**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ
ТЕХНОЛОГІЙ НАВЧАЛЬНО-НАУКОВИЙ
ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ**

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Віртуалізована платформа для тестування програмного забезпечення в різних середовищах»

на здобуття освітнього ступеня бакалавра
зі спеціальності 123 Комп'ютерна інженерія
освітньо-професійної програми «Комп'ютерна інженерія»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Максим ОСТАПЕНКО
(підпис)

Виконав: здобувач вищої освіти групи КІД-41
Максим ОСТАПЕНКО

Керівник: А. М. ДАВИДЕНКО
д.т.н., проф.

Рецензент: Володимир САДОВЕНКО
к.ф.-м.н., доцент *Ім'я, ПРІЗВИЩЕ*

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерна інженерія

Ступінь вищої освіти Бакалавра

Спеціальність 123 Комп'ютерна інженерія

Освітньо-професійна програма «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерної інженерії
_____Наталія ЛАЩЕВСЬКА
«_____» _____2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Остапенко Максим Станіславович

1. Тема кваліфікаційної роботи: «Віртуалізована платформа для тестування програмного забезпечення в різних середовищах»

керівник кваліфікаційної роботи А.М. ДАВИДЕНКО, д.т.н., проф,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» травня 2025 р. № 320.
2. Строк подання кваліфікаційної роботи «17» травня 2025 р.
3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, метод удосконалення параметрів, вимоги до якості друку.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Огляд літератури.
2. Аналіз технологій віртуалізації у тестуванні програмного забезпечення.
3. Фактори, що впливають на ефективність тестування у віртуалізованих середовищах.
4. Розробка методу автоматизації тестування ПЗ у віртуальному середовищі.

1. Методи оптимізації тестування у віртуалізованих середовищах.
2. Експериментальні результати тестування.

2. Перелік ілюстративного матеріалу: *презентація*
    1. Аналіз технологій віртуалізації у тестуванні програмного забезпечення.
    2. Проблематика автоматизованого тестування ПЗ та роль віртуалізованих середовищ.
    3. Методи оптимізації тестування у віртуальних середовищах.
    4. Архітектура розробленої тестової платформи.
    5. Процес розгортання віртуалізованого тестового середовища
    6. Автоматизація тестування: використання Appium, Cucumber та інших інструментів.
    7. Аналіз продуктивності та безпеки тестової платформи
    8. Експериментальні результати тестування.

3. Дата видачі завдання «16» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	16.02-23.02.2025	
2	Вивчення матеріалів для аналізу технологій віртуалізації та автоматизованого тестування ПЗ	23.02-02.03.2025	
3	Дослідження платформ та інструментів для віртуалізованого тестування ПЗ	02.03-09.03.2025	
4	Розробка методу автоматизації тестування у віртуальному середовищі	09.03-19.03.2025	
5	Постановка експерименту: тестування автоматизованої платформи	19.03-29.03.2025	
6	Оформлення роботи: вступ, висновки, реферат	29.03-30.03.2025	
7	Розробка демонстраційних матеріалів	30.03-31.03.2025	
8	Попередній захист роботи	01.04.2025	

Здобувач вищої освіти _____ Максим Остапенко
(підпис)

Керівник
кваліфікаційної роботи _____
А.М. ДАВИДЕНКО
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 72с., 33 рис., 3 табл., 41 джерело.

Об'єкт дослідження – віртуалізована платформа для тестування програмного забезпечення у різних середовищах.

Предмет дослідження – методи автоматизованого тестування у віртуалізованих середовищах, технології віртуалізації та інструменти для забезпечення якості програмного забезпечення.

Мета роботи – розробка та реалізація платформи для автоматизованого тестування програмного забезпечення у віртуалізованих середовищах, що дозволяє створювати ефективні тестові сценарії, перевіряти функціональність API та оцінювати продуктивність додатків.

Методи дослідження:

1. Аналіз літературних джерел та нормативних документів – дослідження сучасних методів автоматизованого тестування, огляд технологій віртуалізації та їх застосування у тестуванні програмного забезпечення.
2. Методи програмного тестування – розгляд різних підходів до автоматизації тестування, аналіз ефективності інструментів для віртуалізованого тестування.
3. Методи моделювання – створення архітектури тестової платформи, розробка сценаріїв тестування та їх впровадження у віртуалізованих середовищах.
4. Порівняльний аналіз – оцінка продуктивності різних інструментів тестування, таких як Appium, Cucumber, Mitmproxy, Mock-server, AVD, Genymotion та Bluestacks.

Галузь використання – автоматизоване тестування програмного забезпечення, розробка мобільних та веб-застосунків, забезпечення якості ПЗ, DevOps, кібербезпека.

КЛЮЧОВІ СЛОВА: ВІРТУАЛІЗАЦІЯ, ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, APP AUTOMATION, CUCUMBER, APPIUM, MITMPROXY, MOCK-SERVER, AVD, GENYMOTION, BLUESTACKS, API ТЕСТУВАННЯ, ВІРТУАЛЬНІ СЕРЕДОВИЩА

ABSTRACT

The textual part of the Bakhaleor's qualification thesis consists of 72 pages, 33 figures, 3 table, and 41 references.

Object of the study – a virtualized platform for software testing in various environments.

Subject of the study – methods of automated testing in virtualized environments, virtualization technologies, and tools for ensuring software quality.

Purpose of the thesis – development and implementation of a platform for automated software testing in virtualized environments, enabling the creation of efficient test scenarios, verification of API functionality, and assessment of application performance.

Research methods:

1. Analysis of literary sources and regulatory documents – exploration of modern methods of automated testing, review of virtualization technologies and their application in software testing.
2. Software testing methods – consideration of various approaches to test automation, analysis of the effectiveness of tools for virtualized testing.
3. Modeling methods – design of the architecture of the testing platform, development of test scenarios and their implementation in virtualized environments.
4. Comparative analysis – performance evaluation of various testing tools such as Appium, Cucumber, Mitmproxy, Mock-server, AVD, Genymotion, and Bluestacks.

Field of application – automated software testing, mobile and web application development, software quality assurance, DevOps, cybersecurity.

KEYWORDS: VIRTUALIZATION, SOFTWARE TESTING, APP AUTOMATION, CUCUMBER, APPIUM, MITMPROXY, MOCK-SERVER, AVD, GENYMOTION, BLUESTACKS, API TESTING, VIRTUAL ENVIRONMENTS

ЗМІСТ

ВСТУП ... 8
1 ВІРТУАЛІЗОВАНІ СЕРЕДОВИЩА ТА АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ... 10
1.1 Основи автоматизованого тестування програмного забезпечення ... 10
1.2 Підходи до використання віртуалізованих середовищ у тестуванні ПЗ ... 14
1.3 Класифікація платформ для віртуалізації та їх застосування ... 17
1.4 Аналіз існуючих технологій та інструментів для тестування у віртуальних середовищах ... 21
Висновок до розділу 1 ... 25
2 ТЕХНОЛОГІЇ ТА ІНСТРУМЕНТИ ВІРТУАЛІЗОВАНОГО ТЕСТУВАННЯ ... 26
2.1 Використання емуляторів та контейнеризації для тестування ... 26
2.2 Інструменти моніторингу та перехоплення мережевого трафіку ... 30
2.3 Тестування безпеки та атаки типу "Людина посередині" (MitM) ... 32
2.4 Емуляція мобільних пристроїв та тестування у віртуальному середовищі ... 35
Висновок до розділу 2 ... 36
3 РЕАЛІЗАЦІЯ ТЕСТОВОЇ ПЛАТФОРМИ У ВІРТУАЛЬНОМУ СЕРЕДОВИЩІ ... 38
3.1 Проектування платформи для автоматизації тестування ПЗ ... 38
3.2 Використання Appium та Cucumber у тестуванні ... 41
3.3 Інтеграція Mitmproxy та Mock-server для аналізу API ... 46
3.4 Опис взаємодії із API та тестування запитів у віртуальному середовищі ... 49
Висновок до розділу 3 ... 57
4 РЕАЛІЗАЦІЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ ТЕСТУВАННЯ У ВІРТУАЛІЗОВАНОМУ СЕРЕДОВИЩІ ... 58
4.1 Створення тестових сценаріїв мовою Gherkin та їх виконання ... 58
4.2 Створення тестових сценаріїв та їх виконання у віртуальному середовищі ... 61
4.3 Аналіз продуктивності тестової платформи та перехоплення трафіку ... 64
4.4 Оптимізація процесу автоматизованого тестування та перспективи розвитку 68
ВИСНОВКИ ... 72
ПЕРЕЛІК ПОСИЛАНЬ ... 73
ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ ... 77

ВСТУП

Сучасний розвиток інформаційних технологій вимагає ефективних інструментів для тестування програмного забезпечення в різних середовищах, оскільки програмні продукти повинні забезпечувати стабільну роботу на різних операційних системах, апаратних платформах та конфігураціях. Традиційні методи тестування часто потребують значних ресурсів, а їх проведення на фізичних пристроях може бути дорогим та трудомістким.

Віртуалізація надає можливість створювати ізольовані середовища для тестування, що дозволяє швидко адаптуватися до змін, масштабувати тестові процеси та знижувати витрати на обладнання. Використання віртуалізованих платформ дає змогу автоматизувати процес тестування, покращити виявлення помилок та підвищити загальну якість програмного забезпечення.

Актуальність даного дослідження полягає у розробці та впровадженні віртуалізованої платформи, яка дозволить ефективно тестувати програмні продукти в різних середовищах, мінімізуючи час та ресурси, необхідні для цього процесу. Це сприятиме покращенню процесу розробки, підвищенню надійності програмних рішень та їхньої сумісності з різними технологічними платформами.

Метою даного дослідження є розробка та реалізація платформи для автоматизованого тестування програмного забезпечення у віртуалізованих середовищах, що дозволяє створювати ефективні тестові сценарії, перевіряти функціональність API та оцінювати продуктивність додатків.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

- аналіз основ автоматизованого тестування програмного забезпечення та визначити його ключові принципи;
- розглянути методи тестування безпеки, зокрема атаки типу

"Людина посередині" (MitM);

- Розробити архітектуру тестової платформи для автоматизації тестування програмного забезпечення;
- провести аналіз продуктивності тестової платформи та перехоплення трафіку.

Об’єкт дослідження – віртуалізована платформа для тестування програмного забезпечення у різних середовищах.

Предмет дослідження – методи автоматизованого тестування у віртуалізованих середовищах, технології віртуалізації та інструменти для забезпечення якості програмного забезпечення.

Методологія дослідження включає аналіз та узагальнення літературних джерел з тематики віртуалізованого тестування програмного забезпечення, дослідження методів автоматизації тестування, моделювання процесів взаємодії тестового середовища із програмним забезпеченням, а також розробку та впровадження тестової платформи для перевірки програм у різних середовищах. У процесі дослідження застосовуються методи математичного моделювання, інструменти контейнеризації та емуляції, а також алгоритми автоматизації тестування для оцінки продуктивності та безпеки програмного забезпечення.

Практична значущість роботи полягає у створенні віртуалізованої тестової платформи, яка дозволить автоматизувати процес тестування програмного забезпечення у різних середовищах. Це сприятиме підвищенню якості тестування, зниженню витрат на апаратне забезпечення, пришвидшенню розробки та випуску програмних продуктів, а також покращенню їхньої сумісності з різними технологічними платформами.

1 ВІРТУАЛІЗОВАНІ СЕРЕДОВИЩА ТА АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Основи автоматизованого тестування програмного забезпечення

Автоматизоване тестування програмного забезпечення – це процес контролю якості, який передбачає використання спеціалізованих інструментів і скриптів для автоматичного виконання тестових сценаріїв з метою перевірки функціональності, продуктивності та стабільності програмного забезпечення. Основною метою автоматизованого тестування є підвищення ефективності процесу виявлення помилок, зменшення впливу людського фактора, а також оптимізація витрат часу та ресурсів, необхідних для забезпечення належного рівня якості програмних продуктів [1].

У процесі автоматизованого тестування програмного забезпечення застосовуються різні типи тестів, кожен з яких має свою специфіку, цілі та рівень абстракції. До основних типів відносять модульні (unit), інтеграційні (integration), інтерфейсні (UI) та API-тести.

Unit-тестування (модульне тестування) полягає у перевірці окремих функцій, методів або класів програми. Основна мета – впевнитися, що конкретна частина коду працює відповідно до вимог. Unit-тести зазвичай пишуться розробниками на етапі розробки і є основою для виявлення помилок на ранніх стадіях [2].

Integration-тестування (інтеграційне тестування) спрямоване на перевірку взаємодії між окремими модулями або компонентами системи. Після того як окремі частини були протестовані за допомогою unit-тестів, необхідно переконатися, що вони правильно працюють разом.

UI-тестування (тестування користувацького інтерфейсу) фокусується на перевірці елементів взаємодії користувача з додатком. Такі тести моделюють дії користувача (наприклад, кліки, введення тексту, навігацію), щоб перевірити коректність відображення інтерфейсу та логіку його поведінки.

API-тестування орієнтоване на перевірку прикладних інтерфейсів (Application Programming Interfaces). Воно дозволяє протестувати логіку обміну даними між клієнтською та серверною частинами додатка, забезпечити відповідність запитів і відповідей встановленим специфікаціям, а також перевірити обробку помилок, авторизацію, аутентифікацію та продуктивність API.

На рис. 1.1 показано, що усі ці типи тестування є складовими частинами автоматизованої стратегії забезпечення якості програмного забезпечення, а їх комплексне використання дає змогу досягти високого рівня надійності, стабільності та безпеки програмного продукту.

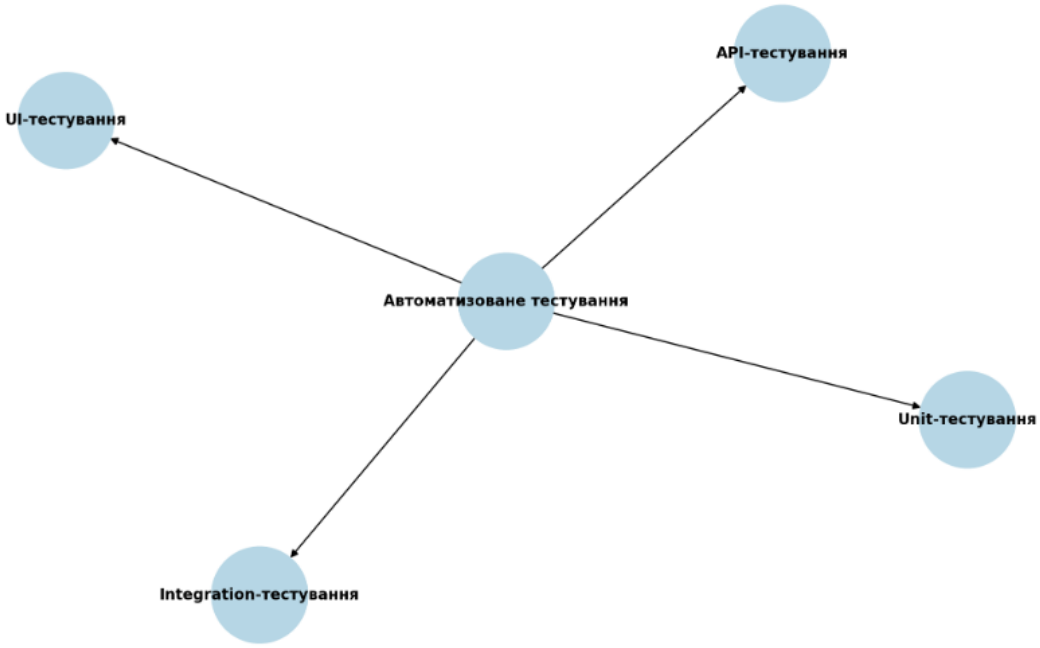


Рисунок 1.1 – Основні типи автоматизованого тестування програмного забезпечення

Автоматизоване тестування програмного забезпечення має низку істотних переваг над ручним тестуванням, що робить його незамінним інструментом у сучасних практиках забезпечення якості. Однією з ключових переваг є висока швидкість виконання тестових сценаріїв, особливо в умовах частих оновлень та релізів. Завдяки автоматизації можливо за лічені хвилини виконати сотні або навіть тисячі тестів, що вручну вимагало б значних часових і людських ресурсів. Це особливо актуально у процесах безперервної інтеграції (CI) та безперервного

розгортання (CD), де ефективність перевірок безпосередньо впливає на швидкість розробки та випуску продукту. Ще однією важливою перевагою автоматизованого тестування є повторюваність і стабільність результатів. На відміну від ручного тестування, де можлива варіативність результатів через втому, неуважність або людські помилки, автоматизовані тести виконуються за фіксованим сценарієм і дають однакові результати при кожному запуску, що підвищує надійність тестування. До того ж автоматизація дозволяє забезпечити значно вищий рівень покриття коду, оскільки вона дає змогу запускати більшу кількість тестів, включаючи як позитивні, так і негативні сценарії, що унеможливлює в ручному режимі через обмеженість часу. Також суттєво зменшуються витрати на рутинну працю тестувальників: автоматизовані скрипти можуть виконувати повторювані дії, даючи можливість фахівцям зосередитися на складніших видах тестування, таких як exploratory testing, аналіз специфічних поведінкових моделей або оцінка юзабіліті [3]. Значною перевагою є можливість тестування у різноманітних середовищах і конфігураціях завдяки використанню емуляторів, віртуальних машин, контейнерів та хмарних платформ. Це особливо актуально для мобільних і веб-застосунків, які мають працювати стабільно на різних пристроях, операційних системах та версіях програмного забезпечення. Крім того, автоматизовані тести легко масштабуються: один і той самий сценарій можна запускати паралельно на декількох середовищах, що значно прискорює процес валідації. Варто також зазначити, що результати автоматизованого тестування легко інтегруються в системи звітності, що забезпечує прозорість процесу тестування для всієї команди розробки. Автоматизоване тестування суттєво знижує загальну вартість контролю якості в довгостроковій перспективі, забезпечуючи вищу швидкість розробки, надійність програмного продукту та стабільність його роботи в різних умовах експлуатації.

У практиці автоматизованого тестування програмного забезпечення активно використовуються численні інструменти, кожен з яких має власну спеціалізацію, функціональні можливості та область застосування. Серед найбільш поширених

засобів можна виділити Selenium, JUnit та TestNG, які широко застосовуються для побудови ефективних і масштабованих тестових рішень.

Selenium є одним із найпопулярніших фреймворків для автоматизації тестування веб-застосунків. Його головна перевага полягає в підтримці багатьох мов програмування (Java, Python, C#, JavaScript тощо), а також можливості інтеграції з різними браузерами та платформами. Selenium WebDriver дозволяє імітувати дії користувача, перевіряти функціональність інтерфейсу та взаємодію з елементами сторінки, що робить його незамінним у тестуванні фронтенду. Крім того, Selenium активно використовується в CI/CD-процесах, оскільки добре інтегрується з Jenkins, Maven та іншими інструментами розробки [4].

JUnit є фреймворком для модульного тестування у середовищі Java. Його основна мета – забезпечити розробників зручним інструментом для написання та виконання unit-тестів. JUnit дозволяє структуровано організовувати тестові кейси, легко додавати нові сценарії та отримувати звіти про успішність їх виконання. Завдяки простому синтаксису та активному розвитку спільноти, JUnit став стандартом де-факто для автоматизації тестування Java-додатків. Його часто застосовують як базову платформу для побудови більш складних систем тестування, що включають перевірку логіки бізнес-процесів, взаємодії компонентів та коректності обробки виняткових ситуацій [5].

TestNG – ще один потужний інструмент для автоматизованого тестування, який також базується на мові Java, але має розширені можливості порівняно з JUnit. TestNG підтримує параметризовані тести, залежності між тестами, групування, конфігурацію виконання та детальну генерацію звітів, що робить його зручним для реалізації інтеграційного, функціонального та навіть навантажувального тестування. Завдяки гнучкості налаштування та розширеній анотації, TestNG дозволяє ефективно управляти складними тестовими наборами та інтегрується з інструментами для побудови (Maven, Gradle) і CI/CD-системами [6].

Таким чином, вибір конкретного інструменту автоматизації тестування залежить від типу застосунку, мовної платформи, вимог до покриття та інтеграції. Selenium, JUnit і TestNG є прикладами надійних та перевірених рішень, які

забезпечують гнучкість, масштабованість та високу якість процесу автоматизованого тестування. У таблиці 1.1 наведено порівняльну характеристику найбільш поширених інструментів автоматизованого тестування, що використовуються для перевірки функціональності, надійності та сумісності програмного забезпечення.

Таблиця 1.1 – Порівняльна таблиця інструментів тестування

Інструмент	Призначення	Мова програмування	Типи тестів	Особливості
Selenium	Автоматизоване тестування веб-інтерфейсів	Java, Python, C#, JavaScript та інші	UI-тести (веб), інтеграційні	Підтримка різних браузерів, інтеграція з CI/CD
JUnit	Unit-тестування для Java	Java	Unit-тести	Простий у використанні, широко підтримується спільнотою
TestNG	Розширене unit та integration-тестування для Java	Java	Unit, integration, functional	Параметризація тестів, залежності між тестами, потужна конфігурація

1.2 Підходи до використання віртуалізованих середовищ у тестуванні ПЗ

Віртуальні машини є компонентом інфраструктури автоматизованого тестування, особливо у випадках, коли необхідно забезпечити ізольовані та контрольовані середовища для запуску програмного забезпечення. До найбільш популярних рішень у цій сфері належать VMware та VirtualBox, які широко використовуються як у промислових, так і в навчальних цілях.

VMware – це комерційне програмне забезпечення для віртуалізації, яке надає широкий спектр інструментів для створення, налаштування та управління віртуальними машинами [7]. Його перевагами є висока продуктивність, підтримка великої кількості операційних систем, гнучкі можливості керування мережевими

налаштуваннями, а також інтеграція з корпоративними інструментами віртуалізації, такими як VMware vSphere. Це робить VMware одним із найнадійніших рішень для створення масштабованих тестових середовищ.



Рисунок 1.2 – Логотип VMware

VirtualBox – це безкоштовне програмне забезпечення з відкритим кодом, яке забезпечує широкі можливості для створення і запуску віртуальних машин на різних хост-системах, зокрема Windows, Linux та macOS. Завдяки простому інтерфейсу та підтримці скриптів для автоматизації, VirtualBox є популярним вибором серед розробників і тестувальників [8]. Його гнучкість дозволяє легко відтворювати різні середовища, налаштовувати симуляцію мережевих взаємодій та швидко змінювати конфігурацію тестового стенду.



Рисунок 1.3 – Логотип VirtualBox

Обидва інструменти забезпечують можливість збереження стану системи, знімків (snapshots), а також експорту й імпорту віртуальних машин, що є зручним для розгортання стандартних конфігурацій у процесі тестування [9]. Застосування віртуальних машин дозволяє істотно скоротити витрати на фізичне обладнання,

підвищити масштабованість тестування та забезпечити відтворюваність результатів у стандартизованому середовищі.

Одним із сучасних підходів до організації тестового середовища є використання контейнеризації, яка значно відрізняється від традиційної віртуалізації. Найбільш популярним інструментом у цій сфері є Docker – платформа з відкритим кодом, яка забезпечує створення, розгортання та управління контейнерами. Контейнери дозволяють ізолювати застосунок разом з усіма його залежностями у компактному середовищі, яке може працювати незалежно від хост-системи.

На відміну від віртуальних машин, які емулюють повноцінну операційну систему з власним ядром, контейнери розділяють ядро хост-системи, що робить їх легшими та швидшими у запуску. Контейнер займає значно менше місця, швидко створюється та знищується, що є критично важливим при автоматизованому тестуванні в рамках CI/CD-процесів. Наприклад, за допомогою Docker можна миттєво створити тестове середовище з точною версією бази даних, необхідним веб-сервером чи сервісом API, а після завершення тестів – так само швидко його видалити [10].

Ключова перевага контейнерів у тестуванні полягає у можливості точного відтворення середовища незалежно від конфігурації машини розробника або тестувальника. Це усуває проблему "працює у мене" (англ. "it works on my machine") та забезпечує стабільність і повторюваність результатів тестування. Крім того, завдяки системі Docker Compose можливо одночасно запускати кілька пов’язаних між собою сервісів (наприклад, застосунок, базу даних та інструмент моніторингу), що спрощує налаштування складної тестової інфраструктури.

Контейнеризація, як технологія, має низку переваг, що роблять її надзвичайно ефективною у процесах тестування програмного забезпечення. Серед ключових переваг слід виокремити масштабованість, ізоляцію та швидкість відновлення середовищ.

Масштабованість забезпечується можливістю швидкого створення та запуску великої кількості контейнерів з однаковими або різними конфігураціями,

що дозволяє ефективно розподіляти навантаження під час паралельного виконання тестів. Ізоляція контейнерів гарантує незалежність окремих середовищ, що дозволяє уникнути конфліктів між компонентами системи, зменшує ризики впливу одного сервісу на інший та забезпечує точність результатів тестування. Швидкість відновлення досягається завдяки можливості швидкого перезапуску або пересоздання контейнера в разі помилки, що значно скорочує час, необхідний для відновлення працездатного стану системи.

Ці переваги в поєднанні формують основу для побудови надійної, гнучкої та автоматизованої інфраструктури тестування, особливо у середовищах з високими вимогами до стабільності, безперервності та швидкості розробки.

1.3 Класифікація платформ для віртуалізації та їх застосування

У сфері інформаційних технологій віртуалізація є ключовою технологією, яка забезпечує ефективне використання ресурсів, ізоляцію середовищ та гнучкість у керуванні інфраструктурою. Залежно від рівня реалізації та призначення, розрізняють кілька основних типів віртуалізації: апаратну, програмну та контейнерну. Кожен із типів має свої особливості, переваги, недоліки та сфери застосування.

Апаратна віртуалізація (hardware virtualization) є найбільш повнофункціональним типом і передбачає створення віртуальної машини, яка повністю імітує фізичне апаратне забезпечення комп’ютера. У цьому випадку використовується гіпервізор (hypervisor) – програмний компонент, що функціонує як посередник між фізичним обладнанням і віртуальними машинами. Гіпервізори бувають двох типів: типу 1 (bare-metal), що працюють безпосередньо на апаратному забезпеченні (наприклад, VMware ESXi, Microsoft Hyper-V, Xen), та типу 2 (hosted), які встановлюються поверх операційної системи (наприклад, VMware Workstation, VirtualBox). Апаратна віртуалізація дозволяє одночасно запускати кілька повноцінних операційних систем, які ізольовані одна від одної, з власним ядром і драйверами. Вона забезпечує високу гнучкість, але потребує

значних обчислювальних ресурсів і часу на створення віртуального середовища [11].

Програмна віртуалізація (software virtualization) полягає у створенні віртуального середовища в межах однієї операційної системи без повної імітації апаратного рівня. Такий підхід може включати емуляцію конкретних компонентів (наприклад, віртуальних принтерів, мережевих інтерфейсів, ізольованих додатків) або виконання програм у захищеному середовищі, яке імітує поведінку системи. Прикладами програмної віртуалізації є Java Virtual Machine (JVM), .NET CLR, Wine для запуску Windows-додатків у Linux. Перевага цього типу віртуалізації – менші витрати ресурсів і швидший запуск, але при цьому можливості ізоляції та сумісності обмежені в порівнянні з апаратною віртуалізацією.

Контейнерна віртуалізація (container-based virtualization) є сучасним підходом, який передбачає ізоляцію програмного середовища на рівні операційної системи. На відміну від апаратної віртуалізації, контейнери не потребують окремого ядра ОС і працюють поверх одного загального ядра, що значно зменшує споживання ресурсів. Контейнери включають усі необхідні для виконання застосунку залежності, такі як бібліотеки, конфігураційні файли та середовища виконання. Найбільш поширеним інструментом контейнеризації є Docker, а також платформи Kubernetes для оркестрації контейнерів у масштабованих кластерах. Контейнерна віртуалізація забезпечує швидкий запуск, легке масштабування, мобільність середовищ та просту інтеграцію в процеси CI/CD, проте обмежена можливістю повної емуляції ОС та рівнем безпеки, який залежить від конфігурації ядра [12].

Вибір типу віртуалізації залежить від поставлених задач: апаратна – для повної ізоляції та гнучкої багатоплатформної підтримки, програмна – для легких сценаріїв або сумісності окремих компонентів, контейнерна – для високої швидкості, масштабованості та інтеграції в хмарні або DevOps-орієнтовані архітектури.

У процесі створення тестових середовищ широко застосовуються різноманітні інструменти віртуалізації та контейнеризації. Серед них найбільш

поширеними є Docker, Vagrant і VirtualBox. Кожна з цих платформ має свої особливості, призначення та область застосування. Їх вибір залежить від завдань розробника або тестувальника, ресурсних обмежень, вимог до ізоляції, швидкості розгортання та сумісності з іншими інструментами.

Docker є провідною платформою контейнеризації, яка дозволяє створювати легковагі, швидко запускані контейнери ізольованих середовищ, що працюють поверх спільного ядра операційної системи. Контейнери Docker забезпечують чудову масштабованість, швидкість запуску, інтеграцію з CI/CD-процесами та портативність. Вони особливо ефективні для мікросервісної архітектури та автоматизованого тестування, де важлива швидка зміна конфігурації.

Vagrant – це інструмент для керування віртуальними машинами, що орієнтований на автоматизацію розгортання середовищ, здебільшого у поєднанні з VirtualBox, VMware чи іншими гіпервізорами. Його основна мета – забезпечити відтворюване середовище для розробки та тестування. Vagrant дозволяє описати конфігурацію віртуальної машини у вигляді коду (Vagrantfile), що спрощує її поширення та підтримку.

VirtualBox – це класичне рішення для повної апаратної віртуалізації. Воно дає змогу запускати повноцінні операційні системи на базі хост-ОС і забезпечує високу сумісність та ізоляцію. VirtualBox часто використовується для тестування взаємодії на рівні ОС, перевірки драйверів, безпеки та сумісності в складних середовищах, але має нижчу швидкість запуску та більші вимоги до ресурсів [13].

У табл. 1.3 наведено порівняльну характеристику трьох популярних платформ – Docker, Vagrant та VirtualBox – які активно використовуються в сучасних процесах розробки та тестування програмного забезпечення. Порівняння охоплює основні технічні параметри, зокрема тип реалізованої віртуалізації, швидкість розгортання середовищ, рівень використання ресурсів, ізоляцію, можливість підтримки графічного інтерфейсу, портативність, а також ступінь інтеграції з інструментами CI/CD. Представлені платформи мають різне функціональне призначення, і вибір між ними залежить від специфіки задач, вимог до продуктивності, гнучкості та відтворюваності середовища тестування.

Таблиця 1.3 – Порівняльна таблиця платформ: Docker, Vagrant, VirtualBox

Характеристика	Docker	Vagrant	VirtualBox
Тип технології	Контейнеризація	Інструмент автоматизації ВМ	Апаратна віртуалізація
Підтримка ОС	Linux, Windows, macOS	Linux, Windows, macOS	Linux, Windows, macOS
Рівень віртуалізації	На рівні ОС (shared kernel)	Повна ОС через гіпервізор	Повна ОС через гіпервізор
Швидкість запуску	Дуже висока (секунди)	Середня (хвилини)	Низька (хвилини)
Використання ресурсів	Низьке	Середнє	Високе
Ізоляція середовищ	Середня	Висока	Висока
Підтримка GUI	Не підтримує напряму	Можлива через гостьову ОС	Підтримується повністю
Портативність	Висока (образи контейнерів)	Середня (Vagrantfile + box)	Низька (великі файли образів)
Інтеграція з CI/CD	Чудова	Хороша	Обмежена
Типові сценарії	Мікросервіси, API, автотестування	Відтворювані середовища розробки	Повна емуляція ОС, сумісність
Формат конфігурації	Dockerfile	Vagrantfile	XML, GUI
Простота використання	Середня	Висока	Середня
Ширина спільноти	Дуже широка	Широка	Широка

Використання віртуалізації та контейнеризації у сфері тестування програмного забезпечення дозволяє підвищити ефективність, надійність і відтворюваність тестових процесів. Залежно від цілей тестування та типу програмного продукту застосовуються різні підходи до побудови тестового середовища.

Одним із типових прикладів є тестування веб-застосунків у різних браузерах. За допомогою контейнерів Docker можна автоматизовано розгортати середовища з попередньо встановленими версіями браузерів (наприклад, Chrome, Firefox), що

дозволяє швидко перевірити сумісність інтерфейсу та функціоналу. У поєднанні з такими інструментами, як Selenium Grid, забезпечується паралельне виконання тестів у кількох середовищах одночасно.

Ще одним сценарієм є тестування мікросервісної архітектури. Контейнери дозволяють запускати кожен мікросервіс у власному ізольованому середовищі, що спрощує перевірку взаємодії між компонентами системи, аналіз API-запитів і стабільності роботи при зміні окремих сервісів. Завдяки використанню Docker Compose можна описати взаємозалежності між сервісами та забезпечити їх одночасний запуск для комплексного тестування.

У випадках, коли необхідно перевірити сумісність програмного забезпечення з різними версіями операційних систем або специфічними конфігураціями, доцільним є застосування віртуальних машин, створених за допомогою VirtualBox або VMware. Це дає змогу моделювати середовища з різними обсягами оперативної пам’яті, роздільною здатністю екрана або встановленими драйверами, що є важливим для desktop-додатків або системного ПЗ.

Також варто згадати про використання контейнерів для навантажувального тестування, коли необхідно імітувати велику кількість одночасних підключень або запитів до серверного застосунку. Створення декількох однотипних контейнерів з генераторами трафіку дозволяє точно оцінити продуктивність, стабільність і здатність системи до масштабування.

1.4 Аналіз існуючих технологій та інструментів для тестування у віртуальних середовищах

У процесі автоматизованого тестування програмного забезпечення важливу роль відіграють спеціалізовані платформи та сервіси, що забезпечують організацію, масштабування та виконання тестів у різних середовищах. Серед найбільш відомих і ефективних рішень можна виокремити Jenkins, Selenium Grid, BrowserStack та SauceLabs.

Jenkins – це одна з найпопулярніших систем для організації процесів

безперервної інтеграції та безперервного розгортання (CI/CD). Це програмне забезпечення з відкритим кодом, яке дозволяє автоматизувати різноманітні завдання, пов’язані з тестуванням, збиранням, перевіркою, доставкою та розгортанням програмного забезпечення. Завдяки розширеній системі плагінів Jenkins легко інтегрується з такими інструментами, як Git, Docker, Maven, JUnit, Selenium та багатьма іншими. У контексті тестування Jenkins дозволяє автоматизовано запускати тести після кожного коміту або з певною періодичністю, зберігати результати, формувати звіти та відстежувати історію виконання. Його гнучкість і можливість масштабування роблять Jenkins базовим компонентом інфраструктури тестування у багатьох компаніях.

Selenium Grid – це розширення до Selenium, яке дозволяє виконувати тести паралельно на кількох машинах та в різних браузерах. Воно реалізує архітектуру “хаб–ноди”, де хаб є центральною точкою управління тестами, а ноди – це інстанції з браузерами, які приймають запити на виконання тестів. Завдяки цій архітектурі Selenium Grid забезпечує розподілене виконання тестів, що суттєво знижує час тестування та дозволяє одночасно перевіряти застосунок у різних середовищах. Такий підхід є особливо корисним при тестуванні інтерфейсів у кількох браузерах, операційних системах або конфігураціях пристроїв. Selenium Grid можна використовувати як локально, так і в хмарному середовищі, зокрема у поєднанні з Docker або Kubernetes для гнучкого масштабування [14] .

BrowserStack – це хмарна платформа для кросбраузерного та кросплатформного тестування, яка надає доступ до широкого спектру реальних пристроїв і браузерів у реальному часі. Головна перевага BrowserStack полягає у можливості тестування без необхідності локального встановлення віртуальних машин чи емуляторів. Платформа дозволяє запускати як ручні, так і автоматизовані тести, включаючи інтеграцію з Selenium, Appium, Cypress, Jenkins та іншими CI/CD-інструментами. BrowserStack підтримує як веб-, так і мобільне тестування на реальних iOS та Android пристроях, що є надзвичайно важливим для виявлення багів, які не завжди можна виявити на емуляторах. Крім того, сервіс надає інструменти для моніторингу, створення скріншотів, запису відео та детального

аналізу результатів тестування.

SauceLabs – ще одна потужна хмарна платформа для автоматизованого тестування, яка дозволяє запускати тести у великій кількості середовищ, включаючи різні браузери, версії операційних систем і мобільні пристрої. На відміну від локальних рішень, SauceLabs забезпечує високу масштабованість і готову інфраструктуру для паралельного запуску тестів. Платформа підтримує інтеграцію з основними фреймворками автоматизованого тестування, такими як Selenium, Appium, Cypress, Espresso, XCUITest тощо. SauceLabs також надає зручну панель керування, звіти про помилки, знімки екранів, відеозаписи сесій тестування та аналітику. Особливістю сервісу є можливість тестування не лише функціональності, а й продуктивності та безпеки програмного забезпечення, що дозволяє проводити комплексне оцінювання якості продукту.

Jenkins, Selenium Grid, BrowserStack і SauceLabs покривають різні аспекти автоматизованого тестування – від управління процесами CI/CD до масштабованого розподіленого тестування в реальних середовищах. Комбіноване використання цих рішень дозволяє побудувати ефективну, надійну й автоматизовану інфраструктуру тестування для сучасних програмних продуктів.

У рамках даного дослідження розглядається можливість використання комплексу інструментів, які забезпечують ефективне автоматизоване тестування у віртуалізованому середовищі. До таких інструментів належать Appium, Cucumber, Mitmproxy та Mock-server. Їх вибір зумовлений поширеністю, гнучкістю, відкритим вихідним кодом та активною підтримкою спільноти.

Appium доцільно використовувати для автоматизації тестування мобільних застосунків, оскільки він дозволяє проводити тести без необхідності вносити зміни у вихідний код. Цей інструмент підтримує різні операційні системи та працює з широким набором мобільних пристроїв і емуляторів [11].

Cucumber дозволяє описувати тестові сценарії у форматі, зрозумілому як технічним спеціалістам, так і представникам бізнесу. Такий підхід полегшує комунікацію між учасниками проєкту та дає змогу формалізувати вимоги до програмного забезпечення у вигляді простих текстових сценаріїв.

Mitmproxy виконує роль посередника для аналізу мережевого трафіку, що є особливо корисним при тестуванні взаємодії з API. Його застосування дозволяє спостерігати, перехоплювати та аналізувати HTTP(S)-запити, що допомагає у виявленні помилок або недокументованої поведінки системи.

Mock-server використовується для створення умовного (віртуального) серверного оточення, коли доступ до реального API є обмеженим або нестабільним. Це дозволяє проводити тестування ізольовано від зовнішніх систем, що підвищує стабільність процесу.

Розглянуті інструменти створюють основу для побудови багаторівневої системи автоматизованого тестування. У наступних розділах планується проаналізувати їхню взаємодію та застосувати у рамках експериментальної частини дослідження.

Висновок до розділу 1

Проведеного теоретичного дослідження було розглянуто основні аспекти автоматизованого тестування програмного забезпечення та особливості використання віртуалізованих середовищ у цьому процесі. Визначено, що автоматизація тестування дозволяє значно підвищити ефективність, швидкість і стабільність перевірки функціональності програмних продуктів, знижуючи залежність від людського фактора та прискорюючи процес розробки.

Проаналізовано основні підходи до віртуалізації, зокрема апаратну, програмну та контейнерну, кожна з яких має свої переваги залежно від цілей тестування. Наведено класифікацію платформ для віртуалізації, таких як VirtualBox, VMware, Docker, Vagrant, а також визначено їх роль у створенні гнучких, масштабованих і ізольованих середовищ для тестування.

Розглянуто сучасні інструменти, що використовуються у віртуалізованому тестуванні, зокрема Jenkins, Selenium Grid, BrowserStack та SauceLabs, які забезпечують ефективне планування, виконання та моніторинг тестів у різних конфігураціях. Окрему увагу приділено підходу, що передбачає використання Appium, Cucumber, Mitmproxy та Mock-server як перспективних інструментів для реалізації автоматизованого багаторівневого тестування у віртуальному середовищі.

2 ТЕХНОЛОГІЇ ТА ІНСТРУМЕНТИ ВІРТУАЛІЗОВАНОГО ТЕСТУВАННЯ

2.1 Використання емуляторів та контейнеризації для тестування

Docker – це платформа контейнеризації з відкритим кодом, яка дозволяє створювати, розгортати та запускати програмне забезпечення в ізольованих середовищах, званих контейнерами (рис. 2.1). Контейнери Docker забезпечують легку, відтворювану та масштабовану інфраструктуру для розробки, тестування та розгортання додатків, що робить цей інструмент надзвичайно популярним у сучасних практиках DevOps.



Рисунок 2.1 – Логотип Docker

Принципи роботи Docker базуються на використанні єдиного ядра операційної системи для всіх контейнерів, що забезпечує їхню легкість і швидкість у порівнянні з традиційними віртуальними машинами. Кожен контейнер містить лише додаток і всі його залежності, але не включає повноцінну ОС. Docker використовує клієнт-серверну архітектуру: клієнт надсилає команди до демона dockerd, який керує створенням, запуском і зупинкою контейнерів. Завдяки цьому підходу контейнери є ізольованими, але водночас використовують спільні ресурси ефективно, що дає змогу запускати десятки чи сотні контейнерів на одному сервері [14].

Елементом екосистеми Docker є образи (images) – це шаблони, на основі яких створюються контейнери. Образ включає все необхідне для запуску програми: код, бібліотеки, залежності, конфігураційні файли тощо. Образи створюються за

допомогою спеціального сценарію – Dockerfile, який описує кроки складання середовища. Наприклад, Dockerfile може вказати базовий образ (наприклад, node, ubuntu, python), скопіювати до нього файли застосунку, встановити залежності та налаштувати команду запуску. Образи можуть бути збережені в локальному сховищі або завантажені в публічні чи приватні реєстри (наприклад, Docker Hub), що дозволяє легко поширювати та повторно використовувати середовища.

Однією переваг Docker є його інтеграція з процесами CI/CD (Continuous Integration/Continuous Delivery). У межах CI/CD-пайплайнів Docker використовується для автоматичного створення, тестування та розгортання контейнеризованих додатків. Наприклад, при кожному коміті до репозиторію система CI (наприклад, Jenkins, GitLab CI, GitHub Actions) може створити новий Docker-образ, виконати в ньому автоматизовані тести та, у разі успіху, передати його на сервер розгортання. Це дозволяє забезпечити стабільність середовища незалежно від того, де саме відбувається запуск: на локальному комп’ютері розробника, у тестовому середовищі чи у продакшн.

Docker поєднується з інструментами моніторингу, масштабування та оркестрації, зокрема з Kubernetes, що дозволяє реалізувати високонавантажені, відмовостійкі системи з автоматичним балансуванням навантаження. Такий підхід забезпечує гнучкість, знижує ризики конфігураційних помилок та дозволяє швидко відновлювати середовище при потребі.

У рамках автоматизованого тестування мобільних застосунків важливим аспектом є перевірка їхньої коректної роботи на різних типах пристроїв, версіях операційної системи та конфігураціях середовища. Для цього широко використовуються емулятори — програмні засоби, які імітують роботу реального мобільного пристрою без необхідності використання фізичного обладнання. Серед найбільш популярних інструментів у цій категорії варто виокремити Android Virtual Device (AVD), Genymotion та Bluestacks [15].

Android Virtual Device (AVD) – це стандартний емулятор, що входить до складу Android Studio (рис. 2.2). Він забезпечує повну інтеграцію з офіційними інструментами розробки Android і дозволяє створювати емулятори з різними

версіями Android, розмірами екрану, обсягом пам’яті та іншими характеристиками. AVD емулює поведінку пристрою на рівні операційної системи, включаючи підтримку сенсорів, камери, геолокації тощо. Його перевагами є гнучкість налаштувань, точна відповідність специфікаціям Android та сумісність з усіма засобами автоматизації, зокрема Appium. Проте AVD вимагає значних ресурсів системи і часто працює повільніше за інші альтернативи.



Рисунок 2.2 –Логотип Android Virtual Device

Genymotion – це комерційний емулятор Android, орієнтований на професійне використання в тестуванні. Він підтримує запуск як на локальній машині, так і у хмарному середовищі, що робить його зручним для інтеграції у CI/CD пайплайни (рис. 2.3). Genymotion відрізняється високою швидкістю роботи завдяки апаратному прискоренню (використання VirtualBox), широкими можливостями конфігурації пристроїв, підтримкою мережевих сценаріїв, емуляцією сенсорів та GPS. Крім того, Genymotion має зручний інтерфейс, підтримує інтеграцію з Appium і дозволяє знімати скріншоти, записувати відео тестових сесій тощо. Його гнучкість та продуктивність роблять його популярним вибором серед тестувальників мобільного ПЗ [15].



Рисунок 2.3 – Логотип Genymotion

Bluestacks – це емулятор Android, який позиціонується передусім як рішення для запуску мобільних застосунків на Windows або macOS. Він орієнтований на кінцевих користувачів, але також може бути використаний для базового тестування застосунків у нестандартних умовах (рис. 2.4). Bluestacks має простий інтерфейс, високу сумісність із великою кількістю застосунків, можливість взаємодії з Google Play та підтримку керування за допомогою миші й клавіатури [13]. Однак у порівнянні з AVD і Genymotion, Bluestacks має обмежену гнучкість конфігурації пристроїв, менш точне відтворення середовища розробки та обмежену підтримку автоматизованого тестування на професійному рівні.



Рисунок 2.4 – Логотип емулятора Bluestacks

У рамках даної роботи для цілей тестування мобільного застосунку буде використовуватись емулятор Genymotion, який забезпечує високу швидкість роботи, зручність налаштування віртуальних пристроїв та сумісність з інструментами автоматизованого тестування, зокрема Appium. Його застосування дозволяє ефективно моделювати поведінку реальних Android-пристроїв у віртуалізованому середовищі, що є важливим при побудові масштабованої та відтворюваної тестової інфраструктури.

2.2 Інструменти моніторингу та перехоплення мережевого трафіку

У процесі тестування програмного забезпечення, зокрема при взаємодії з мережевими сервісами, важливу роль відіграють інструменти для перехоплення, аналізу та діагностики мережевого трафіку. До таких інструментів належать Mitmproxy та Wireshark, які дозволяють тестувальнику досліджувати внутрішні процеси передачі даних, виявляти помилки у взаємодії з API та оцінювати безпеку застосунків.

Mitmproxy – це потужний інтерактивний проксі-сервер з відкритим кодом, що працює за принципом “людина посередині” (англ. Man-in-the-Middle). Він дозволяє перехоплювати, переглядати, модифікувати та повторно надсилати HTTP і HTTPS-запити між клієнтом та сервером (рис. 2.5). У контексті тестування програмного забезпечення Mitmproxy є надзвичайно корисним для відлагодження запитів до API, перевірки вмісту відповідей, тестування надійності обробки помилок та емуляції поведінки сторонніх сервісів. Його застосування не потребує змін у коді тестованого застосунку, що робить його ефективним у поєднанні з мобільними або веб-додатками [14]. Крім того, Mitmproxy може використовуватись для виявлення потенційних вразливостей, пов’язаних із ненадійною обробкою сертифікатів або недостатньо захищеним трафіком.



Рисунок 2.5 – Логотип Mitmproxy

Wireshark – це широко визнаний аналізатор мережевого трафіку, який дає змогу здійснювати глибокий аналіз пакетів, що проходять через мережу (рис. 2.6). На відміну від Mitmproxy, який орієнтований переважно на рівень HTTP(S), Wireshark дозволяє досліджувати низькорівневі протоколи (наприклад, TCP, UDP,

DNS, ARP тощо). Він забезпечує детальний перегляд кожного пакета, дозволяє фільтрувати трафік за різними критеріями, зберігати лог-файли для подальшого аналізу, а також виявляти помилки в конфігурації мережі або нестандартну поведінку програм. У тестуванні програмного забезпечення Wireshark корисний для моніторингу взаємодії додатка з мережею, аналізу затримок, діагностики несправностей або перевірки відповідності мережевих транзакцій очікуваним сценаріям.



Рисунок 2.6 – Логотип Wireshark

Обидва інструменти можуть використовуватись у рамках процесу автоматизованого тестування, зокрема у віртуалізованих середовищах. Вони забезпечують цінну інформацію про роботу системи на рівні мережевої взаємодії, що дозволяє підвищити точність аналізу, виявляти критичні помилки на ранніх етапах та забезпечити відповідність вимогам безпеки й надійності.

У процесі тестування сучасного програмного забезпечення значну роль відіграє аналіз запитів до API (Application Programming Interface), який дозволяє перевірити коректність взаємодії клієнтської та серверної частин системи. Застосування інструментів для моніторингу, перехоплення та аналізу запитів є важливим етапом як у ручному, так і в автоматизованому тестуванні, особливо у випадках, коли програмний продукт активно взаємодіє з зовнішніми сервісами, базами даних або мікросервісною інфраструктурою [15].

Можливості аналізу запитів включають у с ебе фіксацію повного вмісту HTTP/HTTPS-запитів і відповідей, включно з заголовками, параметрами, тілом повідомлень, статус-кодами, часом відповіді та іншими метаданими. Це дозволяє тестувальнику не лише виявити помилки формування запитів, некоректну обробку параметрів або невідповідність формату відповіді, але й оцінити продуктивність та

поведінку системи в умовах навантаження. Аналіз трафіку також дозволяє виявляти неочевидні помилки в логіці роботи API, наприклад, неправильне кешування, некоректну обробку помилок або вразливості, пов’язані із несанкціонованим доступом.

У тестуванні API важливим є використання інструментів, які дозволяють не тільки надсилати запити, але й аналізувати відповіді, перевіряти статус-коди, відповідність JSON-структур, час відгуку та наявність необхідних полів. У цьому контексті активно застосовуються такі засоби, як Postman, Curl, RestAssured, Mitmproxy, Wireshark, а також автоматизовані фреймворки на кшталт Cypress або JUnit з бібліотекою REST-assured. Особливу цінність має інтеграція аналізу запитів у автоматизовані сценарії, що дозволяє перевіряти API під час кожного коміту або оновлення коду в CI/CD-процесі [16].

Є можливість емуляції зовнішніх API з використанням mock-серверів. Це дозволяє ізолювати клієнтський застосунок від реального серверу, який може бути недоступним або перебувати на етапі розробки. За допомогою таких інструментів, як Mock-server, можливо тестувати реакцію системи на специфічні сценарії: помилки автентифікації, нестабільну відповідь, затримку у передачі даних тощо.

2.3 Тестування безпеки та атаки типу "Людина посередині" (MitM)

Атака типу «людина посередині» (англ. Man-in-the-Middle, MitM) – це один із найпоширеніших типів атак у комп’ютерних мережах, суть якого полягає в тому, що зловмисник перехоплює, аналізує або змінює передані між двома сторонами дані без їх відома. Така атака порушує основні принципи інформаційної безпеки: конфіденційність, цілісність та автентичність переданої інформації.

У типовому сценарії MitM-атаки зловмисник розташовується між клієнтом і сервером (наприклад, між мобільним застосунком і веб-сервером), перехоплюючи весь трафік, що передається між ними. Це може відбуватися як у локальній мережі (наприклад, через ARP-spoofing або підміну DNS), так і через компрометований проксі або відкриту точку доступу Wi-Fi [17].

Процес атаки включає кілька етапів. Спершу зловмисник отримує можливість перехоплювати трафік між двома сторонами. Потім, залежно від мети, він може просто спостерігати за передачею даних (пасивна атака) або втручатися у зміст запитів і відповідей (активна атака). У випадку незашифрованого з'єднання (HTTP) зловмисник може зчитувати всі передані логіни, паролі, сесійні токени тощо. Якщо використовується зашифроване з'єднання (HTTPS), зловмисник може застосовувати підроблені сертифікати, змушуючи клієнта "довіритися" фальшивому серверу.

У тестуванні безпеки подібні атаки можуть моделюватися легальними засобами, зокрема за допомогою інструменту Mitmproxy, що дозволяє в безпечних умовах перехоплювати HTTPS-з’єднання, аналізувати заголовки, тіла запитів та відповідей, а також перевіряти реакцію системи на змінений або пошкоджений трафік. Це дає змогу виявити вразливості, пов’язані з обробкою конфіденційної інформації, недостатньою перевіркою сертифікатів або небезпечними HTTP-редиректами.

Для оцінки стійкості програмного забезпечення до мережевих загроз важливим етапом є тестування безпеки з використанням моделювання атак типу «людина посередині» (MitM). Таке тестування дозволяє імітувати дії потенційного зловмисника в контрольованому середовищі, що дає змогу виявити вразливості в логіці взаємодії клієнта з сервером, перевірити налаштування захищеного з’єднання та проаналізувати поведінку застосунку у випадку порушення цілісності трафіку [18].

Одним з найефективніших способів симуляції MitM-атаки є використання інструменту Mitmproxy, який виступає як прозорий проксі-сервер між клієнтом і сервером. Щоб реалізувати атаку, достатньо налаштувати мережу тестового пристрою (наприклад, Android-емулятора або фізичного смартфона) на використання Mitmproxy як проксі. Додатково можна встановити спеціальний SSL-сертифікат Mitmproxy на пристрій для перехоплення HTTPS-трафіку.

У межах симуляції тестувальник може:

– перехоплювати HTTP(S)-запити і відповіді, переглядаючи їхній вміст у зручному форматі;

– модифікувати трафік у реальному часі, змінюючи значення заголовків, тіла запитів або відповіді;

– затримувати або блокувати окремі запити, симулюючи нестабільне мережеве з’єднання;

– перевіряти реакцію застосунку на підміну сертифікатів, відсутність шифрування або помилки в аутентифікації;

– виявляти передачу конфіденційної інформації у відкритому вигляді, наприклад, токенів, паролів або особистих даних [19].

Ще одним інструментом для симуляції подібних атак є Wireshark, який, хоча і не модифікує трафік, дозволяє глибоко проаналізувати мережеві пакети, побудувати граф взаємодії клієнта з сервером, виявити незашифровані передані дані або аномалії в поведінці протоколів.

У більш розширених сценаріях можливе також використання mock-серверів, які емулюють поведінку API з навмисно некоректною логікою або затримками у відповіді. Це дозволяє тестувати стійкість клієнта до непередбачуваних або небезпечних сценаріїв взаємодії.

Застосування таких методів тестування є важливим компонентом комплексної перевірки безпеки застосунку, оскільки дозволяє:

– виявляти уразливості ще на етапі розробки;

– оцінити відповідність стандартам захищеної розробки (OWASP, GDPR тощо);

– зменшити ризики компрометації даних користувача при експлуатації програмного забезпечення.

Симуляція MitM-атак у контрольованому тестовому середовищі є ефективним способом підвищення інформаційної стійкості програмного забезпечення, зокрема у мобільних та веб-застосунках, які активно взаємодіють з мережевими ресурсами.

2.4 Емуляція мобільних пристроїв та тестування у віртуальному середовищі

Для перевірки функціональності мобільних застосунків у рамках автоматизованого тестування активно застосовуються емулятори Android-пристроїв, зокрема Android Virtual Device (AVD), що є частиною Android Studio, та сторонній інструмент Genymotion. Обидва рішення дозволяють проводити тести без необхідності використання фізичних пристроїв, забезпечуючи зручність, ізоляцію середовища та гнучкість налаштувань [20].

Для запуску тестів на вбудованому емуляторі AVD достатньо створити віртуальний пристрій з необхідною версією Android та параметрами (розмір екрану, обсяг пам'яті тощо). Тестування можна здійснювати за допомогою інструментів Android Testing Framework та Espresso. На рисунку 2.7 наведено приклад простого UI-тесту на перевірку тексту у полі інтерфейсу:

```
@RunWith(AndroidJUnit4.class)
public class ExampleInstrumentedTest {

    @Rule
    public ActivityScenarioRule<MainActivity> activityRule =
            new ActivityScenarioRule<>(MainActivity.class);

    @Test
    public void checkTextDisplayed() {
        onView(withId(R.id.helloText)).check(matches(withText("Привіт, світ!")));
    }
}
```

Рисунок 2.7 – Приклад тесту на Android Studio (AVD)

Цей тест перевіряє, чи відображається у зазначеному елементі інтерфейсу очікуваний текст. Запуск можливий безпосередньо з Android Studio на AVD, і результати відображаються в інтегрованій панелі.

Genymotion також підтримує запуск автоматизованих UI-тестів, зокрема у поєднанні з Appium, який дозволяє створювати кросплатформені тести на Java, Python чи інших мовах. На рис. 2.8 зображено приклад тесту з використанням Appium на Java.

```java
public class AppiumTest {
    AndroidDriver driver;

    @Before
    public void setUp() {
        DesiredCapabilities caps = new DesiredCapabilities();
        caps.setCapability("platformName", "Android");
        caps.setCapability("deviceName", "Genymotion");
        caps.setCapability("app", "/path/to/app.apk");

        driver = new AndroidDriver(new URL("http://localhost:4723/wd/hub"), caps);
    }

    @Test
    public void testLoginButton() {
        WebElement loginBtn = driver.findElement(By.id("com.example:id/login_button"));
        loginBtn.click();
        // додаткові перевірки
    }

    @After
    public void tearDown() {
        driver.quit();
    }
}
```

Рисунок 2.8 – приклад тесту з використанням Appium на Java

Genymotion у цьому випадку виступає як віртуальний пристрій, до якого Appium підключається через локальний сервер. Такий підхід дозволяє тестувати застосунки з мінімальною залежністю від фізичних пристроїв, що особливо зручно при інтеграції в CI/CD пайплайни.

Обидва варіанти – AVD і Genymotion – підтримують запуск тестів у фоновому режимі, збереження логів, створення знімків екрана та запис відео, що є важливим для аналізу помилок після завершення тестування.

Висновок до розділу 2

Розглянуто сучасні технології та інструменти, що застосовуються для організації віртуалізованого середовища автоматизованого тестування програмного забезпечення. Проаналізовано переваги використання емуляторів мобільних пристроїв (зокрема Genymotion і AVD), які забезпечують масштабованість, ізоляцію та відтворюваність тестових сценаріїв без залучення фізичного обладнання. Розглянуто роль контейнеризації, зокрема використання Docker, для створення легких та гнучких середовищ тестування, що легко інтегруються у CI/CD-процеси.

Приділено інструментам моніторингу та аналізу мережевого трафіку, серед яких Wireshark та Mitmproxy, що дозволяють виявляти вразливості у взаємодії з API, аналізувати структуру запитів та виявляти потенційні загрози. У межах тестування безпеки описано принципи реалізації та моделювання атак типу «людина посередині» (MitM) у контрольованих умовах з метою оцінки стійкості застосунку до зовнішніх впливів.

Обґрунтовано доцільність використання обраного підходу до побудови тестового середовища, що включає поєднання Genymotion, Appium, Cucumber, Mitmproxy та Mock-server як гнучкої та багатоцільової архітектури для автоматизованого тестування мобільних та мережевих компонентів системи.

3 РЕАЛІЗАЦІЯ ТЕСТОВОЇ ПЛАТФОРМИ У ВІРТУАЛЬНОМУ СЕРЕДОВИЩІ

3.1 Проектування платформи для автоматизації тестування ПЗ

У межах автоматизованого тестування програмного забезпечення, яке реалізовано у вигляді мобільного додатку на платформі Android, було спроєктовано спеціалізоване середовище для перехоплення, аналізу та імітації мережевої активності додатку. Основною метою є моделювання поведінки серверної частини API з одночасною фіксацією параметрів клієнтських запитів та порівнянням їх із очікуваними шаблонами [21].

Архітектура тестового середовища

1. Клієнтська частина.

Роль клієнта виконує мобільний додаток, встановлений на пристрої з ОС Android. Структура запитів, які він надсилає до серверної частини, є сталою — як у форматі, так і в структурі запитань та відповідей. Це дозволяє зафіксувати стандартну поведінку додатку в контрольованому середовищі.

2. Перехоплення трафіку за допомогою mitmproxy.

Для перехоплення HTTP(S)-трафіку, що надсилається додатком, використовується утиліта mitmproxy, встановлена на робочій станції. Вона виступає у ролі посередника між додатком і сервером, дозволяючи детально проаналізувати кожен мережевий запит [21].

Кожен перехоплений запит автоматично розбивається на складові:

- Адреса (URL) запиту
- Тип запиту (GET, POST, PUT, DELETE тощо)
- Заголовки запиту (Headers)
- Тіло запиту (Payload / Data)

Усі ці елементи зберігаються у внутрішню базу для подальшого аналізу та повторного відтворення.

3. Імітація роботи API-сервера.

З метою перевірки реакції мобільного клієнта без використання реального серверного API, було створено локальний проксі-сервер, який генерує шаблонні відповіді на запити. Це дає змогу ізолювати процес тестування від зовнішніх факторів і точно контролювати умови тестування.

Проксі-сервер обробляє запити, отримані через mitmproxy, і повертає заздалегідь підготовлені відповіді згідно зі специфікацією API. У такий спосіб забезпечується повноцінна емуляція серверної логіки.

4. Порівняння очікуваної і фактичної поведінки

Після отримання відповіді додатком, платформа виконує аналіз відповідності між [22]:

- фактичним запитом, надісланим додатком;
- шаблонним запитом, що очікується згідно з технічною документацією;
- відповіддю, яка надходить від реального чи емуляційного сервера.

Це дозволяє не лише протестувати функціональність, але й виявити потенційні відхилення у логіці взаємодії з API.

Переваги розробленої платформи.

- Автоматизація рутинних перевірок – зменшується час на ручне тестування;
- Гнучкість у сценаріях – можна легко моделювати ситуації, коли сервер повертає різні типи помилок або успішних відповідей;
- Безпечне середовище для експериментів – використання ізольованої тестової мережі;
- Масштабованість – архітектура дозволяє додати підтримку нових API-ендпоінтів без значних змін у коді.

У межах розробки платформи для автоматизованого тестування програмного забезпечення було обрано мобільний додаток, який призначений для використання агентами страхової компанії. Основною функцією даного додатку є створення та редагування звітів, що стосуються страхових випадків. Усі сформовані звіти зберігаються у хмарному сховищі, що забезпечує доступ до них з будь-якого

пристрою користувача. Серверна частина програми зберігає не лише ці документи, але й супутню інформацію, таку як дані користувачів, адреси об’єктів, посилання на додаткові ресурси тощо. Для передачі даних між клієнтською частиною додатку та сервером використовується захищене з’єднання на основі протоколу HTTPS, що гарантує конфіденційність і цілісність інформації.

Інтерфейс користувача додатку реалізований у вигляді кількох функціональних екранів. Наприклад, окремі екрани передбачені для перегляду, створення або редагування звітів, управління користувацькими даними та налаштування параметрів програми. Під час виконання дій у додатку клієнтська частина ініціює відповідні HTTP-запити до серверної частини. Саме ці запити є основою для тестування та подальшого аналізу.

Для перехоплення та аналізу мережевого трафіку використовується інструмент mitmproxy, який встановлений на робочій станції. Він виступає в ролі посередника між мобільним додатком та сервером і дозволяє повністю контролювати обмін інформацією. Після того, як додаток відправляє запит до серверу, утиліта перехоплює його, розбиває на структурні частини, такі як адреса (URL), тип запиту (наприклад, GET або POST), заголовки та тіло запиту (наприклад, у форматі JSON). Ці дані зберігаються для подальшого аналізу або повторного використання. Таким чином забезпечується повна прозорість у тестуванні взаємодії між клієнтом і сервером [22-23].

Наступним кроком у процесі тестування є імітація відповіді від серверної частини. З цією метою на тій самій робочій станції створюється локальний прозорий Proxy-сервер, який виконує функцію емуляції API. Він приймає запити, що були перехоплені mitmproxy, та повертає на них заздалегідь підготовлені шаблонні відповіді. Ці відповіді моделюють поведінку справжнього серверного API, яке могло б бути реалізоване на продуктивному середовищі. Таке рішення дозволяє перевірити реакцію мобільного додатку на певні сценарії взаємодії із сервером без необхідності підключення до реального серверного ресурсу.

Під час тестування здійснюється порівняння фактичних запитів, які були сформовані клієнтом, із шаблонними, що очікуються згідно з технічною

документацією. Крім цього, аналізується відповідність отриманих відповідей тій структурі, яку передбачено специфікацією API. Це дозволяє виявити логічні помилки у формуванні запитів, некоректну обробку даних або неочікувану реакцію з боку додатку при отриманні певних типів відповідей. У разі потреби платформа може також залучати збережені відповіді від реального сервера для порівняння з результатами роботи імітованого API.

Проєктована система забезпечує безпечне, ізольоване середовище для тестування, що дозволяє уникнути небажаних впливів на реальні дані або продуктивні сервери. Крім того, вона дозволяє автоматизувати численні повторювані сценарії тестування, зменшуючи витрати часу на ручну перевірку функціональності. Такий підхід також підвищує стабільність та якість програмного забезпечення, оскільки дозволяє заздалегідь виявити потенційні проблеми у логіці взаємодії між клієнтом і сервером [24].

3.2 Використання Appium та Cucumber у тестуванні

Cucumber як інструмент для поведінкового тестування (BDD)

Cucumber — це сучасний інструмент тестування програмного забезпечення, який реалізує підхід Behaviour-Driven Development (BDD), тобто розробку, орієнтовану на поведінку системи. Основна мета цього підходу полягає у забезпеченні спільного розуміння вимог до функціональності програмного продукту між усіма учасниками проєкту: розробниками, тестувальниками, аналітиками та кінцевими користувачами.

Особливістю Cucumber є використання простої, природної мови для опису очікуваної поведінки додатку. Для цього застосовується спеціальна мова специфікацій під назвою Gherkin. Вона дозволяє створювати тестові сценарії, які легко читаються навіть тими, хто не має глибоких технічних знань. Сценарії формуються у вигляді текстових файлів з розширенням .feature, де кожен окремий сценарій описує певну функціональність програмного забезпечення — так звану фічу (feature). Ці файли зберігаються у відповідній директорії проєкту, зазвичай під

назвою features.

Усередині кожного .feature-файлу описується один або декілька сценаріїв взаємодії користувача з системою. Кожен сценарій складається з ключових слів Gherkin, таких як Given, When, Then, які визначають початкові умови, дії користувача та очікувані результати відповідно. Такий формат дозволяє точно і формалізовано описати функціональні вимоги до програмного продукту в термінах, які будуть зрозумілі як технічним спеціалістам, так і бізнес-замовникам [23-25].

Cucumber не лише дозволяє описати очікувану поведінку системи, але й автоматизовано перевірити її. Після створення .feature-файлу, наступним кроком є реалізація кроків сценарію у вигляді програмного коду (наприклад, на мовах Java, JavaScript, Ruby тощо). Таким чином, кожен текстовий опис дії у файлі сценарію відповідає певному методу в коді, який виконує реальну перевірку поведінки додатку.

На рисунку 3.2.1 представлена детальна схема створення тестових сценаріїв у середовищі Cucumber, яка ілюструє повний цикл формування автоматизованого тесту від початкового опису до фактичного виконання.

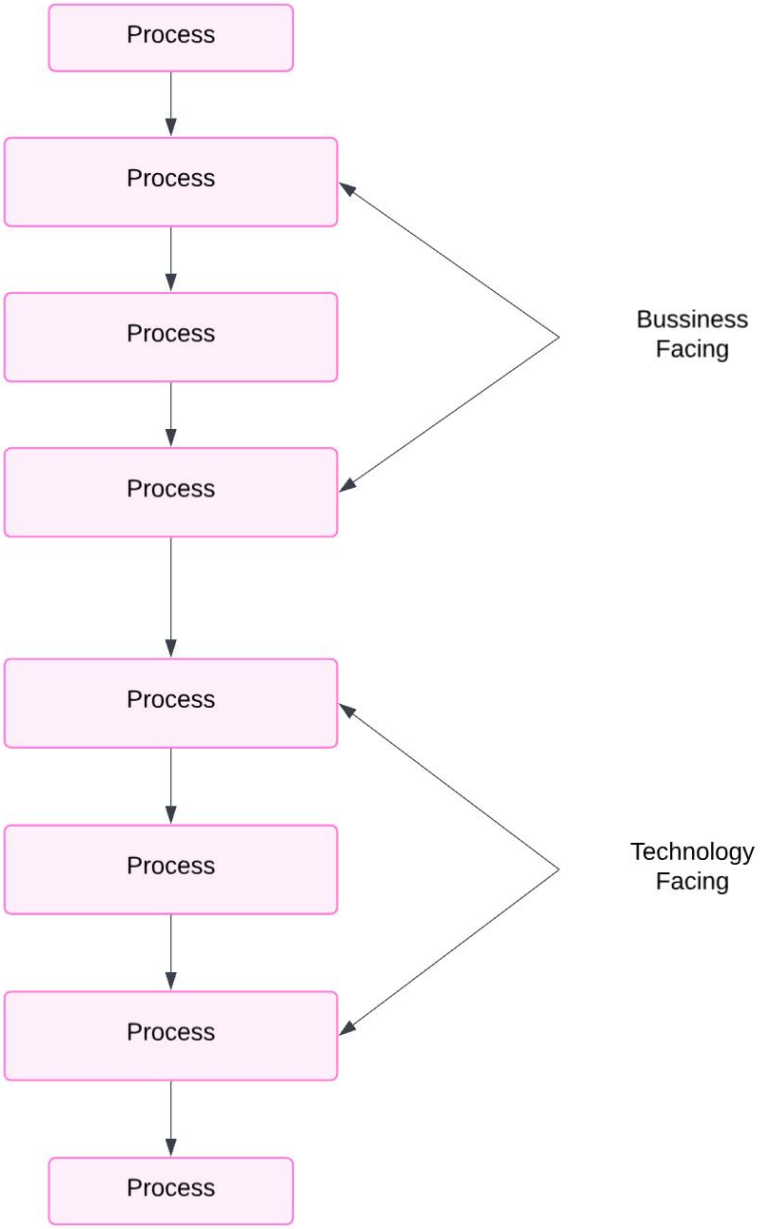


Рисунок 3.1 – Послідовність реалізації поведінкового тесту з використанням Cucumber

У середовищі Cucumber кожен тестовий випадок зазвичай складається з двох окремих складових: файлу з описом сценарію та файлу з реалізацією відповідних дій програмною мовою. Така структура дозволяє чітко розмежувати формулювання вимог до функціональності у зрозумілому для всіх текстовому вигляді (написаному простою англійською мовою) та їхню технічну реалізацію, що виконується засобами програмування. Це, своєю чергою, спрощує командну

роботу над проєктом, оскільки навіть нефахівці в області програмування можуть працювати з текстовими сценаріями, не заглиблюючись у технічні деталі коду [26].

Під час запуску системи тестування Cucumber послідовно обробляє вміст файлу .feature, який містить опис функціональних сценаріїв. Обробка починається з розділу, що позначається ключовим словом Feature, після чого слідує текстовий опис очікуваної поведінки програми. Структура цього файлу гнучка — текст можна вільно редагувати, адаптуючи до конкретних потреб, проте дотримання базового синтаксису Gherkin є обов'язковим.

Кожен описаний сценарій складається з послідовності кроків, які реалізуються у вигляді програмних функцій у файлах, що зазвичай зберігаються у директорії step_definitions. Саме у цих файлах за допомогою коду реалізуються дії, що відповідають крокам на кшталт Given, When, Then, які визначають початкові умови, події та очікувані результати відповідно. Під час виконання тестів Cucumber порівнює кроки, описані у .feature-файлі, з відповідними реалізаціями у step-файлах та викликає відповідні методи [27].

```php
namespace app\tests\unit\models;

use yii\codeception\DbTestCase;
use app\tests\fixtures\UserProfileFixture;

class UserProfileTest extends DbTestCase
{
    public function fixtures()
    {
        return [
            'profiles' => UserProfileFixture::className(),
        ];
    }

    // ...методи тестування...
}
```

Рисунок 3.2 – Структура налаштування фікстур для бази даних

У результаті формується test-suite, який описаний простою мовою, зрозумілою для всіх учасників команди — від розробників до аналітиків. Це дозволяє ефективно узгоджувати логіку поведінки додатку ще до написання коду. Нижче наведено приклад сценарію для перевірки заповнення звіту [27]:

```
Feature: Завершення заповнення звіту
Scenario: Користувач повинен увійти в систему та створити звіт
Given login a@a.aa password 123qwe
And click the Add button
And I enter the value Ann in the Insured Name field
And I enter in the Property Address field the value of the Luxembourg rose
And I enter in the City field the value of Tomsk
And I put 645750 in the Zip Code field
And select Australia from the Country drop-down list
Then click Done button
```

Рисунок 3.3 – Тестовий сценарій заповнення звіту у форматі Cucumber

Ключовим етапом реалізації такого сценарію є написання Step Definitions — частини, де кожен крок сценарію пов’язується з відповідною програмною логікою.

Саме тут відбувається автоматизація: текстовий опис дій перетворюється на виконуваний код. На відповідному рисунку показано, як відбувається взаємодія між .feature-файлом і його технічною реалізацією у вигляді step_definitions.

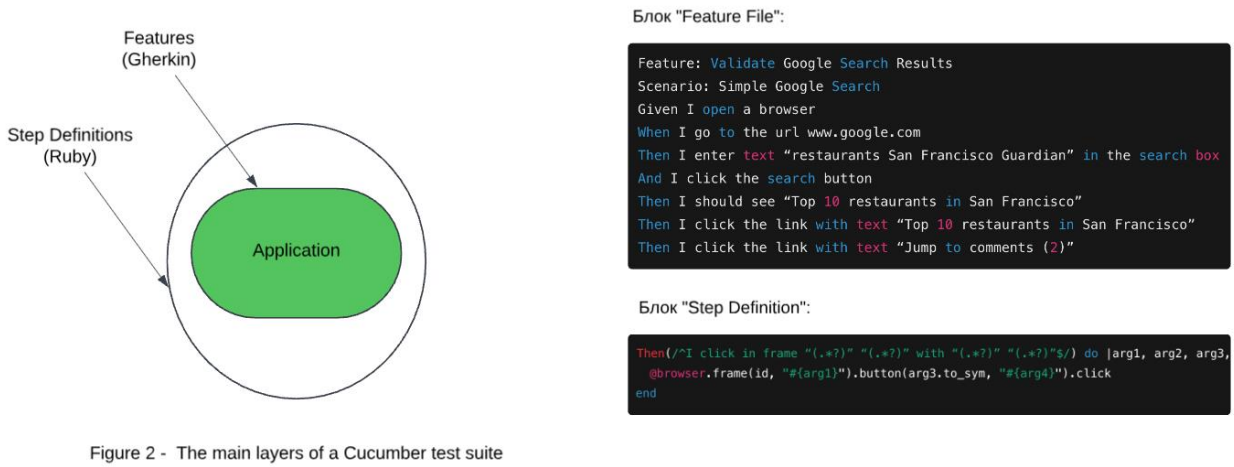


Рисунок 3.4 – Модель взаємозв’язку Feature-файлів і Step Definition

Appium як інструмент для автоматизованого тестування мобільних додатків

Appium є потужним інструментом для автоматизації тестування мобільних додатків на платформах Android та iOS. Його ключовою перевагою є те, що він дозволяє тестувати додатки без необхідності їх попередньої модифікації або вбудовування додаткових елементів. Це забезпечує збереження цілісності тестованого програмного забезпечення.

Перед запуском тестів необхідно встановити всі залежності та драйвери, які відповідають за взаємодію з конкретною платформою. Appium працює як окремий серверний процес, тому після завершення налаштувань його можна запускати незалежно від основного середовища розробки. Одразу після інсталяції можна переходити до створення та виконання тестів [28-29].

Для коректної роботи Appium необхідно вказати адресу та порт, за якими відбуватиметься комунікація між клієнтом і сервером Appium. Ці параметри задаються або у конфігураційному файлі, або безпосередньо у скрипті запуску. Крім того, потрібно вказати шлях до тестованого мобільного додатку (файл .apk для Android або .ipa для iOS), цільову платформу, версію ОС та інші параметри середовища. У разі потреби можуть додаватися зовнішні бібліотеки або утиліти для розширення функціоналу тестування.

Appium підтримує широке коло мов програмування, що дозволяє легко інтегрувати його у вже існуючі середовища автоматизованого тестування, включаючи Cucumber, Selenium, JUnit та інші.

3.3 Інтеграція Mitmproxy та Mock-server для аналізу API

Mitmproxy як інструмент для аналізу мережевої взаємодії

Mitmproxy — це потужний інструмент, який використовується для перехоплення, тестування та налагодження мережевого трафіку між клієнтом і сервером. Він дозволяє в реальному часі відстежувати всі HTTP(S)-запити та

відповіді, що надсилаються й отримуються додатком, що робить його незамінним під час аналізу мережевої взаємодії у веб- та мобільних застосунках.

У контексті даного дипломного проєкту Mitmproxy застосовується як прозорий проксі-сервер для перехоплення та дослідження трафіку між клієнтською частиною мобільного додатку та сервером. Завдяки відкритій структурі і прозорості своєї роботи, утиліта дає змогу не лише переглядати потік запитів і відповідей, а й змінювати їх для імітації різних сценаріїв. Це дозволяє перевірити, як реагує додаток на нестандартні або потенційно шкідливі відповіді від сервера.

Такий підхід сприяє підвищенню безпеки програмного забезпечення і дозволяє заздалегідь виявити потенційні вразливості.

Mitmproxy складається з кількох ключових компонентів [29]:

⇒ ***Mitmproxy*** — основна консольна утиліта, яка дозволяє в онлайн-режимі переглядати, фільтрувати та модифікувати трафік. Вона зберігає історію взаємодії додатку із сервером, відображаючи як вхідні запити, так і відповіді. Цей інструмент є зручним для інтерактивної роботи в процесі дослідження.

⇒ ***Mitmdump*** — скриптовий інтерфейс, орієнтований на автоматизацію. Він працює в командному рядку без інтерфейсу, перехоплюючи трафік та зберігаючи його у текстовий файл. Ці логи можна детально проаналізувати пізніше, що зручно для створення репортів або відтворення сценаріїв.

⇒ ***Libmproxy*** — бібліотека на мові Python, яка є основою функціоналу Mitmproxy. Саме через цю бібліотеку реалізовано гнучке керування трафіком, можливість інтеграції з іншими інструментами та написання скриптів для обробки запитів/відповідей.

Назва ***Mitmproxy*** походить від терміну *MITM* (Man-In-The-Middle), що означає "людина посередині". Цей підхід полягає у вставленні проксі-сервера між клієнтом і сервером, щоб контролювати або модифікувати трафік у двосторонньому порядку. Такий механізм детально описано в пункті 2.2 поточного розділу, де розглянуто принципи роботи MITM-атаки.

У рамках побудованої архітектури, Mitmproxy виконує роль посередника, який "прослуховує" запити клієнта та відповіді сервера, водночас маючи

можливість змінювати або підмінювати вміст. На відповідному рисунку відображено типову схему встановлення з'єднання, в якій Mitmproxy займає проміжне положення між мобільним додатком та серверною частиною, забезпечуючи повний контроль над мережею.

Mock-сервер як засіб для емуляції API-взаємодії

Для реалізації тестування взаємодії мобільного додатку з API, у даному проєкті використовується ***mock-сервер*** — окремий HTTP-сервер, що запускається на визначеному порту та імітує роботу справжнього серверного середовища. Такий підхід дозволяє точно відтворити поведінку API без залучення реального бекенду, що є особливо корисним у процесі розробки, тестування та налагодження додатку.

Mock-сервер конфігурується за допомогою спеціальних правил, які задаються у вигляді очікувань (*expectations*). Вони описують, як сервер повинен реагувати на певні типи запитів. У разі, якщо очікування не задані, сервер за замовчуванням повертає код відповіді `404 Not Found`, що вказує на відсутність обробника для конкретного запиту.

Щоб запит був коректно оброблений mock-сервером, він повинен містити певні параметри. До основних належать [30]:

- ***Path*** — абсолютний шлях до запитуваного ресурсу,
- ***Query string*** — рядок параметрів, що передається у запиті,
- ***Header*** — заголовки, які вказують тип контенту, авторизацію тощо,
- ***Cookie*** — додаткові дані, які можуть передаватись у запиті,
- ***Body*** — основний вміст запиту, що містить дані для обробки.

У відповідь mock-сервер може повертати такі ключові компоненти:

- ***Status code*** — код HTTP-відповіді, який вказує на результат обробки запиту,
- **Body** — тіло відповіді, яке містить потрібні дані у вигляді JSON чи іншого формату,
- ***Headers*** — заголовки відповіді, що описують її структуру,
- ***Cookies*** — додаткові параметри, що можуть передаватися у відповіді.

3.4 Опис взаємодії із API та тестування запитів у віртуальному середовищі

API (Application Programming Interface) використовується для взаємодії між клієнтською частиною додатку та сервером без потреби у прямій участі користувача. Його основне призначення — надання доступу до даних та функціоналу системи у форматі, зручному для автоматизованої обробки. У контексті даного проєкту API забезпечує отримання структурованої інформації у вигляді коду, що слугує базою для автоматизованого формування звітів.

Завдяки API мобільний додаток може звертатися до окремих елементів або ресурсів системи без використання графічного інтерфейсу. Вся взаємодія здійснюється «під капотом», тобто у фоновому режимі. При цьому API абстрагується від конкретної реалізації внутрішньої логіки додатку, надаючи уніфікований інтерфейс доступу до функціональності [31].

Бібліотеки, які реалізують API, містять чітко визначений опис доступних функцій — зокрема, їх сигнатур і семантики. Сигнатура функції — це формалізоване оголошення, яке включає її назву, список вхідних параметрів та тип повернутого значення. Такий підхід дозволяє розробникам чітко розуміти, які дії можна виконати через API та яку відповідь очікувати у відповідь на виклик тієї чи іншої функції.

Функціонування API базується на ієрархічному підході, який відповідає принципам багаторівневої моделі. Структура взаємодії із зовнішніми ресурсами через API реалізована за аналогією з еталонною семирівневою моделлю OSI, яка включає фізичний, канальний, мережевий, транспортний, сеансовий, представницький та прикладний рівні. Такий підхід дозволяє розмежувати функціональність за рівнями, де нижні API можуть бути використані на вищих шарах, забезпечуючи при цьому масштабованість та модульність системи.

Однією з ключових переваг API є можливість обходити традиційний користувацький інтерфейс, виконуючи запити безпосередньо до внутрішньої логіки додатку. Це значно спрощує реалізацію автоматизованих сценаріїв,

дозволяючи системі функціонувати «під капотом», що, власне, і пояснює широку популярність API в сучасній розробці та тестуванні.

API також виконує функцію контролю коректності вхідних запитів. У випадках, коли запит сформовано з помилками або не відповідає специфікації, сервер повертає відповідні коди помилок. У межах реалізованої архітектури передбачено два основні коди [32]:

— ***403001*** — означає, що помилка виникла на стороні клієнта, тобто запит сформований некоректно або порушує правила доступу;

— ***404002*** — сигналізує про те, що сервер не зміг знайти або обробити запит, незважаючи на його формальну правильність.

Серед функцій, реалізованих через API, передбачена процедура авторизації користувача. У даному випадку створення нового користувача не виконується, натомість на електронну пошту надсилається повідомлення з унікальним кодом підтвердження. Цей код генерується автоматично за допомогою сервісної логіки API, а для реалізації подібного сценарію використовується метод **POST**, що відповідає за передачу даних до сервера.

На відповідному рисунку нижче представлено приклад формування запиту авторизації користувача, який демонструє структуру HTTP-запиту та параметри, необхідні для його коректного виконання.

```
POST /api/auth/register
Content-Type: application/json

{
  "email": "john.doe@example.com",
  "password": "4v5mwef3e",
  "fullName": "John Doe",
  "facebookToken": "...",
  "company": "Left Corporation Ltd.",
  "address": "< address entity >"
}
```

Рисунок 3.5 – Формат HTTP-запиту до API при реєстрації/авторизації

Після надсилання запиту сервер формує відповідь, що містить згенерований секретний код, який надсилається користувачеві. Якщо повідомлення не отримано, користувач може повторно ініціювати запит, і система знову згенерує код. Для цього використовується той самий метод ***POST***, що й під час початкового звернення.

```
POST /api/resend-code
Content-Type: application/json

{
  "email": "john.doe@example.com" // required
}
```

Рисунок 3.6 – Формування JSON-запиту з електронною адресою користувача

Створення звіту на сервері здійснюється у два послідовні етапи. Спочатку генерується структура самої форми звіту, після чого виконується передача необхідних параметрів, які використовуються для її заповнення. Такий підхід дозволяє чітко розділити ініціалізацію об'єкта звіту та введення даних у нього.

Після успішного створення звіту сервер повертає відповідь із кодом 200 OK, що свідчить про коректне завершення операції. Для цього процесу використовується метод POST, який передбачає надсилання даних на сервер для створення нового ресурсу. Приклад відповідного запиту наведено на рисунку 3.7 [32].

```
POST /api/reports?query=monthly&limit=10&offset=0
Content-Type: application/json

{
  "reportTitle": "Monthly Financial Summary",
  "createdBy": "john.doe@example.com",
  "date": "2025-04-03",
  "sections": [
    {
      "title": "Revenue",
      "content": "Total revenue for the month was $50,000"
    },
    {
      "title": "Expenses",
      "content": "Operational expenses reached $35,000"
    }
  ]
}
```

Рисунок 3.7 – Структура POST-запиту для формування звіту

Для отримання раніше створеного звіту клієнтська частина звертається до API за допомогою методу GET. На відміну від POST-запиту, який використовується для створення ресурсу, GET-запит призначений для отримання вже наявних даних. У відповідь сервер повертає потрібну інформацію згідно з переданими параметрами. Приклад такого запиту наведено на рисунку 3.8.

```
GET /api/reports/12345
Content-Type: application/json
```

```
{
  "reportId": "12345",
  "reportTitle": "Monthly Financial Summary",
  "createdBy": "john.doe@example.com",
  "date": "2025-04-03",
  "sections": [
    {
      "title": "Revenue",
      "content": "Total revenue for the month was $50,000"
    },
    {
      "title": "Expenses",
      "content": "Operational expenses reached $35,000"
    }
  ]
}
```

Рисунок 3.8 – Структура запиту на витяг звіту через API

Щоб внести зміни до вже створеного звіту, необхідно надіслати запит із тими параметрами, які потребують оновлення. Для часткового оновлення ресурсу, зокрема звіту, використовується метод PATCH, що дозволяє змінити лише окремі поля, не передаючи весь об’єкт повністю. Приклад такого запиту наведено на рисунку 3.9 [33].

```
PATCH /api/reports/12345?query=finance&limit=5&offset=0
Content-Type: application/json

{
  "reportTitle": "Updated Financial Report",
  "date": "2025-04-03",
  "sections": [
    {
      "title": "Revenue",
      "content": "Updated revenue section content"
    }
  ]
}
```

Рисунок 3.9 – Внесення змін у звіт за допомогою PATCH-запиту

У деяких випадках виникає необхідність видалити раніше створений звіт. Для цього мобільний додаток формує запит із використанням методу DELETE, який звертається до відповідного API-ендпоінту. У результаті, якщо запит сформовано правильно, сервер виконує видалення звіту за вказаним ідентифікатором. Приклад такого запиту наведено на рисунку 3.10.

```
DELETE /api/reports/12345
Content-Type: application/json
```

Рисунок 3.10 – Формування запиту з методом DELETE

Перед обробкою основної частини запиту сервер виконує перевірку на дублікати. Якщо файл з ідентичним розміром і хеш-сумою вже існує у сховищі, сервер повертає відповідь з кодом 201 Created, що вказує на повторне використання наявного ресурсу без дублювання [33].

Для більшої інформативності відповіді API включають не лише коди стану, а

й повідомлення з описом проблеми або інструкцією для її вирішення. У разі неправильно сформованого запиту клієнт отримує не просто код помилки, а й текстове пояснення. Для кожного методу API передбачено набір специфічних кодів помилок, що полегшує діагностику.

На рисунку 3.11 зображено приклад ситуації, коли сервер повертає помилку 404 Not Found, що означає, що запитаний ресурс не знайдено.

```
HTTP/1.1 404 Not Found
Content-Type: application/json

{
  "code": 404001,
  "message": "User not found"
}
```

Рисунок 3.11 – Формат обробки помилки "Not Found" у API

Для реалізації процесу тестування API необхідно попередньо налаштувати тестове середовище, а також забезпечити коректну взаємодію між усіма компонентами API-архітектури.

Схема роботи тестового середовища

Усі дані опрацьовуються виключно після надсилання запиту з боку мобільного додатку. Паралельно із запуском самого додатку відбувається ініціалізація mock-сервера, який імітує роботу реального API.

Запуск застосунку виконується у зв’язці з інструментами Appium та Genymotion. Appium надає API для виконання автоматизованих тестових сценаріїв, які зазвичай пишуться мовами Python або Ruby. Genymotion, своєю чергою, виконує роль емулятора Android-середовища для відтворення поведінки додатку в умовах, наближених до реальних [34].

Інструмент mitmproxy використовується здебільшого для логування та

аналізу мережевої логіки. У випадках, коли додаток використовує захищене HTTPS-з’єднання, mitmproxy дозволяє обійти шифрування за рахунок генерації власних SSL-сертифікатів, що дає змогу перехоплювати та аналізувати зашифровані запити й відповіді.

Процес прогону тестів завершується або після успішного виконання всіх етапів сценарію, або у разі виникнення помилки. В обох випадках здійснюється логування результатів, що дозволяє проаналізувати поведінку додатку та виявити потенційні проблеми.

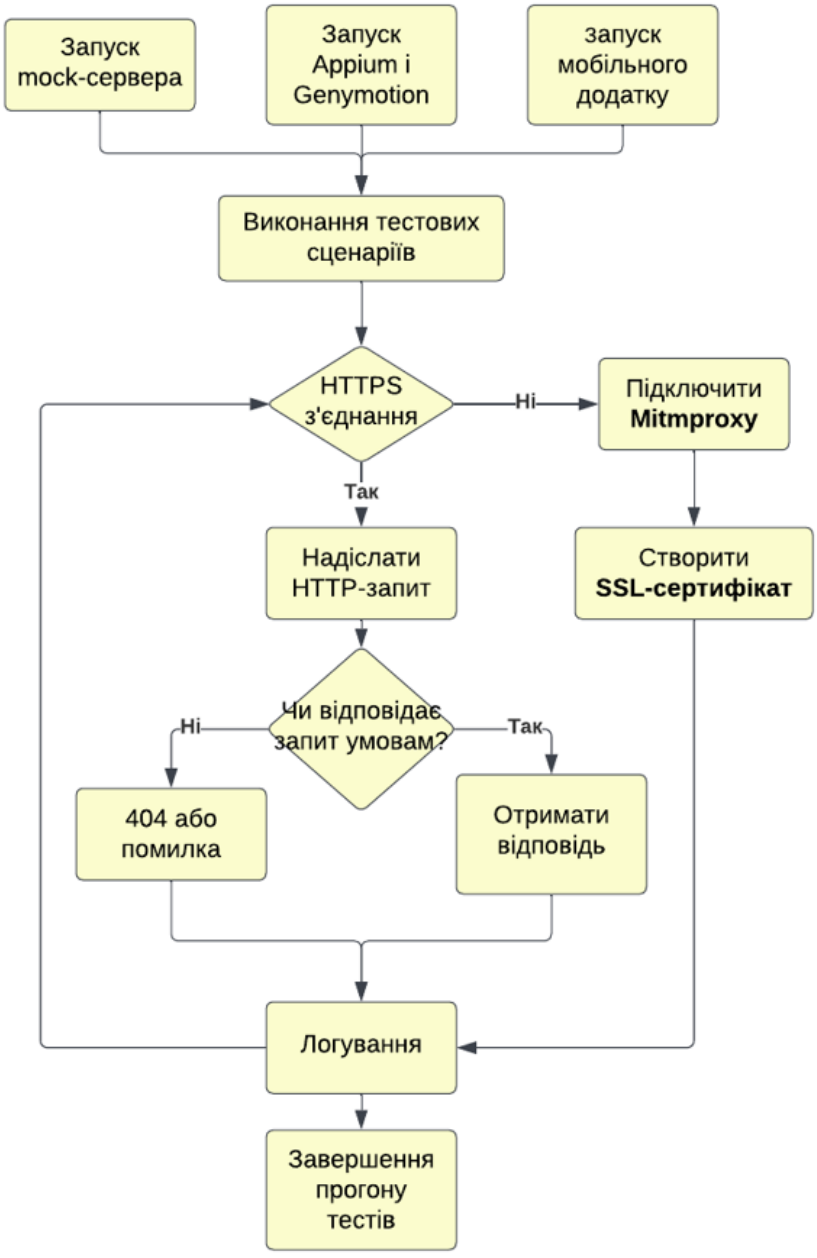


Рисунок 3.11 – Архітектура тестового середовища з використанням Appium і Mock-сервера

Висновок до розділу 3

У цьому розділі було розглянуто архітектуру тестового середовища, побудовану на основі інструментів Appium, Cucumber, Mitmproxy та Mock-server. Детально описано процес проектування платформи для автоматизованого тестування програмного забезпечення, з урахуванням вимог до взаємодії з мобільним додатком через API, а також із використанням BDD-підходу для формалізації поведінки додатку в зрозумілій формі.

В рамках дослідження було розроблено повноцінне тестове середовище з можливістю емуляції пристроїв та перехоплення мережевого трафіку. Реалізовано інструменти для автоматичного запуску тестів, налаштовано перехід запитів до імітованого сервера та перевірку коректності обробки запитів/відповідей. Проаналізовано механізм обробки HTTPS-з'єднань, включаючи використання SSL-сертифікатів у середовищі mitmproxy для перехоплення зашифрованого трафіку.

Також було виокремлено ключові етапи взаємодії з API, описано методи перевірки запитів, обробки помилок та забезпечення безпеки під час тестування. Отримані результати демонструють ефективність обраної методології та дозволяють стверджувати, що така платформа може використовуватися як гнучкий інструмент для перевірки якості мобільних додатків, орієнтованих на роботу з REST API.

4 РЕАЛІЗАЦІЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ ТЕСТУВАННЯ У ВІРТУАЛІЗОВАНОМУ СЕРЕДОВИЩІ

4.1 Створення тестових сценаріїв мовою Gherkin та їх виконання

Написання тестових сценаріїв за допомогою мови Gherkin дозволяє одночасно реалізувати дві важливі задачі: структурування проєктної документації та формування ієрархії автоматизованого тестування [5]. Такий підхід забезпечує зрозуміле представлення функціональності системи та дозволяє ефективно описати логіку перевірок.

Для реалізації процесу тестування необхідно створити відповідні сценарії, які будуть виконуватись з метою перевірки готовності програмного продукту до використання. Мова Gherkin використовує низку ключових термінів, серед яких основними є Feature та Scenario. Ключове слово Feature описує загальну мету або призначення тесту, тобто яку частину поведінки системи він перевіряє. У свою чергу, Scenario визначає конкретну послідовність дій і очікувану поведінку системи у вигляді прикладу використання [36].

Приклад використання конструкції Feature для перевірки доступу до API наведено на рисунку 4.1.

```
Feature: Log in

  In order to access the login page
  As an admin
  I should be redirected to the page with the document
```

Рисунок 4.1 – Опис можливості входу в API за допомогою Feature

Особливістю мови Gherkin є те, що в деяких випадках ключові слова можуть виділятись курсивом, що підкреслює їх роль у структурі сценарію. Водночас

використання таких ключових слів, як Feature, Scenario, Given, When, Then, є рекомендованим для зручності читання, але не завжди обов’язковим для виконання тесту інтерпретатором.

Нижче наведено приклад Scenario, який описує виконання тесту для перевірки авторизації на сайті:

```
Scenario: Login with invalid credentials

  When user is "admin" and password is "1"
  Then the response should not contain the user
```

Рисунок 4.2 – Scenario перевірки некоректного входу до API

На відміну від конструкції Feature, інші ключові слова мови Gherkin, що використовуються всередині сценарію, є обов’язковими. Зокрема, слово Scenario позначає початок опису конкретного тестового випадку у стислому вигляді. Ключове слово When застосовується для опису дії або події, яка ініціюється користувачем або програмним кодом, і яка запускає перевірку [36].

У свою чергу, Then використовується для опису очікуваної реакції системи — тобто правильної поведінки програмного продукту після виконання дії. Саме ці конструкції формують логічну послідовність перевірки в рамках BDD-підходу.

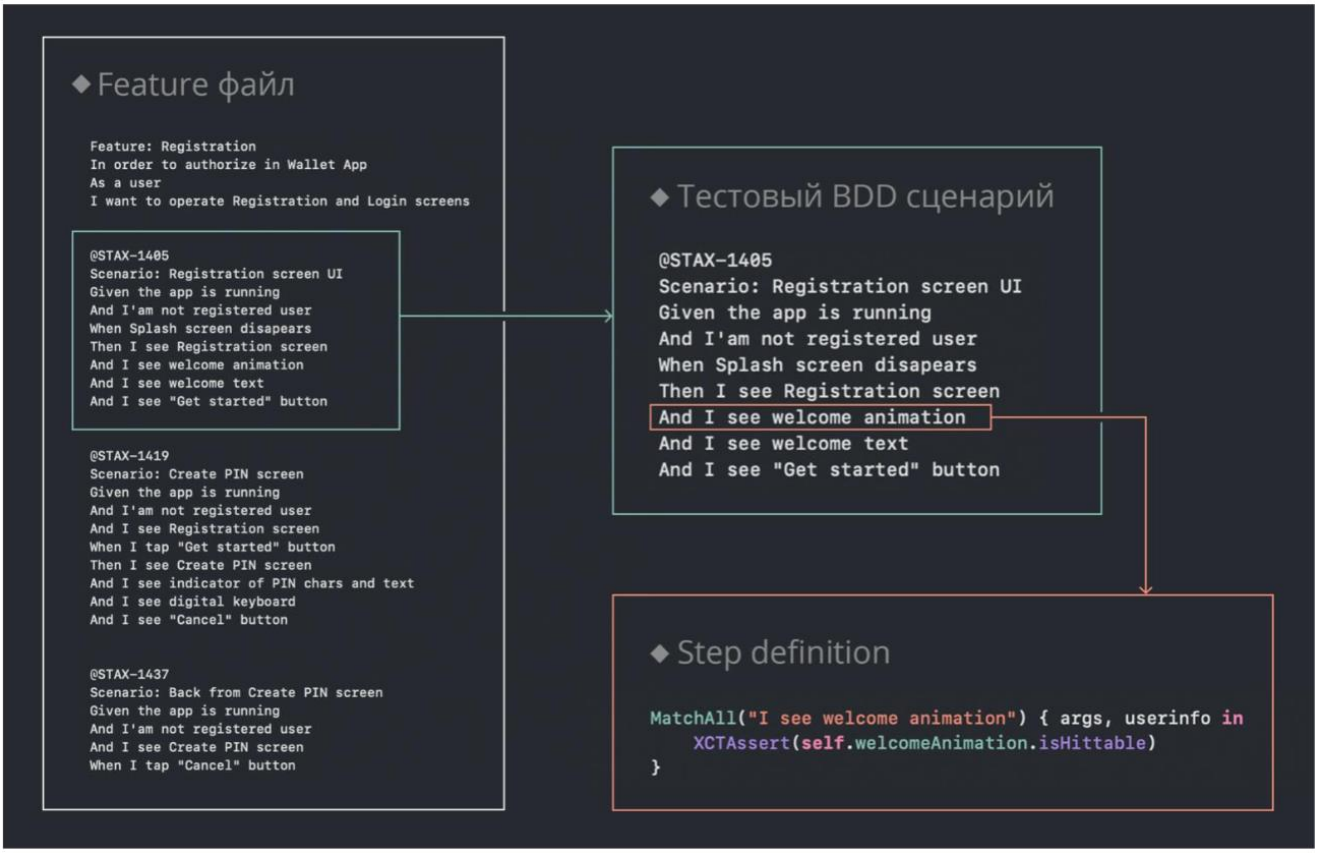


Рисунок 4.3 – Логіка обробки тестових сценаріїв Gherkin

На рисунку 4.3 зображено загальну ієрархію виконання автоматизованого тестування за допомогою мови **Gherkin** та фреймворку **Cucumber**. Дана схема демонструє послідовність основних етапів під час запуску тестових сценаріїв у рамках поведінкового підходу до тестування (BDD).

Опис діаграми:

- Фреймворк **Cucumber** ініціалізує проєкт на тестовому пристрої.
- Далі відбувається покрокове проходження сценаріїв, які описані у фіча-файлах (`*.feature`), починаючи з першого.
- Після зчитування сценарію, Cucumber звертається до кореневого файлу зі step-реалізаціями, де аналізує програмний код і здійснює запуск тестованого застосунку.

```
Given("the app is running") { (args, userInfo) -> Void in
    application.launch()
}
```

Рисунок 4.4 – Код ініціалізації застосунку в BDD-сценарії

Після проходження всіх кроків Given і виконання відповідного програмного коду, фреймворк переходить до обробки першої зустрінутої інструкції When або Then. На цьому етапі вважається, що всі попередні умови виконані, і можна переходити до наступних дій.

Далі Cucumber знову звертається до кореневого файлу, в якому зберігаються відповідні реалізації кроків (Step Definitions), та продовжує виконання сценарію відповідно до описаних дій.

Після завершення кожного сценарію тестований застосунок закривається. Далі фреймворк бере наступний сценарій із поточного фіча-файлу та повторює процес. Це триває доти, доки всі сценарії не будуть виконані [37].

4.2 Створення тестових сценаріїв та їх виконання у віртуальному середовищі

Для початку розробки кроків Steps to reproduce, у раніше створеній кореневій директорії проєкту створюються окремі файли з відповідними назвами. У кожному з них буде міститися програмний код, що реалізує конкретні дії, описані у тестових сценаріях.

Після створення файлів до них поступово додається реалізація кроків для кожного сценарію. Таким чином забезпечується зв’язок між текстовим описом сценарію (файл.feature) та його програмною реалізацією. Приклад написання такого тестового коду наведено на рисунку 4.5.

```
# Запуск Cucumber з профілем Android
$ cucumber -p android
Using the android profile...
Feature: Hi!

  Scenario: Hi from platform
    When I am greeted
    Then I see the platform name

1 scenario (1 passed)
2 steps (2 passed)
0m0.027s

# Запуск Cucumber з профілем iOS
$ cucumber -p ios
Using the ios profile...
Feature: Hi!

  Scenario: Hi from platform
    When I am greeted
    Then I see the platform name

1 scenario (1 passed)
2 steps (2 passed)
0m0.026s
```

Рисунок 4.5– Виконання тестового сценарію з використанням Step Definitions

Cucumber виконує лише ті тестові кроки, які відповідають визначеним регулярним виразам, що асоційовані з ключовими словами Given, When або Then. Це означає, що для успішного запуску тесту в сценарії обов'язково мають бути присутні ці ключові слова, а відповідні функції мають точно відповідати шаблону, який заданий у step definitions [38].

У випадку, якщо в одному тесті описано кілька кроків, які починаються однаково, Cucumber обере лише той, що повністю збігається з регулярним виразом. Таким чином забезпечується точність відповідності між описаними кроками в .feature-файлі та їх реалізацією в коді.

Після завершення виконання тестового сценарію у консоль виводиться

підсумковий результат з інформацією про успішне або неуспішне проходження кожного кроку.

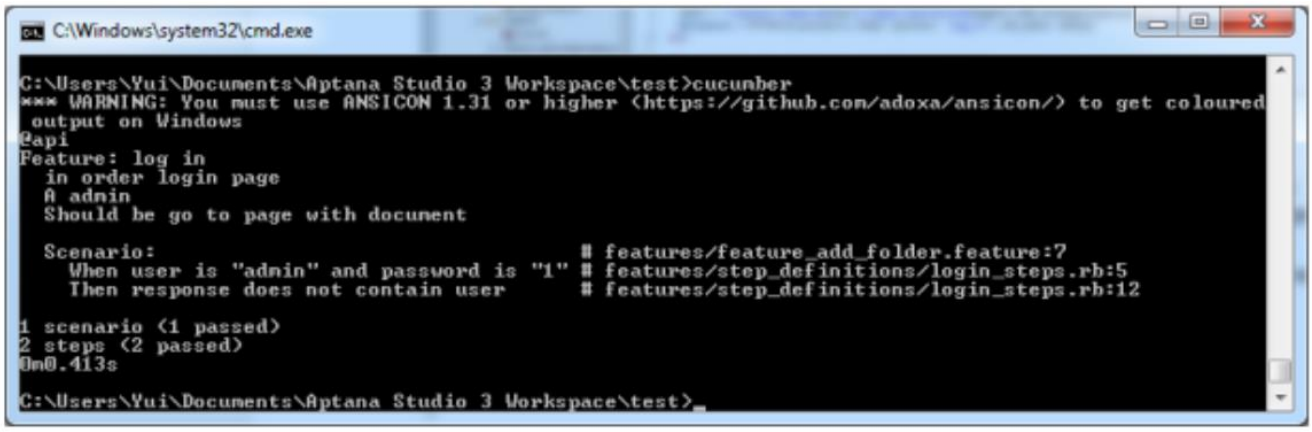


Рисунок 4.6 – Консольний лог після запуску BDD-тесту

Для запуску автоматизованих тестів за допомогою фреймворку **Cucumber** необхідно попередньо налаштувати тестове середовище, яке складається з таких основних компонентів:

– **Appium**

– **Genymotion**

– **Mock-сервер**

Appium — це кросплатформений інструмент з відкритим кодом, призначений для автоматизованого тестування мобільних додатків. Він підтримує нативні, гібридні та веб-застосунки, а також дозволяє запускати тести як на реальних пристроях, так і на емуляторах. Завдяки єдиному API, Appium забезпечує можливість написання універсальних тестів як для Android, так і для iOS-платформ.

Genymotion Desktop — це високопродуктивний емулятор Android, який дає змогу створювати віртуальні пристрої з попередньо налаштованими параметрами. Він підтримує велику кількість сенсорів, а також містить інструменти для взаємодії з додатком у процесі тестування. Genymotion доступний для операційних систем **Windows**, **macOS** та **Linux**, і є зручним рішенням для розробки, перевірки та демонстрації мобільних застосунків [39].

Запуск тестового середовища починається з ініціалізації віртуального пристрою у Genymotion. На цьому пристрої розгортається тестований мобільний додаток. Далі, у зв’язці з Appium, виконується запуск тестових сценаріїв. Усі дії автоматизовано завдяки заздалегідь підготовленим крокам сценарію.

4.3 Аналіз продуктивності тестової платформи та перехоплення трафіку

Після того як тестове середовище та мобільний додаток підготовлено до запуску, наступним кроком є активація mock-сервера, який імітує роботу реального API. Така заглушка виконує низку важливих функцій: вона приймає запити від додатку, формує відповідні відповіді та забезпечує можливість відстеження даних, які надсилаються пристроєм-емулятором під час виконання тестів [39-40].

Mock-сервер перехоплює всі запити, що надходять під час тестування, і обробляє їх відповідно до заздалегідь визначених правил. Щоб імітація API була коректною, ці правила необхідно описати в спеціальному конфігураційному файлі, зазвичай у форматі .rb (Ruby-файл).

У такому файлі вказуються ключові параметри запиту, зокрема:

— HTTP-метод (наприклад, GET, POST, PATCH, DELETE);

— шлях до ресурсу (endpoint);

— заголовки;

— тіло запиту.

Ці параметри виступають у ролі шаблонів, за якими сервер ідентифікує запити та формує відповідну відповідь. Таким чином забезпечується повноцінна емуляція взаємодії між клієнтом та сервером у контрольованому середовищі.

```
include MockServer
include MockServer::Model::DSL

def create_mock_client
  MockServer::MockServerClient.new(
    ENV.fetch("MOCK_SERVER_HOST", 'localhost'),
    ENV.fetch("MOCK_SERVER_PORT", 1080)
  )
end

let(:query_string_parameters) { [] }
let(:movies_resp_body) { '[]' }

before :each do
  @movies_server = create_mock_client
  @movies_server.reset

  @exp = expectation do |exp|
    exp.request do |request|
      request.method = 'GET'
      request.path = '/movies'
      request.headers << header('Accept', 'application/json')
      request.query_string_parameters = query_string_parameters
    end

    exp.response do |response|
      response.status_code = 200
      response.headers << header('Content-Type', 'application/json; charset=utf-8')
      response.body = body(movies_resp_body)
    end
  end

  @movies_server.register(@exp)
end
```

Рисунок 4.7 – Налаштування правила обробки GET-запиту у Mock-сервері

Для перевірки правильності надсилання запитів до імітованого сервера необхідно покрити відповідні endpoints тестовими сценаріями. Це дозволяє переконатися в коректності підключення до API та відповідності структури запиту встановленим правилам. Тестування конкретного endpoint'а гарантує, що програма взаємодіє саме з тим сервером, який передбачений у середовищі тестування, а не з реальним або зовнішнім ресурсом.

Для перевірки коректності послідовності HTTP-запитів, які надсилає мобільний додаток до mock-сервера, слід реалізувати відповідний механізм

відстеження запитів. Один із підходів полягає у використанні спеціального методу, що дозволяє зафіксувати хоча б одноразове звернення до вказаного endpoint’а. Такий механізм забезпечує підтвердження факту, що запит був успішно надісланий, отриманий і оброблений сервером-заглушкою.

Mock-сервер перебуває у стані очікування вхідних запитів. Якщо запит відповідає критеріям, визначеним у заздалегідь сформованих правилах (описаних у конфігураційних файлах), сервер формує відповідь згідно з цими правилами та, за необхідності, передає запит далі — наприклад, до реального сервера чи наступного етапу обробки [40].

Ініціалізація mock-сервера здійснюється через консоль за допомогою команди запуску. Після активації завантажуються встановлені правила обробки запитів. Сервер порівнює параметри вхідних запитів із цими правилами й аналізує їх на відповідність. У випадку, якщо параметри запиту не збігаються з очікуваними, сервер повертає помилку з HTTP-кодом 404 Not Found.

У разі використання незахищеного з'єднання між клієнтським додатком і сервером утиліта mitmproxy дозволяє встановити це з'єднання та виконати його перехоплення. При цьому немає значення, чи клієнтом виступає браузер, чи мобільний застосунок — інструмент функціонує однаково ефективно в обох випадках.

Mitmproxy здійснює аналіз трафіку в режимі реального часу, відображаючи всі перехоплені запити та відповіді у консольному вікні (відповідно до прикладу на рисунку 4.8), або зберігає інформацію у файл з розширенням .txt для подальшого аналізу. Після цього утиліта повертає клієнту згенеровану або переадресовану відповідь на запит.



Рисунок 4.8 – Перехоплення запиту та відповіді у mitmproxy

У процесі тестування може виникнути ситуація, коли з'єднання між клієнтським додатком і сервером є захищеним, тобто відбувається через протокол **HTTPS**. Для того, щоб mitmproxy міг перехоплювати та аналізувати такий трафік, необхідно створити власний SSL-сертифікат і розмістити його у директорії `~/.mitmproxy [41]`.

Після створення сертифіката, при спробі встановити з'єднання через браузер, останній розпізнає такий сертифікат як потенційно небезпечний. Для уникнення подібних попереджень, сертифікат mitmproxy необхідно додати до списку довірених сертифікатів у системі або браузері, що дозволить сприймати його як безпечний під час перевірки.

Послідовність дій при перехопленні HTTPS-з'єднання:

1. Клієнт (додаток) ініціює з'єднання з **mitmproxy**, як із проміжним сервером.
2. У відповідь **mitmproxy** імітує сервер і надсилає клієнту код **200 OK**, що підтверджує встановлення з’єднання.
3. Клієнт встановлює **SSL-з'єднання** з mitmproxy, так само, як це відбувалося б із реальним сервером. Завдяки механізму **Server Name Indication (SNI)** клієнт передає ім’я хоста, до якого хоче підключитися.
4. **Mitmproxy** встановлює окреме SSL-з’єднання з реальним сервером, використовуючи передане ім’я хоста.

5. Сервер надсилає SSL-сертифікат, який містить дані про параметри з’єднання.

6. **Mitmproxy** на основі цього сертифіката створює власний — підроблений, який буде використано для подальшої взаємодії з клієнтом.

7. Клієнт надсилає свій запит через захищене з’єднання, вважаючи, що спілкується з реальним сервером.

8. **Mitmproxy** пересилає запит на справжній сервер по окремому SSL-з’єднанню, отримує відповідь і повертає її назад клієнту.

4.4 Оптимізація процесу автоматизованого тестування та перспективи розвитку

Сучасний світ програмного забезпечення пропонує безліч рішень для реалізації автоматизованого тестування. Однак метод, представлений у даному дипломному проєкті, забезпечує скорочення часу на перевірку коректності роботи програмного продукту, а також ефективну перевірку взаємодії з API-сервером.

Запропонований підхід сприяє покращенню взаємодії між усіма учасниками проєкту [42]:

– бізнес-аналітиками, які формують вимоги,

– розробниками, що реалізують ці вимоги у вигляді програмного коду,

– тестувальниками, які перевіряють відповідність функціоналу очікуваній поведінці,

– спеціалістами з автоматизації, які спрощують і пришвидшують тестування шляхом впровадження BDD-підходу (Behavior Driven Development).

Методологія, описана у роботі, поєднує два підходи:

1. Написання тестових сценаріїв мовою Gherkin, що є легкою для сприйняття всіма членами команди (за умови базового володіння англійською мовою);

2. Реалізація логіки тестування на мові програмування Ruby, що дозволяє перехоплювати HTTP-трафік між додатком і імітованим API-с.

Використання засобів, описаних у даному дипломному проєкті, дозволяє ефективно вирішувати проблему комунікації між замовником і командою розробників. Зокрема, забезпечується якісне тестування програмного продукту навіть в умовах динамічної зміни вимог. Методологія BDD (Behavior Driven Development) фокусується на поведінці системи з точки зору кінцевого користувача, а написання сценаріїв мовою, зрозумілою усім учасникам проєкту, дає змогу узгоджувати очікувану поведінку додатку безпосередньо із замовником. Це значно скорочує витрати часу на уточнення вимог і спрощує подальшу реалізацію функціоналу.

У межах дипломного дослідження були розроблені тестові сценарії за допомогою мови Gherkin, реалізовано модуль інтеграції з використанням Ruby, а також налаштовано повноцінне середовище для виконання автоматизованого тестування. Особливу увагу приділено перехопленню трафіку між клієнтським додатком і зовнішніми API — реалізовано механізм перенаправлення запитів до mock-сервера, який емулює поведінку реального серверу та дозволяє формувати відповідні відповіді на запити.

Крім того, реалізовано автоматизований запуск усіх необхідних компонентів за допомогою скрипту у форматі .bat, що дозволяє запускати повний тестовий процес в один клік. На рисунку 4.6.1 наведено приклад запуску тестів, а також фрагмент роботи мобільного додатку у середовищі емулятора. Для тестування використано емулятор пристрою Pixel 3A з Android 10.0, який створено за допомогою Android Studio SDK [41-42].

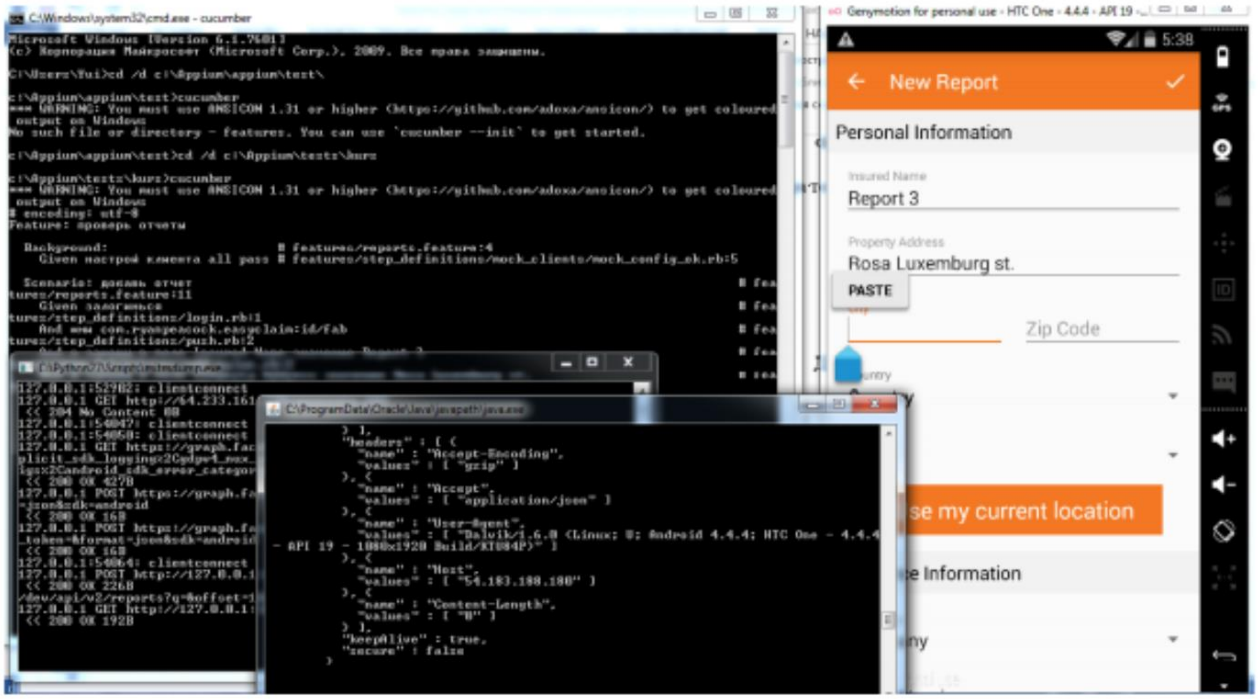


Рисунок 4.9 – Журнал запуску тестів через командний рядок

На основі вхідних даних, представлених на рисунку 4.10, було виконано запуск автоматизованих тестів. Отримані результати також зображено на цьому рисунку.

```
steps
-------------------
/I have a working Gherkin environment/ (SanitySteps.swift:17) - pass
/I use the example Insured name (?:Report 3)/ (SanitySteps.swift:38) - pass
/I use Property Address(?:Rosa Luxemburg st.) - pass
/I use City (?:Rosa) - pass
/The Zip Code is (?:355) - failed

XCTestCase+Gherkin.swift:165: error: -[XCTest_Gherkin_Tests.ExampleFeatures testBasicSteps] : failed - Step definition not found for 'The Zip Code is'
```

Рисунок 4.10 – Лог тестового запуску з частковим проходженням кроків

За допомогою певного набору тестових сценаріїв було виявлено, що значення 355 у полі Zip Code є невалідним. У результаті цього сервер відхилив запит від мобільного додатку через невідповідність правилам валідації.

У випадку, якщо всі обов’язкові поля заповнені коректно, і форму звіту створено відповідно до вимог, запуск автоматизованих тестів дозволяє переконатися в правильності виконання всіх дій. Приклад успішного проходження тестового сценарію наведено на рисунку 4.11.



Рисунок 4.11 – Демонстрація правильного виконання автоматизованого тесту

ВИСНОВКИ

У межах кваліфікаційної роботи було здійснено всебічне дослідження підходів до розробки віртуалізованої платформи для тестування програмного забезпечення у різних середовищах. Розглянуто сучасні методи організації автоматизованого тестування, зокрема з використанням технологій віртуалізації та емуляції, що забезпечують незалежність від апаратного забезпечення та операційної системи.

Проведено класифікацію й аналіз інструментів автоматизації, включаючи Appium, Cucumber, Mitmproxy та Mock-server. Обґрунтовано доцільність їх використання для побудови гнучкого середовища тестування, орієнтованого на мобільні застосунки з можливістю перехоплення, моделювання та перевірки API-запитів у контрольованих умовах.

У процесі реалізації тестових сценаріїв у віртуальному середовищі забезпечено перевірку коректності функціонування застосунку, стабільності API, відповідності форматів запитів/відповідей, а також виявлення критичних помилок на ранніх етапах життєвого циклу. Впроваджено засоби логування та аналізу трафіку, що підвищило прозорість і керованість тестового процесу.

Запропонована платформа довела свою ефективність, масштабованість і адаптивність до різних типів програмного забезпечення. Отримані результати свідчать про її придатність для інтеграції у CI/CD-процеси, що відкриває можливості для реалізації безперервного тестування у динамічних середовищах розробки.

ПЕРЕЛІК ПОСИЛАНЬ

1. Goucher, A. & Riley, M. – Beautiful Testing: Leading Professionals Reveal How They Improve Software (O'Reilly Media, 2009)

2. Fewster, M. & Graham, D. – Software Test Automation: Effective Use of Test Execution Tools (Addison-Wesley, 1999)

3. Smart, J. F. – BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle (Manning, 2014)

4. Fowler, M. – Continuous Integration: Improving Software Quality and Reducing Risk (Addison-Wesley, 2006)

5. Humble, J. & Farley, D. – Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Addison-Wesley, 2010)

6. Nguyen, H. Q., Johnson, B., Hackett, M. – Testing Applications on the Web (Wiley, 2003)

7. Meszaros, G. – xUnit Test Patterns: Refactoring Test Code (Addison-Wesley, 2007)

8. Binstock, A. – Practical Guide to Testing Wireless Smartphone Applications (Wiley, 2010)

9. Desikan, S. & Ramesh, G. – Software Testing: Principles and Practices (Pearson, 2006)

10. Arora, A. – Appium Essentials (Packt Publishing, 2015)

11. Postman Team – Postman: The Complete Guide (Postman, 2020)

12. McGraw, G. – Software Security: Building Security In (Addison-Wesley, 2006)

13. Andress, J. – The Basics of Information Security (Syngress, 2020)

14. Merkel, D. – Docker: Up & Running (O’Reilly Media, 2015)

15. Turnbull, J. – The Docker Book: Containerization is the new virtualization (James Turnbull, 2014)

16. Burns, B. et al. – Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services (O’Reilly, 2018)

17. Boettiger, C. – An Introduction to Docker for Reproducible Research (ACM Digital Library, 2015)

18. Chappell, L. – Wireshark Network Analysis (Protocol Analysis Institute, 2012)

19. Sanders, C. – Practical Packet Analysis: Using Wireshark to Solve Real-World Network Problems (No Starch Press, 2017)

20. Skoudis, E. & Liston, T. – Counter Hack Reloaded: A Step-by-Step Guide to Computer Attacks and Effective Defenses (Prentice Hall, 2006)

21. Automation of software testing – URL: https://www.sciencedirect.com/science/article/pii/S1877050915000058

22. Software that makes software better – URL: https://www.economist.com/technology-quarterly/2008/03/08/software-thatmakes-software-better

23. Shimarov V. A. Definition and quantitative estimation of testing criteria // Software Quality Concern for people. Proceedings of the Fourth European Conference on Software Quality. October 17 – 20, 1994, Basel, Switzerland. pp. 350-360.

24. Prather R., Myers J. P., Jr. The path prefix software testing strategy IEEE Transactions on Software Engineering SE-13, 7 (July, 1987), pp. 761–766.

25. Moranda P.B. Asymptotic Limits to Program Testing, INFOTECH State of Art deport «Software Testing», v/2, 1979, pp. 201-210.

26. McCabe T. J., Schulmeyer G. G. System Testing Aided by Structured Analysis (A Practical Experience) // COMPSAC’82. Proc. IEEE Comput. Soc. 6th International Computer Software and Appl. Conference (Chicago, Ill, Nov. 8-12, 1982). – pp. 523–528

27. The Art of Software Testing / Glenford J. Myers, Revised and Updated by Tom Badgett, Todd M.Thomas, Corey Sandler. – 2nd ed. – Hoboken, New Jersey: John Wiley & Sons, Inc., 2004 – 234 p.

28. Сучасні технології автоматизованого проектування та верифікації програм. Тестування програмного забезпечення. Методичні вказівки до виконання лабораторних робіт. Уклад.: Я.Ю. Дорогий, О.О. Дорога-Іванюк. – К.: НТУУ

«КПІ», 2020. – 87 с.

29. Підсумковий звіт про тестові звіти: Навчіться на прикладі та шаблоні – URL: https://uk.csstricks.net/8223011-testsummary-reports-tutorial-learn-with-example-and-template

30. Arnon Rotem-Gal-Oz. “Beautiful Architecture: Leading Thinkers Reveal the Hidden Beauty in Software Design.” – 1st ed. – Sebastopol, CA: O'Reilly Media, Inc., 2009 – 336 p.

31. Daniel Knott. “Hands-On Mobile App Testing: A Guide for Mobile Testers and Anyone Involved in the Mobile App Business.” – 1st ed. – Shelter Island, NY: Manning Publications, 2015 – 320 p.

32. Сучасні технології автоматизованого проектування і верифікації програм: Конспект лекцій – URL: навч. посіб. для студ. спеціальності 121 «Інженерія програмного забезпечення», спеціалізації «Інженерія програмного забезпечення комп’ютерних систем» / КПІ ім. Ігоря Сікорського; уклад.: Я. Ю. Дорогий, О. О. Дорога-Іванюк. – Електронні текстові дані (1 файл: 3,9 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2020. – 89 с.

33. Narayanan Palani. “Mobile Test Automation with Appium.” – Birmingham, UK: Packt Publishing, 2015 – 150 p.

34. Paul Hammant. “Trunk-Based Development.” – 1st ed. – San Francisco, CA: O'Reilly Media, Inc., 2020 – 200 p.

35. Seb Rose, Matt Wynne, Aslak Hellesøy. “The Cucumber Book: Behaviour-Driven Development for Testers and Developers.” – 2nd ed. – Raleigh, NC: Pragmatic Bookshelf, 2021 – 260 p.

36. Jason Myers, Hobson Lane, et al. “Natural Language Processing in Action.” – 1st ed. – Shelter Island, NY: Manning Publications, 2019 – 500 p.

37. Stack Overflow – URL: http :// stackoverflow. com/

38. GitHub – Iamtodor/Diabetes: App for people, who have diabetes diseases – URL: https://github.com/Iamtodor/Diabetes

39. Ronnie Mitra, Irakli Nadareishvili. “Microservices: Up and Running.” – 1st ed. – Sebastopol, CA: O’Reilly Media, 2020 – 250 p.

40. Michael T. Nygard. “Release It!: Design and Deploy Production-Ready Software.” – 2nd ed. – Raleigh, NC: Pragmatic Bookshelf, 2018 – 378 p.

41. Gregory T. Brown. “API Design for C++.” – Boston, MA: Addison-Wesley, 2011 – 352 p.

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДОДАТОК А

```
virtualized-test-platform/
├── README.md
├── docker-compose.yml
├── appium-server/
│   └── Dockerfile
├── cucumber-tests/
│   ├── features/
│   │   └── basic.feature
│   └── step_definitions/
│       └── steps.js
├── mock-server/
│   ├── mock_server.rb
│   └── expectations/
│       └── example_expectation.rb
├── mitmproxy/
│   ├── mitm_script.py
│   └── docker-compose.override.yml
├── screenshots/
└── reports/
```

cucumber-tests/features/basic.feature

```
Feature: Example test
  Scenario: Filling the form
    Given the app is running
    And I use the example Insured name "Report 3"
    And I use Property Address "Rosa Luxemburg st."
    And I use City "Rosa"
    And The Zip Code is "355"
```

```
    And The Country is "Luxemburg"
    And The age should be "32"
    Then I should see a valid submission
```

cucumber-tests/step_definitions/steps.js

```
const { Given, Then } = require('@cucumber/cucumber');
const assert = require('assert');
Given('the app is running', async function () {
  await this.driver.launchApp();
});
Given('I use the example Insured name {string}', async function (name) {
  const input = await this.driver.$('~InsuredName');
  await input.setValue(name);
});
Given('I use Property Address {string}', async function (address) {
  const input = await this.driver.$('~PropertyAddress');
  await input.setValue(address);
});
Given('I use City {string}', async function (city) {
  const input = await this.driver.$('~City');
  await input.setValue(city);
});
Given('The Zip Code is {string}', async function (zip) {
  const input = await this.driver.$('~ZipCode');
  await input.setValue(zip);
});
Then('I should see a valid submission', async function () {
  const button = await this.driver.$('~Submit');
  await button.click();
  const result = await this.driver.$('~SuccessMessage');
  assert(await result.isDisplayed());
```

```
});
```

mock-server/mock_server.rb

```
require 'mock_server'
include MockServer
include MockServer::Model::DSL
def create_mock_client
  MockServer::MockServerClient.new(
    ENV.fetch("MOCK_SERVER_HOST", 'localhost'),
    ENV.fetch("MOCK_SERVER_PORT", 1080)
  )
end
RSpec.configure do |config|
  config.before(:each) do
    @movies_server = create_mock_client
    @movies_server.reset
    exp = expectation do |exp|
      exp.request do |req|
        req.method = 'GET'
        req.path = '/movies'
        req.headers << header('Accept', 'application/json')
      end
      exp.response do |res|
        res.status_code = 200
        res.headers << header('Content-Type', 'application/json; charset=utf-8')
        res.body = '[{"id":1,"name":"Test Movie"}]'
      end
    end
    @movies_server.register(exp)
  end
end
```

mitmproxy/mitm_script.py

```
def response(flow):
    if "api.example.com" in flow.request.pretty_url:
        flow.response.headers["X-Debug"] = "Intercepted"
        flow.response.content = flow.response.content.replace(b"Original", b"Modified")
```

appium-server/Dockerfile

```
FROM appium/appium:latest
EXPOSE 4723
```

docker-compose.yml

```
version: '3.8'
services:
  appium:
    build: ./appium-server
    ports:
      - "4723:4723"
  mockserver:
    image: jamesdbloom/mockserver
    ports:
      - "1080:1080"
  mitmproxy:
    image: mitmproxy/mitmproxy
    command: mitmweb --mode transparent
    ports:
      - "8080:8080"
    volumes:
      - ./mitmproxy:/home/mitmproxy/.mitmproxy
```