програмування

Можна подумати, що використання методу автоматично реалізує наскрізні проблеми, але це не так. Просто виклик методу не викликає пораду (роботу, яку потрібно виконати). Java використовує механізм на основі проксі, тобто він створює проксі-об'єкт, який обгорне оригінальний об'єкт і буде використовувати поради, які стосуються виклику методу. Об’єкти проксі-сервера можна створити вручну за допомогою фабричного компонента проксі-сервера або за допомогою автоматичної конфігурації проксі-сервера у файлі XML, та видалити їх після завершення виконання. Проксі-об’єкти використовуються для збагачення оригінальної поведінки реального об’єкта.

3.6.2 Загальні терміни в AOP

Рисунок 3.2 – Компоненти AOP

Аспект: клас, який реалізує наскрізні проблеми програми JEE (транзакція, реєстратор тощо), відомий як аспект. Це може бути звичайний клас, налаштований за допомогою конфігурації XML, або через звичайні класи, анотовані @Aspect.

@JoinPoint (Точка приєднання) — це частина або точка, де виконується аспект. У AOP точка з’єднання є методом, де буде виконуватися аспект.

@Advice (Порада), є основною і найважливішою частиною АОП. Порада — це те, де виконується дія аспектом у точці з’єднання. Порада може бути розташована перед точкою з’єднання, після або під час виконання точки з’єднання. Як правило, порада — це перехоплювач, який керує всіма перехопленнями в точці з’єднання

@Pointcut (Точковий зріз) — це вираз або присудок, який відповідає точці спільного об’єднання. Зазвичай регулярний вираз. Порада буде пов’язано з виразом точкового зрізу і виконуватиметься в точці з’єднання.

@Around(Навколо). Це об'єкт, який підлягає пораді. Особливо у АОР реалізує використання проксі часу виконання. Таким чином, цей об'єкт завжди є проксі-об'єктом.

AOP Proxy, об'єкт, створений фреймворком AOP для реалізації аспекту контракту.

3.7 Термін Selenium

Selenium відноситься до набору інструментів, які широко використовуються в спільноті тестування, коли справа доходить до міжбраузерного тестування. Selenium не може автоматизувати настільні програми; його можна використовувати лише у браузерах. Вважається одним з найбільш переважних наборів інструментів для автоматизованого тестування веб-додатків, оскільки забезпечує підтримку популярних веб-браузерів, що робить його дуже потужним. Він підтримує ряд браузерів (Google Chrome 12+, Internet Explorer 7,8,9,10, Safari 5.1+, Opera 11.5, Firefox 3+) та операційних систем (Windows, Mac, Linux/Unix). Selenium також забезпечує сумісність з різними мовами програмування – C#, Java, JavaScript, Ruby, Python, PHP. Тестувальники можуть вибирати, якою мовою розробляти тестові випадки, що робить Selenium дуже сприятливим для його гнучкості.

3.7.1 Компоненти Selenium

Набір компонентів Selenium складається з чотирьох основних компонентів (рис. 3.3):

- Selenium IDE;
- Selenium RC;
- Selenium Webdriver;
- Selenium Grid.

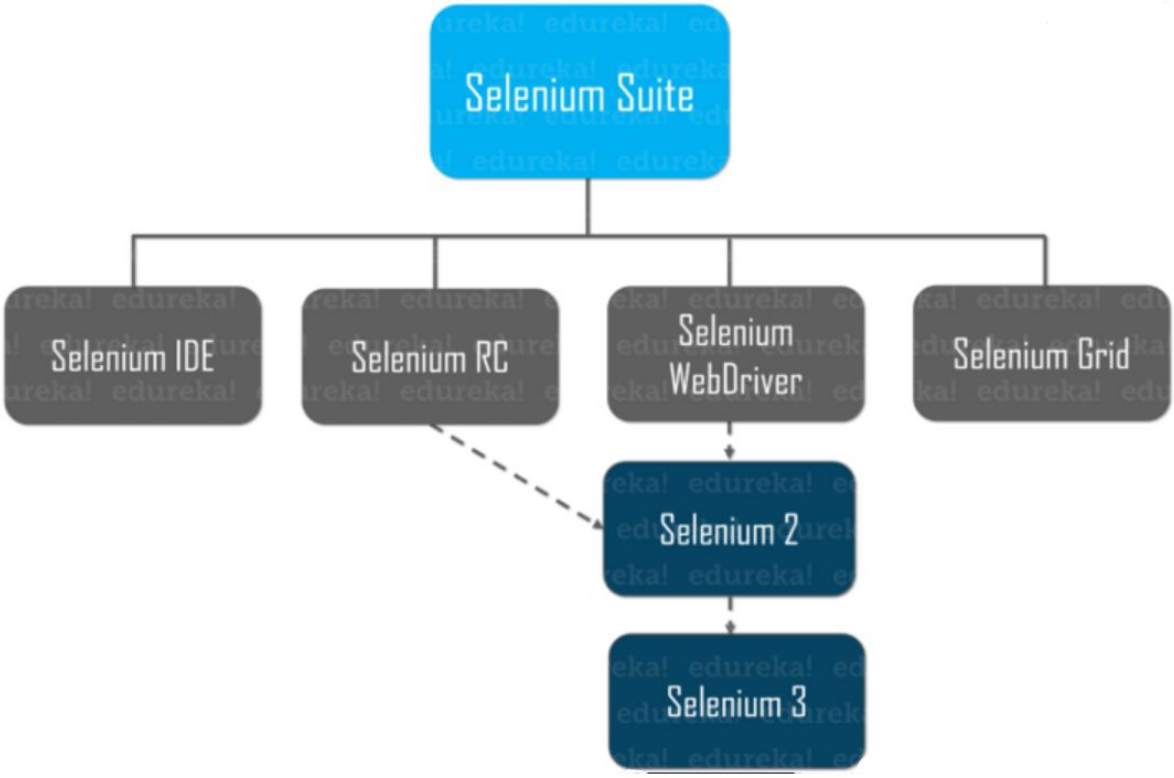

Рисунок 3.3 - Компоненти Selenium

Selenium IDE (Інтегроване середовище розробки) — це інструмент запису/запуску тестів у браузері користувача. Це надбудова або розширення, доступне як для Firefox, так і для Chrome, яке швидко генерує тести завдяки функції запису та відтворення. Нам не потрібно вивчати жодну мову сценаріїв тестів для створення будь-яких функціональних тестів.

У разі роботи з Selenium RC (Remote Control) необхідно добре знати хоча б одну мову програмування. Цей інструмент дозволяє розробляти тести адаптивного дизайну будь-якою мовою сценаріїв на наш вибір. Серверні та клієнтські бібліотеки є двома основними компонентами Selenium RC. Його архітектура складна і має свої обмеження.

Selenium WebDriver — це розширена версія Selenium RC. Він був представлений на ринку, щоб подолати обмеження, з якими стикається Selenium RC. Хоча це розширена версія RC, його архітектура повністю відрізняється від архітектури RC. Так само, як і Selenium RC, Selenium WebDriver також підтримує кілька платформ програмування для забезпечення більшої гнучкості та вимагає знання однієї мови програмування.

Selenium Grid — це інструмент, який використовується для одночасного виконання тестових випадків на різних браузерах, машинах та операційних системах одночасно. Цей інструмент спрощує перевірку на сумісність між браузерами. Існують дві версії Selenium Grid: старіша версія відома як Grid 1, а остання версія відома як Grid 2.

3.7.2 Принцип роботи Selenium Web-driver

Selenium WebDriver — це веб-фреймворк, який дозволяє виконувати міжбраузерні тести. Цей інструмент використовується для автоматизації тестування веб-додатків, щоб перевірити, чи він працює належним чином.

Selenium WebDriver дозволяє вибрати мову програмування для створення тестових скриптів. Як обговорювалося раніше, це прогрес у порівнянні з Selenium RC для подолання кількох обмежень. Selenium WebDriver не здатний обробляти компоненти вікна, але цей недолік можна подолати за допомогою таких інструментів, як Sikuli, Auto IT тощо.

Архітектура WebDriver складається з чотирьох основних компонентів:

- клієнтська бібліотека Selenium;
- протокол JSON через HTTP;

- драйвери браузерів;
- браузери.

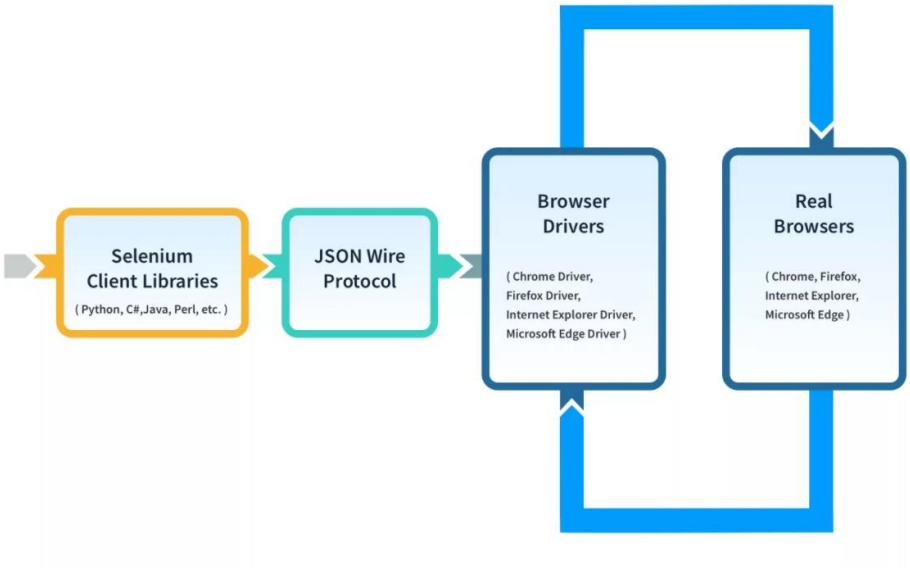

Рисунок 3.4 – Алгоритм роботи Selenium

Клієнтські бібліотеки Selenium забезпечують підтримку кількох бібліотек, таких як Ruby, Python, Java тощо, оскільки прив’язки мов були розроблені розробниками Selenium для забезпечення сумісності для кількох мов. Наприклад, якщо ми хочемо використовувати драйвер браузера та автоматизувати тести на Python. Ми можете завантажити всі підтримувані мовні прив’язки на наш вибір з офіційного сайту Selenium.

Протокол JSON Wire — це абревіатура від JavaScript Object Notation. Це відкритий стандарт, який забезпечує транспортний механізм для передачі даних між клієнтом і сервером в Інтернеті. Він забезпечує підтримку різних структур даних, таких як масиви та об’єкти, що полегшує читання та запис даних із JSON.

JSON служить REST (Representational State Transfer) API, який обмінюється інформацією між серверами HTTP.

Selenium надає драйвери, специфічні для кожного браузера, і без розкриття внутрішньої логіки функціональності браузера, драйвер браузера

взаємодіє з відповідним браузером, встановлюючи безпечне з’єднання. Ці драйвери браузера також є специфічними для мови, яка використовується для автоматизації тестів, як-от C#, Python, Java тощо.

Коли тестовий сценарій виконується за допомогою WebDriver, у фоновому режимі виконуються такі завдання:

- генерується HTTP-запит, який доставляється драйверу браузера для кожної команди Selenium
- запит HTTP отримує драйвер через HTTP-сервер
- усі кроки/інструкції, які потрібно виконати в браузері, вирішує сервер HTTP
- потім сервер HTTP отримує статус виконання і, у свою чергу, відправляє його назад сценаріям автоматизації.

4 УДОСКОНАЛЕННЯ МЕТОДУ ТЕСТУВАННЯ API ВЕБ-ЗАСТОСУНКУ ЧЕРЕЗ UI

Під час тестування веб-додатку було пройдено такі етапи:

- детальний огляд тестуємого об’єкта;
- написання плану тестування;
- створення ручних тестів;
- виконання ручного тестування API через інтерфейс користувача;
- створення фреймворку з авто-тестами;
- отримання результатів тестування;
- аналіз методу тестування API через інтерфейс користувача.

4.1 Детальний огляд об’єкту тестування

Для виконання тестування та наступного аналізу методу тестування API через інтерфейс користувача було використано веб-застосунок Національного університету «Полтавська політехніка імені Юрія Кондратюка» кафедри «Автоматики, електроніки та телекомунікацій» за посиланням «https://nupp.edu.ua/page/kafedra-avtomatiki-yelektroniki-ta-telekomunikatsiy.html». Даний веб-застосунок є інтернет сайтом, головна задача якого повідомляти усіх заінтересованих людей про новини щодо кафедри. Даний сайт має такий функціонал:

- загалну інформацію про кафедру;
- історія кафедри;
- склад кафедри;
- спеціальності;
- науково-дослідницька робота;
- міжнародне співробітництво;

- абітурієнту;
- іноземному студенту;
- студентське життя;
- контактна інформація.

Даний веб-застосунок має 2 інтерфейси. Інтерфейс користувача, та програмний інтерфейс. Користувач взаємодіє з сайтом через інтерфейс користувача. Програмний інтерфейс оброблює дії користувача та повертає різну корисну інформацію. Об’єктом тестування цієї кваліфікаційної роботи магістра є API даного веб-застосунку. Але в нас немає прямого доступу до програмного інтерфейсу. Тому це типова ситуація для використання та аналізу методу тестування API через інтерфейс користувача.

4.1.1 Огляд інтерфейсу користувача веб-додатку

Інтерфейс веб-застосунку є дуже простим у використанні. Коли ми заходим на сайт, ми можемо бачити таку домашню сторінку (рис.4.1).

На домашній сторінці ми маємо майже весь очікуваний функціонал новинного сайті:

- рядок пошуку;
- навігаційне меню з категоріями інформації;
- перемикач мови;
- посилання на сторінки в соціальних мережах.

Саме з цими елементами ми будемо працювати під час тестування веб-застосунку.

Інформація може бути різної категорії. Наприклад, історія кафедри, склад кафедри, спеціальності і інш. Приклади зображення інформації зображено нижче.

Рисунок 4.1 – Домашня сторінка

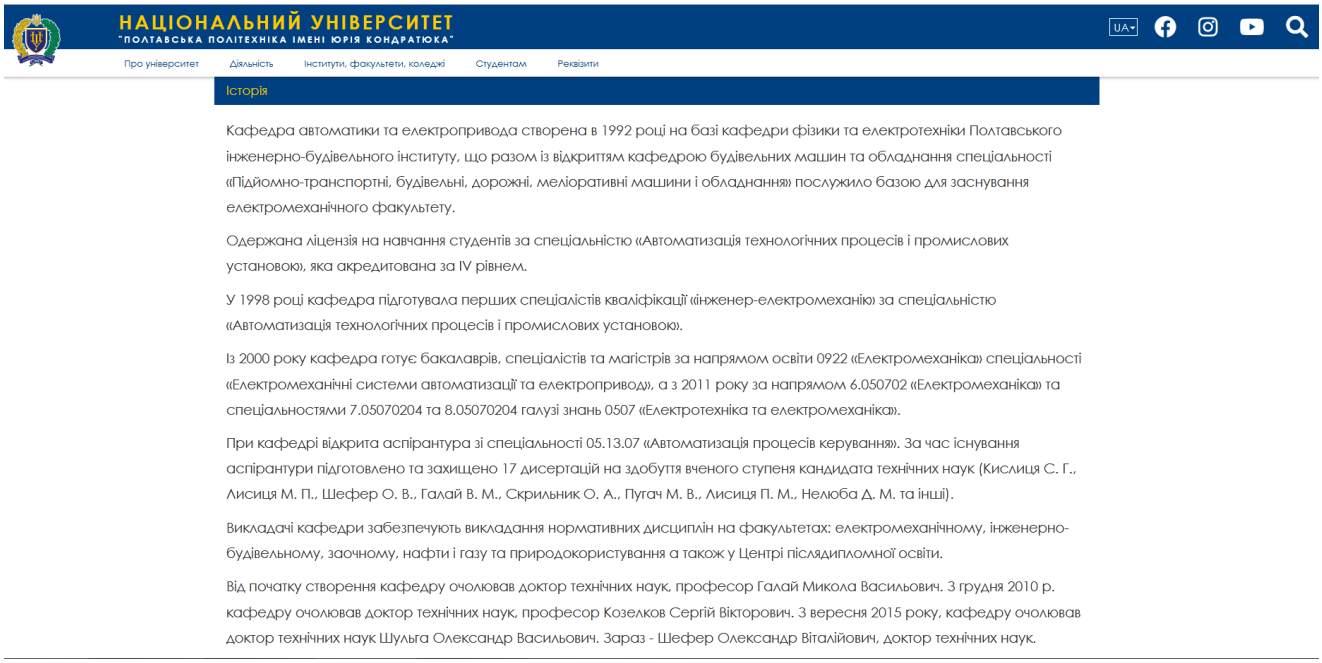

Рисунок 4.2 – Історія кафедри

Рисунок 4.3 – Склад кафедри

Рисунок 4.4 – Спеціальності кафедри

4.2 Створення плану тестування

План тестування – це детальний документ, який описує стратегію тестування, цілі, графік, оцінку, результати та ресурси, необхідні для проведення тестування програмного продукту. План тестування допомагає нам визначити зусилля, необхідні для перевірки якості програми, що тестується. План тестування служить планом для проведення заходів з тестування програмного забезпечення як визначеного процесу, який щохвилинно відстежується та контролюється менеджером тестування. Для проведення тестування веб-застосунку було створено план тестування. Розглянемо декілька його пунктів нижче [7].

Мета тестування: виконати тестування API веб-застосунку Національного університету «Полтавська політехніка імені Юрія Кондратюка» кафедри «Автоматики, електроніки та телекомунікацій» за посиланням «https://nupp.edu.ua/page/kafedra-avtomatiki-yelektroniki-ta-telekomunikatsiy.html» через інтерфейс користувача для виконання подальшого аналізу.

Що буде тестуватись: під час проведення тестування веб-застосунку буде тестуватись сайт взагалі. Тестування буде проводитись через тестування

основних компонентів сайту – це програмний інтерфейс, та інтерфейс користувача.

Що не буде тестуватись: не буде виконано тестування продуктивності компонентів сайту. Бо для проведення такого тестування потрібен прямий доступ до програмного інтерфейсу та згода з розробниками сайту. Без згоди з розробниками таке тестування є незаконним, тому що буде розцінюватись як DDoS атака. Атаки розподіленої мережі часто називають атаками розподіленої відмови в обслуговуванні (DDoS). Цей тип атаки використовує конкретні обмеження пропускної здатності, які застосовуються до будь-яких мережевих ресурсів, таких як інфраструктура, яка забезпечує веб-сайт компанії. DDoS-атака надсилає кілька запитів на атакований веб-ресурс – з метою перевищення можливостей веб-сайту обробляти декілька запитів, та запобігання належному функціонуванню веб-сайту [8].

Стратегія тестування включає в себе такі пункти:

1. Виконання тестування. Тестування виконується вручну та автоматизується. Такий підхід потрібен для аналізу методу тестування API через інтерфейс користувача.

2. Використовувані рівні для тестування. Було використано системний рівень тестування, який включає в себе запуск всій системи для проведення тестування.

3. Використовувані типи та види тестування.

Види тестування які було використано:

- функціональне тестування;
- локалізаційне тестування – робота системи з різними мовами;
- інтернаціоналізаційне тестування – перевірка граматики щодо новинних ресурсів;
- тестування макету – перевірка шрифту, кольору, розміру та розташування елементів інтерфейсу;
- регресивне тестування – виконується коли щось змінюється у коді або оточені застосунку.

Типи тестування які було використано:

- функціональне тестування – перевіряє бізнес-призначення додатку;
- тестування зв’язане зі змінами у коді – виконується при будь-яких змінах у коді.

4. Використані методи для тестування. Було використано метод чорного ящику та метод тестування API через інтерфейс користувача.

5. Збір вимог з налаштуваннями, необхідними для тестового середовища. Для виконання тестування має бути стабільне інтернет з'єднання від 5 мб/с, та стабільний хостинг на якому лежить веб-застосунок.

6. Необхідні інструменти для виконання тестування. Для автоматизування тестів було використано бібліотеку UI-тестування - Selenide. Бібліотека Allure використовується для збору результатів тестування та аналізу методу тестування через інтерфейс користувача.

Ризики які можуть бути під час тестування та варіанти їх рішення:

1. Нестабільний інтернет. Проводити тестування через іншого інтернет-провайдера.

2. Відключення електроживлення, через обстріли країною-агресором. Дочекатися відновлення електропостачання, або перейти у інше приміщення де є електрика.

3. Нестабільна робота комп’ютера або його компонентів. Використовувати інший комп’ютер.

4. Втрата комп’ютера з програмним кодом фреймворку автоматизованого тестування. Використовувати систему контроля версій з онлайн репозиторієм.

5. Оновлення версії інтерфейсу користувача на сайті. Регулярно оновлювати тести для роботи за останньою версією інтерфейсу користувача.

Критерії початку тестування: комп’ютер зі стабільним інтернет з’єднанням, та робоче місце. Критерії зупинки та продовження тестування: з’явлення одного з рисків які було описано вище, та їх подолання. Критерії закінчення тестування: покриття регресії авто-тестами, та 100% проходження авто-тестів.

Метрики які будуть збиратися під час тестування:

- загальна кількість тестів;
- кількість автоматизованих тестів;
- час виконання авто-тестів;
- кількість пройдених тестів.

4.3 Виконання ручного тестування

Після створення тест плану було виконано ручне функціональне тестування застосунку. Для виконання ручного тестування було використано такі техніки тест-дизайну як, Еквівалентний Поділ та Причина / Наслідок.

Еквівалентний Поділ: як приклад, у нас є діапазон допустимих значень від 1 до 10, ви повинні вибрати одне правильне значення всередині інтервалу, скажімо, 5, і одне неправильне значення поза інтервалом - 0.

Причина / Наслідок - це, як правило, введення комбінацій умов (причин) для отримання відповіді від системи (Наслідки). Наприклад, ми перевіряємо можливість додавати клієнта, використовуючи певну екранну форму. Для цього нам необхідно буде ввести кілька полів, таких як «Ім'я», «Адреса», «Номер Телефону», а потім натиснути кнопку «Додати» — це «Причина». Після натискання кнопки «Додати» система додає клієнта в базу даних і показує його номер на екрані — це «Наслідки».

Було використано підхід - Дослідницьке тестування. Дослідницьке тестування – це підхід до тестування програмного забезпечення, який часто описують як одночасне навчання, розробку тесту та виконання. Він зосереджений на виявленні та покладається на вказівки окремого тестувальника, щоб виявити дефекти, які нелегко охопити в рамках інших тестів [9].

Як вже було описано раніше, основною метою тестування даного дипломного проекту є аналіз методу тестування програмного інтерфейсу через інтерфейс користувача. Для використання даного підходу достатньо використовувати

справжні функціональні тести, бо під час використання застосунку буде викликатись його програмний інтерфейс та обробляти запити, які генерує інтерфейс користувача. Для проведення такого тестування було створено 50 тест-кейсів, які перевіряють функціональність застосунку через інтерфейс користувача. Приклад одного з тест-кейсів можна побачити нижче.

- openPage 1 sub-step
- clickOnContactUsButton 3 sub-step
- checkHeadOfDepartmentFullNameContactUsPage (Завідувач кафедри автоматики, електроніки та телекомунікацій Шефер олександр Віталійович) 2 sub-steps
- checkHeadOfDepartmentFullNameContactUsPage (avs075@ukr.net) 4 sub-steps
- checkHeadOfDepartmentFullNameAddress (корпус Ф, ауд. 314) 2 sub-steps
- checkHeadOfDepartmentContactUsPage (Наукові бази Google Scholar, Scopus, ORCID) 4 sub-steps
- returnToHomePage 2 sub-steps
- chargeLenguageToUkrainian 5 sub-steps
- chargeLenguageToEnglish 7 sub-steps

Рисунок 4.5 – Приклад тест-кейсу

На даному рисунку зображено тест, який перевіряє зображення контактної інформації на сторінці «Зв’язок з нами». Тест-кейс було написано у такому форматі Англійською мовою, тому що авто-тести пишуться Англійською мовою, та такого для полегшення автоматизації тестів. Зелена птичка на рисунку біля кожного кроку тесту позначає, що цей крок було успішно пройдено. На основі цього рисунку ми можемо зробити висновок що інформація на сторінці «Зв’язок з нами» зображено як очікувалось – правильно, для різних мов.

4.4 Створення фреймворку з авто-тестами

4.4.1 Алгоритм роботи фреймворку

Першим етапом створення фреймворку з авто-тестами – було розроблення блок-схеми з принципом його роботи. Розглянемо блок-схему нижче (рис. 4.6).

Кроки блок-схеми:

1. Запуск фреймворку з авто-тестами. Він може відбуватись через командну сторінку або через натиснення кнопки запуску у середовище розробки, наприклад Intelij Idea або Eclipse IDE.

2. Після запуску фреймворк зчитує ім’я та версію драйверу (браузеру) з спеціального property файлу у якому лежать необхідні параметри.

3. Після цього фреймворк скачує необхідні файли для обраної версії браузеру.

4. Далі відбувається ініціалізація об’єкту WebDriver з обраними параметрами у Java-коді.

5. Запуск процессу драйверу (браузеру), очищення кешу та встановлення повноекранного режиму зображення.

6. Установку класу Web-Driver до класу з передумовами для проведення тестування авто-тестами. Передумовами – це кроки які роблять застосунок готовим до проведення конкретного тесту або тестів.

7. Виконання авто-тесту, який проходить через драйвер (браузер)

8. Перевірка статусу авто-тесту. Якщо тест пройшов, то фреймворк закінчує свою роботу, якщо тест впав, то фреймворк робить скриншот у місці, де впав тест, потім закінчує свою роботу.

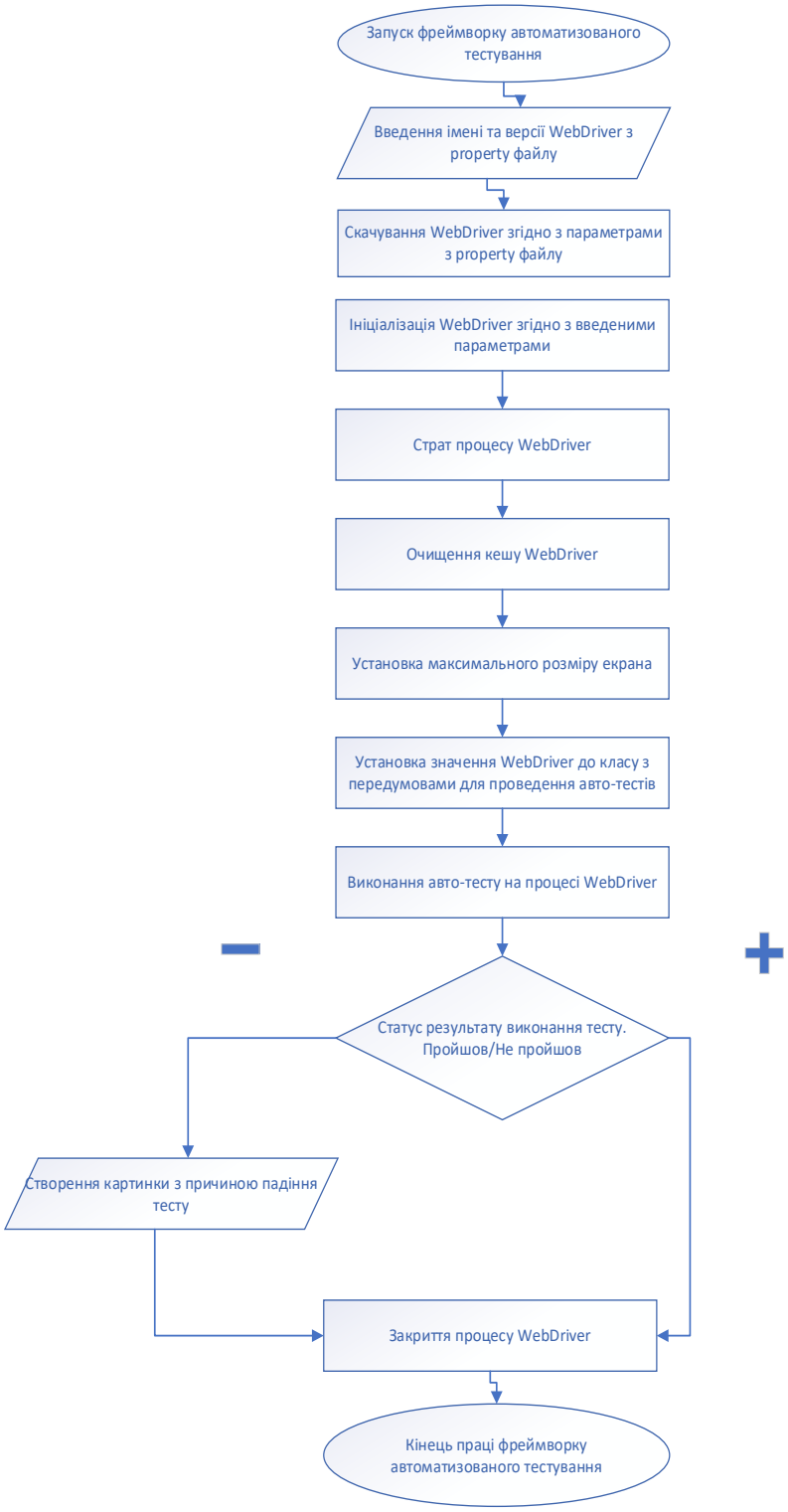

Рисунок 4.6 – Блок-схема фреймворку автоматизованого тестування

4.4.2 Керування контролем версій

Для керування контролем версій даного дипломного проекту було використано GitLab репозиторій. Розглянемо сторінку с проектом нижче.

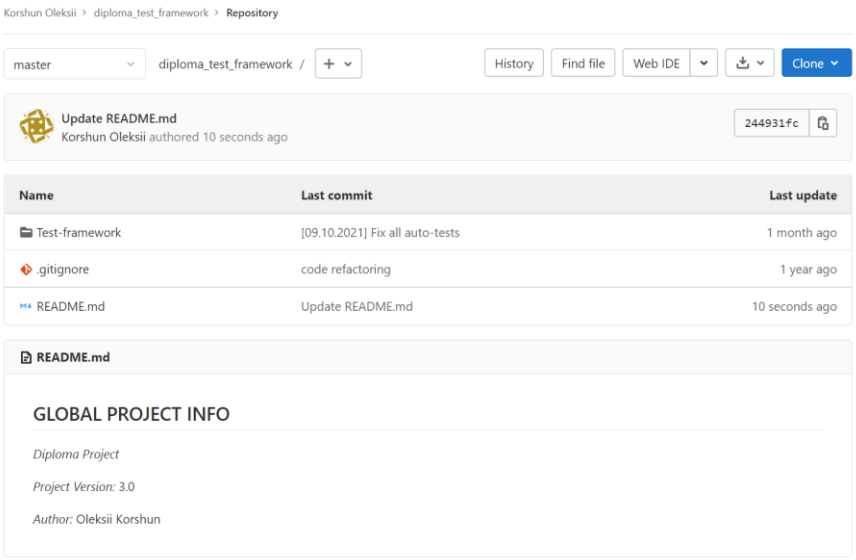

Рисунок 4.7 – Сторінка з репозитарієм проекту

На даній сторінці ми можемо бачити ім’я проекту “diploma_test_project”.

Проектні файли, та загальний опис проекту у якому вказано, що це 3-я версія фреймворку, яка получила стек технологій, про який було написано у розділі «Постановка завдання».

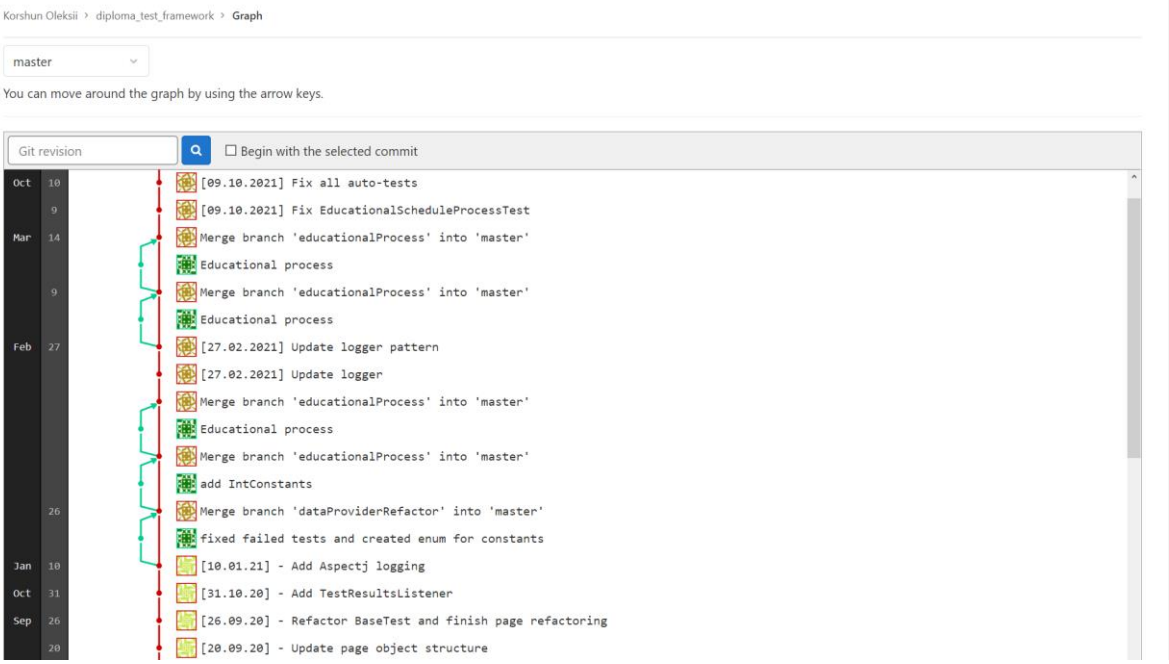

Рисунок 4.8 – Історія оновлювання коду до версії 3.0

Фреймворк має 3 версії. Версія 1.0 вміє лише виконувати авто-тесті, але не вміє генерувати звіт, і має лише 10 авто-тестів. Версія 2.0 вміє генерувати звіт, але має багато коду написаного не по код-конвенції, та має 23 авто-тести. У версії 3.0 було виконано рефакторінг згідно з код-конвенцією. Додано 27 авто-тестів. Також додано код для збору метрик та генерування логів.

4.4.3 Структура проекту

Під час створювання фреймворку за авто-тестами було створено структуру проекту з двома кореневими пакетами «nupp.core» та «ta_tests», які містять у себе інші пакети для різних цілей. Розглянемо структуру пакетів нижче:

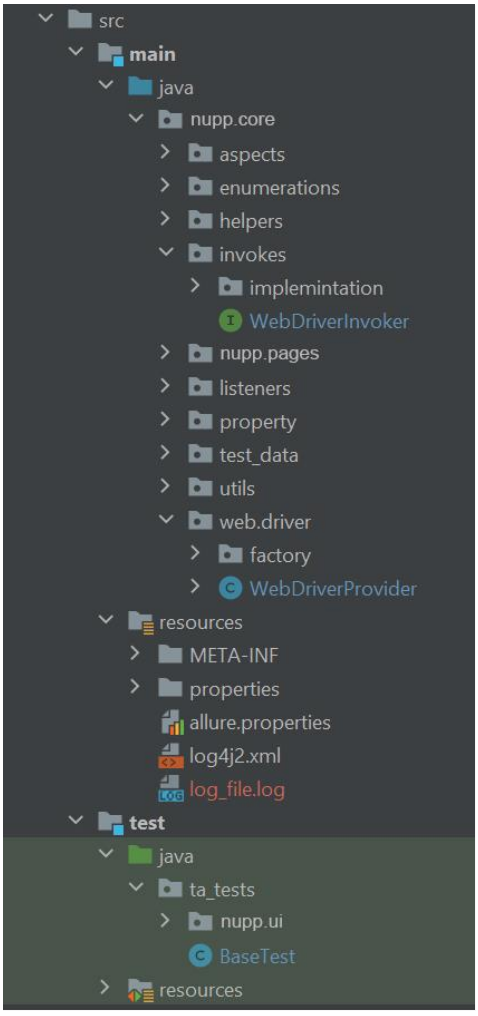

Рисунок 4.9 – Структура проекту

Перший кореневий пакет «nupp.core» містить у себе пакети для таких призначень:

- aspects – реалізація логування через AspectJ бібліотеку;
- enumerations – містить різні enum-об’єкти для запуску тестів;
- helpers – клас з повторюваними діями для тестів;
- invokes – інтерфейс для роботи з різними драйверами;
- invokes.implemintation – імплементація інтерфейсів для роботи з драйверами;
- kpi.pages – зображення сторінок веб-застосунку у вигляді java-об’єктів
- listeners – лісенери для бібліотеки TestNG;
- property – класи для роботи з .property файлами;
- test_data – класи з тестовими даними;
- utils – класи утильні класи;
- web.driver – класи для роботи з Web-Driver;
- web.driver.factory – класи, які генерують параметри запуску Web-Driver;
- main.java.resources – папка, які містить різні файли з конфігураціями.

Другий кореневий пакет «ta_tests» містить у себе класи з реалізацією тестів автоматизованих тестів. Папка test.java.resources - містить у себе конфігураційний файл для запуска тест-комплекту.

4.4.4 Отримання результатів тестування

Під час тестування застосунку було створено та автоматизовано 50 тест-кейсів. Для розгляду та аналізу отриманих результатів було використано генератор звітів о тестування Allure. Розглянемо результати тестування нижче.

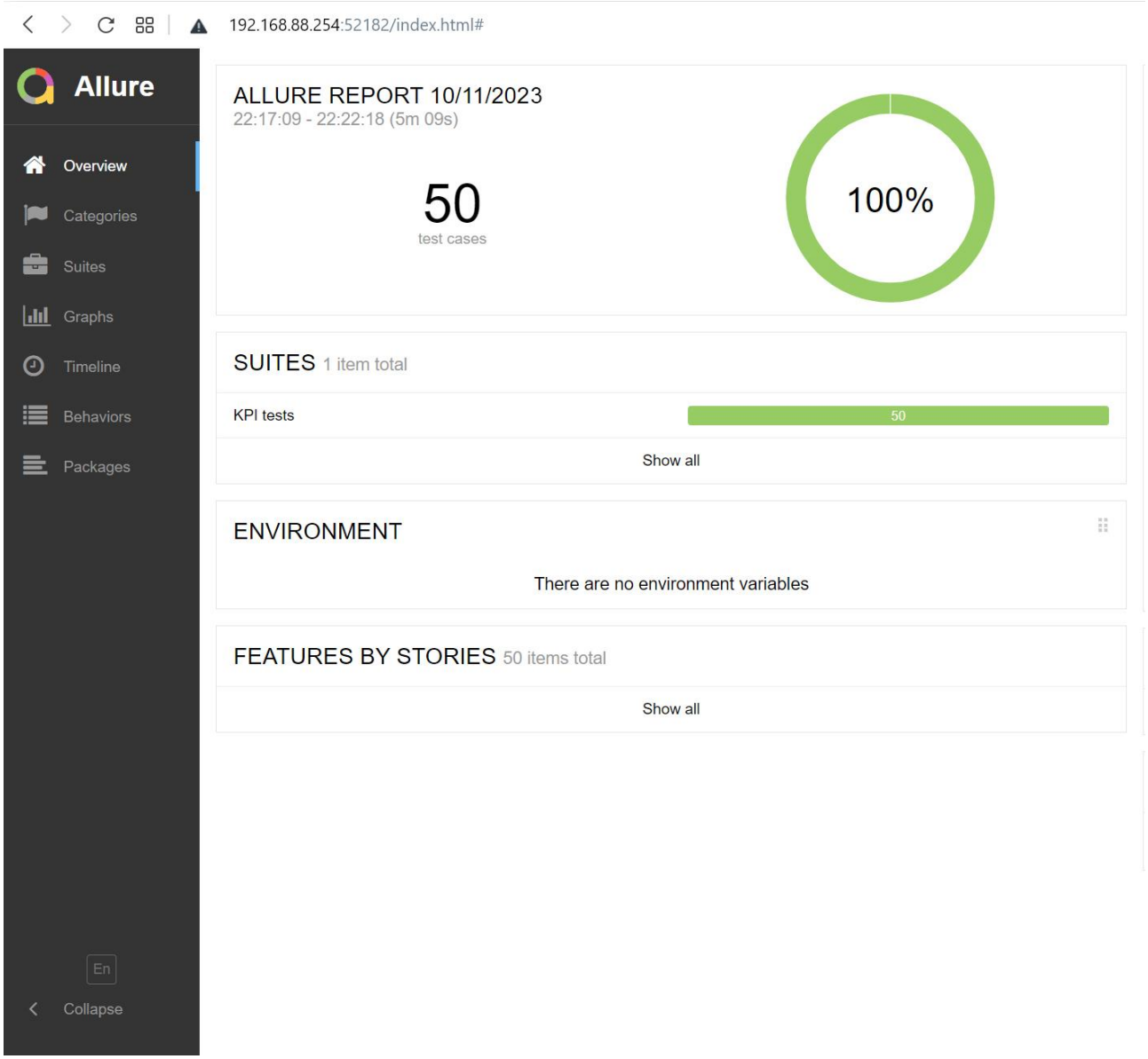

Рисунок 4.10 – Загальна кількість пройдених тестів

На даному рисунку ми можемо бачити, що усі 50 автоматизованих тестів було успішно пройдено, та виконання авто-тестів тривало 5 хвилин 9 секунд.

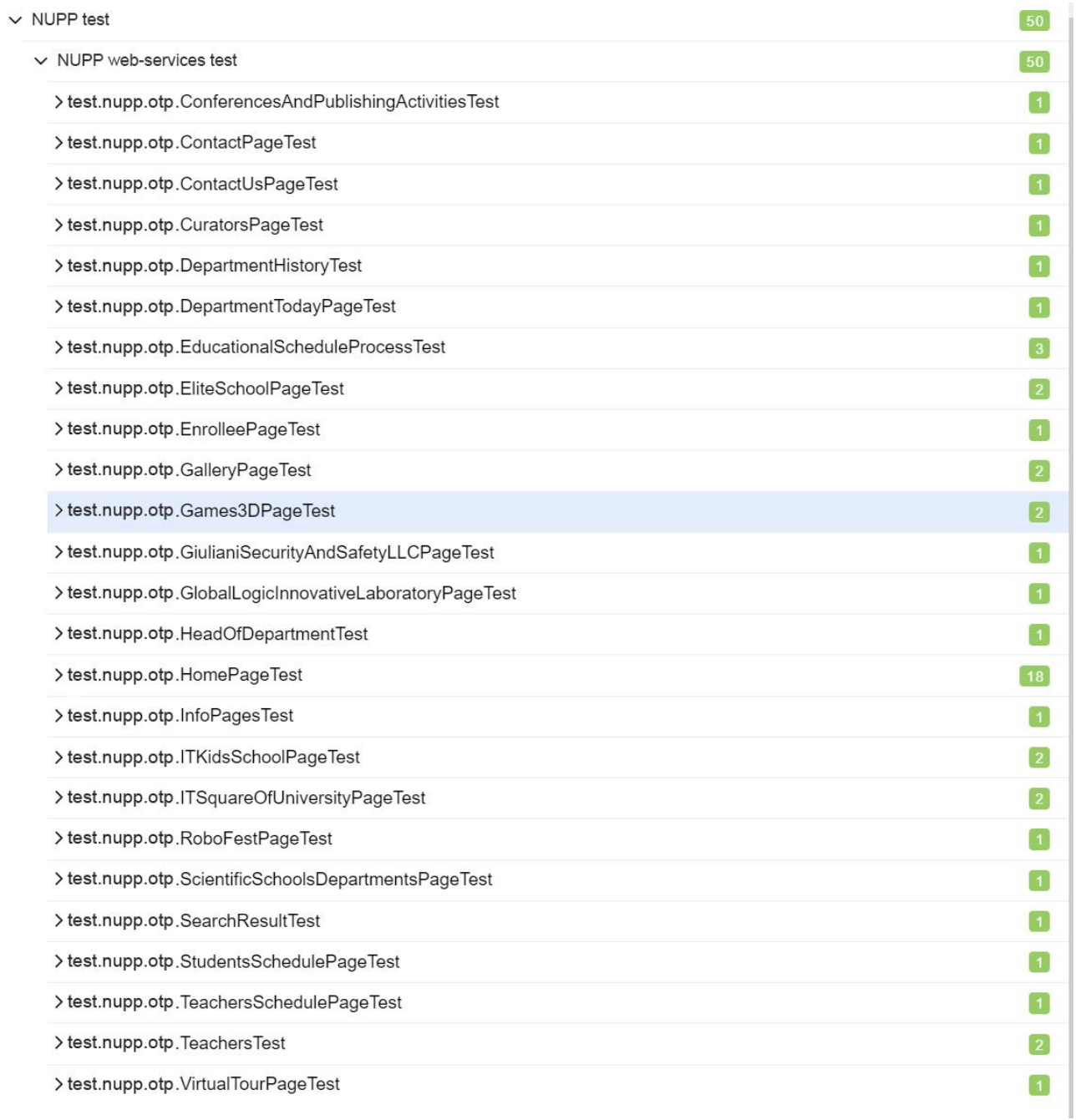

Рисунок 4.11 – Java класи з тестами

На рисунку 4.11 ми можемо бачити список усіх класів з тестам, та у якого класу скільки авто-тестів. Наприклад, у класі HomePageTest 18 тестів. Дані класи знаходяться у пакеті ta_tests.

На основі цього звіту з результатами тестування ми можем зібрати метрики, про які було написано у тест-плані.

- Загальна кількість тестів = 50 тестів
- Кількість автоматизованих тестів = 50 тестів

- Час виконання авто-тестів = 5 мінут 9 секунд
- Кількість пройдених тестів = 50 тестів

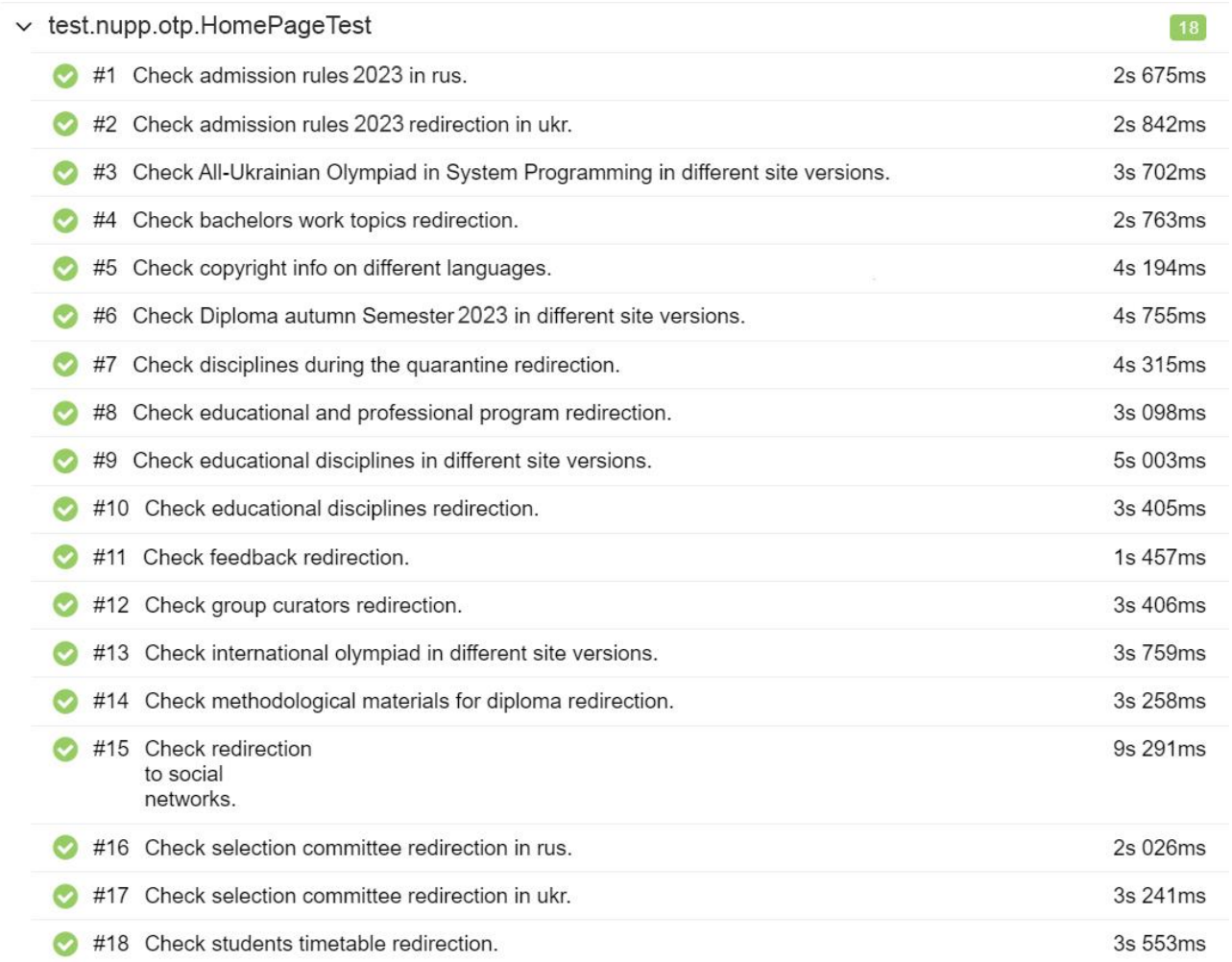

Рисунок 4.12 – Список тест-кейсів конкретного класу

Кожен Java клас з пакету ta_tests призначено для зберігання та запуску коду з авто-тестами. На рисунку 4.12 зображено приклад списку тестів конкретного класу з тестами.

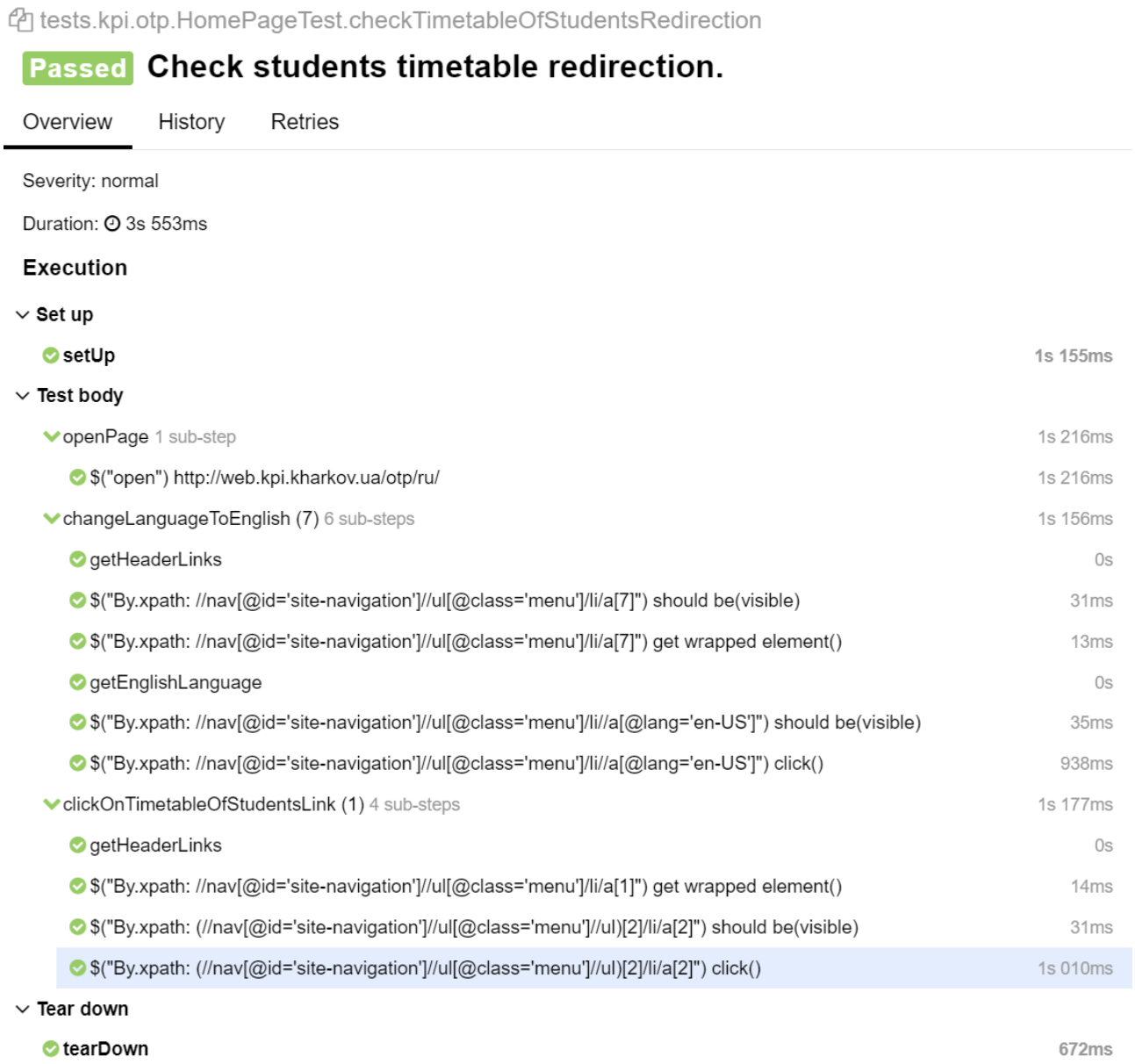

Рисунок 4.13 – Приклад конкретного авто-тесту

На рисунку 4.13 ми можемо бачити кроки конкретного авто-тесту, який знаходиться у класі з тестами «HomePageTests».

4.4.5 Аналіз методу тестування API через інтерфейс користувача

Метрика «кількість автоматизованих тестів» яку було зібрано вище зображує загальний час виконання 50 авто-тестів API через інтерфейс користувача.

Якщо у нас був прямий доступ до API веб застосунку, ми б мали можливість відправляти API запити до серверу та перевіряти відповідь від серверу застосунку у json форматі. Без прямого доступу ми маємо можливість відправляти запити до серверу, але він поверне дані у html форматі. Але нам достатньо цього для перевірки часу реакції від серверу.

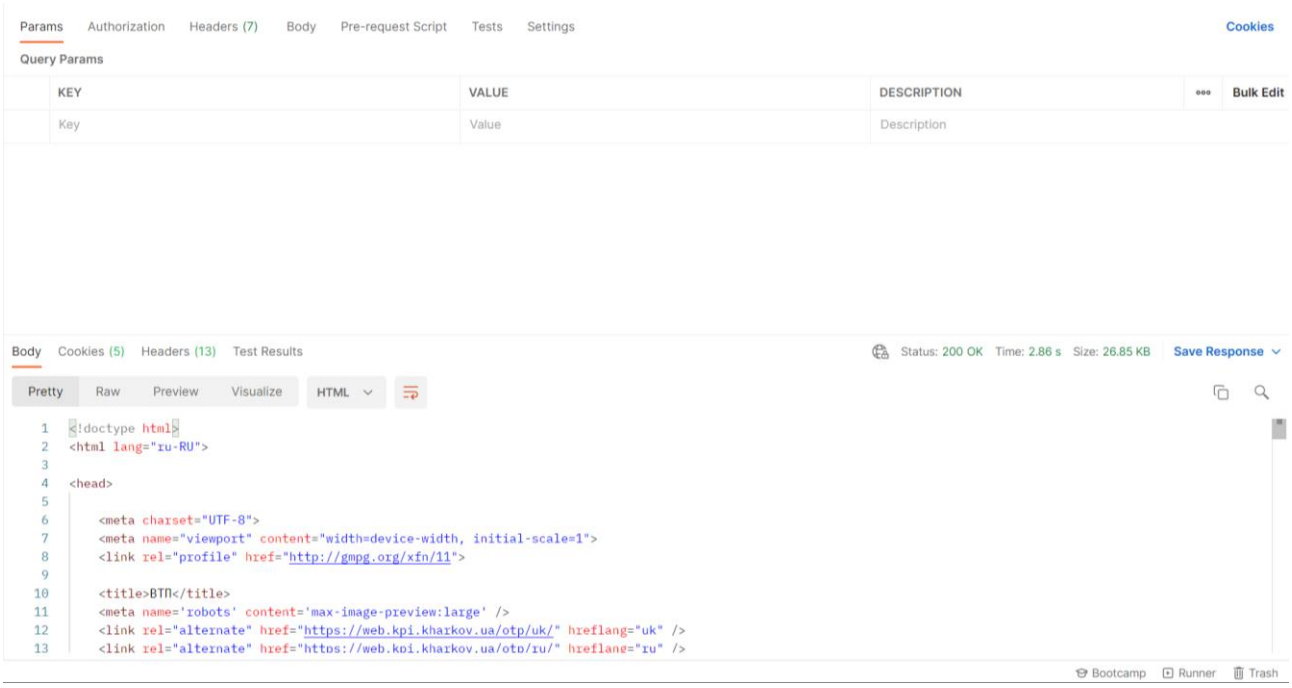

Рисунок 4.14 – API Запит до серверу

На даному рисунку ми можемо бачити, що час реакції від нашого серверу дорівнює 1071 мілісекунд. Підрахування тривалості виконання авто-тестів через API виконується таким чином: 1 тест дорівнює 1 запит. Ми маємо 50 тестів, це позначає що в нас буде 50 запитів до серверу. Кількість одного запиту, як ми бачимо на рисунку, дорівнює 1071 мілісекунд – це приблизно 1 секунда. На основі цього ми можемо зробити висновок, що якщо ми б

виконували тестування через API, то загальний час виконання авто-тестів тривав би 50 секунд.

На основі зібраних даних ми можемо порівняти метод тестування API через інтерфейс користувача з звичайним API тестуванням. Якщо виконувати наші тести через інтерфейс користувача, то загальний час виконання авто-тестів дорівнює 5 мінут 9 секунд. Якщо виконати ти ж самі тесті через API, загальний час виконання авто-тестів дорівнює 50 секунд, що є набагато швидше. Це є основною перевагою тестування через API. Але, при тестуванні через UI використовується UI прошарок, це робить тестове покриття більшим ніж при використанні API. Тому, коли треба обирати підхід до тестування API. Треба дивитись на різні фактори, бо кожен метод тестування є гарним для конкретної ситуації.

4.4.6 Альтернативні сценарії тестування

Існує ряд проблем, пов'язаних із тестуванням API:

1. Відсутність GUI;

2. Обов'язковість перевірки обробки винятків;

3. Необхідність знань у програмуванні для тестувальника.

API має чіткий формат передачі даних, що спрощує процес обміну інформацією.

Для цього використовують запити HTTP.

HTTP – протокол прикладного рівня моделі OSI, що використовується для передачі даних. Спочатку з його допомогою була можлива передача лише гіпертекстових документів у форматі HTML, але згодом це умова була доопрацьована до передачі довільних даних. Протокол HTTP є основою мережі Інтернет, забезпечуючи клієнт-серверне взаємодія додатків.

Взаємодія клієнтської та серверної частин програми здійснюється за допомогою відправки запитів та обробкою отриманих на них відповідей.

HTTP-запит складається з методу HTTP, шляху до ресурсу, версії HTTP - протоколу, заголовка та тіла запиту (необов'язково). HTTP-відповідь містить також версію протоколу, код стану (помилки), повідомлення стану, заголовок та тіло відповіді [6].

HTTP-метод – слово, що визначає операцію, яку виконує клієнт.

Зазвичай є дієсловом (GET, POST, та ін), проте може бути і іменником (OPTIONS, HEAD). Найбільш використовуваними методами HTTP є:

- GET - отримання та читання даних;

– PUT – завантаження інформації на вказаний у запиті URI;

– POST – передача даних від користувача до ресурсу;

– DELETE – видаляє ресурс;

– OPTIONS – визначає можливості веб-сервера.

Розглянемо виконання запитів за допомогою програми Postman.

Postman – додаток, призначений для тестування різних API, а також надсилання різних запитів на сервер. Важливою властивістю програми є можливість створення ази запитів до API, що значно прискорює розробку та оптимізує час та ресурси, часто витрачаються на пошук запиту у пам'яті чи ручне заповнення параметрів. З Postman досить легко працювати, тому багато тестувальників вибирають його як інструмент автоматизації [5]. З метою розуміння структури виконання запиту у Postman як при стандартному виконанні, так і при проведенні тестування, розглянемо ключові поняття, які у ньому. До них відносять Колекції (Collections), Папки (Folders) та Запити (Requests). Колекція – це файл проекту, до якого входять усі запити, у ній є як свої скрипти, так і свої змінні. Також Колекції зберігають історію попередніх запитів. Папка поєднує запити в одну групу всередині колекції, у неї можуть бути свої скрипти, але не змінні. Запити створюються в конструкторі, але також у них є власні скрипти.

Важливо відзначити, що такі тести в Postman можуть мати складнішу структуру. У них може відбуватися перевірка кількох умов, аналіз даних і т.д. Однак, незважаючи на складність складання таких тестів, їх важливою перевагою є значне прискорення процесу розробки, тестування, а також мінімізація людського фактора при перевірки помилок.

Таким чином, можна зробити висновок, що автоматизоване тестування API має більше переваг, ніж недоліків, оскільки його ціна досить швидко окупається, а швидкість перевищує швидкість ручне тестування. Єдиний недолік – необхідність вміння програмувати для тестувальника, але для цього існує безліч інструментів, які можуть допомогти у створенні тестів. Одним із них можна назвати програму Postman, яка допомагає створювати, тестувати та документувати програму за допомогою одного ресурсу.

Насправді, якщо в технічному завданні немає окремо виділеного сценарію використання, то можна об'єднати пункти “альтернативні сценарії” та “негативне тестування”, тому що за фактом після базового тесту ми:

1. Читаємо кожну пропозицію з технічного завдання, «Особливості використання».

2. Продумуємо, як його перевірити: як позитивно, й негативно.

Але для чистоти експерименту спробуємо рознести ці пункти окремо. Тоді, в альтернативі, потрапляють усі додаткові умови, які накладаються на дані, що посилаються або повертаються.

Тестувальник завжди буде «SOAP / REST», змінювати його можна тільки через відповідний метод.

Тестувальника перевірили, але тільки в технічному завданні він вказаний kaps, а за фактом створюється в нижньому реєстрі. Це вже невеликий bag, швидше за все, документації, оскільки некритично і простіше оновити. Зробили нотатку / самі виправили документ, якщо є доступ.

А далі бачимо, що сценарій можемо змінювати лише через відповідний метод. Створили через REST, міняти можна також через REST, через SOAP не можна. І навпаки.

Для початку подивимося, чи є можливість відкоригувати власну картку - ні, відкриваємо в SOAP Ui WSDL проекту - http://users.bugred.ua/tasks/soap/WrapperSoapServer.php?wsdl, і дивимося, яким способом можна вносити зміни. Підійде UpdateUserOneField, він оновлює одне поле користувача.

У Users описані не всі методи, але є опис CreateUser, де можна взяти назви полів.

4.4.7 Негативний сценарій перевірки

За фактом, це перевірка повідомлень про помилки. З бізнесової точки зору дуже зручно, коли всі помилки прописують прямо в технічному завданні. Це можна поділити на «Особливості використання» і «Виняткові ситуації», як у Folks [8]. Тоді тестуємо блок "Виняткові ситуації". Опис Jira Cloud REST API,

виберемо в лівому навігаційному меню якийсь метод, наприклад Delete avatar. Там є опис методу, а потім у блоці Responces перемикання між кодами відповідей.

Жовті коди - це помилки. Відкриваємо кожну, читаємо «Returned if» і виконуємо цю умову - дуже зручно:

А далі йдуть тести, які, за ідеєю, повинні давати помилку:

1. Email без точок у доменній частині — test@mailcom

2. Перевищення довжини email (>320 символів)

3. Відсутність @ у email - testmail.ua

4. Email з пробілами в імені користувача – testuser@mail.ua

5. Email із пробілами в доменній частині — test@ma il.ua

6. Email без імені користувача - @mail.ua

7. Email без доменної частини - test@

8. Некоректний домен першого рівня (припустимо 2-6 букв після крапки: .ua) — test@mail.uauauaua

Можливі ситуації, які треба перевірити:

Заголовок не передано, як система реагує:

- видає помилку → чи зрозуміло, що мені треба зробити? Застосовується стандартна поведінка (якщо мова не передана, використовуй українську)

Заголовок переданий:

- вірно;

- невірно → яка помилка?

Плюс реєстрозалежність. Відповідно до специфікації, всі заголовки мають бути реєстронезалежними. І не має значення спосіб передачі. Можливо CamelCase (перша літера велика, решта дрібних), можливо у верхньому регістрі, причому як значення заголовка, так і його назва:

• Accept: application/json;

• Accept: APPLICATION/JSON;

• ACCEPT: application/json;

• ACCEPT: APPLICATION/JSON;

• ACcePT: APPlicATIon/JSon.

Перевірити ці варіанти дуже важливо. Тому що за реєстронезалежність відповідає розробник, сама по собі з повітря вона не з'явиться. Він має прописати це у коді.

Accept може бути:

• application/json;

• application/xml.

Запит йде безпосередньо на сервер, він проходить через проксі Nginx. А Nginx змінює заголовки на upper case: ACCEPT: APPLICATION/JSON.

Система до такого не готова, вона шукає Accept, не знаходить його і видає помилку. А переданий заголовок ігнорує. Так що перевіряти регістр обов’язково потрібно.

Розробник може використовувати стандартні коди відповідей:

• 2xx - все добре;

• 4xx – помилка на клієнті (у запиті);

• 5xx – помилка на сервері;

А може вигадати свої:

• 570 - порожня відповідь на пошук;

• 571 - знайшли одного ФО;

• 572 – знайшли кілька ІП;

• 573 – знайшли лише ІП;

• …

Розробник може навіть перекрити стандартні:

400 - Bad Request

↓

400 - Internal Server Error

Але так краще не робити =))) Просто така можливість є.

Якщо розробник не налаштував коди, то він побачить помилку стандартний код: 400 Bad Request. Неправильний заголовок? Помилка 400.

Неправильне тіло? Помилка 400. Проблеми у бізнес-логіці? Помилка 400 ... Ну і як тут розібратися, що потрібно виправляти?

Або ще "краще", розробник може сказати "завжди повертай код 200". І це дуже погано, адже саме за кодом ми орієнтуємося насамперед: все добре чи щось погано?

Тому статус-код перевіряємо завжди, звертаємо на нього увагу.

Підсумовуючи, можна зазначити, що тестування API є важливим етапом у розробці програмного забезпечення з багатьох причин:

1. Перевірка відповідності вимогам: тестування API дозволяє переконатися, що API відповідає вимогам, визначеним у специфікації чи документації. Це допомагає виявити помилки та недоліки в API на ранніх стадіях розробки і виправити їх до того, як вони призведуть до більш серйозних проблем у продукті.

2. Забезпечення функціональності: тестування API дозволяє переконатися, що API працює правильно та забезпечує необхідну функціональність. Це важливо, тому що API може використовуватись іншими додатками або системами, і будь-які проблеми в API можуть призвести до неправильної роботи цих додатків чи систем.

3. Забезпечення безпеки: тестування API допомагає виявити вразливості у безпеці API, такі як можливість ін'єкцій, підробки або обходу аутентифікації. Це дозволяє розробникам виправити ці уразливості до того, як вони будуть використані зловмисниками.

4. Оптимізація продуктивності: тестування API дозволяє переконатися, що API працює ефективно і не викликає затримок або перевантажень у системі. Це особливо важливо у випадках, коли API використовується для передачі великої кількості даних або обробляє велику кількість запитів.

5. Спрощення інтеграції: тестування API дозволяє виявити та виправити проблеми в інтеграції з іншими програмами або системами.

Це спрощує процес інтеграції та зменшує ризик проблем у процесі інтеграції [2].

Тестування API є невід'ємною частиною розробки програмного забезпечення, оскільки API є ключовим компонентом для взаємодії між різними додатками та системами. Правильне тестування API допомагає забезпечити безпеку, функціональність, продуктивність та надійність API, що в свою чергу покращує якість та надійність всього додатка чи системи.

Нижче наведено найбільш важливі типи тестування API:

1. Перевірка правильності роботи API, відповідності очікуваному результату та правильності обробки вхідних параметрів.

2. Перевірка здатності API обробляти великі обсяги запитів та тестування продуктивності в умовах високого навантаження.

3. Перевірка коректності процесу аутентифікації та авторизації, включаючи перевірку правильності видачі токенів та керування доступом.

4. Перевірка наявності та ефективності заходів захисту від загроз безпеки, таких як ін'єкції, підробки та обхід автентифікації.

5. Перевірка стабільності API, його здібності працювати без збоїв та зупинок.

6. Перевірка сумісності API з різними платформами, браузерами та пристроями.

7. Перевірка можливості API працювати ефективно в умовах масштабування, коли збільшується кількість користувачів та обсяги даних.

8. Перевірка можливості API обробляти помилки та відновлюватися після збоїв або відмов.

9. Перевірка продуктивності API, включаючи час відгуку та час обробки запитів [3].

Кожен із цих типів тестування важливий для забезпечення якості та надійності API і має бути включений у процес тестування API.

Тестування API може проводитись як вручну, так і за допомогою спеціалізованих інструментів для тестування API, таких як Postman, SoapUI, Swagger та інші.

Інструменти та фреймворки для тестування API – це програмні засоби, що дозволяють автоматизувати тестування API, спрощують процес створення, відправлення та обробки запитів та відповідей на API, та дозволяють швидко та точно виявляти проблеми в API [1].

Деякі з найбільш популярних інструментів та фреймворків для тестування API включають:

1. Postman – популярний інструмент для тестування API, який дозволяє створювати запити API, надсилати їх на сервер та аналізувати відповіді. Postman також дозволяє автоматизувати тестування API за допомогою колекцій запитів, скриптів та тестових сценаріїв.

2. SoapUI – інструмент для тестування веб-сервісів, який дозволяє створювати та відправляти SOAP- та REST-запити на сервер та аналізувати відповіді. SoapUI також дозволяє автоматизувати тестування API за допомогою скриптів та тестових сценаріїв.

3. Swagger – інструмент для створення, документування та тестування API. Swagger надає можливість створювати API- специфікації у форматі OpenAPI та автоматично генерувати код клієнтів та серверів на основі цих специфікацій.

4. RestAssured – це Java-фреймворк для тестування REST API, який дозволяє створювати та надсилати запити на сервер та аналізувати відповіді. RestAssured також дозволяє автоматизувати тестування API з за допомогою Java-коду.

5. JMeter – це інструмент для тестування продуктивності, який може використовуватися для тестування API в умовах високої навантаження. JMeter дозволяє створювати та відправляти запити на сервер та аналізувати час відгуку та час обробки запитів.

Ці інструменти та фреймворки допомагають скоротити час та зусилля, необхідні для тестування API, підвищують точність та надійність тестування і дозволяють забезпечити більш високу якість API.

4.4.8 Архітектура мережевої взаємодії компонентів

Архітектурний стиль мережевої взаємодії компонентів розподіленої програми REST не є протоколом. Це набір правил того, як розробнику організувати написання коду серверної програми, щоб усі системи ефективно обмінювалися даними, і програму можна було масштабувати.

У стилі REST можна виділити кілька характерних рис:

1. Прив'язка до методів HTTP і, відповідно, використання лише цього протоколу передачі даних.

2. Використання різних форматів передачі даних, таких як JSON, XML, HTML. Найчастіше використовується JSON через його легкість, простоту і людину читання.

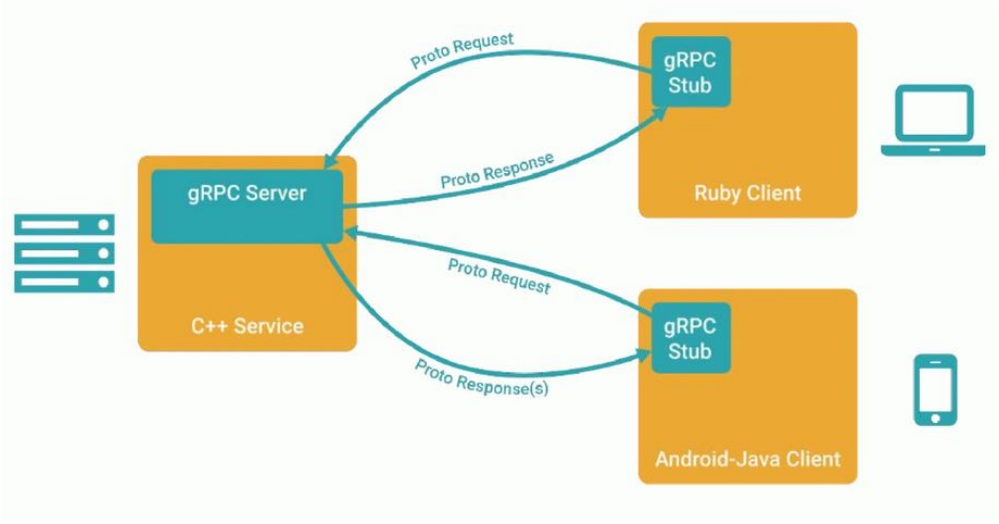

Рисунок 4.15 – Архітектурний стиль мережевої взаємодії компонентів розподіленої програми REST (gRPC)

Система віддаленого виклику процедур, розроблена Google. Як транспорт gRPC використовує протокол HTTP v2. Найчастіше gRPC допомагає організувати взаємодію у мікросервісній архітектурі. Реалізація спілкування через gRPC відбувається так: клієнтська програма може безпосередньо викликати метод серверної програми як би це був локальний метод клієнта.

Відмінна риса реалізації додатків за допомогою gRPC API – наявність компонента gRPC STUB. Це модуль, який конвертує дані з людиночитаних у бінарні файли та передає їх між клієнтом та сервером. Ще одна невід'ємна частина gRPC API - це proto файл - опис того, які методи є в додатку, і взаємозв'язок між запитом і відповіддю [8].

Познайомившись із прикладами реалізації API, стає зрозуміло, що вони є структурою виду «запит - відповідь». У запиті потрібно передавати деякі параметри, у відповідь на які надходять відповідні дані.

4.4.9 Інструменти для тестування API

Існують десятки різноманітних інструментів, за допомогою яких тестують API [9]. Для їх порівняння, сформулюємо основні критерії:

1) Можливість надіслати власні параметри запиту.
2) Принципова можливість звернутися до ендпойнтів програми.
3) Здатність зберігати тест-кейси.
4) Швидкість написання тестів.
5) Швидкість освоєння інструмента.
6) Можливості автоматизації перевірки очікуваних результатів.
7) Додаткові можливості для автоматизації тестування.
8) Можливості використання змінних оточення.
9) Здатність використовувати мок-сервіси, заглушки тощо.
10) Особливості запуску готових тестів.
11) Інтеграція із системами CI/CD.
12) Додатковий функціонал, що полегшує тестування.

До стандартного функціоналу сваггероподібних систем входять: відправка запитів, перегляд відповідей, можливість подивитися готові приклади запитів та відповідей, варіанти авторизації, можливість експорту відповіді у файл та генерація cURL запиту.

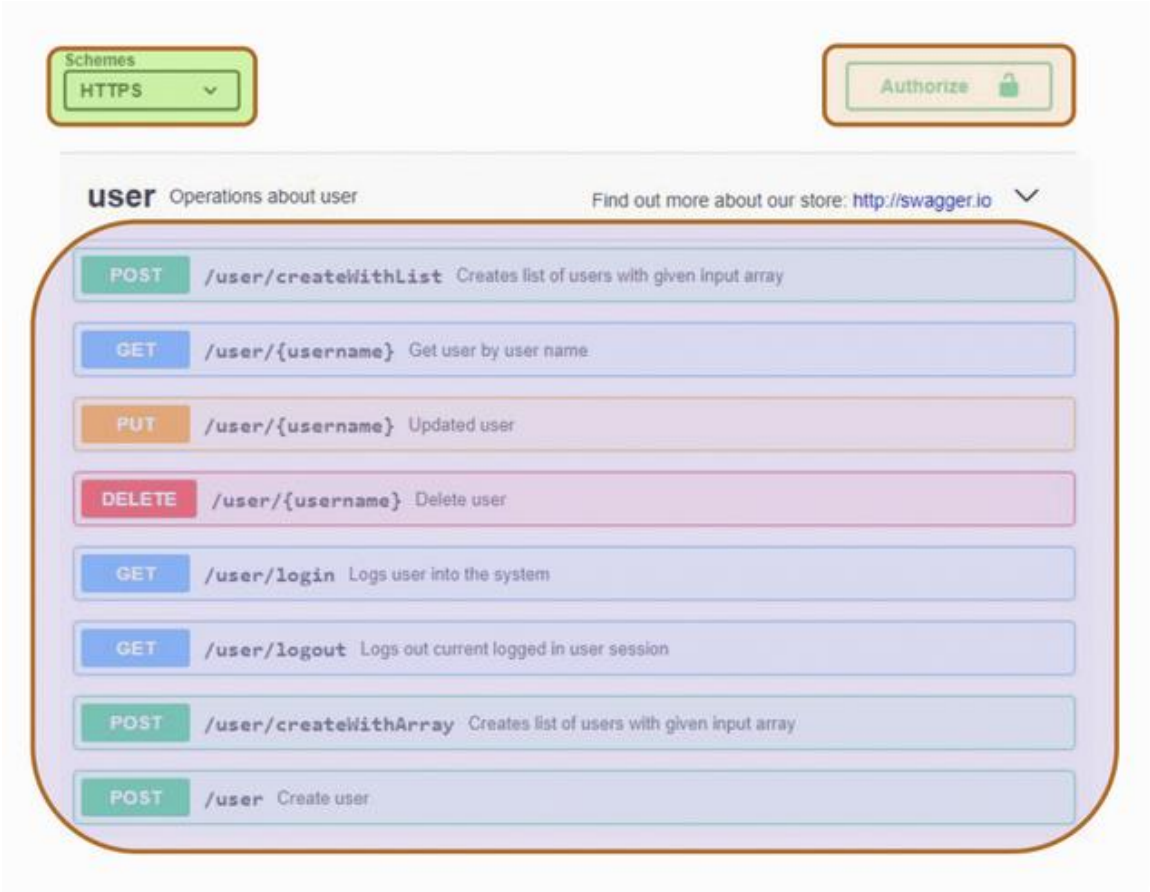

Рисунок 4.16 – Стандартний функціонал сваггероподібних систем

Переваги сваггероподібних систем:

1. Повністю готові інструменти, в яких вже створено всі звернення до ендпойнтів, тіла запитів та інші параметри.

2. Швидкі та прості у освоєнні та використанні через елементарний функціонал.

3. Містять приклади відповідей і опис параметрів, що в них приходять.

4. Не потрібні установки на пристрій користувача, вони вже розгорнуті на стенді.

5. Генерують cURL, який можна додати до баг-репорту, щоб розробник міг швидше відтворити баг та виправити його.

Недоліки сваггероподібних систем:

1. Вони зберігають лише заздалегідь прописані розробником параметри запитів, які можуть бути надмірними чи навпаки, неповними.

2. Не можна зберігати параметри запитів. Тобто після поновлення сторінки сваггероподібної системи пропаде все, що ви туди ввели. Таким чином, параметри запитів, які використовуються у тест-кейсах, доведеться зберігати в окремих документах, а це незручно.

3. Не можна створювати колекції запитів, оскільки неможливо зберігати варіанти запитів.

4. Не можна створювати скрипти-перевірки, очікуваний результат з фактичним доведеться звіряти очима/руками.

5. Неможливі прогони, тобто автоматизований послідовний запуск запитів.

6. Не можна створювати та використовувати жодні оточення, змінні.

7. Корисність описів у сваггероподібних системах дуже відрізняється від проекту до проекту.

Коли вибирати сваггероподібні системи? Так як це не інструмент тестування, а інтерактивна документація API, такі системи частіше використовуються для ознайомлення з ендпойнтами та дослідницьких перевірок. В інших випадках використовувати сваггероподібні системи не рекомендується, хіба що ви працюєте на специфічному проекті та інші інструменти використовувати неможливо. Наприклад, закритий проект, де ви працюєте на віртуальних машинах та віддалених робочих столах, і при цьому немає можливості встановити жодний з API-клієнтів. Тоді все, що ви можете використовувати Swagger або подібну систему, щоб тестувати ендпойнти сервісу.

Інструменти середнього рівня складності. Це, як правило, десктопні програми, які дозволяють звернутися до ендпойнтів. Також вони надають додаткові можливості використання змінних, рандомайзерів, скриптів та інших функцій.

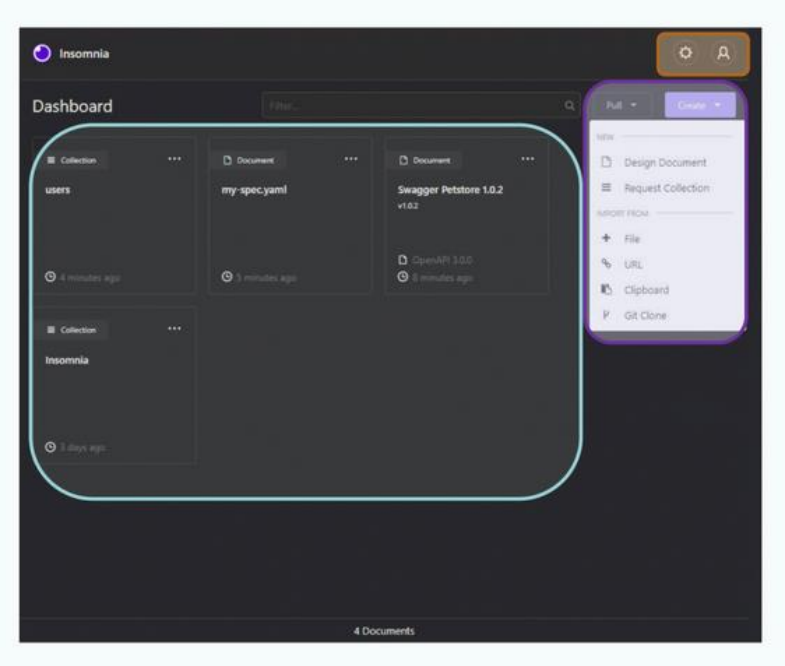

Рисунок 4.17 – Діалогове вікно функціоналу Insomnia

При відкритті Insomnia ми бачимо головну сторінку, на якій можна налаштувати саму програму і свій обліковий запис, меню створення проектів, а також список вже створених. Додати новий проект можна у різний спосіб: імпортування з локального файлу, «підтягування» із зовнішнього ресурсу за URL, копіювання API проекту з Git.

При відкритті проекту стає доступним функціонал додавання нових та список створених запитів. Вікно запиту дає можливість вводити і зберігати параметри, що відправляються. Подивитися і зберегти як приклад відповідь, що приходить, можна у вікні праворуч.

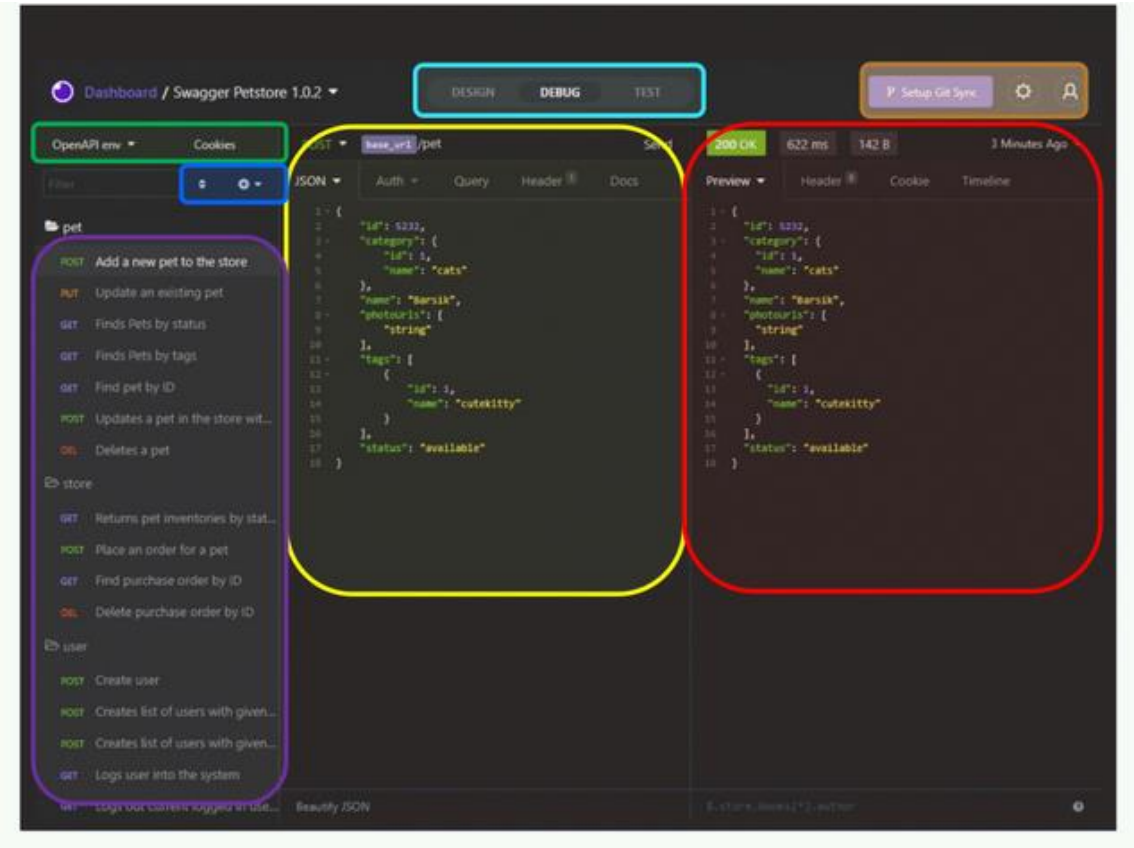

Рисунок 4.18 – Варіант передачі даних у функціоналі Insomnia

Insomnia підтримує широкий набір можливостей для звернень до різних API: набір методів HTTP/s, звернення до gRPC API, а також кастомні методи зі своїми параметрами. Варіантів форматів повідомлень, що передаються теж багато: JSON, XML, YAML, EDN, бінарні файли, GraphQL Query.

Важлива особливість Insomnia — можливість створювати змінні оточення, які підтягуватимуться або автоматично генеруватимуться залежно від налаштування. Значення зі змінних можна використовувати в тестах, щоб автоматизувати їх та спростити собі роботу.

На сторінці проекту у вкладці “Test” додаються перевірки до запитів, що відправляються, що порівнюють відповідь з очікуваним результатом. Запит із доданими перевірками називається Тестом. Декілька запитів можна об'єднати в Тест-сьют.

Insomnia підтримує можливість запускати тести списками за допомогою раннера. При цьому відображаються запити, що надсилаються, і результати перевірок.

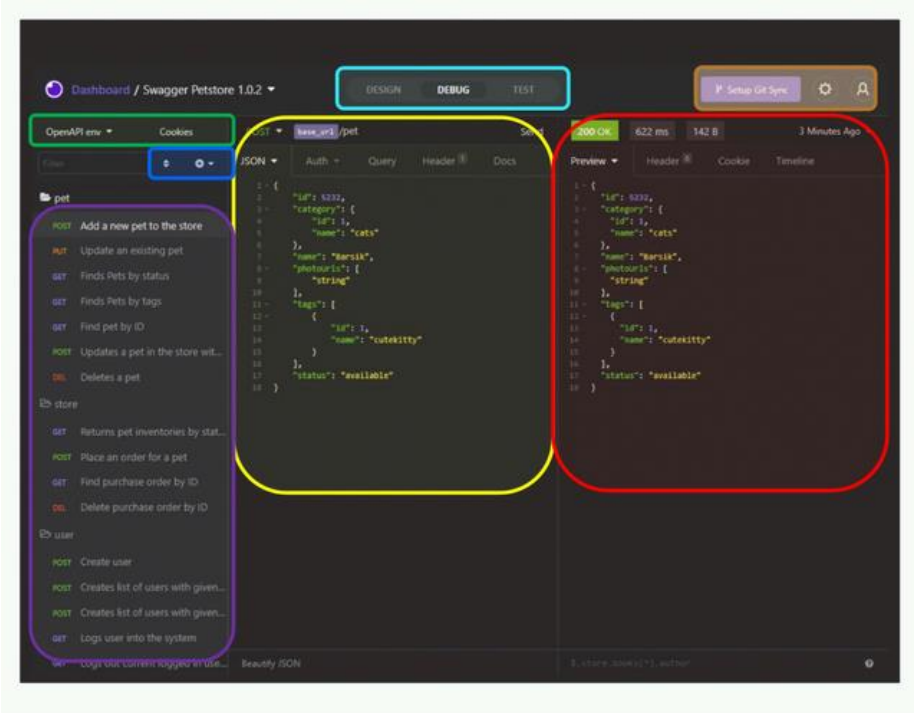

Рисунок 4.19 – Приклад запуску тестів списками за допомогою раннера Insomnia

Переваги Insomnia:

1. Легкий у освоєнні інтерфейс щодо інших подібних API-клієнтів.

2. Кросплатформність.

3. Можливість завантаження проектів із Swagger. Можемо по URL імпортувати проект, і у нас з'являться ті самі списки з описами, як це було в Swagger.

4. Можливість налаштування оточення.

5. Можливість створення динамічних змінних.

6. Можливість створювати та проганяти тест-кейси.

7. Можливість підтримки версійності кейсів із підключенням їх до Git.

8. Можливість тестування різних API, таких як REST, SOAP, gRPC, GraphQL та інші.

Недоліки Insomnia:

1. Створення тест-кейсів із кількох запитів утруднено. Якщо тест-кейс складається з кількох послідовних кроків/запитів, то він позначатиметься як тест-сьют, відповідно, побудувати ієрархію кейсів складно через один рівень вкладеності (папка (тест-сьют) – запит).

2. Неможливий паралельний запуск запитів та кейсів.

3. Налаштування змінних оточення складне щодо інших API-клієнтів.

4. Невеликий вибір вбудованих скриптів-рандомайзерів, порівняно з іншими інструментами.

5. Не вбудований список перевірок до відповідей.

Використовуйте Insomnia, коли потрібно тестувати різні види API, використовувати оточення, змінні, скрипти, і при цьому функції Postman вам надмірні. Або якщо вам потрібний зручний API-клієнт для тестування gRPC.

Postman дуже популярний API-клієнт. Хоча його інтерфейс і лякає насиченістю, розібратися в ньому досить легко. На домашній сторінці пропонується вибрати робочий простір. Це своєрідна «папка», до якої додаються колекції запитів, оточення, змінні, мок-заглушки та інше. Такий поділ дуже зручний, якщо у вас є кілька проектів або сервісів, що тестуються.

У кожному робочому просторі є:

• список колекцій у вікні “Collections”;

• документація API, сюди можна імпортувати Swagger;

• вікно, в якому редагуються оточення та змінні;

• функціонал створення мок-заглушок.

«Монітор» – функція, яка дозволяє за таймером запускати колекції запитів. Причому це можна робити хмарно, без необхідності тримати свій комп'ютер увімкненим.

Вікно запиту складається з поля для вказівки HTTP-методу та URL-адреси запиту, налаштування параметрів (заголовки і тіло), Pre-request Script - скриптів, які будуть виконуватися до відправлення повідомлення, Tests - коду, який буде виконуватися після виконання запиту.

Помаранчевим кольором виділяються змінні Postman. Вони бувають автоматично генеруються (ім'я починається з $, наприклад $randomInt — генерує рандомне число) і створені користувачем (куди можна записати все що завгодно).

В даному випадку створюється змінна 'CreateTime' в яку записується час, що генерується за допомогою JS Pre-request Script.

У вкладці Test ми бачимо код, який звіряє очікуваний та фактичний результат. Перевірки в Postman починаються з конструкції pm.test. Першим тестом ми перевіряємо статус-код. Якщо він дорівнює 200, то у вкладці Test results з'явиться повідомлення “Status code is 200”.

Також до вкладки Test можна додати код, який виконує будь-які дії - наприклад, що дістає з відповіді значення, яке зберігається в довільному полі, і передає його в змінну оточення.

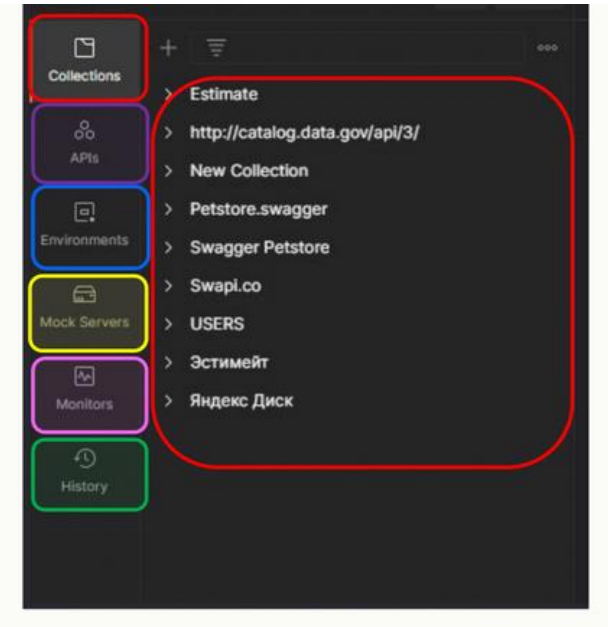

Рисунок 4.20 – Тестовий функціонал Postman

Переваги Postman:

1. Щодо простий інтерфейс.
2. Можливість підключення до Swagger.
3. Можливість створювати робочі простори та колекції.

4. Підтримка мок-серверів та прогонів колекцій (ручні та періодичні).

5. Окремий компонент-перехоплювач запитів.

6. Оточення, змінні та можливість їх налаштовувати, створювати, редагувати.

7. Можливість тестування різних видів API, як REST, SOAP тощо.

8. Підтримка інтеграції з Git.

9. Дуже багато вбудованих скриптів-рандомайзерів.

10. Можливість написання кастомних скриптів перевірок за допомогою JavaScript.

11. Дуже багато вбудованих варіантів перевірок у сніпетах.

12. Можливість написання кастомних перевірок з допомогою JS.

13. Можливість запускати колекції з командного рядка за допомогою інструмента-компаньйона Newman.

14. Можливість інтеграції Newman із системами безперервного складання. Наприклад, ми можемо створити колекцію в Postman, вона передається Newman, інтегрується з системами безперервного складання, і Newman запускає цю колекцію при оновленні версії стенда, програми або інших подібних ситуаціях.

Недоліки Postman:

1. Колекції та оточення до них експортуються окремо.

2. Неможливий паралельний запуск кейсів у Postman. Наприклад, ви тестуєте асинхронний сервіс і є певний етап очікування між створенням завдання та отриманням на нього результату. У цьому випадку краще, щоб ваш інструмент тестування дозволяв запускати перевірки паралельно і чекати на результат у кожному випадку окремо. Інакше ви чекатимете їх дуже довго, особливо коли багато тест-кейсів. Прогін такої колекції буде розтягнутим.

3. Раннер, який послідовно запускає запити, бачить саме список запитів, а чи не кейси. Це несе у собі ризики, коли кейс складається з кількох запитів. Наприклад, папка є тест-кейс. У вас десять папок, у яких десять тест-кейсів, і в кожному по два-три кроки. Коли ви запускаєте раннер, вам видно 30

послідовних запитів. Тому в деяких запитах у різних тест-кейсах вам потрібно писати скрипти, які у разі провалу автоматично очистять деякі змінні від своїх значень. Тоді наступний кейс пройде без помилок і не залишиться значень, які сприймаються наступним кейсом. Це впливає на незалежність тест-кейсів один від одного.

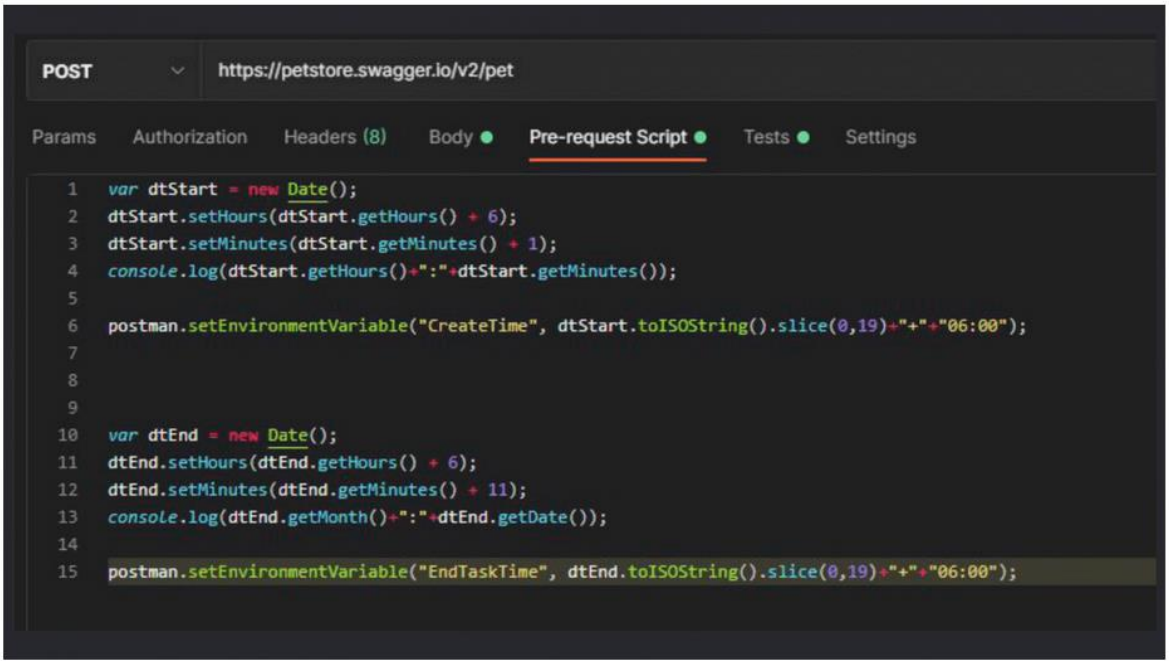

Рисунок 4.21 – Тест-кейс Postman

Postman слід використовувати, коли вам потрібен зручний та потужний API-клієнт для тестування різних видів API з можливістю використовувати просунуті змінні та скрипти у запитах. При цьому не важливим є паралельний запуск кейсів у проекті і ви розумієте особливості побудови кейсів з декількох запитів.

Тестовий функціонал SOAPUI був створений для тестування SOAP сервісів, але згодом став застосовуватись і для REST API. Інструмент дозволяє створювати та імпортувати проекти, у ньому є меню роботи з тест-кейсами, список усіх проектів та меню дій над поточним.

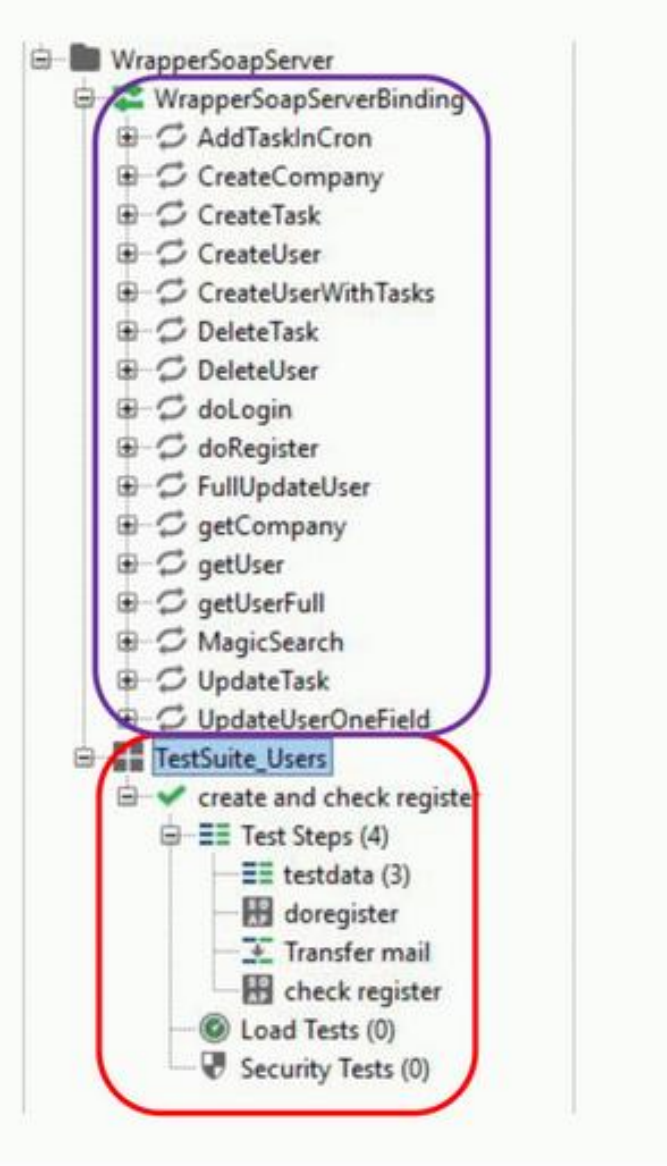

Рисунок 4.22 – Тестування SOAP сервісів

При відкритті проекту стає доступним список усіх запитів, що існують у вашому сервісі, та список тест-кейсів. Це одна з відмінних рис SOAPUI – запити в бібліотеці та тести існують окремо. Тобто ми описали 15 запитів, а потім можемо у будь-яких комбінаціях використовувати їх у тестах, ставити у різному порядку залежно від потреби.

Тест-сьюти складаються з тест-кейсів, а тест-кейс — з кроків. Кроки можуть бути як запитами, і службовими діями.

Запити налаштовуються в окремому вікні, де можна додати і зберегти дані, надіслати запит, подивитися відповідь на нього, а також прикріпити перевірки очікуваного результату (в даному випадку перевірки Contains).

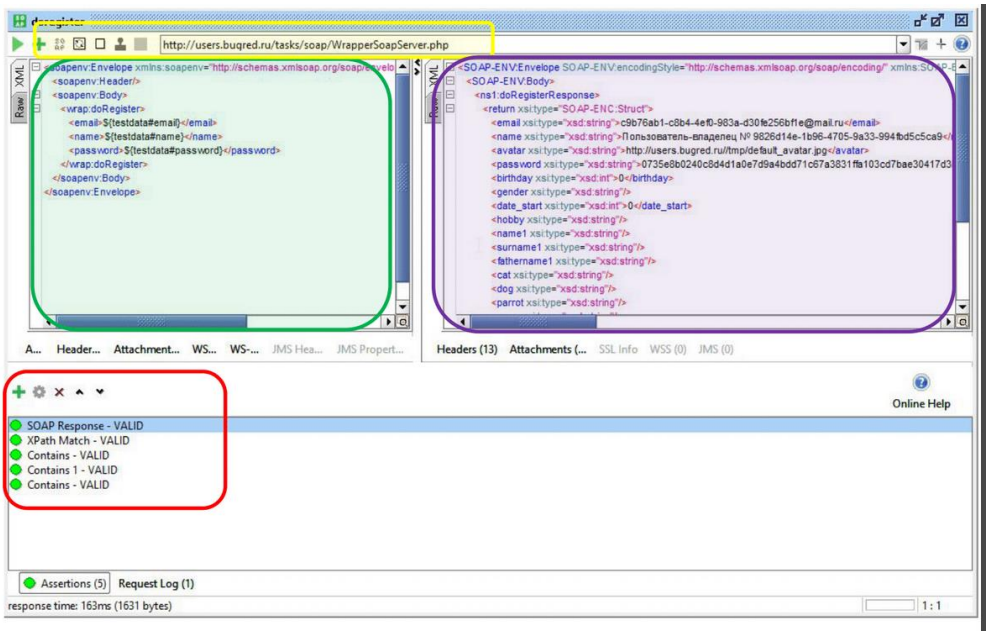

Рисунок 4.22 – Запити на налаштування SOAP сервісів

До службових дій відносяться, наприклад, таблиці з тестовими даними (testdata), які за допомогою змінних підтягуватимуться у запити. Таблиці мають просту структуру “name-value”, причому value може бути як статичним, а й генеруватися скриптом.

Ще одним видом службових дій є кроки трансферу (Transfer mail), що виконують передачу даних між запитами або між запитом і таблицею з даними. Структура передачі також проста “source-target”. Єдина особливість тут: грамотно вказати з допомогою мови парсингу поля обміну даними.

Інтуїтивно зрозумілий функціонал запуску тест-кейсів складається з: кнопки «старт/стоп» та перемикача послідовного або паралельного запуску перевірок. При натисканні на кнопку “Start” відображається список запущених кейсів, а також результат їхнього проходження. Відкривши провалений кейс можна побачити, на якому кроці він «упав» і чому (у даному прикладі — тому

що немає значення, яке передається в “mail”). При цьому наступний крок не запускався і це одна з особливостей SOAPUI. Він бачить саме тест-кейси, а не перелік кроків. Якщо якийсь крок у тест-кейсі провалився, інші кроки він не запускає і переходить далі. Наступні кейси працюють тільки зі змінними, скриптами та тестовими даними всередині них самих, не бачачи змінні з інших кейсів.

Налаштування послідовного або паралельного запуску кейсів з коробки - це також відмінна особливість SOAPUI. У багатьох випадках це заощаджує ваш час.

Переваги SOAPUI:

1. Можливість створення проектів за допомогою Swagger (для REST API) та файлів WSDL (для SOAP API).

2. Можливість тестувати REST та SOAP.

3. Легкий експорт та імпорт проектів одним файлом. Тобто ви експортуєте цей файл один раз, передаєте колезі, він встановлює його та імпортує собі. Немає необхідності переносити окремо змінні, оточення та самі запити, як у Postman.

4. Можливість використовувати оточення та змінні.

5. Можливість написання кастомних скриптів для перевірки очікуваного та фактичного результату.

6. Вбудовані варіанти перевірок.

7. Паралельний запуск кейсів, що дуже важливо.

8. Раннер бачить саме список кейсів, а не список запитів та список кроків у них.

9. Можна зробити тести навантаження.

10. Можна написати мок-заглушки.

11. Є можливість використовувати функціонал проксі.

Недоліки SOAPUI:

1. Складний інтерфейс щодо попередніх API-клієнтів.

2. Не підтримує gRPC та GraphQL у безкоштовній версії.

3. Не підтримує інтеграцію з Git у безкоштовній версії.

4. Не підтримує інтеграцію із системами безперервного складання у безкоштовній версії.

SOAPUI підійде, якщо у вас є багато тест-кейсів з кількох запитів, а також потрібний паралельний запуск цих кейсів. Коли у вас є кейси з вилкою результатів. Якщо на проекті немає потреби тестувати API крім REST та SOAP. Коли вам не обов'язково інтегрувати тести із системами безперервного складання.

Бібліотеки це не самостійні інструменти, вони написані для використання в мовах програмування. Так як на розгляд великої кількості існуючих бібліотек тестування API потрібно багато часу. Розглянемо їх типовий функціонал з прикладу популярної Rest Assured.

За допомогою public void оголошується тест "whenCreatePerson_thenStatus201" в given() задаються попередні умови запиту, блок when() відправляє запит до ендпойнту, а then() відбувається звірка очікуваного і фактичного результату. У цьому прикладі перевіряється, що при успішному створенні користувача надходить статус-код 201 (Created).

Наступний тест трохи складніший: тут оновлюється попередньо створений користувач. При цьому перевіряються два очікування: статус-код 200 і в тілі поле “name” матиме значення “Michail”.

Переваги Rest Assured:

1. Повна інтеграція із системами CI/CD та можливість автоматично запускати перевірки при появі нової версії програми на стенді.

2. Можливість підключити інструмент Allure Reporting для автоматизованого формування звітів про тестування.

3. Можливість реалізувати практично будь-які перевірки, які можете написати самі, та практично будь-які передзапитові скрипти.

4. Швидкість роботи тестів, написаних за допомогою бібліотек, зокрема Rest Assured, швидше, ніж у API-клієнтів.

Мінуси Rest Assured:

1. Потрібно вміти писати код, причому добре. Rest Assured використовує мову Java. Перший тест із підручника написати не складно, але створити тести, які дійсно принесуть користь вашому проекту — це не так просто.

2. Офіційні туторіали для фреймворків і бібліотек підходять лише дрібних проектів чи написання тестів а-ля “Hello, world!”. Найкращі патерни написання кейсів доведеться шукати самостійно за форумами та іншими джерелами.

Фреймворки варто вибирати, якщо ви вже вмієте добре писати код або маєте намір прокачатися в цій справі. Коли хочете глибокої автоматизації та інтеграції кейсів з CI/CD. Якщо в кейсах багато логік, котрим потрібні складні скрипти. Якщо для вашого проекту важливим є збільшення швидкості повторного тестування, насамперед — регресійного.

Порівняння реального використання інструментів

На проекті автора статті послідовно запроваджувалося використання інструментів тестування API.

Сервіс назвемо умовно «Квартет», оскільки він включає 4 мікросервіси. Тип роботи – асинхронний. На цьому проекті відбулася послідовна зміна інструментів Postman - SOAPUI - Rest assured. Swagger присутній, але для тестування він не використовувався, оскільки проходження кейсів з його допомогою, у нашому сервісі, не дуже скоріше використання графічного інтерфейсу. Проходження кейсів за допомогою GUI візьмемо за початкову точку порівняння.

Написання кожного тест-кейсу зайняло 6 хвилин. 200 кейсів - це 1200 хвилин або 20 годин. Ручний прогін кожного end to end кейсу в середньому займає 4 хвилини з урахуванням швидкості роботи фронту, бека та QA спеціаліста. 200 кейсів це 800 хвилин або 13,5 години.

Тобто, якщо проводити регрес вручну, на це щоразу йтиме 13-14 годин. І це без урахування часу на баг-репорти, без поправки на втому фахівця тощо. Звичайно ж, такий час неприйнятний в умовах обмеженої команди QA.

Першим інструментом став Postman. Швидкість занурення в інструмент з моменту встановлення до написання першого кейсу, який можна використовувати — 2 години. Час створення першого повністю автоматизованого кейсу, який чекає побудови файлу, перевіряє очікуваний результат тощо. - 8:00. Тобто, освоєння інструменту зайняло 10 годин. Після цього написання та налагодження кейсів почали займати по 10 хвилин у середньому, тобто на 200 кейсів пішло 34 години.

Автоматизоване проходження такого кейсу в Postman займає 50 секунд, тобто 200 кейсів проходять за 2 години 47 хвилин. Начебто все добре, але через особливості роботи системи, 50 секунд йде тільки на успішний кейс, а на неуспішний — до 30 хвилин. Якщо провалюватимуться всі 200, то таке проходження займе до 100 годин. Звичайно, такого ніколи не траплялося, але теоретична можливість такого сценарію змушувала шукати інструмент із паралельним запуском кейсів. Тим більше, що сам сервіс міг опрацьовувати результати паралельно.

Ним став SOAPUI. Час занурення в інструмент із встановлення до написання першого корисного кейсу становив 3 години. Пілотний автоматизований кейс було написано за 10 годин. 200 кейсів у SOAP UI були написані за 44 години. Так, робити тест-кейси в цьому інструменті трохи довше через своєрідний інтерфейс. Але паралельний запуск повністю окупає цей недолік. Кейси запускалися порціями по 100 і з урахуванням особливостей роботи сервісу це давало 18 хвилин на порцію у разі успішного проходження та 46 хвилин при неуспішному. Тобто 200 успішних кейсів проходили за 36-37 хвилин, 200 неуспішних – 1 годину 32 хвилини. Це був успіх, який дозволив не стежити за прогоном, щоб вчасно зупинити його за кількох неуспішних кейсів поспіль. Але SOAPUI не вистачає звітності про тестування, а головне – немає інтеграції з CI/CD, а цього дуже хотілося.

Тому перейшли на Rest Assured у зв'язці з JUnit 5. Занурення зайняло 10 годин, ще 11 на написання пілотного кейсу. 200 кейсів було створено за 42 години. JUnit вміє запускати кейси паралельно, тому запускали тими ж

порціями по 100, що дало проходження 200 успішних кейсів за 35 хвилин в середньому, неуспішних — за 1 годину 29 хвилин. На вигляд прискорення незначне, але не варто забувати, що з'явилася звітність про кожен прогін і головне — кейси автоматично запускалися при спрацюванні певного тригера. Регрес став повністю автоматизованим.

ВИСНОВКИ

Дивлячись на результати дослідження методу тестування API через інтерфейс користувача ми можемо зробити висновок, що цей метод вирішує проблему до тестування API, коли немає до нього прямого доступу.

Якщо робити співвідношення виконаної розробки з вітчизняними та світовими аналогами, ми можемо порівняти наш метод тестування на основі зібраних результатів тестування зі звичайним API тестуванням яке використовується всюди. Для порівняння розглянемо недоліки та переваги методу тестування API через інтерфейс користувача зі звичайним API тестуванням.

Недоліком методу тестування API через інтерфейс користувача є загальний час виконання тестів, який є 5 хвилин 9 секунд замість 50 секунд як при тестування через API. Також його недоліком є час написання авто-тестів, бо час необхідний на написання тестів інтерфейсу користувача є більше ніж час написання звичайних API тестів. Останній недолік який буде розглянуто – це підтримка тестів. API тести легше підтримувати, вони легше адаптуються до змін.

Перевагою методу тестування API через інтерфейс користувача є розширення тестового покриття, бо тестується більше компонентів системи, та тести більш поглядають на сценарії реального використання додатку користувачем. Також, його перевагою є рішення проблемі з отриманням доступу, кола за якихось причин в нас не має можливості отримати прямий доступ до API, а тестування вже потрібно починати. Наступною перевагою є більша зрозумілість того, що трапиться у тестах для замовників продукту, бо в тестах інтерфейсу користувача ми можем бачити сценарії реального використання продукту, яке викликає API.

Дані переваги вказують на те, що іноді метод тестування API через інтерфейс користувача може зберегти час та гроші замовників, коли є великі проблеми з отриманням доступу до API.

Щодо подальшої роботи в даному напрямку – подана стаття про цей метод до наукового фахового збірника наукових праць «Системи управління, навігації та зв’язку». Бо не всі тестувальники знають про такий обхідний путь отримання доступу до API.

Отже, на основі результатів дипломної роботи ми можемо зробити висновок, якщо в нас збігаються такі фактори як:

- немає прямого доступу до API;
- немає можливості запросити доступ до API;
- або немає часу на очікування доступу до API;
- у системі є інтерфейс користувача;
- інтерфейс користувача системи є простим;
- є можливість виконувати тестування API через інтерфейс користувача.

Ми можемо зробити висновок, що метод тестування API через інтерфейс користувача є гарним рішенням.

СПИСОК ІНФОРМАЦІЙНИХ ДЖЕРЕЛ

1. https://qagroup.com.ua/publications/korotko-pro-ari-ta-jogo-testuvannia/
2. [Електронний ресурс] - https://www.ibm.com/cloud/learn/api#toc-types-of-a-DMAvqUq6

3. [Електронний ресурс] - https://www.functionize.com/blog/myth-or-fact-api-testing-is-easier-than-ui-testing/

4. [Електронний ресурс] - https://cutt.ly/fTXpoMu.

5. [Електронний ресурс] - https://economictimes.indiatimes.com/definition/testng.

6. [Електронний ресурс] - https://www.javatpoint.com/testng-tutorial

7. [Електронний ресурс] - https://www.geeksforgeeks.org/introduction-to-java.

8. [Електронний ресурс] - https://www.guru99.com/what-everybody-ought-to-know-about-test-planing.html.

9. [Електронний ресурс] - https://www.browserstack.com/guide/selenium-webdriver-tutorial.

10. [Електронний ресурс] - https://www.atlassian.com/continuous-delivery/software-testing/exploratory-testing.

11. [Електронний ресурс] What Is Automation Testing (Ultimate Guide To Start Test Automation) //Software Testing Help: сайт – Режим доступу: https://www.softwaretestinghelp.com/automation-testing-tutorial-1.

12. Postman [Електронний ресурс] //Офиційний сайт Postman – Режим доступу: https://www.getpostman.com.

13. Lisa Crispin, Janet Gregory. Agile testing. Training course for the whole team - MYTH, 2019, 530 pp.

14. Elfrid Dustin, Jeff Raschka, John Paul. Automated Software Testing - Laurie 2003, 592 pp.

ДОДАТКИ

1 ANALYTICAL REVIEW

1.1 API, its significance for modern information and communication technologies

API is a set of software code that provides data transfer between one software product and another [1]. It also contains data sharing terms.

An application programming interface allows companies to expose the data and functionality of their applications to external third-party developers, business partners, and internal departments within their companies. It allows services and products to interact with each other and use each other's data and functionality through a documented interface. Developers don't need to know how the API is implemented; they simply use an interface to communicate with other products and services. Over the last decade, the use of APIs has grown to such an extent that many of the most popular web applications today would not be possible without APIs.

1.2 Principle of API operation

The API works according to this principle. The client application initiates an API call to obtain information, also known as a request. This request is handled from the application to the web server via a Uniform Resource Identifier (URI) API and includes the request verb, headers, and sometimes the request body. After receiving a valid request, the API makes a call to an external application or web server. The server sends an API response with the requested information. The API passes the data to the original requesting application. Although the data transfer will vary depending on the web service used, this process of requests and responses occurs through an API.

While a user interface is designed to be used by humans, an API is designed to be used by a computer or program. APIs provide security by design because their intermediary position facilitates the abstraction of functionality between two

systems—an API endpoint separates the consumer application from the infrastructure that provides the service. API calls typically include authorization credentials to reduce the risk of server attacks, and an API gateway can restrict access to minimize security threats. In addition, HTTP headers, cookies, or query string parameters provide additional layers of data security during exchange. For example, consider the API offered by a payment processing service. Customers can enter their card details in the app interface for the e-commerce store. The payment processor does not require access to the user's bank account; The API creates a unique token for this transaction and includes it in the API call to the server. This provides a higher level of security against potential hacking threats.

1.3 Purpose of the API

One of the most important purposes of the API is to improve collaborative work. The average enterprise uses nearly 1,200 cloud applications, many of which are disabled. APIs provide integration so that these platforms and applications can interact seamlessly with each other. With this integration, companies can automate workflows and improve collaboration in the workplace. Without APIs, many businesses would lack connectivity and suffer from information silos that compromise performance and productivity.

The next purpose of the API is simpler innovation. APIs offer flexibility, allowing companies to connect with new business partners, offer new services in an existing market, and ultimately gain access to new markets that can generate huge returns and drive digital transformation. For example, Stripe started as an API with just seven lines of code. Since then, the company has partnered with many of the world's largest businesses, diversifying into credit and corporate cards, and was recently valued at $36 billion.

Another purpose of API is data monetization. Many companies choose to offer APIs for free, at least initially, so they can build an audience of developers around their brand and build relationships with potential business partners. However, if an

API provides access to valuable digital assets, you can monetize it by selling access (called the API economy). When AccuWeather launched its self-service developer portal to sell a wide range of API packages, it took just 10 months to attract 24,000 developers, sell 11,000 API keys, and build a thriving community in the process.

The final purpose of the API is additional security. As mentioned above, APIs create an additional layer of protection between your data and the server. Developers can further strengthen API security by using tokens, signatures, and Transport Layer Security (TLS) encryption; by implementing API gateways to manage and authenticate traffic, and by practicing effective API management.

1.4 API examples

Because APIs allow companies to open access to their resources while maintaining security and control, they have become a valuable aspect of modern business. Here are some popular examples of APIs you may encounter.

The first is universal logins. A popular example of an API is a feature that allows people to log into websites using their Facebook, Twitter, or Google login credentials. This handy feature allows any website to use an API from one of the most popular services to quickly authenticate a user, saving you the time of creating a new profile for each website service.

Next - Third Party Payment Processing. For example, the ubiquitous "Pay With" feature we see on e-commerce websites works through an API. It allows people to pay for products online without revealing any sensitive data or giving access to unauthorized persons.

Travel Booking Comparison: Travel booking sites aggregate thousands of flights, showing you the cheapest options for every date and destination. This service is made possible by APIs that give app users access to the latest hotel and airline availability information. By autonomously exchanging data and queries, APIs

dramatically reduce the time and effort required to check flight availability or accommodation.

One of the most common examples of a good API is Google Maps. In addition to the core APIs that display static or interactive maps, the app uses other APIs and features to provide users with directions or points of interest. Using geolocation and several layers of data, we can communicate with the Maps API to plot travel routes or track objects in transit, such as a vehicle.

Twitter: Each tweet contains descriptive basic attributes, including author, unique identifier, message, timestamp when it was posted, and geolocation metadata. Twitter makes public tweets and replies available to developers and allows developers to publish tweets through the company's API.

1.5 Types of APIs

Today, most APIs are Web APIs that expose application data and functionality over the Internet. There are four main types of web APIs.

Open APIs are open source application programming interfaces that can be accessed using the HTTP protocol. Also known as public APIs, they define API endpoints and request and response formats.

Partner APIs are application programming interfaces available to or from strategic business partners. Typically, developers can access these APIs in self-service mode through the public API Developer Portal. However, they will need to complete the onboarding process and obtain credentials to access partner APIs.

Internal APIs are application programming interfaces that remain hidden from external users. These private APIs are not accessible to users outside the company, but instead are intended to improve productivity and communication between different internal development teams.

Composite APIs combine multiple data or service APIs. These services allow developers to access multiple endpoints in a single call. Composite APIs are useful in

microservices architectures where a single task may require information from multiple sources.

1.6 Types of API protocols

As the use of Web APIs has expanded, certain protocols have been developed to provide users with a set of defined rules that define accepted data types and commands. In effect, these APIs facilitate the standardized exchange of information.

SOAP (Simple Object Access Protocol) is an API protocol built from XML that allows users to send and receive data via SMTP and HTTP. The SOAP API makes it easier to exchange information between applications or software components running in different environments or written in different languages.

XML-RPC is a protocol that relies on a specific XML format for data transfer, while SOAP uses its own XML format. XML-RPC is older than SOAP, but much simpler and relatively lightweight because it uses minimal bandwidth.

JSON-RPC is a protocol similar to XML-RPC in that they are both remote procedure calls (RPCs), but this protocol uses JSON instead of XML to transfer data. Both protocols are simple. Although calls can take multiple parameters, they expect only one result.

REST (Representational State Transfer) is a set of principles for web API architecture, which means that there are no official standards (unlike those that have a protocol). To be a REST API (also known as a RESTful API), an interface must conform to certain architectural constraints. It is possible to build RESTful APIs using SOAP protocols, but the two standards are generally seen as competing specifications. It is the analysis of the method of automating API testing through the user interface for a web application with REST architecture that is discussed in this diploma project [1].