Міністерство освіти і науки України
Кам’янець-Подільський національний університет імені Івана Огієнка
Фізико-математичний факультет
Кафедра комп’ютерних наук

Дипломна робота
магістра

з теми: «**ДОСЛІДЖЕННЯ ДОПОМІЖНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ТЕСТУВАННЯ ВЕБ-ДОДАТКІВ»**

Виконав: студент 2 курсу, групи KN1-B22
спеціальності 122 Комп’ютерні науки
Чорний Роман Анатолійович

Керівник: **Моцик Р.В.**,
кандидат педагогічних наук, доцент,
доцент кафедри комп’ютерних наук

Рецензент: **Оптасюк С.В.,**
кандидат фізико-математичних наук,
доцент, завідувач кафедри фізики

Кам’янець-Подільський – 2023

ЗМІСТ

ВСТУП

Комп’ютерні технології все глибше проникають у наше повсякденне життя. Програмне забезпечення здійснює управління роботою безлічі приладів навколо нас – від мобільних телефонів і персональних комп’ютерів (ПК), до побутових приладів, кредитних карток і автомобілів. У будь-якому випадку, всі ми зустрічалися з тими чи іншими помилками у роботі програм: текстовий редактор намертво «завис» під час роботи з документом, банкомат «з'їв» картку або сайт ніяк не завантажиться – все це, аж ніяк не полегшує нам життя. Саме тому, у зв’язку з «діджиталізацією» усіх сфер людської діяльності та стрімким розвитком IT- ринку, на передній план виступають питання кібербезпеки, захисту від цифрових загроз та надійності програмних продуктів.

Усі програмні продукти, що виробляються людиною можуть містити помилки. Однак не всі помилки однаково безпечні – для різних програмних систем, рівні ризику можуть відрізнятися. Деякі з них можуть бути незначними, в той час як інші мати руйнівні наслідки. Саме тому, будь-який програмний продукт потребує перевірки – тестування, перш ніж його можна буде ефективно і безпечно використовувати. Те ж саме справедливо і для програмного забезпечення. Всебічна перевірка працездатності програмного забезпечення, виявлення недоліків під час розробки, впровадження, функціонування та підтримки, забезпечується комплексом заходів його тестування. Тестування, проведене на всіх етапах розробки, дозволяє значно поліпшити якість, надійність і продуктивність системи. Під час перевірки, команда тестувальників пересвідчується в тому, що програмний продукт належним чином виконує всі задокументовані функції і, одночасно, не робить того, що не повинен. Для забезпечення високої якості кінцевого IT-продукту, критично важливим є включення тестування в життєвий цикл розробки програмного забезпечення. Особливе значення має впровадження тестування саме на ранніх стадіях роботи над проєктом, оскільки такий підхід дозволяє значно знизити витрати на усунення виявлених помилок. Будь-які IT-рішення,

стосовно яких не застосовуються перевірочні заходи, можуть обернутися суттєвими економічними та іміджевими втратами. Значні обсяги об’єктів перевірки обумовлюють широке використання допоміжного забезпечення. Але, для підвищення якості вихідного програмного забезпечення, зазвичай недостатньо використання одного методу тестування. Саме тому нагально постає потреба не тільки у перевірці програмного продукту, а й у розвитку методів його тестування. А саме, у побудові цілої системи тестування програмного забезпечення, з обов’язковим використанням допоміжних інструментів – програм, додатків, розширень веб-тестування або тестування веб-сайту – це техніка тестування програмного забезпечення, яка допомагає забезпечити функціональність і якість програми відповідно до вимог. Перед релізом веб-додатка, веб-тестування має виявити всі основні проблеми, включаючи функціональні невідповідності, порушення безпеки, проблеми з інтеграцією, виклики навколишнього середовища або навантаження на трафік.

Актуальність теми дослідження. Засоби тестування програмного забезпечення можуть допомогти фахівцям переконатися, що програмне забезпечення на стадії розробки відповідає очікуванням проєкту. Тестування програмного забезпечення також може підтвердити якість і безпеку продукту. Водночас, для початківця, який зацікавлений в тому, щоб стати розробником або тестувальником програмного забезпечення, буде корисно дізнатися про інструменти, які можна використовувати для полегшення процесу забезпечення якості.

Оскільки тестування, якість продукту є одними з основних критеріїв для успішної реалізації продукту, то знання і досвід у використанні засобів тестування відіграє велику роль у працевлаштуванні, та рівні кваліфікації працівника.

Об’єкт дослідження: Об’єктом дослідження є інструменти тестування веб-додатків.

Предмет дослідження: Предметом дослідження є аналіз і розробка допоміжного програмного забезпечення для тестування веб-додатків.

Мета дослідження. Аналіз особливостей допоміжного програмного забезпечення для тестування веб-додатків. Розробка покрокової інструкції розробки валідатора HTML-документа на мові програмування Python.

Для досягнення поставленої мети необхідно вирішити такі **завдання**:

- Проаналізувати дослідження з теми (наукові видання, програмні продукти);
- Пошук та оцінку нових інструментів для тестування веб-додатків, які можуть бути більш ефективними або забезпечувати більші можливості порівняно з існуючими рішеннями.
- Удосконалення процесу тестування веб-додатків, включаючи автоматизацію, оптимізацію тестових сценаріїв та зменшення витрат часу і ресурсів.
- Створення та опис детальної покрокової інструкції із розробки валідатора HTML-документа на мові програмування Python, та розробка на основі цього валідатора HTML-документа;

Методи дослідження. Методологічна основа дослідження допоміжного програмного забезпечення для тестування веб-додатків базується на різних підходах і методах дослідження: дослідження базується на аналізі даних про використання програмного забезпечення для тестування в реальних умовах. Це дозволяє визначити, як програмне забезпечення використовується в практиці, і як його можна вдосконалити; дослідження конкретних випадків використання програмного забезпечення для тестування веб-додатків. Цей підхід дозволяє вивчити досвід і результати відповідних проектів; дослідження, спрямовані на розробку та впровадження нових методів та інструментів для покращення процесу тестування веб-додатків.

Інформаційною базою дослідження є ресурси дослідження допоміжного програмного забезпечення для тестування веб-додатків включають в себе різноманітні види інформації та дані, що стосуються тестування веб-додатків та інструментів для цього тестування: огляд літератури; документація і технічні специфікації програмного забезпечення;

відгуки інших користувачів та результати анкетування і опитувань; демонстраційні матеріали.

Практичне значення. Оскільки тестування, якість програмного продукту є одними з основних критеріїв для успішної реалізації продукту, то знання і досвід у використанні засобів тестування відіграє велику роль у рівні кваліфікації працівника, а відповідно і в працевлаштуванні. Для початківця, який зацікавлений в тому, щоб стати розробником або тестувальником програмного забезпечення, буде корисно знати і вміти використовувати інструменти, які полегшують процес забезпечення якості продукту.

Апробація результатів. За темою дослідження опубліковано статтю «Засоби тестування веб-додатків як запит та вимога сьогодення» у Збірнику наукових праць студентів та магістрантів Кам'янець-Подільського національного університету імені Івана Огієнка та подано до друку тези «Роль і місце допоміжного програмного забезпечення у тестуванні Веб-додатків» до Збірника наукових праць студентів та магістрантів К-ПНУ. Фізико-математичні науки. Випуск 13 .

Структура роботи. Дипломна робота складається із вступу, трьох розділів (2 теоретичних і 1 практичного), висновків, 19 рисунків, 2 таблиць, одного додатку, списку використаних джерел, обсяг роботи становить 67 сторінок.

РОЗДІЛ І. АНАЛІЗ ВИДІВ ТЕСТУВАННЯ ВЕБ-ДОДАТКІВ

1.1 Основні визначення і терміни

У цьому розділі наведемо основні визначення, відповідно до стандартів, які використовуються як в наукових, так і в прикладних сферах стосовно програмного забезпечення (далі – ПЗ) та його якості.

QA (Quality Assurance) – Забезпечення якості, це процеси і методи, які забезпечують якість продукту чи послуги.

Тестування (Testing) – Процес перевірки програмного продукту на відповідність специфікаціям та виявлення помилок.

Тест-кейс (Test Case) – Документ, який містить опис конкретного тесту, його кроки та очікуваний результат.

Автоматизоване тестування (Automated Testing) – Використання програм для виконання тестів замість ручного виконання.

Юніт-тестування (Unit Testing) – Тестування окремих компонентів або функцій програми для перевірки їх роботи.

Інтеграційне тестування (Integration Testing) – Тестування взаємодії між компонентами програми.

Функціональне тестування (Functional Testing) – Перевірка, чи виконує програма задані функції.

Навантажувальне тестування (Load Testing) – Визначення, як програма працює під великим навантаженням.

Стрес-тестування (Stress Testing) – Визначення меж програми, під якими вона перестає працювати стабільно.

Помилка (Bug) – Виявлена проблема чи невідповідність програми вимогам.

Дефект (Defect) – Помилка або проблема, виявлена під час тестування.

Трасування дефекту (Defect Tracking) – Процес відстеження та управління дефектами в програмі.

Регресійне тестування (Regression Testing) – Повторне тестування програми після внесення змін для переконання, що вони не вплинули на існуючий функціонал.

Тестування користувацького інтерфейсу (UI Testing) – Перевірка користувацького інтерфейсу програми на відповідність дизайну та взаємодію з користувачем.

Чорний ящик (Black Box Testing) – Тестування, де тестер не має інформації про внутрішню структуру програми і тестує її як самостійний блок.

Білий ящик (White Box Testing) – Тестування, де тестер має доступ до внутрішньої структури програми і використовує цю інформацію для тестування.

Модульне тестування (Module Testing) – Тестування окремих модулів або компонентів програми.

Артефакти тестування (Testing Artifacts) – Усі документи і матеріали, що створюються під час процесу тестування, такі як тест-кейси, звіти, журнали тощо.

Збій (Malfunction) – прояв несправності, зазвичай в роботі устаткування.

Відмова (Failure) – порушення нормального функціонування системи, повна або часткова втрата працездатності системи (або підсистеми).

Якість (Quality) – ступінь відповідності системи, компоненту або процесу заданим вимогам, потребам або очікуванням користувача. З метою визначення добротності системи, компоненту або процесу використовують так звані атрибути якості – характеристики, що відображають дану властивість

Перевірка на валідність (Validation) – процес, що дозволяє визначити, наскільки точно з позицій потенційного користувача деяка модель представляє задану суть реального миру.

Верифікація (Verification) – процес, який дозволяє визначити, що розроблене програмне забезпечення точно реалізує концептуальний опис даної системи. Або, як ще кажуть, процес перевірки відповідності системи заданими стандартами. Верифікація успішна, якщо отримані дані збігаються з

очікуваними, заздалегідь визначеними як правильні. Зазначимо, що вона може бути неформальною, тобто тестер визначає успішність на основі своїх знань [1, 2, 3].

Методи забезпечення якості є техніками, що гарантують досягнення певних показників якості при їх застосуванні.

Методи контролю якості дозволяють переконатися, що певні характеристики якості ПЗ досягнуті. Самі по собі вони не можуть допомогти їх досягненню, вони лише дають змогу визначити, чи вдалося отримати в результаті те, що хотілося, чи ні, а також знайти помилки, дефекти і відхилення від вимог.

1.2. Види тестування.

Вид тестування, згідно з даними ISTQB (International Software Testing Qualifications Board) – це засіб чіткого визначення мети конкретного рівня для програми або проєкту [5].

У процесі розробки ПЗ тестування ПЗ зазвичай відбувається на декількох рівнях інтеграції: поблочне тестування, перевірка взаємодії (інтеграційне тестування) та системне тестування. Відповідно до етапів розробки ПЗ прийнято виділяти три фази тестування: модульне, інтеграційне і системне, також виділяють функціональне та нефункціональне тестування.

Вид тестування сфокусований на конкретну мету тестування, яка може бути перевіркою функції, що виконується компонентом або системою в цілому. Мета тестування може бути спрямована на перевірку елементів нефункціонального тестування (надійність, зручність використання), структури, архітектури компонентів або системи в цілому, а також на елементи, в залежності від змін в системі, наприклад, перевірка виправлення конкретного дефекту (підтверджувальне або повторне тестування) або перевірка випадкових змін (регресійне тестування).

Модульне тестування, тестування модуля, або автономне тестування (module testing, unit testing) – контроль окремого програмного модуля,

зазвичай в ізольованому середовищі (тобто ізольовано від решти всіх модулів). Під модулем розуміється логічно замкнутий фрагмент програми, який може бути викликаний через його інтерфейс. Модуль перевіряється на відповідність своїм специфікаціям і внутрішню логіку.

Інтеграційне тестування або тестування взаємодій (integration testing) - контроль взаємодії між частинами системи (модулями, компонентами, підсистемами). Системне тестування або комплексне тестування (system testing) – контроль та/або випробування всього програмного забезпечення, як повної системи, в цільовому середовищі, тобто підтвердження того, що доступ до всіх компонентів системи і взаємодія з ними несуперечливі і передбачені згідно специфікацій системи.

Вочевидь, що фази не є взаємозамінними і, наприклад, проведення модульного тестування не гарантує правильності інтеграційного тестування, бо правильність функціонування окремих компонент не гарантує правильності їх взаємодії, як між собою, так і з системою в цілому.

Функціональне тестування – один із видів тестування, спрямованого на перевірку відповідностей функціональних вимог ПЗ його реальним характеристикам. Основним завданням функціонального тестування є підтвердження того, що програмний продукт, який розробляється, володіє усім необхідним замовнику функціоналом.

Елементи функціонального тестування:

- підготовка тестових даних виходячи з описаної документації;
- бізнес-вимоги, як частина функціонального тестування;
- отримання результатів на основі специфікації;
- проходження тест-кейсів;
- аналіз фактичних та очікуваних результатів.

Функціональне тестування може бути проведено відповідно до специфікації, а також і на основі бізнес-процесу, тобто відповідно до знань системи.

Переваги функціонального тестування:

- в рамках тестування ми «копіюємо» безпосереднє використання системи;
- тестування, як правило, проводиться в умовах близьких до реальних.

Недоліки:

- існує ймовірність пропустити кілька помилок логіки програмного забезпечення під час перевірки функціоналу програми.

Нефункціональне тестування направлено на перевірку тих аспектів ПЗ, які можуть бути описані в документації, але не відносяться до функцій програмних продуктів.

Нефункціональне тестування – це вид тестування програмного забезпечення, який спрямований на перевірку аспектів, які не стосуються безпосередньо функціональності програми, але важливі для забезпечення її якості та ефективності. Цей тип тестування оцінює властивості, такі як продуктивність, надійність, безпека, витрати ресурсів і інші характеристики, які визначаються нефункціональними вимогами.

1.2.1. Функціональне тестування.

Основна мета функціонального тестування полягає в тому, щоб переконатися, що програма виконує свої функції відповідно до вимог та специфікацій, на основі яких вона була розроблена.

Під час функціонального тестування перевіряються різні аспекти функціональності програмного забезпечення, включаючи:

- Тестується функціональність програми, як програма виконує свої функції та чи відповідає вона специфікаціям і вимогам.
- Перевірка вхідні та вихідних даних, коректності обробки вхідних даних і виводу результатів.
- Оцінка зручності інтерфейсу користувача, якість відображення інформації, інтуїтивність для користувачів. В цей аспект ми включаємо перевірку макету, та верстки веб-додатка.

- Перевірка того, чи відповідає програма встановленим вимогам та специфікаціям проекту.
- Взаємодія інтегрованих компонентів: Тестування взаємодії різних компонентів, модулів чи систем, що утворюють програму.
- Перевірка того, як програма управляє різними робочими процесами, включаючи потоки даних та логіку додатку.
- Тестування того, як програма реагує на непередбачені ситуації та помилки, включаючи коректну обробку винятків. Написання позитивних та негативних тест-кейсів.
- Перевірка заходів безпеки програми, включаючи захист від вразливостей, таких як атаки на внесення SQL-запитів, перехоплення даних тощо.
- Тестування сумісності програми з різними операційними системами, браузерами та іншими залежними програмами. Наприклад перевірка роботи веб-додатка на різних браузерах, різних операційних системах та девайсах [12].

Розглянемо приклад застосування функціонального тестування:

Специфікація: Програма призначена для додавання двох цілих чисел. Кожен з доданків – не більш, ніж двозначне ціле число. Програма запитує у користувача два числа, після чого виводить результат. Виділимо класи еквівалентності (Табл. 1.1.) [2].

Таблиця 1.1. Класи еквівалентності

	Класи коректних даних	Класи некоректних даних	Граничні й спеціальні значення
Перший доданок	від -99 до -10 від -9 до -10 від 1 до 9 від 10 до 99	> 99 < -99 1.5 “Q”	0, 1, -1, 9, -9 10, -10 99, -99 100, -100

Другий доданок	-‖-‖-	-‖-‖-	-‖-‖-
Сума	від -198 до -100 від -99 до -1 0 від 1 до 99 Від 100 до 198	> 198 < -198 “некоректні дані”	(-99, -99) (-49, -51) (99, -99) (49, 51)

Можна виділити наступні допоміжні програмні забезпечення для тестування веб-додатків під час функціонального тестування, які можуть спростити, покращити та пришвидшити процес тестування:

- Розширення браузерів, які стануть у нагоді під час функцінального тестування (PixelPerfect (Рис.1.1), Bug magnet, Fake filler, Google Lighthouse, Web Developer, Eye Dropper);

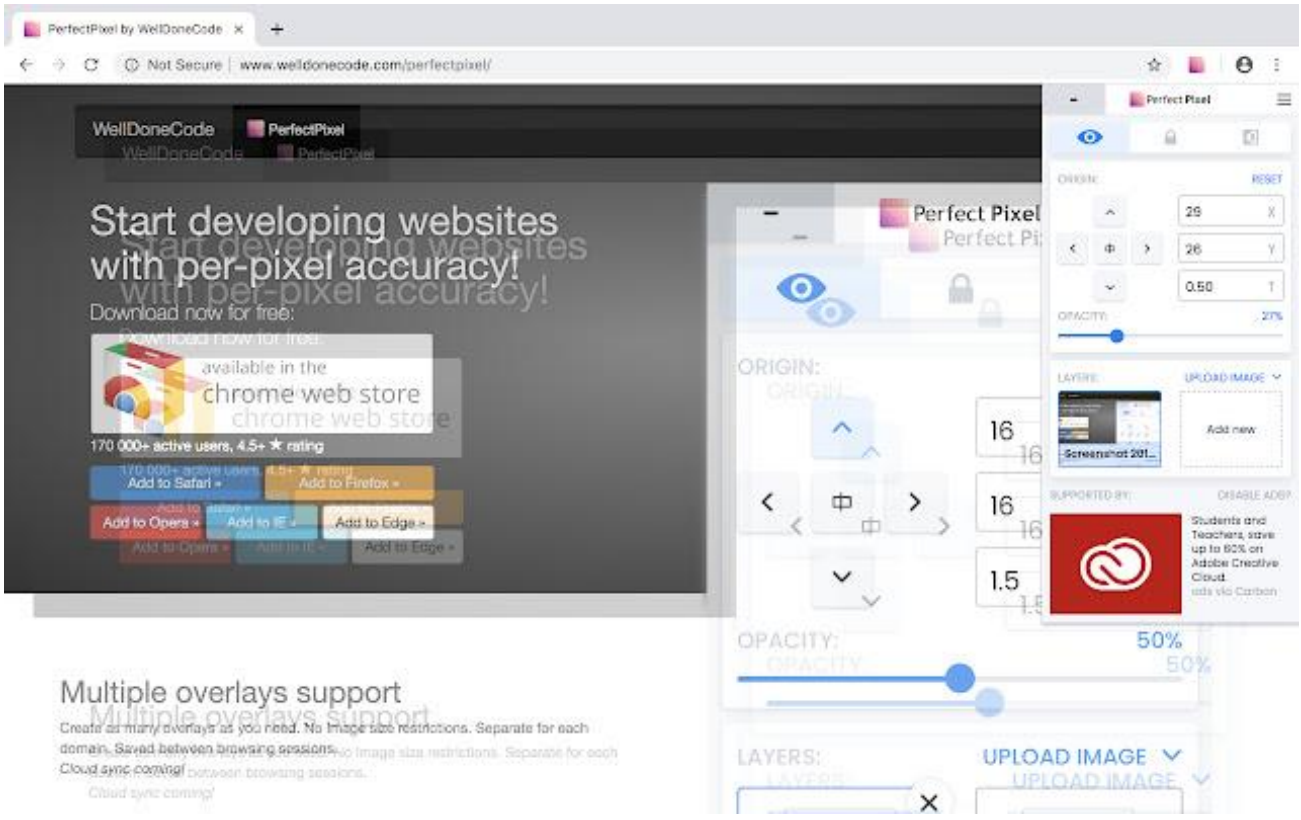


Рис. 1.1. Вигляд застосування розширення PixelPerfect

- Програмне забезпечення для фіксування багів, шляхом запису відео чи знімоків екрану (Lightshot, Joxi, Monosnap, ShareX (Рис. 1.2.);

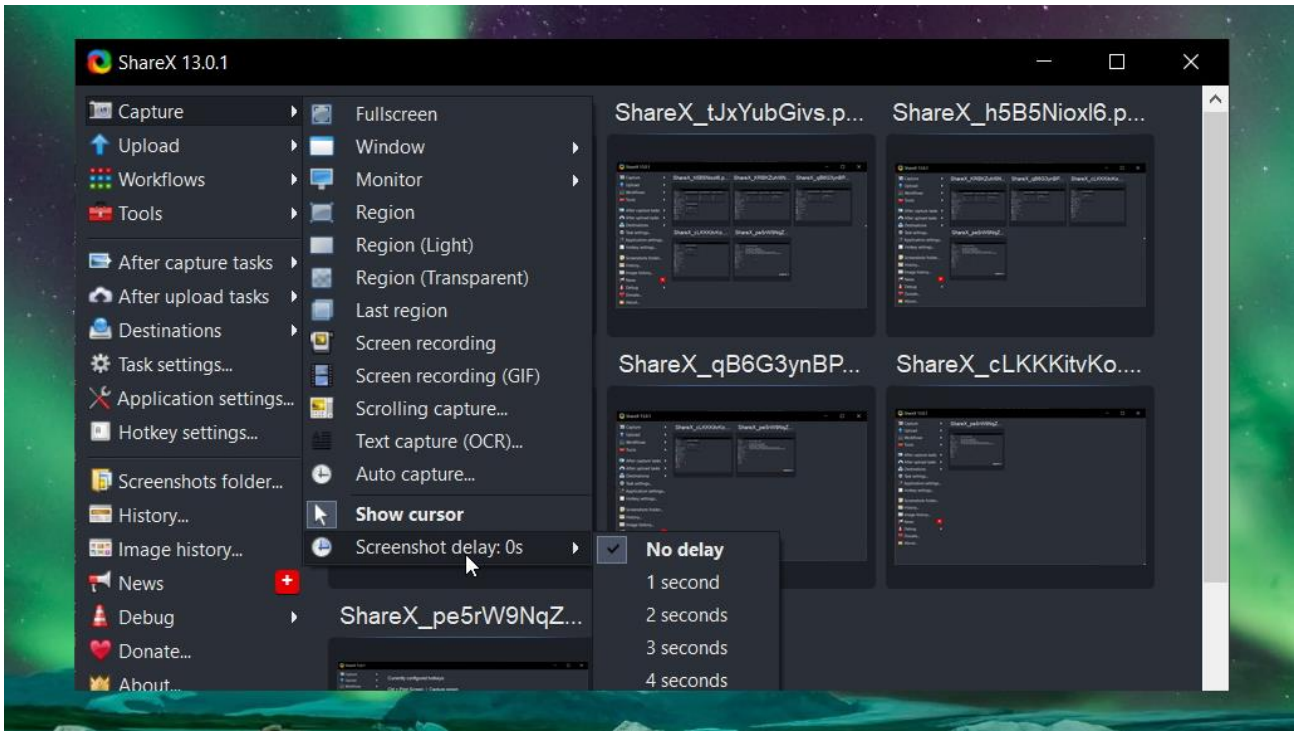


Рис. 1.2. Вигляд інтерфейсу програми для скріншотів ShareX

- Компаратори, для порівняння зображень, текстових даних (Diffchecker, Beyond compare (Рис. 1.3.);

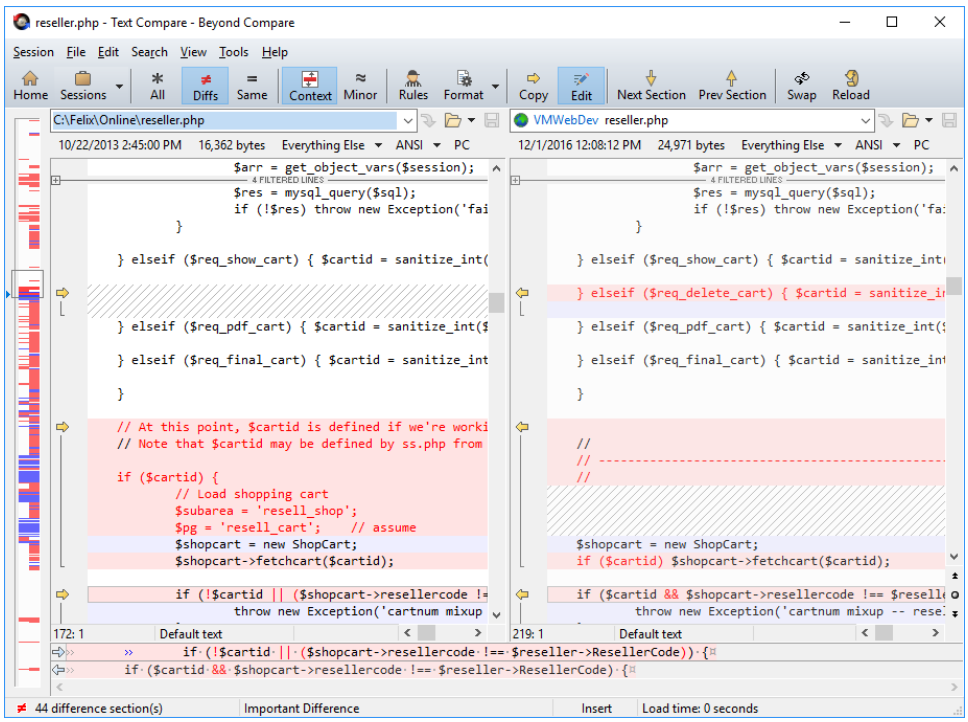


Рис. 1.3. Вигляд інтерфейсу компаратора Beyond Compare

- Програмне забезпечення для тестування API веб-додатків (Postman, Jmeter (Рис. 1.4.), TestMace);

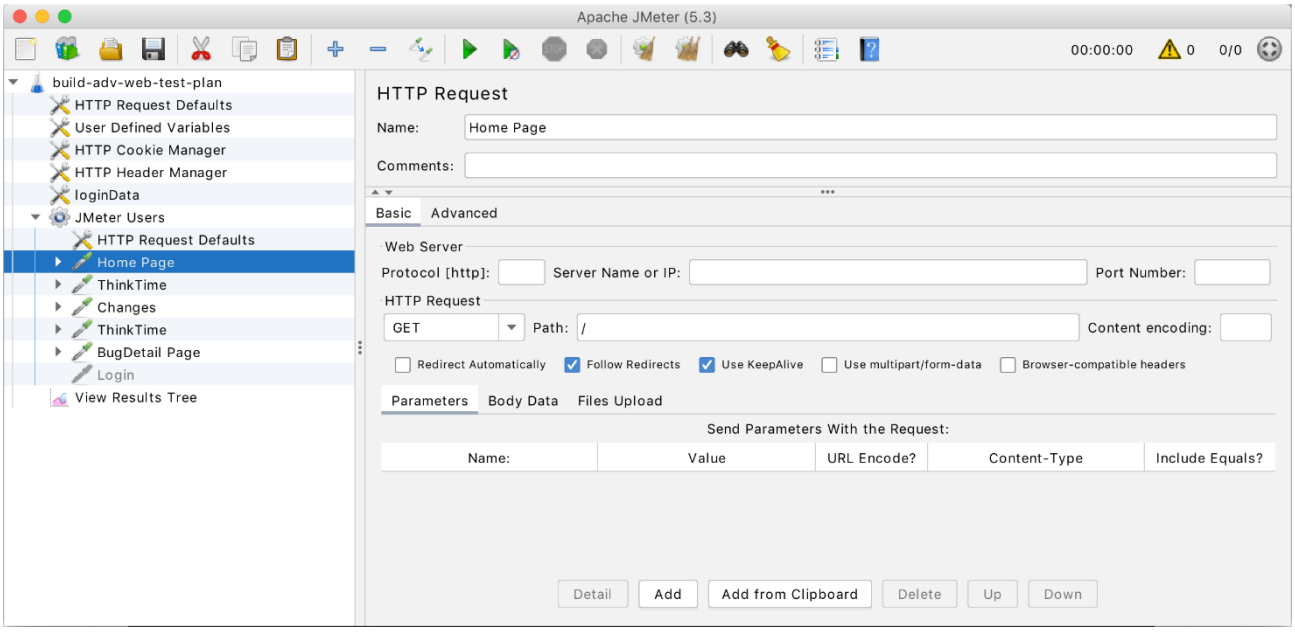


Рис. 1.4. Інтерфейс програми Jmeter, для тестування API веб-додатків

- Програмне забезпечення для автоматизації тестування веб додатків (Preflight, TestRail (Рис. 1.5.).

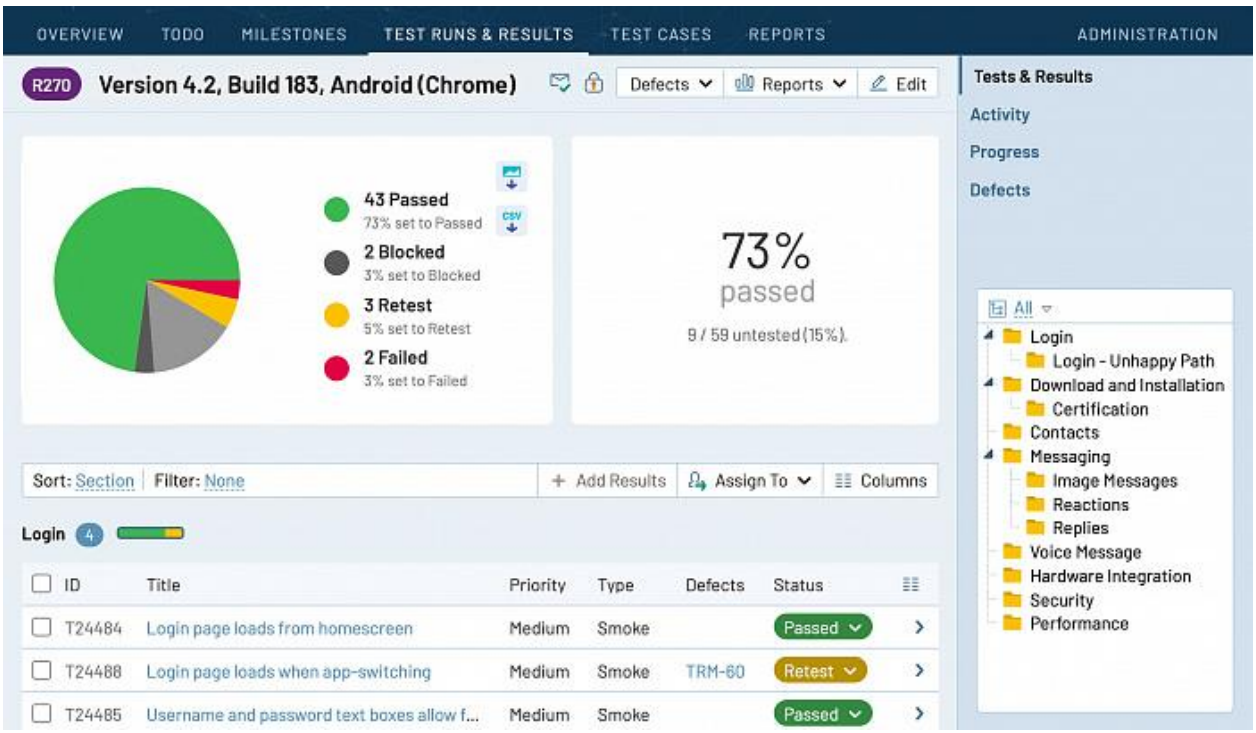


Рис. 1.5. Інтерфейс програми TestRail, для автоматизації тестування веб додатків

Перелічені засоби тестування веб-додатків можуть допомогти фахівцям переконатися, що програмне забезпечення на стадії функціонального тестування відповідає очікуванням проекту.

1.2.2. Нефункціональне тестування веб-додатків.

Нефункціональне тестування веб-додатків включає в себе перевірку аспектів, які не стосуються безпосередньо функціональності додатка, але впливають на його якість, ефективність та здатність працювати в різних умовах. Ось деякі типи нефункціонального тестування для веб-додатків:

- Тестування продуктивності (Performance Testing):
    - Тестування завантаження (Load Testing): Перевірка реакції веб-додатка на велику кількість користувачів або запитів.
    - Стерс тест (Stress Testing): Оцінка того, як додаток веде себе при перевищенні максимального навантаження.
    - Тестування ефективності (Efficiency Testing): Вимірювання ресурсів, які використовуються додатком (CPU, пам'ять, мережа) під час роботи.
- Тестування безпеки (Security Testing):
    - Тестування на вразливості (Vulnerability Testing): Пошук вразливостей, таких як SQL-ін'єкція, перехоплення сесій, ін'єкція коду тощо.
    - Тестування автентифікації та авторизації (Authentication and Authorization Testing): Перевірка заходів безпеки, пов'язаних з автентифікацією та визначенням прав доступу користувачів.
- Тестування сумісності (Compatibility Testing):
    - Тестування браузерів (Browser Compatibility Testing): Перевірка працездатності додатка в різних веб-браузерах.
    - Тестування операційних систем (OS Compatibility Testing): Перевірка сумісності з різними операційними системами.

- Тестування доступності (Accessibility Testing) - тестування доступності для користувачів з обмеженими можливостями. (Disability Testing) - перевірка того, як додаток відповідає вимогам щодо доступності для всіх користувачів.
- Тестування навантаження (Scalability Testing): Оцінка того, наскільки додаток може масштабуватися і працювати в умовах зі зростаючим обсягом користувачів чи даних.
- Тестування відновлення після аварій (Recovery Testing): Перевірка можливості відновлення роботи веб-додатка після виникнення непередбачених ситуацій чи збоїв.
- Тестування швидкості завантаження (Page Load Testing): Вимірювання часу, необхідного для завантаження сторінок веб-додатка.
- Тестування стабільності (Reliability Testing): Перевірка стабільності та надійності додатка під час тривалого використання.
- Тестування відгуку (Usability Testing): Оцінка користувацького досвіду, зокрема швидкості та комфорту взаємодії з веб-додатком [13].

Можна виділити наступні допоміжні програмні забезпечення для тестування веб-додатків під час функціонального тестування, які можуть спростити, покращити та пришвидшити процес тестування:

- Веб-сервіси для валідації, та перевірки оптимізації веб-додатків (validator.w3.org, PageSpeed (Рис. 1.6.).

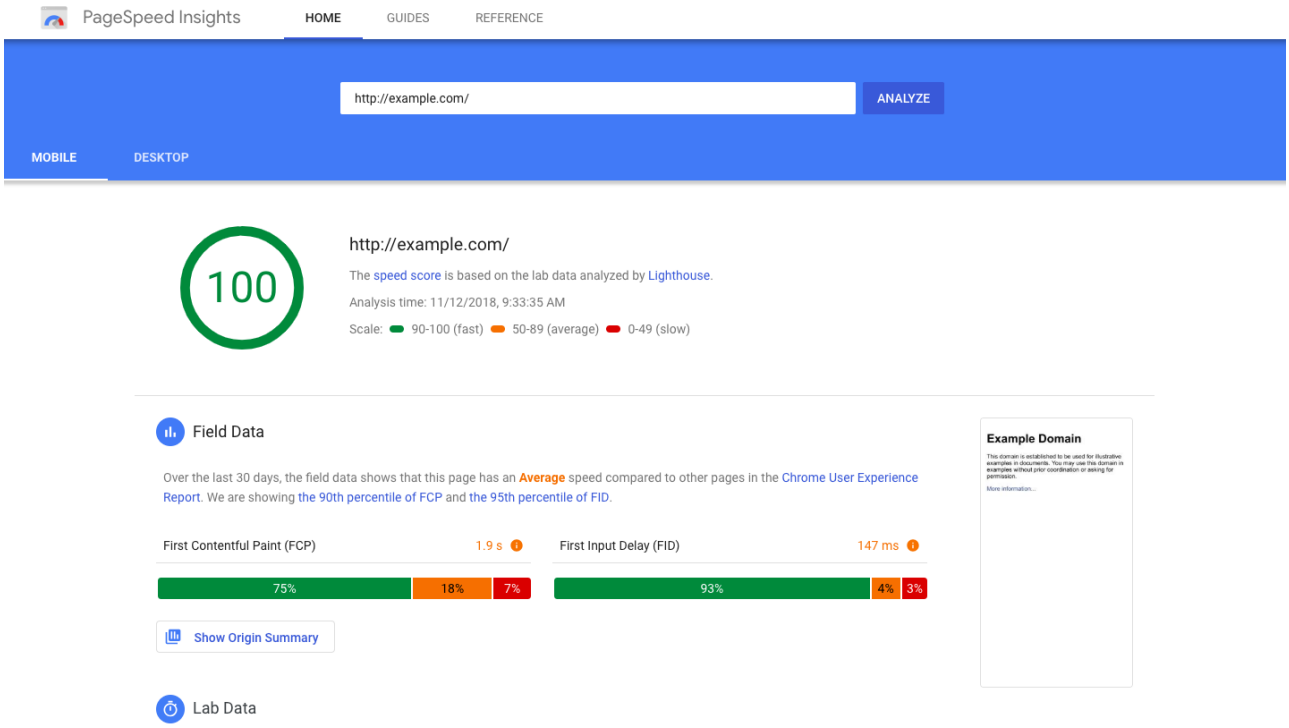


Рис. 1.6. Інтерфейс веб-сервісу для аналізу оптимізації веб-додатків

- Програмне забезпечення для тестування кросбраузерності веб-додатків (Lambda (Рис. 1.7.), Android Studio, Browser stack, Sauce labs)
-

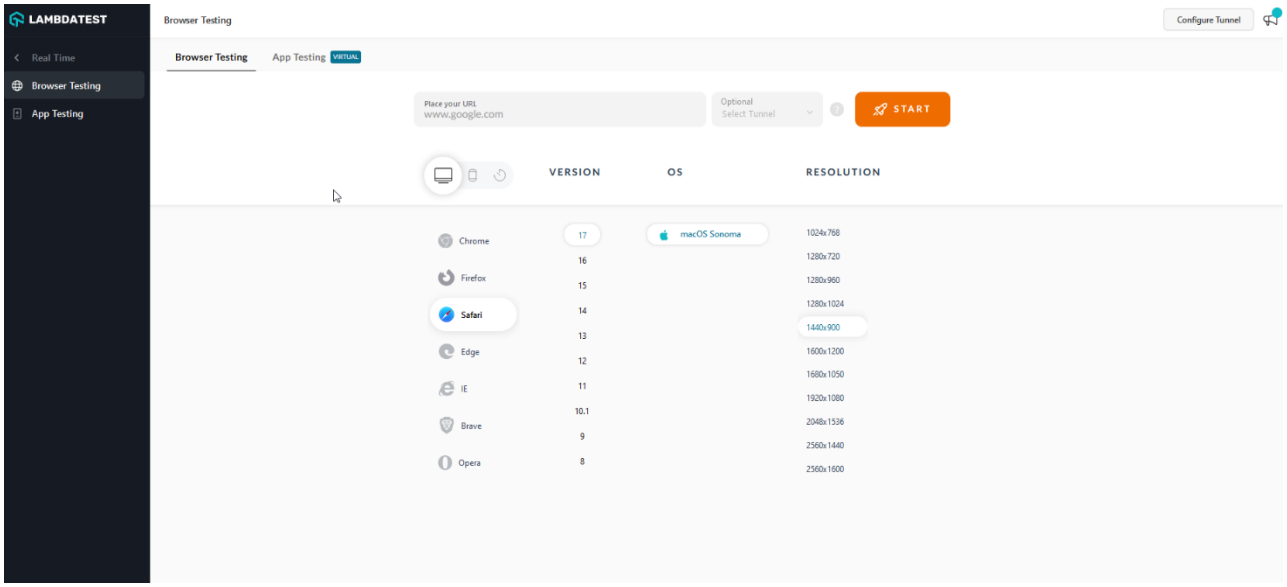


Рис. 1.7. Інтерфейс веб-сервісу для тестування кросбраузерності веб-додатків

- Програми для стрес тестів, тестування навантаження, тестування стабільності та часу відновлення після аварії (Postman (Рис. 1.8.), Jmeter, TestMace)

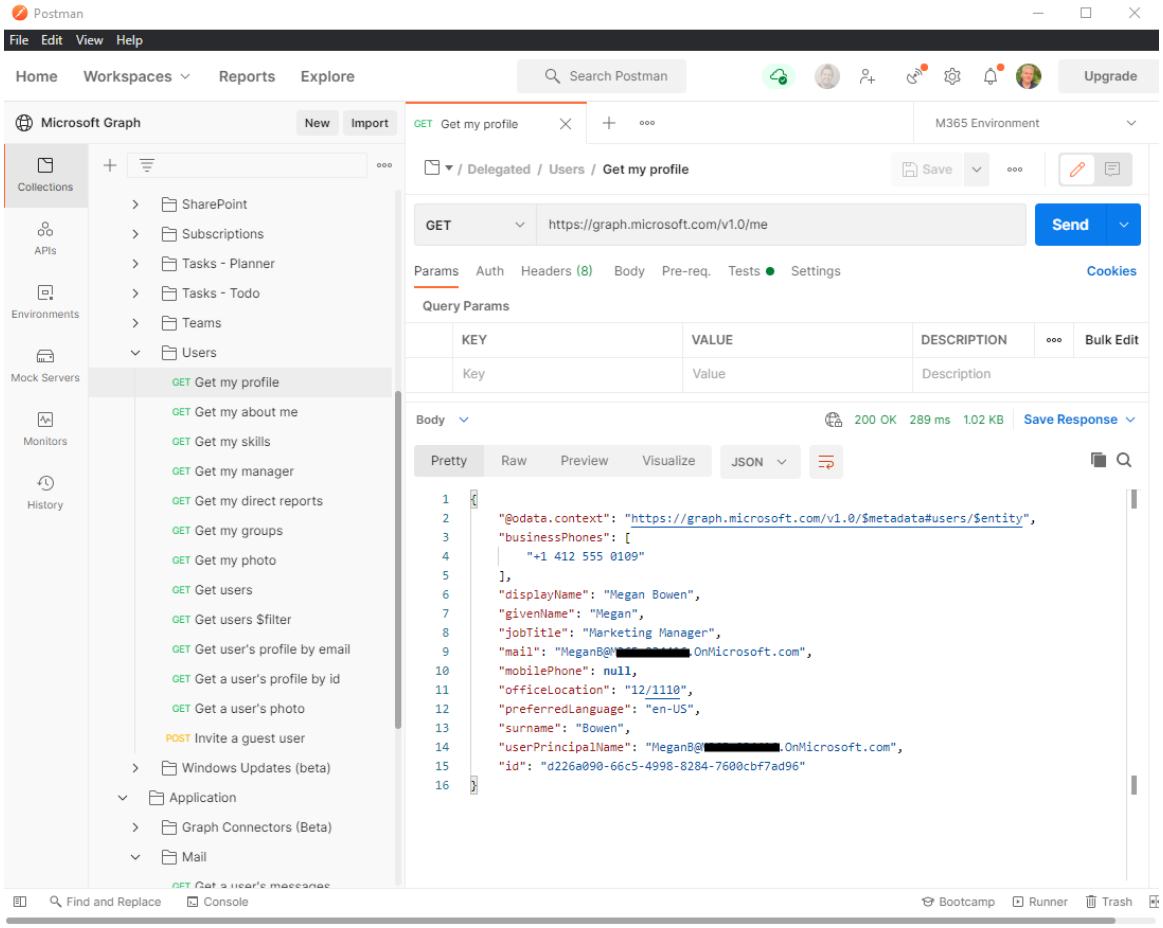


Рис. 1.8. Інтерфейс програми Postman, для стрес тестів веб-додатків.

- Проксі-сервери для перевірки і зміни трафіку (Fiddler (Рис. 1.9), Charles)

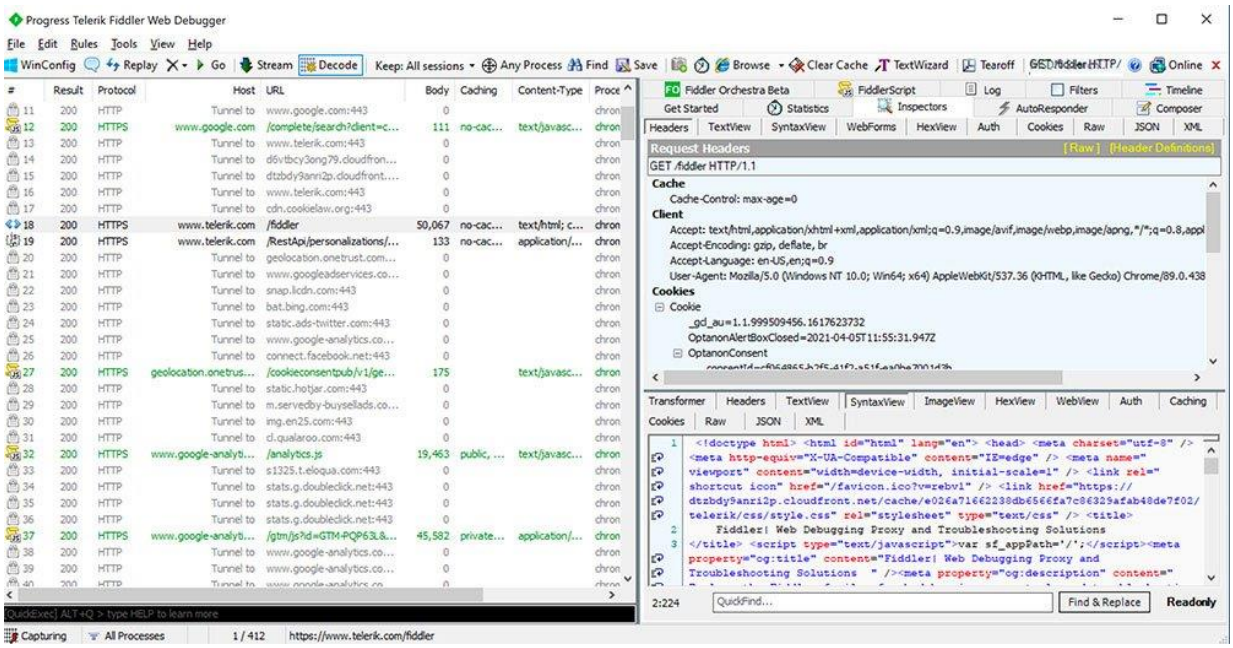


Рис.1.9. Інтерфейс програми Fiddler, для перевірки зміни трафіку

- Для тестування безпеки (Burp Suite (Рис. 1.10), OWASP ZAP, OpenVAS, WebScarab)

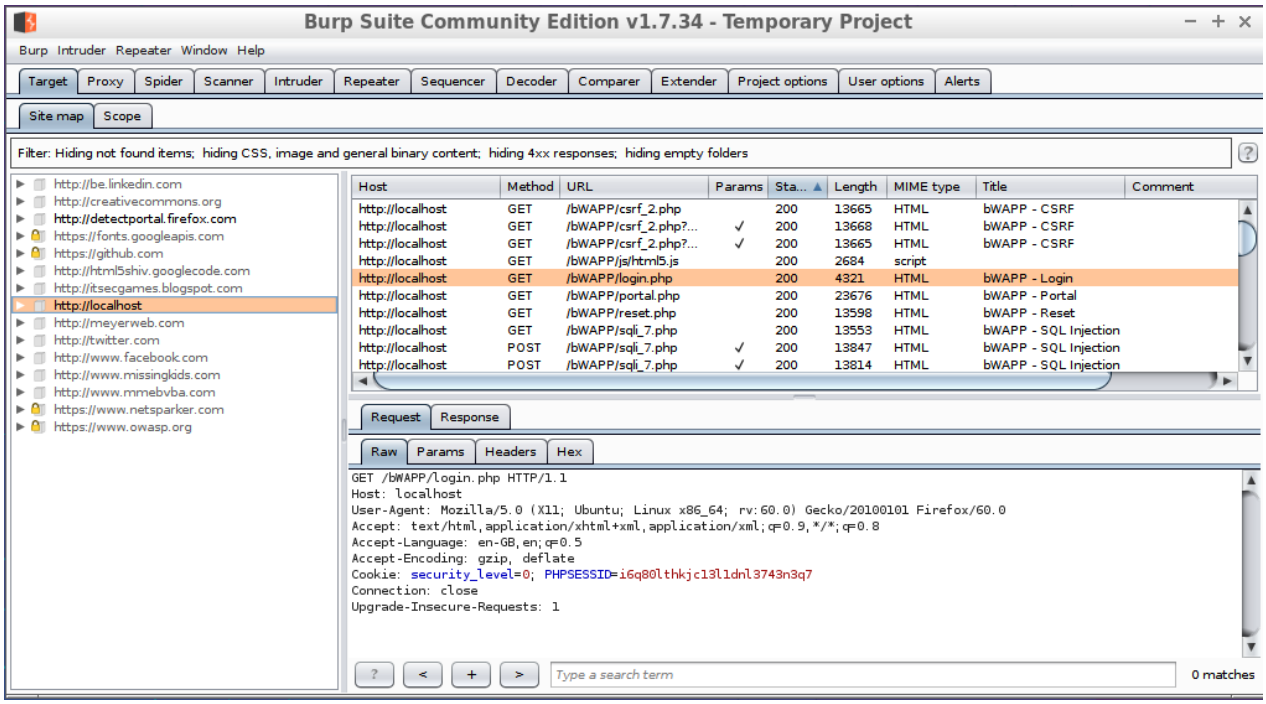


Рис. 1.10. Інтерфейс програми Burp Suite, для тестування безпеки веб-додатків

Перелічені засоби тестування веб-додатків можуть допомогти фахівцям переконатися, що програмне забезпечення на стадії нефункціонального тестування відповідає очікуванням проекту.

Висновки до першого розділу

У розділі було розглянуто різні аспекти тестування веб-додатків, включаючи типи і види тестування, а також акцентувався на функціональному та нефункціональному тестуванні. Крім того, було наведено перелік програм і інструментів, які можуть бути використані для забезпечення безпеки і якості веб-додатків відповідно до їхнього призначення.

Функціональне тестування є ключовою частиною процесу забезпечення якості програмного забезпечення і включає в себе оцінку того, як програма виконує свої функції та чи відповідає вона вимогам та специфікаціям.

З функціонального тестування виділяються такі типи, як перевірка функціональності веб-додатка, перевірка вхідних та вихідних даних, оцінка зручності інтерфейсу користувача. Перевірка відповідності специфікації проекту, перевірка інтеграції, та інші. Кожен з цих типів має свою унікальну роль у впевненість у функціональності та відповідності вимогам веб-додатку.

Нефункціональне тестування, з іншого боку, охоплює аспекти, які не стосуються безпосередньо функціональності, але впливають на якість та ефективність додатка. Серед таких видів тестування можна виділити тестування продуктивності, безпеки, сумісності, доступності, кросбраузерності, та інші.

Для забезпечення безпеки веб-додатків було наведено перелік інструментів, які можуть бути використані для виявлення вразливостей і здійснення сканування веб-додатків. Кожен з цих інструментів має свої особливості та можливості для виявлення різних видів вразливостей.

У цілому, розділ надає вичерпний огляд методів та інструментів для тестування веб-додатків, допомагаючи розуміти різні аспекти тестування та вибір правильних інструментів для забезпечення безпеки і надійності веб-додатків.

РОЗДІЛ 2. АНАЛІЗ МЕТОДІВ І МОДЕЛЕЙ ТЕСТУВАННЯ ВЕБ-ДОДАТКІВ

На сьогодні тестування програмного забезпечення – один з найбільш дорогих етапів життєвого циклу програмного забезпечення, на нього відводиться від 50% до 65% загальних витрат. У царині кодування ПЗ широкого розповсюдження набули різноманітні CASEзасоби, які дозволяють прискорити процеси створення коду. На жаль, в галузі тестування відчувається нестача таких засобів і більшість зусиль витрачається на ручне тестування. Зазвичай, для проведення тестування застосовуються методи структурного («білий ящик») та функціонального («чорний ящик») тестування [1]. Розглянемо їх докладніше. При функціональному тестуванні вихідний код програми не доступний. Суть полягає в перевірці відповідності поведінки програми її зовнішній специфікації. Критерієм повноти тестування вважається перебір всіх можливих значень вхідних даних, що здійснити на практиці надзвичайно важко. При структурному тестуванні текст програми відкритий для аналізу. Суть даного методу полягає в перевірці внутрішньої логіки ПЗ. Повним тестуванням у цьому випадку буде таке, що приведе до перебору всіх можливих шляхів на графі передач керування програми. Число таких шляхів може досягати десятків тисяч. Крім того, виникає питання про створення тестів, що забезпечують дане покриття. Здійснити повне всеохоплююче тестування навіть простої програми вкрай важко, а часом і неможливо в силу обмеженості часу й ресурсів. Отже, необхідно мати певні критерії за якими мають обиратися контрольні приклади та критерії зупинки процесу тестування.

2.1. Метод “Black Box”

Тестування, безумовно, передбачає численну кількість видів, методів, способів перевірок між реальною та очікуваною поведінкою системи. Одними із фундаментальних видів є тестування чорного та білого ящиків, з якими варто познайомитися ближче.

Тестування чорного ящику (Black Box Testing) – це метод тестування програмного забезпечення, за якого функціональні можливості ПЗ перевіряються без знання внутрішньої структури коду та деталей реалізації системи. Тестування Black Box в основному зосереджується на вхідній та вихідній інформації продукту й повністю базується на вимогах і специфікаціях програмного забезпечення. Воно також відоме як зовнішнє тестування, закрите тестування або ж поведінкове тестування.

Таким «чорним ящиком» може бути будь-який ресурс, будь то операційна система, сайт, додаток чи база даних. Даний вид тестування не потребує знань та відомостей про архітектуру чи будову продукту, тому фактично перевірки здійснюються від імені звичайного споживача.

До найбільш відомих різновидів тестування чорного ящику відносять функціональне тестування, нефункціональне тестування, регресійне тестування.

Можна виділити наступні ключові переваги тестування чорного ящику:

- є більш ефективним способом тестування великих частин продукту через досить низькі вимоги до технічного боку тестування;
- займає порівняно невеликий проміжок часу;
- не вимагає технічного досвіду чи спеціалізованих знань;
- низька деталізація документації;
- є менш ресурсозатратним порівняно з тестуванням білого ящику.

Недоліки тестування чорного ящику:

- можна перевірити лише обмежену кількість вхідних даних;
- складно спроектувати тест-кейси, якщо тестувальник не володіє специфікацією продукту;
- більш обмежене за масштабом та, відповідно, результатами, тестування.

Завданням функціонального тестування є перевірка відповідності програми своїм специфікаціям. При даному підході текст програми не доступний, і програма розглядається як «чорний ящик». Найпоширенішими

видами функціонального тестування є методи випадкового тестування, еквівалентної розбивки й аналізу граничних умов [6].

Випадкове (стохастичне) тестування – відповідно до даного методу створюється необхідна кількість незалежних тестів, у яких вхідні дані генеруються випадковим чином. Недоліком даного методу є загальна кількість тестів, які необхідно згенерувати відповідно до вимог надійності, до того ж забезпечивши незалежність цих тестів. Так, наприклад, для забезпечення надійності програмного забезпечення з імовірністю відмови не більше 10-5 і з помилкою не більше 5%, потрібно згенерувати 299 572 тестів. З метою скорочення кількості необхідних тестів Маєрсом було запропоновано розглядати розбиття безлічі вихідних даних на еквівалентні класи [8].

Тестування за класами еквівалентності - відповідно до даної методики необхідно розбити множину значень вхідних даних на кінцеве число підмножин (які будуть називатися класами еквівалентності), щоб кожний тест, що є представником певного класу, був еквівалентним будь-якому іншому тесту цього класу. Два тести є еквівалентними, якщо вони виявляють ті самі помилки.

Проектування тестів за методом класів еквівалентності проводиться у два етапи: - виділення за специфікацією класів еквівалентності; - побудова множини тестів.

На першому етапі відбувається вибір зі специфікації кожної вхідної умови та розбиття її на дві або більше групи, що відповідають так званим – правильним класам еквівалентності (ПКЕ) та – неправильним класам еквівалентності (НКЕ), тобто множинам допустимих для програми й недопустимих значень вхідних даних. Цей процес залежить від вигляду вхідної умови.

Наприклад: якщо вхідна умова описує множину (|x|<=0.5), то визначається один ПКЕ (-0.5<=x <=0.5) і два НКЕ (x< -0.5; x>0.5).

На другому етапі методу класів еквівалентності виділені класи використовуються для побудови тестів. Для НКЕ тести проектуються таким

чином, що кожен тест покриває один і тільки один НКЕ, доки всі НКЕ не будуть покриті.

Метод класів еквівалентності дозволяє значно скоротити кількість тестів у порівнянні з методом випадкового тестування, але також має свої недоліки. Основний з них - це складність виділення класів еквівалентності, особливо НКЕ, а також можливий пропуск певних типів високоефективних тестів (тобто тестів, що характеризуються великою ймовірністю виявлення помилок). Так, наприклад, мінімальні й максимальні припустимі значення вхідних параметрів дозволяють виявити більшість помилок, пов'язаних з відповідностями й переповненнями типів даних. Для вирішення даної проблеми був запропонований метод аналізу граничних умов [2, 19].

Метод аналізу граничних умов - під граничними умовами розуміють ситуації, що виникають безпосередньо на границі певної вхідної або вихідної умови, вище або нижче її. Метод аналізу граничних умов відрізняється від методу класів еквівалентності наступним:

- вибір будь-якого представника класу еквівалентності здійснюється таким чином, щоб перевірити тестом кожну границю цього класу;
- при побудові тестів розглядаються не тільки вхідні умови, але й вихідні (тобто певні специфіковані обмеження на значення вхідних даних).

Загальні правила методу аналізу граничних умов – побудувати тести для границь множини допустимих значень вхідних даних і тести з недопустимими значеннями, що відповідають незначному виходу за межі цієї множини.

Наприклад, для множини [-1.0; 1.0] будуються тести -1.0; 1.0; - 1.001; 1.001; Зауважимо, що на практиці з метою локалізації несправностей створюють також тести, що відповідають допустимим значенням, тобто є внутрішніми для множини та ті, що незначно відхиляються від граничних значень: -1.0; 1.0; -1.001; 1.001;0.999;- 0.999.

Якщо множина допустимих значень вхідних даних дискретна, то будуються тести для мінімального й максимального значення вхідних умов і тести для значень, більших або менших цих величин.

Наприклад, якщо вхідний файл може містити від 1 до 255 записів, то вибираються тести для порожнього файлу та файлу, що містить 1, 254, 255 і 256 записів.

Аналіз граничних умов – один з найбільш корисних методів проектування тестів. Але він часто виявляється неефективним через те, що граничні умови іноді ледь вловимі, а їхнє виявлення досить важке.

2.2. Метод “White Box”

Якщо тестування чорного ящику – це тестування без доступу до «нутрощів» продукту, то на противагу йому тестування білого ящику (White Box Testing) – це метод тестування, який перевіряє внутрішнє функціонування системи. У цьому методі тестування перевіряються внутрішня структура, будова і код програмного забезпечення. У тестуванні білого ящика код доступний тестувальникам, тому його також називають тестуванням чистого ящика, тестування відкритого ящика, тестування прозорого ящика, тестування на основі коду та тестування скляного ящика.

До найбільш відомих різновидів тестування білого ящику відносять модульне тестування, циклічне тестування та тестування на витік пам’яті.

Загальний алгоритм проведення тестування білого ящику:

- вивчити та зрозуміти код ресурсу, який тестують – на даному етапі важливо приділити увагу базовим принципам побудови коду, його логічності, простоті, і найперше – безпеці. Безпека є найбільш частою метою тестування білого ящику, адже вразливість продукту закладається саме на рівні коду;
- створити та пройти тест-кейси – на цьому етапі слід перевірити структуру та правильність побудови коду. Це відбувається найчастіше за допомогою написання автоматизованих тестових випадків перевірки коду, або ж за допомогою мануальних тестів. Звичайно, й автоматизація, і мануальні перевірки вимагають фундаментальних знань тестувальника у програмуванні.

Переваги білого ящику:

- значна оптимізація коду продукту шляхом аналізу та пошуку прихованих помилок;
- відносно легкий процес автоматизації більшості перевірок;
- значно ретельніші перевірки базових шляхів та зв’язків у продукті;
- можливість розпочати тестування на найперших етапах життєвого циклу тестування ПЗ, навіть якщо фронт-енд ще не реалізовано.

Серед недоліків тестування білого ящику виділяємо такі:

- даний вид тестування є досить ресурсозатратним;
- складність виконання через високі вимоги професійної кваліфікації тестувальників;
- більш обмежений вибір фахівців для виконання;
- процес займає тривалий час.

Структурне тестування, або тестування «білого ящика», - це методика аналізу вихідного коду програми. Існує три різновиди структурного тестування: тестування на основі потоку керування програми, на основі потоку даних та мутаційне тестування.

При використанні першого типу тестується логіка програми, що представлена у вигляді графа керування: вершинами є оператори, а гілками - переходи між ними. При тестування на основі потоку даних увага приділяється взаємозв'язкам між змінними. Виділяються вершини, у яких змінна ініціалізується та в яких використовується, і вивчаються переходи й взаємозв'язки між такими вершинами. Мутаційне тестування полягає у внесенні несправностей у вихідний код програми та порівняння роботи вихідної програми та програми мутанта. Оскільки здійснити вичерпне структурне тестування вкрай важко, необхідно вибрати такі критерії його повноти, які допускали б їхню просту перевірку й полегшували б цілеспрямований підбір тестів. Зупинимося на цьому докладніше.

Мутаційне тестування є різновидом тестування білого ящика, для його здійснення необхідний доступ до вихідного коду програми [4]. Мутаційний

критерій ґрунтується на штучному внесенні помилок у програму. У мутаційному критерії приймається припущення про те, що програмісти пишуть майже коректні програми, що відрізняються від правильних незначними помилками в арифметичних операціях, перестановками індексів, некоректними границями циклів, невірними константними значеннями та ін. Для виправлення дефектів подібного роду, у програму вносяться дрібні помилки (мутації). Програми, що відрізняються від вихідних програм, штучно внесеними помилками називають мутантами. Як правило, мутант відрізняється від вихідної програми невеликим числом мутацій. У вихідній програмі можуть піддаватися мутаціям ділянки коду пов'язані з перерахованими вище дефектами (змінюються значення змінних, модифікуються індекси й границі циклів, вносяться мутації в умови). Таким чином, з первісної програми шляхом внесення n числа мутацій одержують k мутантів, n>=k (як мінімум одна мутація на один мутанта). Якщо сформована множина тестових наборів виявляє всі мутації у всіх мутантах, то воно відповідає мутаційному критерію. Якщо тестування вихідної програми та мутанту на заданій множині тестових наборів не виявило помилок, то програма оголошується еквівалентною мутанту. У випадку мутаційного тестування важливо створити таке число мутантів, яке б охоплювало всі можливі ділянки прояву помилок. Вважається, на основі дрібних помилок можна оцінити загальне число помилок, що залишилися в програмі [3, 19].

2.3. Метод “Grey Box”

Тестування сірим ящиком, або "Grey Box Testing", представляє собою комбінацію методів тестування білим ящиком і чорним ящиком. У цьому методі тестувальник має часткове знання внутрішньої структури системи, що дозволяє враховувати як зовнішні, так і внутрішні аспекти під час тестування.

Основні риси тестування сірим ящиком включають:

- Чорний ящик інтерфейсу – тестувальник дивиться на систему як на "чорний ящик" в тих випадках, коли необхідне врахування зовнішнього

поведінки без докладного знання про внутрішню реалізацію. Це дозволяє визначити, чи відповідає програма вхідним даним і генерує очікувані результати.

- Білий ящик часткового знання – тестувальник має обмежене знання внутрішніх структур і деталей системи. Це може включати обмежене знання коду, але не завершене розуміння всіх аспектів внутрішньої реалізації.
- Тестування функціональності та структури – тестування сірим ящиком спрямоване на перевірку функціональних аспектів системи, таких як зовнішня функціональність та взаємодія з користувачем, а також на структурні аспекти, враховуючи обмежене знання внутрішньої реалізації.
- Тестування безпеки – в контексті тестування безпеки, тестування сірим ящиком може включати аспекти аналізу зовнішнього поведінки системи, спрямовані на виявлення потенційних вразливостей.
- Тестування зв'язку між компонентами – тестування сірим ящиком також може враховувати взаємодію між різними компонентами системи, не вдаваючись при цьому в деталі внутрішньої реалізації цих компонентів.

Цей метод дозволяє тестувальникам зберігати деяку незалежність від внутрішньої реалізації системи, але при цьому надає можливість враховувати технічні деталі там, де це необхідно.

Переваги тестування методом "Grey Box":

- Метод "Grey Box" надає гнучкість та збалансований підхід, дозволяючи враховувати як зовнішню функціональність, так і обмежені внутрішні деталі.

Наприклад: Тестувальник може здійснювати тестування з точки зору кінцевого користувача, враховуючи його потреби, але при цьому мати деяке знання внутрішньої реалізації для ефективного виявлення помилок.

- Знання обмеженої кількості внутрішніх деталей може полегшити процес тестування і зробити його ефективнішим у порівнянні з тестуванням білим ящиком.

Наприклад: Тестувальники можуть швидко перевірити зовнішню функціональність системи, не витрачаючи час на вивчення всіх внутрішніх деталей.

- Завдяки комбінації обох підходів (чорного і білого ящиків), тестування сірим ящиком може дозволити досягнути більшого покриття коду і функціональності [19].

Підсумуємо ключові відмінності описаних двох видів тестування у порівняльній таблиці (Табл. 2.1.):

Таблиця 2.1. Порівняння методів тестування

Ознака порівняння	**Тестування чорного ящику**	**Тестування білого ящику**	**Тестування сірого ящику**
1	2	3	4
Знання програмування	Не потрібне	Обов’язкове	Частково
Ким виконується	Виконується кінцевим користувачем, розробником і тестувальником	Виконується тестувальником з відповідними знаннями і розробниками	Виконується тестувальником з відповідними знаннями і розробниками
База тестування	Тестування базується на зовнішніх очікуваннях; внутрішня поведінка програми невідома	Внутрішні принципи функціонування відомі і можуть бути перевіреними	Внутрішні принципи функціонування відомі і можуть бути перевіреними
Час виконання	Менш вичерпний і трудомісткий вид тестування	Вичерпний і трудомісткий вид	Менш вичерпний і трудомісткий вид тестування
Використання	Ідеально підходить для вищих рівнів тестування, таких як системне тестування, приймальне тестування	Найкраще підходить для нижчого рівня тестування, наприклад модульного тестування, інтеграційного тестування	Одночасно задіює як вищі рівні так і нижчі рівні тестування

1	2	3	4
Ключова перевага	Добре підходить та ефективний для великих частин продукту та/або для масштабних проєктів	Дозволяє видалити зайві рядки коду, які можуть принести приховані дефекти	Надає гнучкість та збалансований підхід, дозволяючи враховувати як зовнішню функціональність, так і обмежені внутрішні деталі.
Техніки тестування	Класи еквівалентності (Equivalence partitioning), аналіз граничних значень (Boundary value analysis)	Покриття заяви (Statement Coverage), покриття філій (Branch coverage), покриття шляху (Path coverage)	Комбіновані
Ключова мета	Перевірити функціональні можливості тестованої системи	Перевірити якість коду продукту	Комбіновано

Висновки до другого розділу

Класичні методи (Чорний ящик, Білий ящик, сірий ящик) структурного й функціонального тестування мають певні обмеження при застосуванні, розглянемо їх детальніше. Функціональне тестування за допомогою методу Чорного ящика характеризується дуже великою кількістю необхідних тестів, більш того, відразу піднімається питання про забезпечення незалежності цих тестів. Якщо ж обмежувати кількість, то, все одно, виникають складності як при виділенні класів еквівалентності, так і границь, при цьому може бути пропущений ряд високоефективних тестів, а зловмисна логіка не виявляється. До того ж, варто помітити, що методи функціонального тестування не дозволяють локалізувати несправності. Функціональне тестування застосовується тільки коли ПЗ вже створене, тобто на останніх етапах життєвого циклу ПЗ. Якщо функціональне тестування виявить погану якість створення ПЗ, то доводиться вертатися на попередні етапи розробки, що спричиняє як фінансові збитки так і часові витрати.

Структурне тестування перевіряє внутрішню логіку програми, що дозволяє локалізувати несправності. На жаль, із зростанням розміру вихідного

коду програми повноцінне структурне тестування стає все складнішим. А спуск по представленій ієрархічній структурі взаємозв'язків критеріїв приводить до пропущення певного типу помилок і, відповідно, до втрати якості. Можливості застосування структурного тестування для різних фаз тестування обмежені. Якщо для модульного тестування, в силу невеликих розмірів вихідного коду, представлені методи застосовні, хоча й з тими або іншими зазначеними вище складностями й обмеженнями, то для інтеграційного й системного тестування сфера застосування наведених класичних методів украй обмежена в силу різкого зростання вихідного коду або взагалі відсутності такого. Більше того, можливість застосування структурного тестування залежить і від обраної парадигми програмування: якщо для процедурного програмування методи структурного тестування застосовні; для об‘єктно-орієнтованого застосування обмежене через зростання обсягу вихідного коду; то для компонентно-базованого програмування - дані методи часто стають недосяжними [3, 5, 6]. При компонентно-базованому програмуванні, компоненти в основному представлені як «чорні ящики» і доступні лише їхні автомати станів (на UML). Таким чином, класичні методи структурного тестування, для яких рівень абстракції - це рівень операторів, стають не застосовні.

Кожен з цих методів має свої унікальні особливості та застосування, що робить їх важливими інструментами для розробників та тестувальників програмного забезпечення.

Метод White box тестування, заснований на внутрішніх аспектах програми, дозволяє виявити помилки на рівні коду та логіки, що робить його незамінним для забезпечення високої якості програм. Black box тестування, у свою чергу, фокусується на зовнішньому поведінці системи, що спрощує процес тестування функціональності та взаємодії з користувачем. Гібридний метод - Grey box тестування - поєднує переваги обох підходів, дозволяючи більш глибоко та повноцінно тестувати програму.

Безперечно, що кожен метод має свої переваги та недоліки. White box тестування може виявитися витратним у великих проектах, тоді як Black box може пропустити деякі деталі внутрішньої логіки. Grey box тестування, хоча і є компромісом, вимагає додаткового зусилля для організації.

Наприклад, використання White box тестування може бути особливо корисним при розробці критичних застосунків, де важлива безпека та надійність. У той час як Black box тестування може бути ефективним для забезпечення функціональності та коректної взаємодії із зовнішніми системами.

Загальний висновок полягає в тому, що вибір методу тестування повинен бути здійснений на основі конкретних вимог та характеристик проекту. Комбінування різних методів може забезпечити комплексний та ефективний підхід до забезпечення якості програмного продукту.

РОЗДІЛ ІІІ. РОЗРОБКА ПОКРОКОВОЇ ІНСТРУКЦІЇ ВАЛІДАТОРА HTML-ДОКУМЕНТУ

3.1. Валідація HTML-документу, як необхідний компонент тестування веб-додатка

HTML (Hypertext Markup Language) є мовою розмітки, яка використовується для створення структури веб-сторінок. HTML складається з тегів, які оточують різні елементи контенту, такі як текст, зображення, посилання та інші [1].

Основний синтаксис HTML виглядає наступним чином (Рис. 3.1.):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Заголовок сторінки</title>
</head>
<body>
    <h1>Привіт, світ!</h1>
    <p>Це приклад абзацу тексту.</p>
    <a href="https://www.example.com">Посилання на приклад</a>
</body>
</html>
```

Рис. 3.1. Вигляд основного синтаксису HTML

- HTML документ складається з різних компонентів, які визначають структуру і зміст веб-сторінки. Основні складові HTML документа включають:
- Елемент <!DOCTYPE html>: Це декларація типу документа, яка повідомляє браузеру, яка версія HTML використовується на сторінці.
- Елемент <html>: Визначає початок і кінець HTML документа.
- Елемент <head>: Містить інформацію про документ, таку як метатеги, зовнішні таблиці стилів, сценарії та інше.
- Метатеги (<meta>): Використовуються для встановлення характеристик документа, таких як кодування символів та налаштування відображення на мобільних пристроях.

- Елемент <title>: Визначає заголовок сторінки, який відображається в заголовку вкладки браузера або на панелі завдань.
- Елемент <body>: Містить зміст сторінки, такий як тексти, зображення, посилання, форми, скрипти та інші елементи.
- Текстові елементи: Такі як <h1>, , <a>, <img>, <ul>, <ol>, <li> і багато інших, які визначають заголовки, абзаци, посилання, зображення та списки.
- Коментарі: Коментарі дозволяють розміщувати коментарі в HTML коді, які не відображаються у вихідному HTML документі.

Ці компоненти спільно визначають структуру і вміст HTML документа, дозволяючи браузерам інтерпретувати і відображати сторінку користувачам.

HTML документи повинні відповідати певним вимогам та правилам, щоб забезпечити коректне їхнє відображення в браузерах та взаємодію з різними інструментами. Основні вимоги та правила включають:

- HTML документ повинен розпочинатися декларацією типу документа (DOCTYPE), яка вказує браузеру на використання конкретної версії HTML.
- HTML документ повинен мати кореневий елемент <html>, який визначає початок і кінець HTML документа.
- Усі теги повинні бути закриті правильно. Наприклад: Це абзац тексту
- Атрибути тегів повинні бути вказані в подвійних або одинарних лапках, а їх значення повинні бути правильно визначені.
- HTML документ повинен бути валідним, тобто відповідати стандартам HTML.

Валідація HTML документа - це процес перевірки коду HTML на відповідність стандартам і синтаксичні правила. Правильно написаний і валідний HTML забезпечує правильне відображення сторінок у браузерах та сприяє кращій доступності та оптимізації для пошукових систем.

Існує кілька способів валідації HTML:

- Онлайн-валідатори: Спеціальні інтернет-сервіси, які дозволяють вам вставляти код HTML безпосередньо в інтерфейс веб-сторінки та отримувати звіт про помилки. Найпопулярніший з них - це W3C Markup Validation Service.
- Браузерні розширення: Деякі браузерні розширення надають можливість валідації HTML прямо з браузера, наприклад, розширення для Google Chrome.
- Локальні інструменти: Ви можете встановити локальні інструменти, такі як html-validator або tidy, або використовувати власноруч створену упргграму для перевірки HTML-коду.

Після того, як ви використаєте один із цих методів, вам буде надано звіт про помилки, які потрібно виправити в коді, щоб зробити його валідним.

Важливо валідувати HTML перед публікацією веб-сторінки, оскільки це може допомогти уникнути проблем з відображенням на різних пристроях та браузерах, а також підвищити загальну якість вашого веб-сайту. Особливо коректність та валідність HTML-коду можуть впливати на результати пошукових систем таким чином:

- Пошукові системи використовують роботів (пошукові боти), щоб індексувати вміст веб-сайтів. Якщо HTML-код має валідну структуру, це полегшує роботів правильно індексувати ваш вміст. Валідний HTML може сприяти кращій індексації вашого вмісту пошуковими системами.
- Зрозумілість контенту - правильна розмітка HTML допомагає пошуковим системам зрозуміти структуру і зміст вашого веб-сайту. Це може позитивно вплинути на сприйняття ключових слів, заголовків та інших елементів сторінки.
- Валідний HTML може позитивно вплинути на адаптивність вашого веб-сайту, дозволяючи йому правильно відображатися на різних пристроях. Це також може вплинути на швидкість завантаження сторінок, що є фактором ранжування для багатьох пошукових систем.

- Доступність: Правильний HTML сприяє поліпшенню доступності веб-сайту для користувачів з обмеженими можливостями. Це може включати в себе коректне визначення ролей, використання альтернативних текстів для зображень і тому подібне. Ці аспекти можуть враховуватися пошуковими системами при визначенні рейтингу.

3.2. Розробка валідатора HTML-документа

Зважаючи на важливість використання валідатора HTML-документа під час тестування веб-додатка, та наслідки недотримання вимог до HTML-коду, вважаємо за необхідне створити власний валідатор HTML-документу. Створення власного валідатора HTML може бути важливим з точки зору індивідуальних потреб та конкретних обставин проекту. Основна причина в цьому полягає в тому, щоб мати контроль над процесом валідації та можливість відстроювати його під специфічні вимоги та внутрішні стандарти. Актуальність цього підходу визначається необхідністю максимальної гнучкості та індивідуальними потребами розробників у конкретному проекті.

Створення власного валідатора HTML-документу має й інші переваги та причини:

- Створення власного валідатора дозволяє вам визначити власні внутрішні правила та стандарти для HTML документів, що може бути важливим, якщо у вас є конкретні вимоги або стандарти для вашого проекту чи організації.
- Ви можете кастомізувати валідатор так, щоб він враховував ваші конкретні потреби. Наприклад, ви можете додати власні правила валідації або видаляти ті, які не є важливими для вашого проекту.
- Створення власного валідатора дозволяє вам валідувати HTML документи локально, без необхідності відправляти їх на зовнішній сервіс для перевірки. Це може бути зручно, особливо при розробці та тестуванні.

- Ви можете налаштувати свій валідатор так, щоб обробляти помилки в спосіб, який вам підходить. Наприклад, генерувати деталізовані звіти про помилки або автоматично виправляти певні типи помилок.
- Створення власного валідатора може допомогти вам інтегрувати його з вашим робочим процесом розробки, дозволяючи автоматизувати валідацію та забезпечувати високу якість HTML коду.
- Освітній аспект: Створення власного валідатора може бути освітнім процесом, що дозволяє глибше зрозуміти принципи роботи HTML та механізми валідації. Це може бути корисним для навчання чи розвитку у галузі веб-розробки.
- Якщо ви працюєте над проектом або для компанії, де існує Non-Disclosure Agreement (NDA) (угода про нерозголошення), то створення власного валідатора HTML може бути пов'язане з вимогою зберігати конфіденційність щодо структури і особливостей HTML-коду. Власний валідатор може бути створений з метою внутрішнього використання, де зовнішні сервіси валідації не можуть або не повинні бути використані через обмеження, зазначені в NDA. Такий підхід дозволяє забезпечити контроль за конфіденційністю і забезпечити відповідність угоді про нерозголошення.

3.3. Покрокова інструкція розробки валідатора HTML-документа на мові програмування Python

Створення валідатора HTML-документа за допомогою мови програмування Python є цікавим завданням. В цій частині дипломної роботи буде наведено конкретні кроки та методи, які необхідно виконати для розробки програми.

Мова програмування Python була обрана з багатьох причин, наприклад: спрощений синтаксис і читабельність, що полегшує розробку та обслуговування коду; широкий вибір бібліотек для обробки HTML, веб-скрапінгу та роботи з мережевими запитам. [16] BeautifulSoup – одна з таких

бібліотек; активна спільнота, що означає багато документації, прикладів коду, та підтримки з різних питань; Python легко інтегрується з іншими технологіями, що може бути важливим для проектів, де використовуються інші мови програмування або інструменти; мова Python є портативною мовою програмування, що дозволяє запускати код на різних платформах без необхідності змін. Це робить його зручним для проектів, які мають різні середовища виконання.

Перш за все необхідно встановити Python. Для цього скачуємо встановлювач з відповідного розділу на офіційному сайті Python, у нашому випадку це останній реліз 3.12.0 для операційної системи Windows (Рис. 3.2.). Та встановити його https://www.python.org/downloads/windows/.

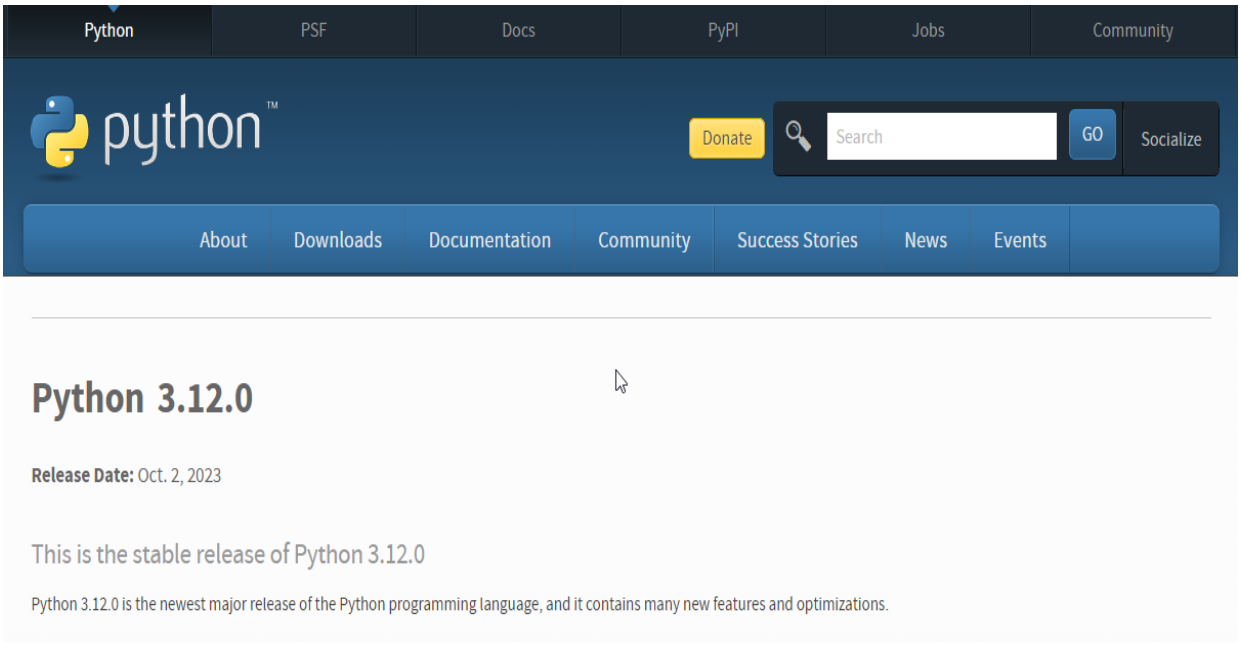
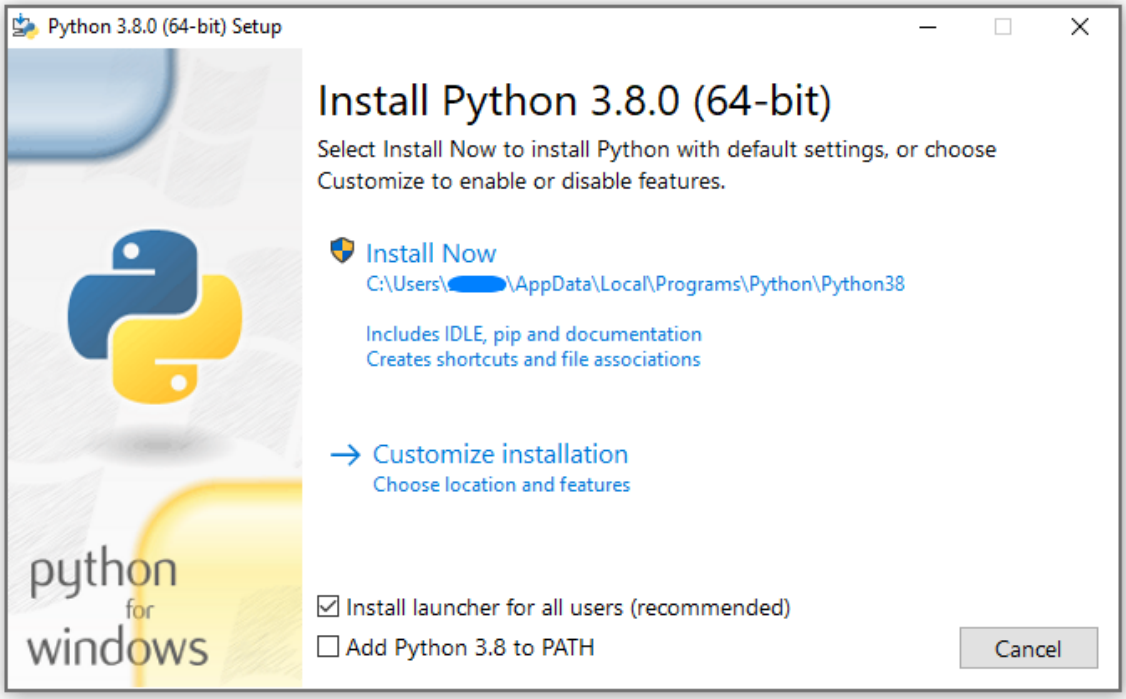


Рис. 3.2. Скріншот з офіційного сайту Python

Після чого необхідно встановити Python (Рис. 3.3.), надаючи всі необхідні дозволи, та вмикаючи усі необхідні налаштування для коректної роботи Python, під час програмування.



h

Рис. 3.3. Вигляд інсталятора Python

Після чого нам необхідно налаштувати середовище програмування у IDE Visual Studio Code, для початку програмування (Рис.3.4.).

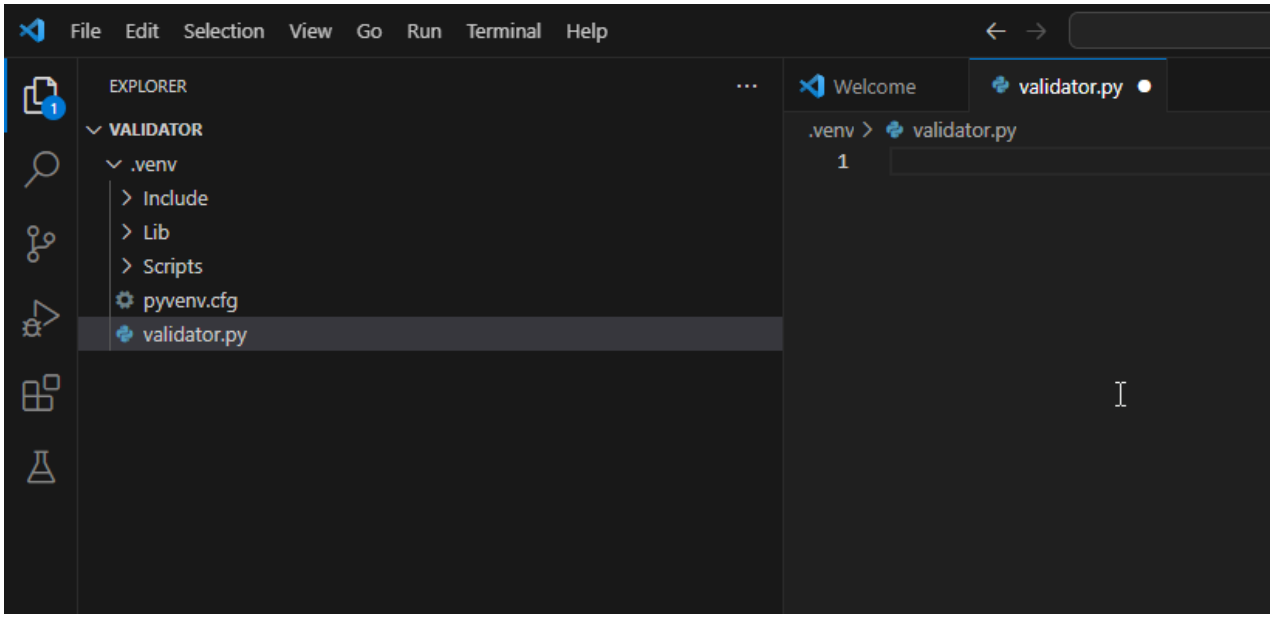


Рис. 3.4. Вигляд налаштованого середовища Python в IDE Visual Studio Code

Важливим моментом є встановлення пакетів Python beautifulsoup4 та html5validator. beautifulsoup4 та html5validator – це пакети Python, які використовуються для обробки та роботи з HTML-документами.

Призначення beautifulsoup4 – парсинг HTML та XML даних. Вона дозволяє легко вилучати інформацію з HTML-документів, проводити навігацію по структурі DOM (Document Object Model) та взаємодіяти з різними елементами веб-сторінок. Встановлення виконується за допомогою pip. Введіть в терміналі чи командному рядку таку команду: “pip install beautifulsoup4”.

Призначення html5validator – валідація HTML-коду відповідно до стандартів HTML5. Він дозволяє перевіряти HTML-документи на відповідність сучасним стандартам та виправляти можливі помилки. Встановлення також виконується за допомогою pip. Введіть в терміналі чи командному рядку таку команду: “pip install html5validator”.

Наступним кроком буде парсинг HTML-документа. Для цього ми будеми виокристовувати beautifulsoup4. Приклад (Рис.3.5.):

validator.py

```
from bs4 import BeautifulSoup

html_content = """
<!DOCTYPE html>
<html>
<head>
    <title>Мій веб-сайт</title>
</head>
<body>
    <h1>Вітаємо на нашому веб-сайті</h1>
    <p>Це приклад HTML-коду</p>
</body>
</html>
"""

soup = BeautifulSoup(html_content, 'html.parser')
```

Рис. 3.5. Використання beautifulsoup4 для парсингу HTML-коду

Після того як відбувся парсинг HTML-документу, за допомогою пакету html5validator валідуємо вихідні дані (Рис.3.6.).

validator.py

```python
from html5validator import HTML5Validator

validator = HTML5Validator()

result = validator.validate_fragment(str())
if result.is_valid:
    print("HTML код валідний")
else:
    print("HTML код не валідний:")
    print(result.messages)
```

Рис. 3.6. Використання html5validator для валідації HTML-коду

Користуючись однією із переваг власноруч створеного валідатора ми можемо встановити власні правила валідації. Для цього їх необхідно додати до html5validator. Наприклад ми можемо визначити власні атрибути, які повинні бути у певних тегах (Рис.3.7.).

.venv > validator.py > ...

```python
from html5validator import ValidationRules

class MyValidationRules(ValidationRules):
    def validate_my_attribute(self, attrs, valid_values):
        my_value = attrs.get('my_attribute')
        if my_value not in valid_values:
            return f"Помилка: my_attribute повинен бути одним із {valid_values}"

validator = html5validator(rules=MyValidationRules())
```

Рис. 3.7. Приклад кастомізації валідації HTML-коду

І на останок ми можемо інтегрувати цей процес у проєкт який тестуємо. Один з шляхів – створення окремого Python-модуля, та імпорт його у проєкт [16]. Або ж якщо наявна система неперервної інтеграції/поставки (CI/CD), можна включити валідатор у процес автоматизованої збірки та тестування. Наприклад, можна додати виклик валідатора до скрипта CI/CD-пайплайну.

3.4 Валідатор HTML-документу на мові програмування Python

Структура проекту валідатора на мові програмування Python (Дод. 1) виглядає наступним чином (Рис. 3.8):

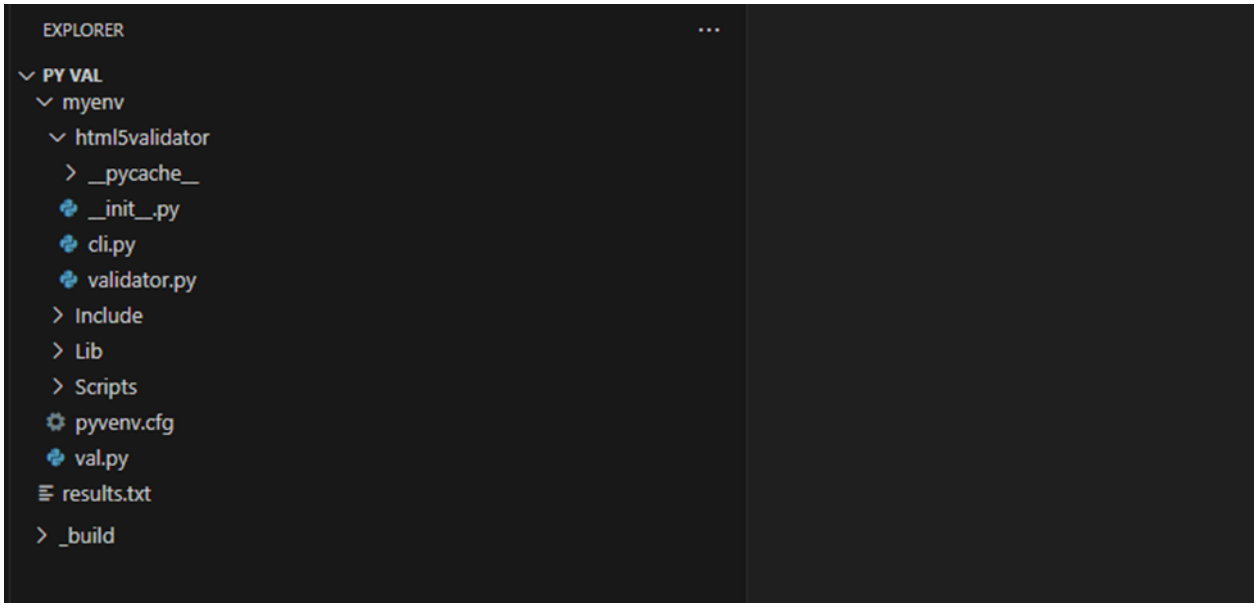


Рис. 3.8 Структура проекту валідатора на мові програмування Python

1. Папка “_pycache_”, вона створюється Python-інтерпретатором для зберігання скомпільованих версій файлів модулів. Коли Ви імпортуєте модуль, Python зазвичай компілює (перетворює в байт-код) вихідний код модуля у спеціальний байт-код, щоб зберегти час при наступних імпортах.
2. Файл “__init__.py”, він використовується для того, щоб Python розглядав каталог як пакет, а не просто як звичайний каталог із набором модулів. Це файл ініціалізації пакету, і його наявність вказує Python, що директорія, в якій знаходиться цей файл, повинна трактуватися як пакет.
3. Файл cli.py - він використовується для реалізації інтерфейсу командного рядка (CLI - Command Line Interface) в програмах на мові програмування Python. CLI є текстовим інтерфейсом, який дозволяє користувачам взаємодіяти з програмою, використовуючи команди, які вони вводять у командному рядку.

4. Файл “validator.py” є основним класом валідатора. Файл містить методи для перевірки HTML5-коду на відповідність стандартам та інші відповідні функції. Він відповідає за основну логіку валідації, яка використовується в рамках програми.

Для того щоб перевірити HTML-документ необхідно запустити програму в віртуальному середовищі Python, та за допомогою команди “html5validator html_docs > results.txt”, в терміналі віртуального середовища, перевірити HTMl-документ. В команді “html5validator” буде відповідати за виклик інструменту для перевірки HTML-документу, “html_docs” — директорією де знаходяться HTML-документи, які необхідно перевірити, “> results.txt” — частина команди яка відповідає за перенаправлення виводу результатів перевірки у новостворений текстовий документ “results.txt”. Відповідь складається з локації файлу, який перевіряється, рядка на якому виникла помилка, та тексту самої помилки. Так виглядає результат перевірки HTML-документу головної сторінки сайту Wikipedia https://uk.wikipedia.org/wiki/ (рис. 3.9).

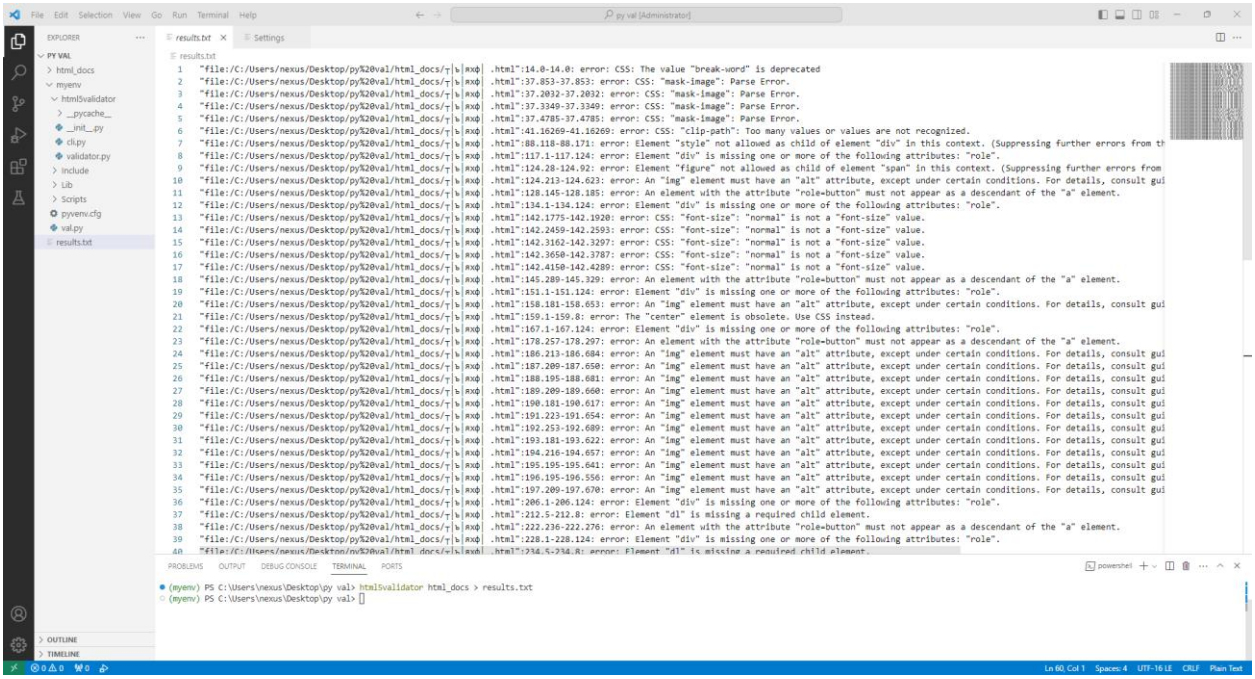


Рис. 3.9 результат перевірки HTML – документу головної сторінки сайту Wikipedia

Висновки до третього розділу

У даному розділі дипломної роботи досліджено ключові аспекти, пов’язані з валідацією HTML-документів у контексті тестування веб-додатків. Відзначена необхідність валідації HTML-документу як важливого етапу тестування, оскільки цей процес сприяє виявленню та усуненню можливих помилок та неполадок, що можуть виникнути під час роботи веб-додатка. Необхідність полягає у:

- Забезпечення правильного відображення;
- Підвищення доступності;
- Покращення оптимізації для пошукових систем (SEO);
- Підтримка мобільних пристроїв;
- Сумісність інтерфейсів та бібліотек;
- Зменшення ризику помилок.

Розділ також розглянув доцільність розробки власного валідатора HTML-документу. Виявлено, що розробка власного валідатора може бути ефективною стратегією, особливо коли вимоги до валідації специфічні для конкретного проекту. Власний валідатор дозволяє здійснювати гнучкий та індивідуальний контроль над правильністю HTML-структури в межах проекту. Переваги та причини:

- Контроль внутрішніх правил;
- Кастомізація;
- Локальна валідація;
- Налаштування обробки помилок;
- Інтеграція з робочим процесом;
- Освітній аспект;
- Валідація в умовах NDA.

Особливу увагу приділено створенню покрокової інструкції для розробки валідатора HTML-документу на мові програмування Python. Встановлено, що Python, завдяки своїй простоті та потужній функціональності, є відмінним вибором для створення інструментів валідації.

Програмна реалізація валідатора на Python дозволяє легко інтегрувати його у тестові сценарії веб-додатків та забезпечує високу точність виявлення помилок.

Узагальнюючи, валідація HTML-документів, розгляд різних підходів до розробки валідаторів та використання мови програмування Python вказують на важливість системного та індивідуального підходу до цього етапу тестування для забезпечення якості веб-додатків.

ВИСНОВКИ

Дипломна робота присвячена аналізу, створенню та дослідженню допоміжних засобів для тестування веб-додатків, розглядає основні визначення та терміни, пов’язані з тестуванням, визначає різновиди тестування, зокрема функціональне та нефункціональне.

Функціональне тестування виявляється ключовою частиною процесу забезпечення якості програмного забезпечення, оцінюючи, як програма виконує свої функції та чи відповідає вимогам. Зазначаються різні типи функціонального тестування, такі як перевірка функціональності, валідація вхідних та вихідних даних та оцінка зручності інтерфейсу користувача.

Нефункціональне тестування охоплює аспекти, не пов'язані безпосередньо з функціональністю, але впливають на якість і ефективність додатка, такі як продуктивність, безпека, сумісність, доступність і кросбраузерність.

Під час виконання магістерської роботи виконано усі поставленні завдання, а саме:

1. Проаналізовано дослідження з теми: наукові видання, програмні продукти та було виявлено, що тестування веб-додатків відіграє важливу роль у забезпеченні їхньої ефективності та надійності. Наукові видання та програмні продукти, які були розглянуті, висвітлили різні аспекти цього процесу, включаючи використання різних методів та підходів до тестування веб додатків; використання актуальних та відповідних допоміжних засобів тестування.

2. Здійснено пошук та оцінку інструментів для тестування веб-додатків, які можуть бути більш ефективними або забезпечувати більші можливості порівняно з існуючими рішеннями, а саме під час проведення пошуку та оцінки нових інструментів для тестування веб-додатків були виявлені декілька потенційно ефективних та інноваційних рішень. Серед них варто відзначити:, Postman, TestRail, TestMace, Fiddler. Ці інструменти полегшують процеси та покращують якість веб-додатків. Важливо обрати той

інструмент, який найбільше відповідає конкретним потребам та вимогам проєкту.

3. Удосконалення процесу тестування веб-додатків, включаючи автоматизацію, оптимізацію тестових сценаріїв та зменшення витрат часу і ресурсів. Автоматизація тестування HTML-документів за допомогою кастомного валідатора дозволяє швидко виявляти та виправляти помилки в коді, забезпечуючи високу якість та відповідність вимогам проєкту.

4. Створення та опис детальної покрокової інструкції із розробки валідатора HTML-документа на мові програмування Python, та розробка на основі цього валідатора HTML-документа. Досліджено доцільність розробки власного валідатора HTML-документа та визначено, що це може бути ефективною стратегією, особливо коли вимоги до валідації специфічні для конкретного проекту. Власний валідатор дозволяє здійснювати гнучкий та індивідуальний контроль над правильністю HTML-структури в межах проекту, забезпечуючи контроль внутрішніх правил, кастомізацію, локальну валідацію та інші переваги.

Особлива увага приділена розробці валідатора HTML-документа на мові програмування Python. Встановлено, що Python, завдяки своїй простоті та потужній функціональності, є відмінним вибором для створення інструментів валідації. Програмна реалізація валідатора на Python легко інтегрується у тестові сценарії веб-додатків та забезпечує високу точність виявлення помилок.

Узагальнюючи, валідація HTML-документів важлива для забезпечення якості веб-додатків. Розгляд різних підходів до розробки валідаторів та використання мови програмування Python підкреслює важливість системного та індивідуального підходу до цього етапу тестування.

Загальна мета магістерської роботи – досягнута. Було висвітлено, проаналізовано та досліджено інформацію про різні аспекти тестування веб-додатків та пропозиції щодо розробки валідатора HTML-документів для поліпшення якості програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Дідковська М.В. Дослідження та аналіз графічних моделей функціональних вимог до Web-проектів [Текст] / М.В. Дідковська // Наукові вісті. 2007. №6. С.49-54.

2. Дідковська М. В. Тестування: Критерії та методи. / Марина В. Дідковська. Київ, 2010. 96 с.

3. Дідковська М. В. Тестування: Основні визначення, аксіоми та принципи. Текст лекцій. Частина І / М. В. Дідковська, Ю. О. Тимошенко. Київ, 2010. 61 с.

4. Ляшко С.Ю. Автоматизоване тестування веб-додатків [Текст]: робота на здобуття кваліфікаційного ступеня магістра; спец.: 051 - економіка / С.Ю. Ляшко; наук. керівник К.Г. Гриценко. - Суми: СумДУ, 2019. 61 с.

5. Потужний І. Р. Дослідження засобів тестування та розробка системи автоматизації тестування веб-додатку : магістерська робота ; спец. 122 “Комп’ютерні науки“ / І. Р. Потужний ; наук. кер. А. Ю. Гайда. Херсон : ХФ НУК, 2020. 99 с.

6. Чорний Р.А. Засоби тестування веб-додатків як запит та вимога сьогодення /Збірник наукових праць студентів та магістрантів Кам’янець-Подільського національного університету імені Івана Огієнка. [Електронний ресурс]. Кам’янець-Подільський: Кам’янець-Подільський національний університет імені Івана Огієнка, 2023. Вип. 17. 315 с. 261-263 С.

7. A. Juntunen, E. Jalonen, and S. Luukkainen, “HTML 5 in Mobile Devices – Drivers and Restraints,” in 2013 46th Hawaii International Conference on System Sciences (HICSS), 2013, pp. 1053–1062.

8. Amitava M. Fundamentals of Quality Control and Improvement / Mitra Amitava, 2016. 816 с.

9. Chester Parrott, Doris Carver, "Lodestone: A Streaming Approach to Behavior Modeling and Load Testing", 2020 3rd International Conference on Data Intelligence and Security (ICDIS), pp.109-116, 2020.

10. Elassy, N. (2015), "The concepts of quality, quality assurance and quality enhancement", Quality Assurance in Education, Vol. 23 No. 3, pp. 250-261. https://doi.org/10.1108/QAE-11-2012-0046

11. Galin D. Software Quality Assurance / Daniel Galin. – Harlow: Pearson Education Limited, 2004. 590 с.

12. Kendo UI, “HTML5 Adoption Fact or Fiction,” Sep. 2012.

13. Leahy, M.J., Thielsen, V.A., Millington, M.J., Austin, B., & Fleming, A. (2009). Quality assurance and program evaluation: Terms, models, and applications. Journal of Rehabilitation Administration, 33(2), 69-82.

14. Software Testing Methodologies [Електронний ресурс]. Режим доступу: https://www.tutorialspoint.com/software_testing_dictionary/test_approach.htm/

15. Test Approach [Електронний ресурс]. Режим доступу: https://smartbear.com/learn/automated-testing/software-testing-methodologies/

16. White/black/grey box-тестування [Електронний ресурс] // QALight – Режим доступу до ресурсу: https://qalight.ua/baza-znaniy/white-black-grey-box-testuvannya/.

ДОДАТКИ

ДОДАТОК А

Код валідатора HTML-документа на мові програмування Python

1. Файл “__init__.py”:

```
"""Validate HTML5 files."""

# flake8: noqa

__version__ = "0.4.2"

from .validator import Validator, JavaNotFoundException
```

2. Файл “cli.py”:

```
"""Command line tool for HTML5 validation. Return code is 0 for valid HTML5.

Arguments that are unknown to html5validator are passed as arguments

to `vnu.jar`.

"""

from .validator import Validator, all_files

import argparse

import logging

import sys

import yaml


from . import __version__ as VERSION
```

```
LOGGER = logging.getLogger(__name__)


def parse_yaml(filename, starter):
    """Parses yaml config file"""
    converted_namespace = vars(starter)
    with open(filename, encoding='utf8') as yaml_file:
        yaml_contents = yaml.safe_load(yaml_file)
    LOGGER.debug(yaml_contents)
    for item in yaml_contents.keys():
        if item == "vnu":
            pass
        converted_namespace[item] = yaml_contents[item]
    extras = [item for item in yaml_contents.get("vnu", [])]
    return argparse.Namespace(**converted_namespace), extras


def main():
    """Main function of html5validator"""
    vnu_help, _ = Validator().run_vnu(['--help'])

    parser = argparse.ArgumentParser(
        description='[v' + VERSION + '] ' + __doc__,
        prog='html5validator',
```

```
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog='''

This html5validator uses vnu.jar to check the files.
It has many options that are documented in the
vnu.jar help below.

================= VNU help ========================
        ''' + vnu_help,
    )
    parser.add_argument('files', nargs='*', default=None,
                        help='specify files to check')

    parser.add_argument('--root', default='.',
                        help='start directory to search for files to validate')
    parser.add_argument('--match', nargs='+',
                        help=('match file pattern in search '
                              '(default: "*.html" or '
                              '"*.html *.css" if --also-check-css is used)'))
    parser.add_argument('--blacklist', type=str, nargs='*',
                        help='directory names to skip in search', default=[])
```

```
parser.add_argument('--show-warnings', dest='errors_only',
                    action='store_false', default=True,
                    help='show warnings and count them as errors')
parser.add_argument('--no-langdetect', dest='detect_language',
                    action='store_false', default=True,
                    help='disable language detection')
parser.add_argument('--no-vnu-stdout', dest='vnu_stdout',
                    action='store_false', default=True,
                    help='do not use --stdout with vnu.jar')
parser.add_argument('--no-asciiquotes', dest='vnu_asciiquotes',
                    action='store_false', default=True,
                    help='do not use --asciiquotes with vnu.jar')
parser.add_argument('--format', choices=['gnu', 'xml', 'json', 'text'],
                    help='output format', default=None)


parser.add_argument('--ignore', nargs='*', default=None,
                    type=lambda s: (s.decode('utf-8')
                                    if isinstance(s, bytes) else s),
                    help='ignore messages containing the given strings')
parser.add_argument('--ignore-re', nargs='*', default=None,
                    type=lambda s: (s.decode('utf-8')
                                    if isinstance(s, bytes) else s),
```

```
                dest='ignore_re',
                help='regular expression of messages to ignore')
parser.add_argument("--config", help="Path to a config file for options")
parser.add_argument('-l', action='store_const', const=2048,
                dest='stack_size',
                help=('run on larger files: sets Java '
                    'stack size to 2048k'))
parser.add_argument('-ll', action='store_const', const=8192,
                dest='stack_size',
                help=('run on larger files: sets Java '
                    'stack size to 8192k'))
parser.add_argument('-lll', action='store_const', const=32768,
                dest='stack_size',
                help=('run on larger files: sets Java '
                    'stack size to 32768k'))

parser.add_argument('--log', default='WARNING',
                help=('log level: DEBUG, INFO or WARNING '
                    '(default: WARNING)'))
parser.add_argument('--log-file', dest="log_file",
                help=("Name for log file. If no name supplied then no "
                    "log file will be created"))
```

```
parser.add_argument('--version', action='version',
                    version='%(prog)s ' + VERSION)
args, extra_args = parser.parse_known_args()

if args.config is not None:
    args, extra_args = parse_yaml(args.config, args)
if args.vnu_stdout:
    extra_args.append('--stdout')
if args.vnu_asciiquotes:
    extra_args.append('--asciiquotes')
if args.match is None:
    args.match = ['*.html']

    # append to match
    if '--also-check-css' in extra_args or '--css' in extra_args:
        args.match.append('*.css')
    if '--also-check-svg' in extra_args or '--svg' in extra_args:
        args.match.append('*.svg')

    # overwrite match
    if '--skip-non-css' in extra_args:
```

```
        args.match = ['*.css']
    if '--skip-non-svg' in extra_args:
        args.match = ['*.svg']


if args.log_file is None:
    logging.basicConfig(level=getattr(logging, args.log))


else:
    logging.basicConfig(level=getattr(logging, args.log),
                        handlers=[
                        logging.FileHandler(f"{args.log_file}.log",
                                            mode="w")])


validator = Validator(ignore=args.ignore,
                      ignore_re=args.ignore_re,
                      errors_only=args.errors_only,
                      detect_language=args.detect_language,
                      format=args.format,
                      stack_size=args.stack_size,
                      vnu_args=extra_args)
if args.files:
    files = args.files
```

```
    else:
        files = all_files(
            directory=args.root,
            match=args.match,
            blacklist=args.blacklist)
    if len(files) == 0:
        LOGGER.error("There are no files to check")
        sys.exit(1)
    LOGGER.info('Files to validate: \n  {0}'.format('\n  '.join(files)))
    LOGGER.info(f'Number of files: {len(files)}')

    error_count = validator.validate(files)
    sys.exit(min(error_count, 255))

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

3. Файл “validator.py”:

```
"""The main validator class."""

import errno
import fnmatch
```

```
import logging
import os
import re
from typing import List, Tuple, Optional
import subprocess
import sys
import vnujar

LOGGER = logging.getLogger(__name__)

DEFAULT_IGNORE_RE: List[str] = [
    r'\APicked up _JAVA_OPTIONS:.*',
    r'\ADocument checking completed. No errors found.*',
]

DEFAULT_IGNORE: List[str] = [
    '{"messages":[]}'
]

DEFAULT_IGNORE_XML: List[str] = [
    '</messages>',
    '<?xml version=\'1.0\' encoding=\'utf-8\'?>',
```

```
    '<messages xmlns="http://n.validator.nu/messages/">'
]


class JavaNotFoundException(Exception):
    """Error raised is there is no Java found"""
    def __str__(self):
        return ('Missing Java Runtime Environment on this system. '
                'The command "java" must be available.')



def all_files(
        directory: str = '.',
        match: str = '*.html',
        blacklist: Optional[List[str]] = None,
        skip_invisible: bool = True) -> List:
    if blacklist is None:
        blacklist = []
    if not isinstance(match, list):
        match = [match]

    files = []
    for root, dirnames, filenames in os.walk(directory):
```

```
    # filter out blacklisted directory names
    for b in blacklist:
        if b in dirnames:
            dirnames.remove(b)
        if b in filenames:
            filenames.remove(b)

    if skip_invisible:
        # filter out directory names starting with '.'
        invisible_dirs = [d for d in dirnames if d[0] == '.']
        for d in invisible_dirs:
            dirnames.remove(d)

    for pattern in match:
        for filename in fnmatch.filter(filenames, pattern):
            if skip_invisible and filename[0] == '.':
                # filter out invisible files
                continue
            files.append(os.path.join(root, filename))

return files
```

```
def _cygwin_path_convert(filepath) -> str:
    return subprocess.check_output(
        ['cygpath', '-w', filepath], shell=False).strip().decode('utf8')



def _normalize_string(s) -> str:
    s = s.replace("“", "")
    s = s.replace("”", "")
    return s

class Validator:


    def __init__(self,
                 ignore=None, ignore_re=None,
                 errors_only=False, detect_language=True, format=None,
                 stack_size=None, vnu_args=None):
        self.ignore = ignore if ignore else []
        self.ignore_re = ignore_re if ignore_re else []

        # java options
        self.stack_size = stack_size
```

```
# vnu options
self.errors_only = errors_only
self.detect_language = detect_language
self.format = format
self.vnu_args = vnu_args

# add default ignore_re
self.ignore_re += DEFAULT_IGNORE_RE

# add default ignore
self.ignore += DEFAULT_IGNORE

# process fancy quotes in ignore
self.ignore = [_normalize_string(s) for s in self.ignore]
self.ignore_re = [_normalize_string(s) for s in self.ignore_re]

# Determine jar location.
self.vnu_jar_location = (vnujar.__file__
                         .replace('__init__.pyc', 'vnu.jar')
                         .replace('__init__.py', 'vnu.jar'))
if sys.platform == 'cygwin':
```

```
        self.vnu_jar_location = _cygwin_path_convert(
            self.vnu_jar_location)

def _java_options(self) -> List[str]:
    java_options = []

    if self.stack_size is not None:
        java_options.append(f'-Xss{self.stack_size}k')

    return java_options

def _vnu_options(self) -> List[str]:
    vnu_options = []

    if self.errors_only:
        vnu_options.append('--errors-only')
    if not self.detect_language:
        vnu_options.append('--no-langdetect')
    if self.format is not None:
        vnu_options.append('--format')
        vnu_options.append(self.format)
    if self.vnu_args is not None:
```

```
        vnu_options += self.vnu_args


    return vnu_options


def run_vnu(self, arguments) -> Tuple[str, str]:
    try:
        cmd = (['java'] + self._java_options()
               + ['-jar', self.vnu_jar_location]
               + arguments)
        LOGGER.debug(cmd)
        p = subprocess.Popen(
            cmd,
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE
        )
        stdout, stderr = p.communicate()
    except OSError as e:
        if e.errno == errno.ENOENT:
            raise JavaNotFoundException()
        else:
            raise
    except subprocess.CalledProcessError as error:
```

```
            raise (error.output.decode('utf-8'))

        return stdout.decode('utf-8'), stderr.decode('utf-8')

    def validate(self, files):
        if sys.platform == 'cygwin':
            files = [_cygwin_path_convert(f) for f in files]

        stdout, stderr = self.run_vnu(self._vnu_options() + files)

        # process fancy quotes into standard quotes
        stdout = _normalize_string(stdout)
        stderr = _normalize_string(stderr)

        err = stdout.splitlines() + stderr.splitlines()

        # Removes any empty items in the list
        err = list(filter(None, err))

        # Prevents removal of xml tags if there are errors
        if self.format == "xml" and len(err) < 4:
            self.ignore = DEFAULT_IGNORE_XML
```

```
LOGGER.debug(err)


for ignored in self.ignore:
    err = [line for line in err if ignored not in line]
for ignored in self.ignore_re:
    regex = re.compile(ignored)
    err = [line for line in err if not regex.search(line)]


if err:
    for line in err:
        print(line)
else:
    LOGGER.info('All good.')
return len(err)
```