НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ

імені адмірала Макарова

Навчально-науковий інститут автоматики та електротехніки

(повне найменування інституту, факультету)

Кафедра комп'ютеризованих систем управління

(повна назва кафедри)

«Допущений до захисту»

Завідувач кафедри

_____

«___» _____ 2022 р.

Пояснювальна записка

до кваліфікаційної роботи

на здобуття першого (бакалаврського) рівня вищої освіти

(освітньо-кваліфікаційний рівень)

на тему: Автоматизація процесів тестування веб-сервісу на основі Java

Виконав студент **4** курсу, **4341** групи спеціальності «**151 - Автоматизація та комп'ютерно-інтегровані технології»**

Бойчук А.М.

(прізвище та ініціали)

Керівник **Герасін О. С.**

(прізвище та ініціали)

Рецензент **Гаврилов С.О.**

(прізвище та ініціали)

м. Миколаїв – 2022 року

АНОТАЦІЯ

Кваліфікаційна робота присвячена розробці фреймворку для автоматизації тестування веб-сервісу на основі Java.

Робота характеризується послідовною структурою. У вступній частині описується основна проблематика кваліфікаційної роботи. В першому розділі розглянуто сучасний стан проблеми автоматизації процесів тестування веб-сервісів, основні завдання автоматизованого тестування керуючих програм, класифікацію різновидів тестування та автоматизації тестування програмного забезпечення, а також проведено аналіз переваг та недоліків існуючих методик тестування. В другому розділі проведено складання документації для тестування веб-сервісів. В третьому розділі розроблено програмні засоби автоматизованого тестування веб-сервісів на основі Java. В четвертому розділі розглянуті питання охорони праці. За результатом проведеної роботи зроблено висновки.

Обсяг кваліфікаційної роботи становить 82 сторінки, робота містить 43 рисунки, 11 таблиць та 16 використаних джерел.

ABSTRACT

Qualification work is devoted to the development of a framework for automating the testing of web services based on Java.

The work is characterized by a consistent structure. The introductory part describes the main issues of qualification work. The first section considers the current state of the problem of automation of web services testing processes, the main tasks of automated testing of control programs, classification of types of testing, automation of software testing and analysis of advantages and disadvantages of existing testing methods. In the second section, documentation for testing web services was compiled. The third section develops software tools for automated testing of web services based on Java. In the fourth section the issues of labor protection were considered. As a result of this work, conclusions were drawn.

The volume of qualifying work is 82 pages, the work contains 43 figures, 11 tables and 16 sources use.

ЗМІСТ

ВСТУП

Програмування сьогодні перейшло з розряду мистецтва, ставши ремеслом для багатьох мільйонів фахівців. На жаль, у такому поспіху розробники часто ігнорують необхідність забезпечення належної якості своїх продуктів, наражаючи цим користувачів на невиправданий ризик.

Забезпечення якості (Quality Assurance) є найширшим з усіх понять і є сукупністю заходів, що охоплюють абсолютно всі етапи розробки, випуску та експлуатації програмного забезпечення. Це активності виконуються на всіх етапах життєвого циклу програмного забезпечення (ПЗ) для забезпечення необхідного рівня якості продукту, що випускається.

Контроль якості (Quality Control) - це дії, що проводяться над продуктом у процесі розробки, для отримання інформації про його актуальний стан: наскільки продукт готовий і чи він відповідає вимогам якості в кожний конкретний проміжок часу.

Тестування програмного забезпечення (Software Testing) - це одна з технік контролю якості, що включає в себе активності з планування тестових дій, дизайну тестів, виконання цих тестів та аналізу отриманих даних. В даний час реальні програмні продукти нерідко розробляються в стислі терміни і при обмежених бюджетах проектів.

Основним аспектом, що доводить актуальність застосування тестування в процесі розробки програмного забезпечення, є мінімізація непередбачуваних витрат як розробника, так і споживача товару. Ці витрати пов'язані з порушенням процесу розробки та застосування програмного продукту, викликаного необхідністю усунення знайдених у програмі помилок – дефектів. Дефекти, виявлені та усунені вже на ранній стадії

розробки, обходяться розробнику та клієнту в десятки і навіть сотні разів дешевші, ніж такі ж, але які вже були виявлені в період комерційного використання продукту.[1]

					151.4341.КР.ПЗ.Р1	
Змн	*Арк*	*№ докум.*	*Підпис*	*Дата*		
Розроб.		*Бойчук А. М..*			ОГЛЯД СУЧАСНОГО СТАНУ ПРОБЛЕМИ АВТОМАТИЗАЦІЇ ПРОЦЕСІВ ТЕСТУВАННЯ ВЕБ-СЕРВІСІВ	*Літ.* / *Арк* 9 / *Аркушів*
Норм. контр		*Вінниченко І.Л.*				НУК ім. адм. Макарова
Керівник		*Герасін О.С.*				
Зав. Каф.		*Черно О.О.*				

РОЗДІЛ 1
ОГЛЯД СУЧАСНОГО СТАНУ ПРОБЛЕМИ АВТОМАТИЗАЦІЇ ПРОЦЕСІВ ТЕСТУВАННЯ ВЕБ-СЕРВІСІВ

QA Automation (автоматизація тестування) – це сучасний напрямок діяльності, який виник з розробки програмного забезпечення. Галузь з'явилася із симбіозу QA-інженерів та розробників, які пишуть код — на стику цих двох спеціальностей народилася така дисципліна.

Потреба автоматизації тестування виникла через постійно зростаючий розмір самих сервісів. Обсяг роботи настільки великий, що жодна команда тестувальників із цим не впорається.

Коли в рік виходить десять релізів дев'ять із них перевіряються чудово, а на десятий — допускаються якісь баги. Начебто люди ті самі, але згодом людський фактор накопичується і проявляє себе в невідповідний момент. Паралельно з цим виросли тестувальники, які захотіли підключити технології, щоб програма перевіряла програму. Суть автоматизації — створити такий програмний комплекс, який братиме код гри, запускатиме на ньому набір тестів і видаватиме результат: чи все гаразд.

Автотести дозволяють тестувати найстабільніші програми скільки завгодно – машина може тестувати в будь-який момент часу, тому можна поставити процес на ніч і вранці побачити результат. При цьому тести дуже добре масштабуються: можна не закуповувати пристрої для тестування, віртуально запустити тест на багатьох платформах і відразу отримати результат.

Завдання автоматизації полягає не у заміні тестувальників. Вона дозволяє звільнити той ресурс QA, зайнятий на прогоні регресу. Зазвичай необхідність витрачати ресурси на старе призводить до того, що нове недотестовано і виходить у гіршій якості, ніж могло б. В результаті користувачі програмного продукту незадоволені.[2]

1.1. **Основні завдання автоматизованого тестування керуючих програм**

Кожен успішний керівник дотримується думки «Заощадив – значить заробив» прагнучи скоротити неявні виробничі витрати. Хоча багато хто з них не помітний на перший погляд, вони відчутно позначаються на фінансовому результаті. До неявних витрат відносяться підвищені витрати через неефективне тестування програм, що розробляються.

Будь-яка більш-менш значуща модернізація програмного коду вимагає повторної перевірки його функціонування. Випробування, що застосовуються для цього, часто є типовими і можуть багаторазово повторюватися в послідовних циклах модернізації програми. Це твердження є правильним і для настільних систем, і для «важких» інформаційних комплексів – від веб-порталів, що грають представницьку роль, до систем фінансового обліку, саме тому дедалі більше керівників усвідомлюють актуальність автоматизованого тестування.

Основна мета автоматизації тестування – скорочення витрат на випробування програми після модернізації. Однотипні перевірки, що періодично повторюються, забирають багато часу в циклі розробки. Автоматизація скорочує етап тестування та звільняє головний ресурс компанії – робочий час спеціалістів. Інша не менш очевидна перевага такого тестування – підвищення якості випробувань, що гарантує надійність продукту. Адже збитки від дефектів, виявлених лише на стадії промислової експлуатації, можуть бути дуже високими. А незадоволеність замовників від них взагалі важко піддається обчисленню.

Досі більшість компаній все ще практикують тестування без використання засобів автоматизації. Іноді для цього формується команда спеціалістів. Але ще частіше тестуванням займаються самі розробники. Вони намагаються виконати всілякі дії користувачів та перевіряють, чи досягнуто в програмі очікуваного результату. Тестування ведеться, здебільшого, з технічного боку на предмет «працює – не працює», безвідносно до очікувань

бізнесу. Така практика ручного тестування, не дозволяє комплексно перевіряти повнофункціональні системи за відведені проектом терміни, що призводить до різноманітних негативних наслідків.

Доцільно навести приклад ситуації із негативними наслідками. Банк пропонує клієнтам скористатися послугою "Електронний гаманець". Нехай через дефект програмного забезпечення клієнту показаний неправильний баланс рахунку. В результаті, цей клієнт навряд чи ще раз скористається цією послугою. Інший приклад. Оператор мобільного зв'язку використовує автоматизовану інформаційну систему, що цілодобово реєструє платежі абонентів. Приховані дефекти цієї системи можуть призвести до великих фінансових втрат.

Саме тоді коли підприємці зтикаються з подібними ситуація, вони починаються задумуватися про автоматизацію тестування. Насамперед, автоматизація дозволяє підвищити надійність програмного забезпечення (далі ПО) та знизити ризик виявлення дефектів на стадії промислової експлуатації. Підвищується точність тестування та можливість знаходити більше дефектів на ранніх етапах. Стає можливим виявляти та усувати вузькі місця продуктивності системи протягом усього життєвого циклу розробки. За допомогою автоматизації можна побачити точну картину продуктивності системи на всіх рівнях, включаючи введення у промислову експлуатацію.

Технологічно автоматизація дозволяє використовувати одні й самі тести багаторазово. Тести виконуються швидше, що дозволяє знизити витрати (вивільняється час, який раніше витрачався на ручне тестування). Багаторазово збільшується тестове покриття. Є можливість проводити перевірки нового або модифікованого додатка на відповідність встановленим вимогам щодо функціональності та продуктивності.

Автоматизація вирішує проблему підготовки вихідних даних для тестування, які є документами, вхідними повідомленнями тощо. Для комплексного тестування необхідно створити значну кількість наборів цих даних. При ручному тестуванні звичайно обмежують вихідні дані, виходячи з

деяких припущень. При автоматизованому тестуванні немає потреби обмежувати набори вихідних даних, що дозволяє охопити набагато більше функцій продукту. Можна проводити тестування одразу усіх тест кейсів і в порівнянні з ручним тестуванням, автоматизовані тести можна провести набагато швидше.

Важливий вид випробувань – тестування навантаження. Проводити його вручну дуже важко. Хоча всім відомі випадки ручного навантажувального тестування, коли команда з 10 осіб одночасно намагалася навантажити систему, але це більше жарт ніж реальність, адже ніхто не буде робити такого в реальному житті. У той же час існує інструментарій для випробування програмних систем під такими ж навантаженнями, як під час експлуатації, а також під піковими навантаженнями. Цей інструментарій дозволяє одночасно емулювати дії будь-якої кількості користувачів у системі. Система, що працює в режимі 24x7 та обслуговує мільйони клієнтів, має бути виключно надійною. Засоби автоматизованого тестування дозволяють виявити межу надійності та стійкості.[3]

1.2. Класифікація різновидів тестування

Тестування програмного забезпечення є невід'ємною частиною циклу розробки програмного забезпечення. Тестування – це те, як можна бути впевненим у функціональності, продуктивності та користувальницькому досвіді. Незалежно від того, проводяться тести - вручну або за допомогою автоматизації, чим раніше і частіше ви зможете проводити тести, тим більша ймовірність того, що визначити помилки, не тільки рятуючи вас та вашу команду від потенційних пожежних навчань пізніше, але й гарантуючи, що програмний додаток був ретельно розглянутий і перевірений, перш ніж він буде перед користувачами. Якщо проблеми переносяться у виробниче середовище, то дорожче і витратніше вони виправлятимуться.

Тестування програмного забезпечення може бути розбите на два різні типи: функціональне та не функціональне тестування. Різні аспекти програми вимагають різних типів тестування, таких як тестування продуктивності,

тестування масштабованості, інтеграційне тестування, унітарне тестування та багато іншого. Кожен з цих типів тестування програмного забезпечення пропонує відмінну видимість у вашому додатку, від коду до досвіду користувача. Давайте докладно розглянемо деякі з найпоширеніших типів тестування програмного забезпечення.

Функціональне тестування. Функціональне тестування проводиться для перевірки критично важливих для бізнесу функцій, функціональності та зручності використання. Функціональне тестування гарантує, що функції програмного забезпечення та функціональні можливості поводяться так, як очікувалося, без будь-яких збоїв. В основному перевіряється вся програма на специфікації, згадані в документі «Специфікація вимог до програмного забезпечення». Типи функціональних тестів включають унітарне тестування, тестування інтерфейсу, регресійні випробування, а також багато з них.

Тестування одиниці. Одноразове тестування фокусується на тестуванні окремих частин/одиниць програмної програми на початку. Будь-яка функція, процедура, метод або модуль можуть бути одиницею для проходження модульного тестування для визначення його правильності та очікуваної поведінки. Унітарне тестування є першим тестуванням, що розробники виконують на етапі розробки.

Інтеграційне тестування. Інтеграційне тестування включає тестування різних модулів програми. Програмне застосування складається з різних підмодулів, які працюють разом для різних функцій. Метою інтеграційного тестування є перевірка спільної інтеграції різних модулів та виявлення пов'язаних з ними помилок та проблем.

Не функціональне тестування. Не функціональне тестування подібно до функціонального тестування, однак основна відмінність полягає в тому, що ці функції перевіряються під навантаженням на продуктивність спостерігача, надійність, зручність використання, масштабованість і т.д. Не функціональні випробування, такі як тестування навантаження та стресу, зазвичай проводяться з використанням засобів автоматизації та рішень, таких

як LoadView. Крім тестування продуктивності, типи не функціонального тестування включають тестування установки, тестування надійності і тестування безпеки.

Тестування продуктивності. Тестування продуктивності - це тип не функціонального тестування, який проводиться для визначення швидкості, стабільності і масштабованості програмного застосування. Як випливає з назви, загальна мета цього тестування полягає у перевірці продуктивності програми щодо різних системних та мережевих критеріїв, таких як використання процесора, швидкість завантаження сторінок, пік обробки трафіку, використання серверних ресурсів тощо. В рамках тестування продуктивності існує кілька інших типів тестування, таких як тестування навантаження та стрес-тестування.

Всі типи тестування зосереджені на надійності та готовності програмних додатків, однак, краще розуміти різницю між ними на деяких прикладах. Допустимо, у вас є веб-сайт електронної комерції / програма зі стандартними функціональними можливостями. Ось кілька прикладів тестування продуктивності, функціонального тестування, інтеграційного тестування та питомого тестування:

Якщо треба перевірити, як сайт працюватиме, коли велика кількість користувачів приходить на нього, наприклад, під час сезону продажів, потрібно виконати тестування завантаженості, які підпадають під категорію тестування продуктивності. Це допоможе вам виявити проблеми швидкості та стабільності та усунути потенційні вузькі місця продуктивності.

Припустимо, що необхідно перевірити вхідні дані та вихід для кожної функціональності, такий як реєстрація, увійти, додати в кошик, виїзд, обробку платежів, записи в базу даних тощо, відповідно до тестових випадків, написаних у документі SRS. У цьому випадку потрібно виконати функціональне тестування.

Якщо треба перевірити функціональність кошика з інтеграцією каси та платіжного модуля, щоб побачити, якщо кількість елементів, доданих до

кошика успішно придбана з правильною оплатою, вам потрібно зробити інтеграційне тестування.

Якщо написали модуль для завантаження продукту і хочете перевірити, чи правильно він і продукти успішно додаються без будь-яких помилок або дефектів, вам потрібно зробити модуль завантаження продукту.

Підсумовуючи, можна сказати, що тестування продуктивності проводиться для перевірки продуктивності веб-сайту. Функціональне тестування проводиться перевірки всіх функціональних можливостей. Інтеграційне тестування проводиться для перевірки взаємодії між різними модулями, а модульне тестування проводиться для перевірки окремих частин коду на правильність.

1.3 Аналіз переваг та недоліків існуючих методик тестування

На основі огляду різновидів тестування можна сформулювати їх переваги та недоліки.

Для тестування продуктивності сильними якостями є наступні:

- оцінка швидкості та масштабованості веб-сайту/додатку;
- визначення вузьких місць покращення продуктивності;
- виявлення помилки, які упускаються з уваги при функціональному тестуванні;
- оптимізація системи та удосконалення функцій;
- забезпечення надійності сайту під великим навантаженням;

Для функціонального тестування характерні наступні позитивні риси:

- переконатися, що веб-сайт/програма не має дефектів;
- забезпечує очікувану поведінку всіх функціональних можливостей;
- гарантує правильність архітектури за необхідної безпеки;
- покращує загальну якість та функціональність;
- мінімізує бізнес-ризики, пов'язані з веб-сайтом/додатком;

Перевагами інтеграційного тестування є:

- гарантує, що всі модулі програм добре інтегровані і працюють, як очікувалося разом;
- виявляє взаємопов'язані проблеми та конфлікти, щоб вирішити їх на ранній стадії, перш ніж вони створять велику проблему;
- перевіряє функціональність, надійність та стабільність між різними модулями;
- виявляє втрачені винятки для покращення якості коду;
- підтримує конвеєр CI/CD;

В свою чергу, тестування одиниці дозволяє отримати наступні покращення:

- раннє виявлення помилок у нещодавно розроблених функціональних можливостях чи функціях;
- зводить до мінімуму витрати на тестування з виявлення проблем на ранніх стадіях;
- покращує якість коду з кращим рефакторингом коду;
- підтримує процес гнучкої розробки;
- спрощує інтеграцію та дозволяє хорошу документацію;

Оскільки всі ці типи тестів покращують функціональні можливості та покращують користувальницький досвід, у них немає жодних недоліків. Єдине, що ви можете розглянути недолік, загалом це час і вартість, пов'язані з тестуванням. Тестування вимагає зусиль та ресурсів, і існує ризик, пов'язаний із неточними результатами випробувань, але не робити веб-сайт / тестування додатків поставить вас у компрометуючий стан, який може завадити вашому бізнесу та репутації значно.

Тестування продуктивності є обов'язковим у всіх середовищах розробки та виробництва, щоб переконатися, що ваш веб-сайт / додаток може витримати очікуване навантаження користувача. Функціональне тестування має бути зроблене з кожною збіркою для перевірки всіх змін та функцій відповідно до специфікацій та вимог. Інтеграційне тестування має бути зроблено при інтеграції нового фрагмента коду з іншим модулем, щоб

переконатися, що немає конфліктів та працювати разом правильно. Одноразове тестування має бути зроблено розробниками щоразу, коли вони роблять написання будь-якого коду для перевірки правильного введення та виведення.[4]

1.4 Автоматизація тестування програмного забезпечення

Безперервне тестування прискорює постачання програмного забезпечення, роблячи весь процес тестування швидшим. А завдяки негайному зворотному зв'язку, який допомагає вже на ранніх етапах виявляти помилки та інші проблеми в додатку, гарантує, що команди розробки будуть створювати високоякісні та надійні програми. Крім того, сама здатність організувати та проводити ефективне тестування може значно знизити витрати в компанії, як за рахунок економії часу розробників, так і внаслідок створення добротного конвеєра поставки, в якому вони можуть швидко вносити зміни до коду з мінімальними ризиками порушення працездатності програми у продуктивному середовищі.

Головним елементом безперервного тестування є його автоматизація, що дає безліч переваг:

- швидке отримання зворотного зв'язку;
- акуратне та ретельне тестування;
- високе покриття тестами;
- швидке виявлення помилок;
- повторне використання тестів;
- коротші терміни постачання;
- адаптація для DevOps;
- економія часу та фінансових витрат;

Незважаючи на перераховані вище переваги, початкові вкладення автоматизації тестування можуть бути дуже високі. Придбання ПЗ, витрати на навчання роботі з ним, проектування та створення автоматизованих тестів – все це потребує чималих часу та грошей. Однак, як тільки ви починаєте все

активніше розробляти нові функції у своєму продукті, ручне тестування зрештою виходить дорожчим, а автоматичне — дешевшим. (рис. 1)

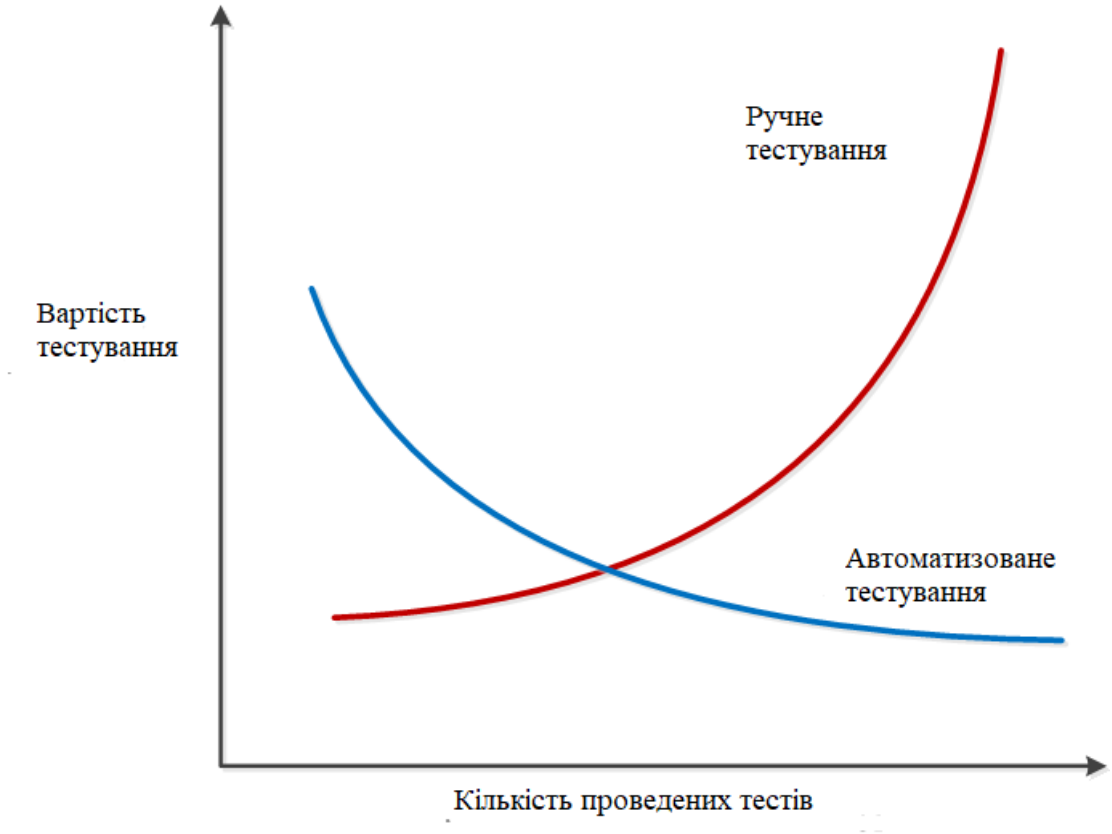


Рис. 1.1 Графік співвідношення вартості ручного та автоматизованого тестування при збільшенні кількості тестів

Незважаючи на все, слід розуміти – не все потрібно автоматизувати і не все можна автоматизувати. Тому важливо ретельно оцінити, вивчити та проаналізувати свої вимоги, перш ніж вирішити, як найкраще організувати автоматизацію тестування.

Практично кожна команда розробників працює над проектом, який критично залежить від термінів, а значить, часу на застосування всіх передових практик завжди не вистачає. Те саме стосується стратегії тестування, оскільки тестування як вид діяльності не завжди є пріоритетом для команд розробки. Потрібно спробувати знайти баланс і зробити правильний вибір залежно від типу програми, часових рамок, використовуваного ПЗ для тестування і наявних ресурсів. Розглянемо важливі типи тестів, які можна автоматизувати.

Модульне тестування. Це відмінний спосіб приступити до автоматизації тестування, оскільки модульні тести спрямовані лише на частину коду, в ході якого він перевіряється на працездатність, і не залежать від інших частин програми. Таким чином, розробники одержують більше інформації про роботу створеної функціональності. Завдяки сучасній культурі тестування багато команд використовують методологію розробки через тестування (test-driven development, TDD), коли вони починають складати тести до написання коду. Таким чином, гарантується якість і коду, і тестів.

Пріоритетні функції. Якщо у вас у плані десятки функцій та стислі терміни на їх розробку, ви можете виділити серед них ті, що мають високу ймовірність збоїв. Тестування подібних функцій слід розпочинати якомога раніше.

Регресійні та інтеграційні тести. Інтеграційні тести використовуються для визначення того, чи працюють окремі модулі у додатку як група, а регресійні тести перевіряють, що функції програми працюють належним чином. Ці два тести зазвичай виконуються після змін/покращень програми, тому тестувальники постійно проводять ці тести. Автоматизація таких тестів зберігає дуже багато часу, вивільняючи його до виконання інших типів тестів.

Навантажувальні тести та тести продуктивності. Для тестів продуктивності та навантажувальних тестів немає альтернативи у вигляді ручного тестування, оскільки необхідно моделювати сотні або тисячі користувачів, що працюють у різних умовах: з-під різних браузерів, у різних часових поясах, що використовують різні операційні системи тощо.

Тестові сценарії, що повторюються. Це дуже важливі тести, які команди розробки змушені запускати мало не постійно. Наприклад, працездатність функції входу в систему - вона забезпечує можливість користуватися програмою, впливаючи на її доступність. Тому краще автоматизувати тестування та заощадити час тестувальників та розробників.

Базова функціональність (димові випробування. На відміну від інших тестів, димові тести не такі складні та відносно легко реалізовані. У цьому проходження цих тестів має вирішальне значення. Вони інформують команди розробки про те, чи правильно працюють базові функції програми, наприклад: чи відкривається вікно входу в програму, чи можуть користувачі увійти в систему, чи доступний API, чи є програма з різних місць і т.п.

Тести які не потрібно автоматизувати. Дедалі більше дізнаючись про переваги автоматизації тестування та глибоко переймаючись ними, можна подумати про те, чому б не автоматизувати взагалі всі тести? Відповідь у вигляді "не потрібно намагатися автоматизувати все" йде врозріз з DevOps-мисленням, в якому явна установка на автоматизацію всього і вся. Перед плануванням автоматизації тестування необхідно врахувати кілька факторів. Ось приклади тестів та сценаріїв, для яких не потрібна автоматизація.

Досвід користувача (UX). Ця область тестування може бути автоматизована. Багато аспектів UX-проектування вимагають ручного, довгого та стомлюючого тестування. Наприклад, коли розробники хочуть зрозуміти, наскільки легко користувачі можуть зареєструватися на веб-сайті, або перевірити, які набори полів дають кращу видимість профілів користувачів. Подібні тести мають бути проведені вручну.

Стадії ранньої розробки. Коли якась функція щойно розробляється, в її код постійно вносяться зміни, а це може ускладнити складання та тесту. На ручне тестування цих функцій потрібно менше часу, тому слід дочекатися стабільної версії.

Функціональність, що не має великої важливості. Автоматизація тестування вимагає часу та зусиль, тому слід автоматизувати тестування не всіх функцій, що розробляються в рамках проекту, а лише найважливіших функцій. Низькопріоритетні можна залишити осторонь та продовжити тестувати їх вручну.

Тести без зрозумілих результатів. Командам розробки необхідно знати очікуваний результат кожного входу функції. Якщо результати незрозумілі,

то автоматизація не надасть необхідних доказів того, що функція працює належним чином.

Тести, які неможливо повністю автоматизувати. Якщо автоматизована половина тесту, а інша половина так і залишилася виконуваною вручну, то це призводить до складності та додаткових витрат, оскільки проведення такого тесту вимагає багато часу, а достовірність його під великим питанням. Було б більш раціонально продовжувати тестування таких функцій вручну.

*Фреймворки автоматизованого тестування***.** У кожній команді розробки та поставки ПЗ група QA відповідає за розробку, впровадження та виконання тестів. Для кожного типу тестування має бути визначений тестовий сценарій, принципи, правила та інструменти для проведення. Фреймворк тестування – це набір цих посібників, інструментів та практик, який допомагає інженерам-тестувальникам ефективно виконувати тестові сценарії.

Існують різні фреймворки для різних цілей тестування. Ось деякі з найпопулярніших типів фреймворків для автоматизованого тестування:

- модульний: програма розділена на окремі модулі, і кожен модуль тестується в ізольованому стані;
- лінійний: складання та виконання тестових скриптів. Тестувальники пишуть тестові сценарії послідовно, виконуючи їх для кожного окремого тест-кейсу;
- бібліотечна архітектура: створений на основі модульного фреймворку тестування, з тією різницею, що містить функції для багаторазового використання;
- тестування, що керується даними: тестові скрипти виконуються і верифікуються на основі даних, які зберігаються в центральному сховищі даних або базі даних (SQL, ODBC-ресурси, csv або xls файли);
- тестування за ключовими словами: у даному фреймворку не обов'язково мати навички програмування, оскільки ключові слова, які

використовуються під час створення тестів, відокремлені від технічного коду. Тестувальнику достатньо мати уявлення про весь набір дій, реалізованих у фреймворку;

- гібридний: комбінація із різних фреймворків;

Головна мета всіх команд розробників програмного забезпечення – забезпечити швидке постачання якісного та надійного програмного продукту. Щоб забезпечити швидкий та ефективний процес постачання, потрібне безперервне тестування. Автоматизація - ключ до того, щоб програмне забезпечення, що розроблялося, могло швидко пройти через всі стадії конвеєра розробки і надати клієнтам свої функції. Однак це не означає, що команди повинні вкладати весь свій час і ресурси в автоматизацію тестування. Команди повинні розуміти, що можна і потрібно автоматизувати, а що не потрібно. Правильний вибір охоплення тестів на ранніх етапах розробки має велике значення.[5]

Висновки за розділом 1

Баланс ручного та автоматизованого тестування дозволяє постійно контролювати якість IT-продукту. Деякі проекти перевіряють вручну, інші завдання можна вирішити за допомогою автоматизації. Тестування вручну використовують у випадках, коли люди незамінні, наприклад, якщо потрібна локалізація, опис помилок, ручна перевірка зручності. У невеликих проектах тести пишуть самі розробники.

Ручне тестування у комплексі з автоматизацією проводять під час створення великих IT-продуктів, де працює кілька команд (наприклад, у банківських додатках), де є складні алгоритми та бізнес-логіка. Цей метод допомагає вибудувати процеси тестування продукту та знизити ризик дорогих помилок, що буває особливо важливим при роботі в умовах жорсткого графіка релізів.

					151.4341.КР.ПЗ.Р2		
Змн	*Арк*	*№ докум.*	*Підпис*	*Дата*			
Розроб.		*Бойчук А. М.*			СКЛАДАННЯ ТЕСТОВОЇ ДОКУМЕНТАЦІЇ ДЛЯ ВЕБ-СЕРВІСУ	*Літ.*	*Арк* 24 / *Аркушів*
Норм. контр		*Вінниченко І.Л.*				НУК ім. адм. Макарова	
Керівник		*Герасін О.С.*					
Зав. Каф.		*Черно О.О.*					

РОЗДІЛ 2
СКЛАДАННЯ ТЕСТОВОЇ ДОКУМЕНТАЦІЇ ДЛЯ ВЕБ-СЕРВІСУ

Тестова документація - це набір документів, що створюються перед початком процесу тестування та безпосередньо в процесі тестування. Ці документи описують покриття тестами та процес виконання тестів, у яких вказуються необхідні для тестування аспекти, наводиться основна термінологія тощо.

У тестовій документації будь-який член команди може знайти повну інформацію про всі дії, пов'язані з тестуванням (і про вже виконані, і про заплановані).

Якщо тестування не документується, це заважає бачити повну картину проекту. Без чітких цілей, покрокового плану їх досягнення і вказівки всіх важливих умов очікуваний результат буде незрозумілий. У таких умовах у всіх може бути різне розуміння загальної мети та кінцевого продукту.

Тестова документація визначає, що для нас важливо і чому, які дії ми маємо виконати і скільки часу ми маємо. Нарешті, у документації зазначено, чого має досягти команда та що сигналізує про закінчення процесу.

І QA-інженери, і клієнти можуть мати на меті отримання на виході додатку, який взагалі не має багів. Але це не досяжна мета, а тільки мрія. Тому має сенс обговорити, що визначатиме кінець фази QA.

Наприклад, можемо протестувати весь функціонал один раз. Або тестувати програму доти, поки не залишиться критичних помилок. Або швидко перевірити критично важливі для бізнесу функції, що дозволить встигнути до кінцевого терміну.

Як правило, спроби обговорити все це у листуванні чи телефонних переговорах є неефективними. Вся справа у людському факторі. У когось просто занадто багато інформації, яку потрібно обробити та запам'ятати. Хтось може неправильно зрозуміти домовленості. В результаті кожного з'являється своя інтерпретація вимог і цілей.

Відсутність документації може вплинути на роботу тестувальників. Це особливо відчувається при роботі зі складними продуктами або при мінливих вимогах.

Нерозуміння того, як і чому має поводитися та чи інша функція, призводить до більшої кількості помилок. Неправильне розміщення пріоритетів може призвести до пропуску багів та надання неповних звітів. Перелік прикладів можна продовжувати.

2.1 Написання тест плану

Тест-план - це документ, який описує всі роботи, які виконуватиме команда тестування на проекті. Він містить ризики, перелік необхідних ресурсів, порядок, опис різних процесів тестування. Тест-план потрібен, щоб дати всім зацікавленим у проекті людям розуміння:

- тип ПЗ, що безпосередньо тестується;
- як саме буде проходити тестування;
- коли треба тестувати;
- ким буде проводитись тестування;
- які ресурси потрібні для процесу тестування;

Тест-план створюють на початковій стадії проекту, коли йде збір вимог, формується технічне завдання, стає зрозумілим обсяг роботи та перелік завдань. Не тільки стейкхолдери, а й головний тестувальник (або просто найдосвідченіший тестувальник на проекті), який пише цей документ, стикається з питаннями. Відповіді на них допомагають прояснити замовнику, які види та рівні тестування потрібні у конкретному випадку. І, власне, «продати» цей сервіс із максимальною прозорістю процесів.

Тест-план покриває все тестування на проекті: функціональне, автоматизоване, тестування безпеки тощо. Загальний документ називають Master Test Plan. Однак у деяких ситуаціях він може покривати лише один або кілька рівнів тестування. Це Level Test Plan.

Без тест-плану звісно можна обійтись, і взагалі Тест-план доцільно писати для тривалих проектів. Якщо ваш проект розрахований на місяць-два,

часу на велику документацію немає і ви впевнені, що не доведеться вводити в проект нових випробувачів, тест-стратегії буде достатньо. Вона може бути як складовою частиною тест-плану, так і окремим документом, в якому описано, як саме буде проводитись тестування.

Наприклад, на одному з проектів розпочали розробку з бекенд-частини, тому фокус тестування був зміщений на API та роботу з базою даних. Відповідні види тестування було відображено у стратегії, яка була частиною тест-плану. Через кілька місяців замовник вирішив додати UI, і тут тест-стратегія була представлена як окремий документ, в якому основний фокус був спрямований на тестування інтерфейсу користувача. Це дозволило акцентувати увагу замовника безпосередньо на змінах у тестових підходах та швидше отримати його згоду на них.

У довгострокових проектах тест-план допомагає вибудовувати довірчі стосунки з клієнтом, показуючи, що саме робитиме команда тестування. Особливо корисно створювати таку документацію, якщо клієнт є новим. Якщо ви вже співпрацюєте із замовником багато років і працюєте над типовими проектами (наприклад, e-commerce), то найчастіше тест-стратегії буде достатньо.

Якщо все-таки необхідно створювати тест-план, одразу треба дізнатися у клієнта: можливо, він має шаблон, якому варто слідувати. Якщо ні, можете використовувати стандарти: IEEE 829 Test Plan, RUP Test Plan.

Без тест-плану можна обійтися, але з ним працюється краще. І ось чому:

- цей документ захищає команду тестування від несподіваних питань клієнта;
- він допомагає лідеру тестувальників на старті проекту зрозуміти, які витрати треба запланувати на розширення або навчання команди, скільки часу; крім того, це хороший гід для ознайомлення новачків із процесами на довгостроковому проекті;

Отже, цей проект не буде затягуватись на пів року чи більше, але для цілісної картини було вирішено, що написання тест-плану просто необхідно. Кожен подібний документ складається із переліку типових розділів, зокрема:

- вступ;
- обсяг роботи;
- критерії сприймання;
- критичні фактори успіху;
- ресурси;
- тестова документація;
- стратегія тестування;
- календар тестування;
- історія документу;

Розпочати написання тест-плану слід зі вступу. У ньому слід описати сам функціонал який буде тестуватися. Так як функціонал не досить великий, щоб у ньому були моменти які не слід протестувати то момент описання чому якийсь функціонал не буде протестовано буде упущено.

Об’єктом тестування для цього проекту буде цифрова бібліотека, розглянемо його типові розділи.

Вступ тест-плану. Цифрова бібліотека - розподілена інформаційна система, що дозволяє надійно зберігати та ефективно використовувати різноманітні колекції електронних документів через глобальні мережі передачі даних у зручному для кінцевого користувача вигляді.

Цифрова бібліотека – це ще й упорядкована колекція різноманітних електронних документів, забезпечених засобами навігації та пошуку. Може бути веб-сайтом, де поступово накопичуються різні тексти (частіше літературні, але також будь-які інші, аж до комп'ютерних програм) та медіа-файли, кожен з яких самодостатній і в будь-який момент може бути затребуваний читачем.

Цифрові бібліотеки можуть бути універсальними, що прагнуть найбільш широкого вибору матеріалу (як Бібліотека Максима Мошкова), і

більш спеціалізованими, як Фундаментальна електронна бібліотека або проект Мережева Словесність, націлений на збирання авторів і типів тексту, що найбільш яскраво заявляють про себе саме в Інтернеті. Походження терміна: " Цифрова бібліотека". Відразу слід зазначити, що термін " Цифрова бібліотека" має вітчизняне походження. Він став широко використовуватися нашій країні як еквівалент англомовного терміна Digital Library.

Author (Автор). Даний розділ необхідний для створення Авторів, Видалення, Оновлення даних а також пошуку авторів по жанрам, книгам, ідентифікатору та одразу усіх.

Таблиця 2.1 – Таблиця адрес які будуть перевірятися для Автору

Функціонал	*Метод*	*Адреса*
Create Author	*POST*	*/api/library/author*
Delete Author	*DELETE*	*/api/library/author*
Get All authors	*GET*	*/api/library/authors*
Get Author by Book	*GET*	*/api/library/book*
Get Authors by Genre	*GET*	*/api/library/genre*
Get Author	*GET*	*/api/library/author*
Update Author	*PATCH*	*/api/library/author*

Book. Даний розділ необхідний для Створення, Видалення, Оновлення книги а також пошуку книги за різними параметрами.

Таблиця 2.2 – Таблиця адрес які будуть перевірятися для Книг

Функціонал	*Метод*	*Адреса*
Create book	*POST*	*/api/library/book*
Delete book	*DELETE*	*/api/library/book*
Get All books	*GET*	*/api/library/books*
Update book	*PATCH*	*/api/library/book*

Genre. Даний розділ написано для Створення, Видалення, Оновлення книги а також пошуку жанру за різними параметрами.

Таблиця 2.3 – Таблиця адрес які будуть перевірятися для Жанрів

Функціонал	*Метод*	*Адреса*
Create genre	*POST*	*/api/library/genre*
Delete genre	*DELETE*	*/api/library/genre*
Get All genre	*GET*	*/api/library/genres*
Update genre	*PATCH*	*/api/library/genre*

Обсяг роботи повинен включати в себе перелік функцій які будуть протестовані.

Наступні розділи не потребують додатково опису так як все що необхідно про них знати буде зрозуміло від самої назви.

Критерії приймання:

• усі необхідні тестові артефакти зібрані та виконані: Checklists та bug reports;

• функціонал не повинен мати відомих багів з пріоритетом Critical, Major та багів, які суттєво впливають на роботу функціоналу до кінця тестування;

Критичні фактори успіху:

- Доступ до документації.
- Доступ до сервісу.
- Доступ до комунікації.
- Доступ до Бази Даних.

Ресурси. Усі ресурси розподіляються на Project manager та Tea, Leader, показані у таблиці 2.5

Таблиця 2.5 - Ключові ресурси проекту

	Роль	*Ім'я*
	Project Manager	*Євген Рибаченко*
	Team Leader	*Ольга Коцюруба*

Таблиця 2.4 – Таблиця обсягу роботи

	Ім'я сервісу	***Назва функції***
	Epam Library	***Library*** • *Створення автора* • *Видалення автора* • *Дістати всіх авторів* • *Дістати автора за книгою* • *Дістати книгу за жанром* • *Дістати книгу за ідентифікатором* • *Оновити автора* • *Створити книгу* • *Видалити книгу* • *Дістати всі книги за жанром* • *Дістати всі книги* • *Дістати книги за жанром і автором* • *Оновити книгу* • *Створити жанр* • *Видалити жанр* • *Дістати всі жанри* • *Дістати жанри за автором* • *Дістати жанри за книгами* • *Оновлення жанру*

Команда тестувальників. Список тестувальника наведений у таблиці 2.6

Тестові інструменти. Список використаних інструментів наведений у таблиці 2.7

Таблиця 2.6 Таблиця тестувальника

	Роль	*Ім'я*	*Локація*	*Зона відповідальності*
	QA	*Адріан Миколайович*	*Миколаїв*	*Написання тестового плану. Написання тест кейсів, Виконання тест кейсів, написання баг репортів а також розробка та підтримка тестового фреймворку.*

Таблиця 2.7 - Тестові інструменти

	Інструмент	*Коментар*
	Jira	*Відстеження багів та документації*
	Google Sheets	*Створення тестової документації та тест кейсів*
	IntelliJ IDEA	*Створення тестового фреймворку*
	GitHub	*Система для контролю версій та зберігання фреймворку*

Тестова документація. Складові частини тестової документації наведені у табл. 2.8

Стратегія тестування. Функціонал буде тестуватися на основі "чорної скриньки" без знання внутрішнього коду, але із розумінням повної структури всього продукту. Застосовуватиметься функціональне та регресійне тестування.

Команда тестування може призупинити тестування, якщо відбудеться будь-яка з наступних дій:

• помилка у функції, яка перешкоджає її перевірці;

• виникла серйозна проблема, яка не дозволяє продовжити тестування;

• критичні проблеми з оточенням;

Таблиця 2.8 - Таблиця тестової документації

#	Назва	Відповідальна особа	Коли використовується	Де знайти
1	Test Plan	Адріан	Перед початком тестування	Google disk
3	Test Cases	Адріан	Перед початком тестування	Google disk
4	Bug report	Адріан	При знаходженні помилок	Jira

Мета тестування полягає в тому, щоб бути впевненим, що функціонал «Цифрова Бібліотека» працює коректно.

Застосовуватимуться такі методи тестування: Manual testing – тестування нового функціоналу та створення документації; Automation Testing – основний метод тестування.

Види тестування

Functional testing – це тип тестування, який перевіряє, що кожна функція сервісу працює відповідно до вимог.

Integration Testing - призначено для перевірки зв'язку між компонентами, і навіть взаємодії з різними частинами системи (операційної системою, устаткуванням чи зв'язку між різними системами).

Рівні тестування

Smoke Test виконується для швидкої оцінки готовності продукту до подальшого більш глибокого та ретельного тестування (Можлива майбутня автоматизація кейсів). Він включає тестування основних функцій функції Fleet Events.

Якщо Smoke Test провалений, команда тестування відправляє повідомлення і зупиняє тестування, доки не буде доступна виправлена версія продукту.

Critical Path Test - знайти помилки, які можуть вплинути на основні функціональні можливості програми, які найбільш важливі для користувачів продукту.

Extended Test's - знайти помилки, пов'язані з нетиповими, але все ще можливими та ймовірними сценаріями використання (наприклад, введення неправильних даних у поля, граничне тестування тощо).

Трекінг багів та іншої документації з функціоналу проводиметься за допомогою Jira, яка міститиме всю необхідну інформацію по знайденим багам та функціоналу. При цьому можна визначити серйозність бага:

- blocker - блокує розробку та/або тестування, подальша робота неможлива;
- critical - збої, втрата даних, серйозний витік пам'яті;
- major – значна втрата функції;
- minor – незначна втрата функції або інша проблема, коли є простий обхідний шлях;
- trivial - Косметична проблема, така як неправильно написані слова або вирівняний текст;

Календар тестування та історія документу наведені у табл. 2.9 і табл 2.10 відповідно.

Таблиця 2.9 - Календар тестування

#	*Активність*	*Початок*	*Кінець*	*Розміщення*	*Запланований час*

1	*Test plan creation*			*Google Disk*	*5 days*
2	*Writing test cases*			*Google Disk*	*5 days*
3	*Testing and writing bug reports*			*Jira*	*2 days*

Таблиця 2.10 - Історія документу

Історія документу					
Версія	*Опис змін*	*Автор*	*Дата зміни*	*Затверджувач*	
				Ім'я	*Дата підтвердження*
1	*Створення тестового плану*	*Адріан*	*08.06.2022*	*Адріан*	*08.06.2022*

2.2 Розробка тест кейсів

Тест-кейс можна порівняти з інструкцією – це послідовність кроків, що призводять до якогось результату. Тест-кейс краще не робити надмірним. Тестувальники найчастіше добре знають свій проект, тому досконало писати тест-кейс немає потреби. Тест-кейс має бути коротким і зрозумілим, так щоб інший тестувальник, або інший спеціаліст у команді зміг швидко пройти по ньому і перевірити, що все відбувається так, як потрібно.

Тест-кейси можна формувати в послідовному сценарії, щоб перевірити, як гравець пройде по цьому функціоналу від початку до кінця. Тест-кейси можна групувати у смислові блоки. Наприклад, якщо в грі запускається якась подія, формується набір тест-кейсів для перевірки цієї події.

Тест-кейси краще писати за вимогами гейм-дизайнерського документа. Але якщо функціонал вже готовий, а вимог тест-кейсів по ньому не написано, написання тест кейсів зайвим не буде.

Стандартний тест кейсу має 5 частин, тобто 5 атрибутів тест кейсу:

- порядковий номер тест кейсу;
- назва тест кейсу. З нього має бути зрозуміло, у чому суть тест кейсу;
- передумови тест кейсу. Це умови, необхідні для проведення тест кейсу. Вони мають бути виконані ще до запуску тест-кейс;

Нехай компанія здає самокати у щохвилину оренду. Потрібно провести тест кейс функції, яка повідомляє користувача про те, що заряд акумулятора самокату. Передумовою тест кейсу буде те, що самокат повинен перебувати у стані оренди[6]

- Порядок дій у тест кейсі та опис дій у тест кейсі
- Очікуваний результат - тест кейс

2.3 Написання чек-листу

Чек-лист - список, що містить низку необхідних перевірок для будь-якої роботи.

Важливість чек листів важко переоцінити. Яким би досвідченим не був співробітник, поспіхом він може легко забути важливу деталь.

У тестуванні чек-лист – це перелік перевірок для тестування продукту. Чек-листи влаштовані дуже просто. Будь-який містить перелік блоків, секцій, сторінок, інших елементів, які слід протестувати, наприклад:

Таблиця 2.11 - Тест кейс написаний під час роботи на проектом

Id	*Case Name*	*Steps*	*Expected result*
BOOK_1	*Search book using author and genre*	*Create new author*	*Author created Successful*
		Create new Genre	*Genre created Successful*
		Create new Book using author Id and genre id from previous steps	*Book created Successful*
		Get book using author if and genre id from previous step	*Returns all books in that genre from author*

Виконані пункти відзначаються статусами, наприклад: “Passed”, “Failed”, “Blocked”, “Skipped”, “Not run”. Ці статуси також можуть мати свій колір. (див. рис. 2.1)

Рис. 2.1 Статуси та їх кольори

Переваги використання чек-листів:

- покращення уявлення про систему загалом, наочність статусу її готовності;
- розуміння обсягу виконаної та майбутньої роботи з тестування;
- уникнути повторення в перевірках і не пропустити нічого важливого в процесі тестування;

Можна виділити два види чек-листів: спеціальні та універсальні.

Спеціальні чек-листи створюються та використовуються для конкретних проектів, тому пункти такого чек-листа відповідають специфіці проекту. Тестувальник за спеціальним чек-листом перевіряє можливість

виконати унікальну дію, передбачену вимогами. Ось приклади пунктів спеціального чек-листа:

- при наведенні курсору на пункт меню "Товари", повинен змінюватися колір на синій. Покажчик повинен змінювати форму на pointer;
- якщо користувач відкрив сторінку "Ваш кошик" і в кошику присутній хоча б один товар, то має відображатися повідомлення;

Такі чек-листи не підходять для використання на інших проектах.

Універсальні чек-листи підходять для тестування проектів одного типу. Перевірка по універсальному чек-листу не прив'язується до графічних елементів або конкретної реалізації, а перевіряється можливість користувача виконати дію. Для універсального чек-листа складається абстрактний перелік перевірок. Пункти універсального чек-листа можуть бути такими:

- користувач може перейти до розділу “Товари”;
- оплата має здійснюватися;
- товар повинен додаватися до кошика;
- посилання при наведенні наголошуються;
- валідатор верстки показує відсутність помилок тощо;

Універсальні чек-листи можна використати повторно на проектах одного типу. Багато агентств є такі універсальні чек-листи, за ними визначається загальний рівень якості продукту.[7]

2.4 Створення баг-репорту

Баг-репорт оформляється, коли баг вже локалізовано і його можна повторити. Якщо баг плаває, потрібно намагатися його повторити або занести в систему, де фіксуються баги, як плаваючий баг. Ключовий момент, що баг можна повторити та відтворити, лише тоді його заносять у систему з багами, де зберігаються баг-репорти. Якщо створити та оформити якийсь баг, і розробник не зможе його відтворити, то тут з'явиться багато запитань. Тому краще відразу перевірити на декількох пристроях, якщо це можливо і подивитися на різних операційних системах, на різних дозволах екрану, тобто максимально локалізувати проблему.[8]

У баг-репорті обов'язково мають бути:

- детальний опис проблеми – що, де, коли сталося;
- важливість дефекту, який показує тестувальник, а вже пріоритет з виправлення цієї помилки показує менеджер або команда з розробників;
- умови відтворення – версія гри, версія операційної системи та інші унікальні умови, які можуть допомогти розробнику швидко знайти баг, усунути та передати завдання на тестування;
- алгоритм відтворення – покрокові передумови, які необхідні для відтворення бага;
- докази – скріни, відео, логи із пристроїв;

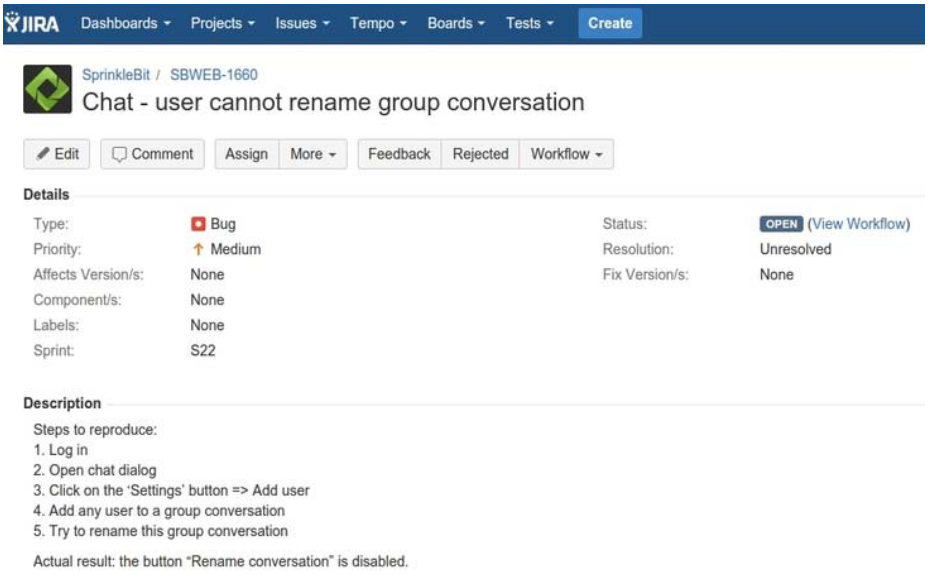


Рис. 2.2 Приклад баг репорту створеного в системі Jira

2.5 Оформлення звіту про тестування

Звіт про тестуванні пишеться, коли функціонал вже перевірено і реліз чи предреліз показує результат виконаної роботи.

Звіт про тестування може бути представлений як текст, таблиця, графік чи діаграма, якщо це дозволяє інструмент.[8]

Складова звіту про тестування:

- хто тестував (склад команди);
- коли тестував (дати проведення тестів);

- як тестував (процес тестування, опис застосовуваних методів та технологій);
- які виникли проблеми під час тестування;

Висновки за розділом 2

Кожна компанія сама визначає, чи варто створювати тестову документацію. QA-фахівці можуть рекомендувати клієнтам це зробити, але останнє слово залишається за клієнтами.

Щодо переваг документування робочого процесу, то вони цілком очевидні. Описані документи допомагають упорядкувати наявну інформацію. Завдяки цьому навіть новачок у команді зможе легко розібратися, що до чого. І хоча створення документації потребує додаткового часу, її відсутність призведе до значно більших часових витрат.

					151.4341.КР.ПЗ.Р3	
Змн	*Арк*	*№ докум.*	*Підпис*	*Дата*		
Розроб.		*Бойчук А. М.*			РОЗРОБКА ПРОГРАМНИХ ЗАСОБІВ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ВЕБ-СЕРВІСУ НА ОСНОВІ JAVA	*Літ.* / *Арк* 41 / *Аркушів*
Норм. контр		*Вінниченко І.Л.*				НУК ім. адм. Макарова
Керівник		*Герасін О.С.*				
Зав. Каф.		*Черно О.О.*				

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНИХ ЗАСОБІВ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ВЕБ-СЕРВІСУ НА ОСНОВІ JAVA

3.1 Особливості автоматизації процесів тестування за допомогою мови програмування Java.

Автоматизоване тестування (АТ) найбільше ефективно, коли реалізовано за допомогою фреймворку. Незважаючи на те, що в АТ термін фреймворк найчастіше використовується для опису сукупності об'єктів, що формує інструмент модульного тестування, тут буде здебільшого сфокусована на фреймворках іншого роду. Розглянемо типи фреймворків, які можуть бути визначені як сукупність абстрактних понять, процесів, процедур та середовищ, за допомогою яких автоматичні тести проектуються, створюються та реалізуються. Крім того, це визначення фреймворку включає фізичні об'єкти, що використовуються для створення тестів та їх реалізації, а також для організації логічної взаємодії між компонентами.

Автоматизоване тестування (і, отже, фреймворки) розвивалося роками, формуючи та ускладнюючись з кожною новою фазою еволюції. Ці фази можуть бути описані в термінах трьох поколінь, кожне з яких має набір недоліків і переваг, завдяки яким кожне з них залишається актуальним, незважаючи на нові розробки.[9]

Сервіси, які необхідно протестувати можна написати на будь-якій мові програмування(МП), наприклад на Java, Python, C#, Assembly, Scala або будь-якою іншою доступною мовою програмування. Фреймворки для автоматизованого тестування на Java автоматизуватимуть ручні тести нічим не гірші за фреймворки на Python. Коли мова заходить про автоматизацію веб-сервісів, немає різниці на якому фреймворку буде проводитись тестування.

Яка МП підійде залежить лише від ситуації. Розберемо плюси та мінуси тих та інших фреймворків.

Плюси Python:

- важко стати на шлях тестувальника, не знаючи Python. Якщо і є така мова, яку потрібно знати, щоб вміти автоматизувати все, то це безумовно Python. Ви можете автоматизувати розгортання оточення, використовувати його для сканування портів або проводити тестування на безпеку;
- у порівнянні з навантаженим синтаксисом Java його до дуже просто використовувати та читати. Крім того, відомий факт, що на один рядок на Python припадає 10 рядків на Java;
- багато інших людей використовують Python з тієї ж причини, що і ви, з цього можна зробити висновок, що хтось вже написав код, який вам потрібен, а ви можете його імпортувати;
- навчання та підтримка. В Інтернеті можна знайти купу навчальних та корисних матеріалів. В цілому, люди прийшли до думки, що матеріали з Python зрозуміліші, ніж з будь-якої іншої мови.

Мінуси Python:

- python створювався, щоб бути простим, універсальним і давати можливість писати скрипти прямо з інтерпретатора, тому він не так добре працює з IDE, як та ж Java. Він настільки простий та універсальний, що IDE не може зрозуміти, що ви робите, коли починаєте створювати об'єкти та передавати їх між методами. Це неприємна особливість, яка може зіграти свою роль, якщо ви дійсно захочете використати IDE для створення свого фреймворку;
- підтримка в офісі. Іноді краще мати локальну підтримку. Якщо у вас в колективі ніхто не знає Python, то отримати невідкладну допомогу у вирішенні питань буде не від кого;

Плюси Java:

- чудові IDE. Одне задоволення писати на Java у такому середовищі, як, наприклад, IDE від IntelliJ. IDE виконує за вас більшу частину роботи, навіть беручи проблеми від складного синтаксису. Функції

авто доповнення коду зроблять за вас величезну кількість роботи, поки вам буде здаватися, що ви набрали на клавіатурі всього пару символів;

- pageFactory. PageFactory Java спрощує код для автоматизації на Selenium і дозволяє писати прості для розуміння тести;
- більшість тестувальників працює з Java-розробниками. Якщо ви з чимось застрягнете, через пару столів від вас завжди буде хтось, хто простягне вам руку допомоги. Це дуже допомагає на лінії навчання, і дає вам переваги знань та досвіду колег. Не встигнете озирнутися, як станете професіоналом;

Мінуси Java:

- непросто читати код Java в порівнянні з простим англійським Python. А ще у Java дуже крута крива навчання, і документація виявляється не завжди корисною. Однак допомогу з багатьох питань можна знайти онлайн (наприклад, StackOverflow);
- проблема з null pointer. Коли Java видає вам повідомлення про помилку і виводить stack trace не завжди просто зрозуміти, в чому справа, і часом ця інформація виявляється зайвою. IntelliJ допомагає там, де може, але незрозумілі повідомлення про помилки можуть перетворити дебаг на розлад.[10]

3.2 Побудова фреймворку для автоматизації тестування веб-сервісу

3.2.1 Вибір інструменту для управління та складання проекту

У світі Java є три основні інструменти складання: Ant, Maven та Gradle. Після декількох років розробки Ant майже зник, Maven також занепадає, а розробка Gradle бурхливо розвивається. Зараз відбувається спад популярності Maven та hike Gradle. Основні функції Maven в основному розділені на 5 пунктів: система управління залежностями, багатомодульна побудова, узгоджена структура проекту, узгоджена модель побудови та механізм модулів, що підключаються. Maven та Gradle мають свої переваги у використанні та використовують їх відповідно до сценарію використання.

Maven необхідно запровадити залежність pom.xml (див. рис. 3.1).

```xml
<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-web -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.1.5.RELEASE</version>
</dependency>
```

Рис. 3.1 Залежності pom.xml

Для роботи з Gradle необхідно запровадити build.gradle (див. рис. 3.2).

```
implementation 'org.springframework.boot:spring-boot-starter-web'
```

Рис. 3.2 Залежності build.gradle

Серед переваг Gradle можна виділити те, шо він має еквівалентні комбінації Maven та Ant. А серед недоліків те, що для посилань на підкласи мікросервісів та мультипроектів він не так гарно структурований, як Maven.

Архітектура проекту з Maven:

- структура/залежність проекту визначається файлом pom.xml;
- виробничий код зберігається в src/main/java;
- тестовий код зберігається в src/test/java;
- процес складання та життєвий цикл:
  - три стандартні життєві цикли (життєвий цикл);
  - найменша одиниця операції – це мета;
  - плагіни можуть пов'язувати свої цілі із певною фазою життєвого циклу;
- управління пакетами та транзитивні залежності:
  - унікальні координати пакета визначаються groupId/artifactId/version;
  - посилання надходить із центрального складу.;
  - транзитивна залежність;
  - коли використання пакета залежить від інших пакетів, Maven автоматично імпортує всі залежні пакети;
- коли є конфлікт залежностей Maven покладається на демодуляцію, щоб слідувати двом принципам:
  - Найближчий шлях;

- порядок визначення;

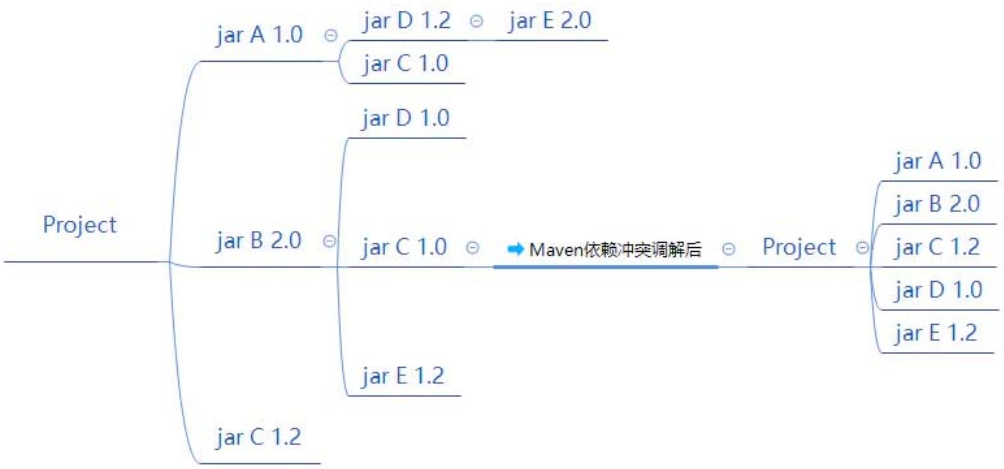


Рис. 3.3 Конфлікт залежностей Maven

Архітектура проекту з Gradle:

- структура / залежність проекту визначаються build.gradle;
- виробничий код зберігається в src/main/java;
- тестовий код зберігається в src/test/java;
- процес складання та життєвий цикл:
  - життєвий цикл не показаний;
  - найменша діюча одиниця - це завдання, і завдання можуть залежати один від одного;
  - може динамічно створювати завдання;
- управління пакетами та транзитивні залежності:
  - використовуйте компонентну систему Ivy, яка є розширеним набором компонентної системи Maven;
  - ant ivy - більший склад, ніж склад Maven;
  - сумісний з репозиторіями Maven;
- коли є конфлікт залежностей:
  - вирішення конфліктів Gradle не є обов'язковим Нова версія (Нові версії зазвичай сумісні);

Підбиваючи підсумки, доцільно дати коротку характеристику фреймворкам.

Maven -

- стабільний та надійний, з безліччю плагінів. (Протягом багатьох років версія підтримувалась на рівні 3.XX, і лише невелике оновлення було випущено протягом тривалого часу, що вказує на її стабільність та меншу кількість помилок);
- трохи багатослівний, логіка, що настроюється, більш проблематична (Maven використовує xml для конфігурації, недоліки xml громіздкі і будуть відображені в Maven(див. рис. 3.4));

Рис. 3.4 Xml файл

- у великомасштабних проектах поступово виникатимуть проблеми із продуктивністю;
- проекти, створені за допомогою Maven, відбуваються через кілька життєвих процесів. Механізму внутрішнього хешування немає, і що довше проект буде перекомпонований, то більше часу це займе;
- оскільки розробка Maven в основному залежить від підтримки спільноти, більше немає засобів для продовження розробки та підтримки Maven, що призводить до припинення розробки;

Gradle:

- gradle використовує логіку коду для побудови, що робить його більш гнучким;
- всередині Gradle є механізм хешування (коли введення та виведення файлу не змінилися, вважається, що це незмінний код, а висновок виконується безпосередньо. Але коли ви змінюєте залежну версію пакета, вона іноді не оновлюється, що також є проблемою з механізмом хешування), який буде працювати швидше;
- активна розробка, надто багато версій;

Отже, в даній роботі буде використовуватись Maven через свою надійність та гнучкість.

3.2.2 Підключення бібліотек до фреймворку на основі Maven

Бібліотека в розумінні Maven є артефактом, необхідним для складання програми.

У кожного артефакту, як зазначалося вище, є groupId, artifactId та version. Потрібно лише вказати maven, що цей артефакт є залежністю проекту. Список залежностей повинен бути обернутий тегом залежностей. Кожна окрема залежність повинна бути обгорнута тегом. Всередині тегу dependency в тегах groupId, artifactId і version необхідно вказати значення відповідних параметрів. Нижче на рисунку 3.5 показано як і які були підключені бібліотеки для проекту.

Залежності підключеного до проекту артефакту maven знайде сам, сам складе список всіх бібліотек, необхідних проекту, включаючи залежності залежностей та залежності залежностей залежностей, сам скачає їх на пристрій, на якому відбувається складання і сам додасть їх до classpath. І це все дуже сильно полегшує роботу в подальшому.[11]

3.2.3 Налаштування логування

Логування – це можливість стежити за процесом виконання бізнес-логіки проекту.

Допустимо, що існує якийсь WEB-проект, і він виконує певну задачу, зараз не важливо що саме. Допустимо це Інтернет магазин, на якому при оформленні замовлення потрібно надіслати на пошту покупцю звіт про його покупку, але поштовий сервер вийшов з ладу, і програмного листа не надійшло.

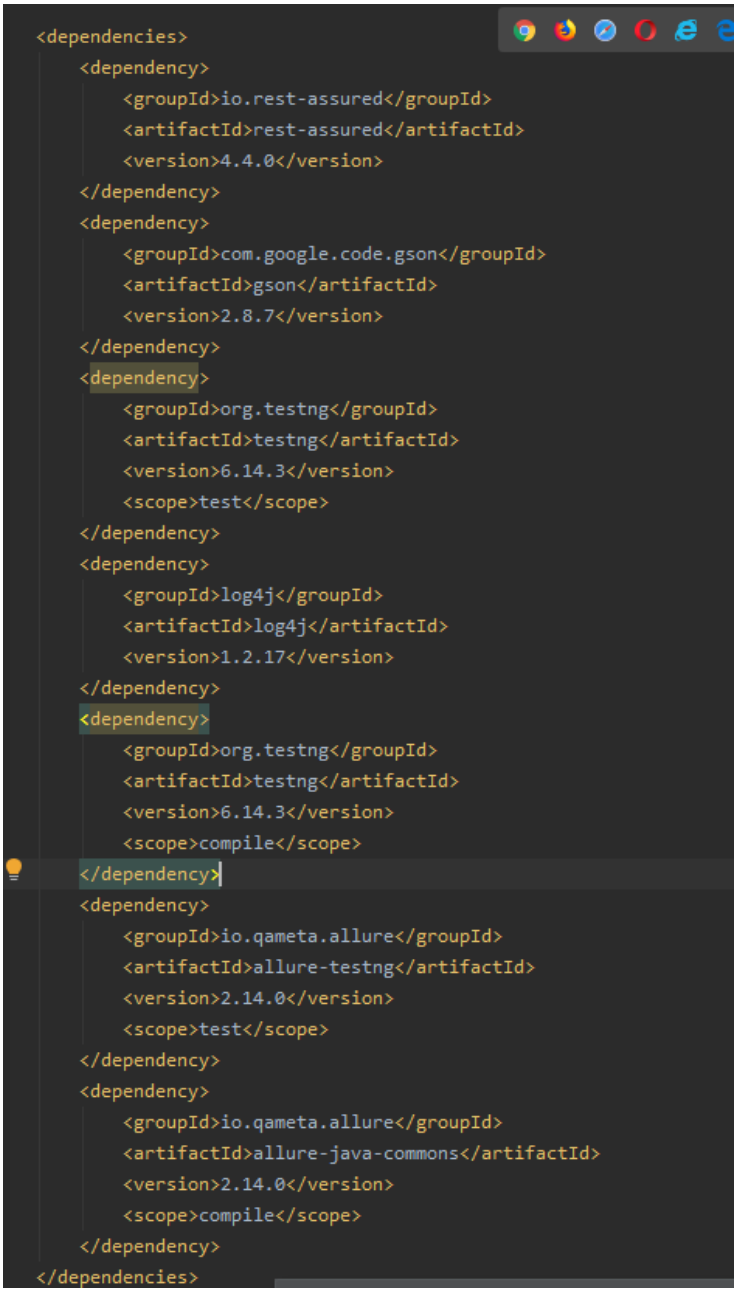


Рис. 3.5 Список підключених бібліотек

Людина, яка адмініструє магазин почне розбиратися в чому ж проблема. Недосвідчена людина довго шукатиме проблему, а досвідчена – відразу піде дивитись логи сервера, але там всі логи сервера і знайти те, що потрібно складно.

В цьому випадку рішення наступне: виводити потрібні вам логи в окремий файл. Але як зрозуміти, які з усіх логів, що приходять в загальний лог сервера, потрібні? Для цього потрібно реалізувати свою систему логування, де можна вказати, які логи куди виводити, або ж налаштувати рівні логування.

Для початку необхідно додати залежність, що ми вже зробили раніше підключивши залежність log4j.(див. рис. 3.6)

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

Рис. 3.6 Залежність log4j.

Щоб гнучко керувати логуванням, варто створити в resources/ файл log4j.properties(див. рис. 3.7):

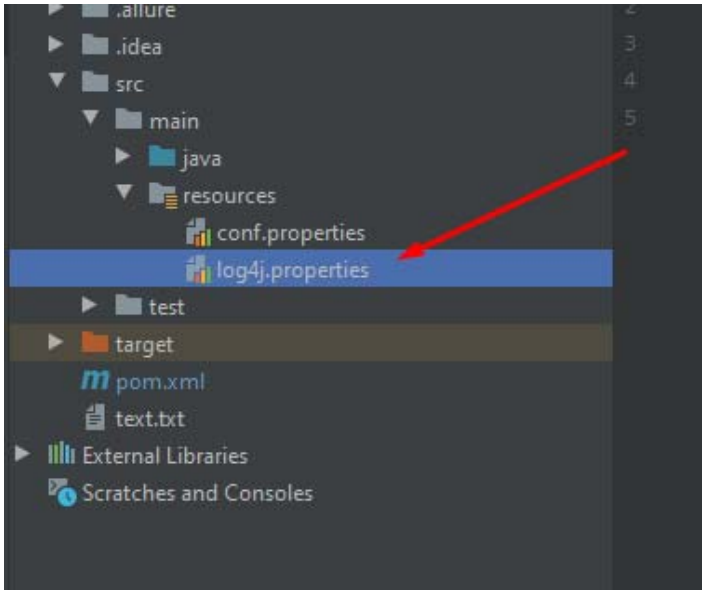


Рис. 3.7 Розташування файлу для налаштування log4j.

Тепер до цього файлу додамо пару рядків конфігурацій(див. рис. 3.8):

```
log4j.rootLogger=INFO,stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm} %-5p %c{1} - %m%n
```

Рис. 3.8 Конфігурація log4j.

Значення:

- %d{yyyy-MM-dd HH:mm:ss} – виводить дату у форматі 2014-01-14 23:55:57;
- %-5p – виводить рівень лога (ERROR, DEBUG, INFO …), цифра 5 означає що завжди використовувати 5 символів решта доповниться пробілами, а мінус (-), що позиціонування з лівої сторони;
- %c{1} – категорія, у дужках вказується скільки рівнів видавати. Так як у нас 1 рівень то писатиметься тільки ім'я класу;
- %m – повідомлення, яке було передано в лог;
- %n – перехід до нового рядка;

Тепер можна додавати логування до проекту. Нижче приклад логування дій(див. рис. 3.9).[12]

```
@Step("Send Request {method} to {endpoint} with body - {body}")
private static Response sendRequest(Method method, String endpoint, String body, String contentType) {
    String url = ServiceConfig.HOST + endpoint;
    RequestSpecification spec = given();
    if(contentType != null) spec.contentType(contentType);
    if(body != null) spec.body(body);
    Response response = spec.request(method, url);
    LOG.info(String.format("Send %s request to %s with body : %s", method, endpoint, body));
    return response;
}
```

Рис. 3.9 Приклад використання log4j.

3.2.4 Налаштування та написання методів для тестування веб-сервісів за допомогою REST Assured.

Rest Assured - це технологія з відкритим вихідним кодом для тестування автоматизації REST API, заснована на бібліотеці на основі java.

Rest Assured взаємодіє з Rest API в режимі безголового клієнта, ми можемо покращити той самий запит, додавши різні рівні для формування запиту та створення HTTP-запиту за допомогою різних дієслів HTTPS на сервері.

Вбудована бібліотека Rest Assured надає величезну кількість методів та утиліт для перевірки відповіді, отриманої від сервера, такого як повідомлення про стан, код стану та тіло відповіді.

Щоб розпочати роботу з REST Assured, необхідно просто додати залежність у фреймворк(див. Рис. 3.10).

```
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <version>4.4.0</version>
</dependency>
```

Рис. 3.10 Залежність log4j.

Так виглядає простий тест, який перевірятиме, що сервіс надає статус 200 OK при зверненні до нього(див. рис. 3.11):

```
given()
    .baseUri("http://cookiemonster.com")
    .when()
    .get("/cookies")
    .then()
    .assertThat()
    .statusCode(200);
```

Рис. 3.11 Простий тест написаний з використанням REST Assured

Ключові слова given, when і then формують запит: given визначає, що буде відправлено в запиті, when - з яким методом і на який ендпоінт відправляємо запит, а then - як перевіряється відповідь. Крім цього, можна отримати тіло відповіді у вигляді об'єкта типу JsonPath або XmlPath, щоб потім використовувати отримані дані.

Реальні тести зазвичай більші та складніші. До запитів додаються заголовки, куки, авторизація, тіло запиту. І якщо API не складається з десятків унікальних ресурсів, кожен з яких вимагає особливих параметрів, треба буде десь зберігати готові шаблони, щоб додавати їх потім до конкретного виклику в тесті(див. рис. 3.12).

```
@Step("Get books with options {options}")
public BaseResponse<Book> getBooks(ListOptions options) {
    EndpointBuilder endpoint = new EndpointBuilder().pathParameter("books");
    if (options.orderType != null) endpoint.queryParam( param: "orderType", options.orderType);
    endpoint
            .queryParam( param: "page", options.page)
            .queryParam( param: "pagination", options.pagination)
            .queryParam( param: "size", options.size);
    if (options.sortBy != null) endpoint.queryParam( param: "sortBy", options.sortBy);
    return new BaseResponse<>(HttpClient.get(endpoint.get(), contentType: null), Book.class);
}
```

Рис. 3.12 Приклад методу написаного за допомогою Rest Assured

3.2.5 Налаштування TestNG

TestNG – це фреймворк для тестування, написаний на Java, він взяв багато чого з JUnit та NUnit, але він не лише успадкувався від існуючої функціональності Junit, а також впровадив нові інноваційні функції, які роблять його потужним, простим у використанні.

TestNG призначений для:

- unit тестування;
- функціональне тестування;
- інтеграційне тестування і т.д.;

Можливості TestNG:

- анотації. (Інструкції);
- використання XML для гнучкого конфігурування тестів;
- підтримка data-driven тестування (за допомогою інструкції @DataProvider);
- залежні методи тестування серверних додатків;
- підтримується в Eclipse, IDEA, Ant, Maven, Netbean, Hudson;
- тестування коду проходить багатопоточно, що дає безпеку та швидкодію;
- легкий перехід від Junit;

Для того щоб почати працювати з TestNG необхідно додати залежності, і створити тестові файли за наступним шляхом: src/test/java/testName (див. рис. 3.13 та 3.14 відповідно).

```
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>6.14.3</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>6.14.3</version>
    <scope>compile</scope>
</dependency>
```

Рис. 3.13 Залежності TestNG

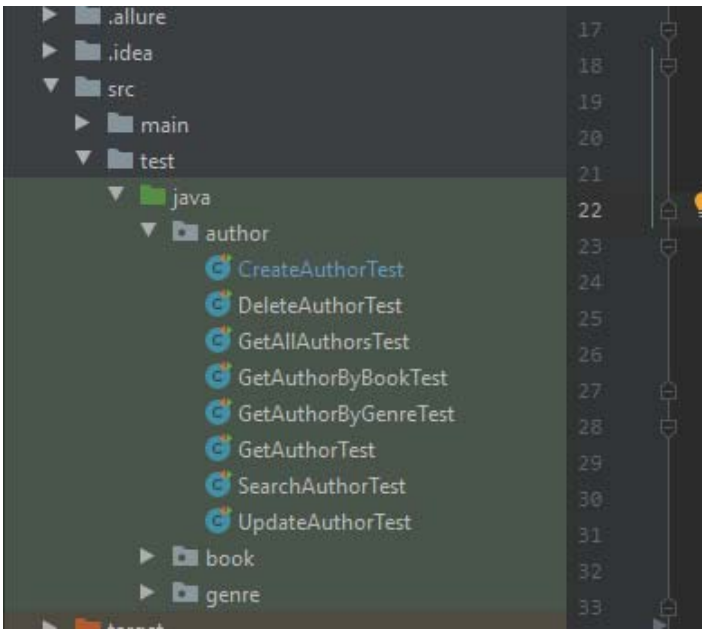


Рис. 3.14 Шлях для створення та зберігання тестів

Кожен тестовий клас виглядає наступним чином(див. рис. 3.15):

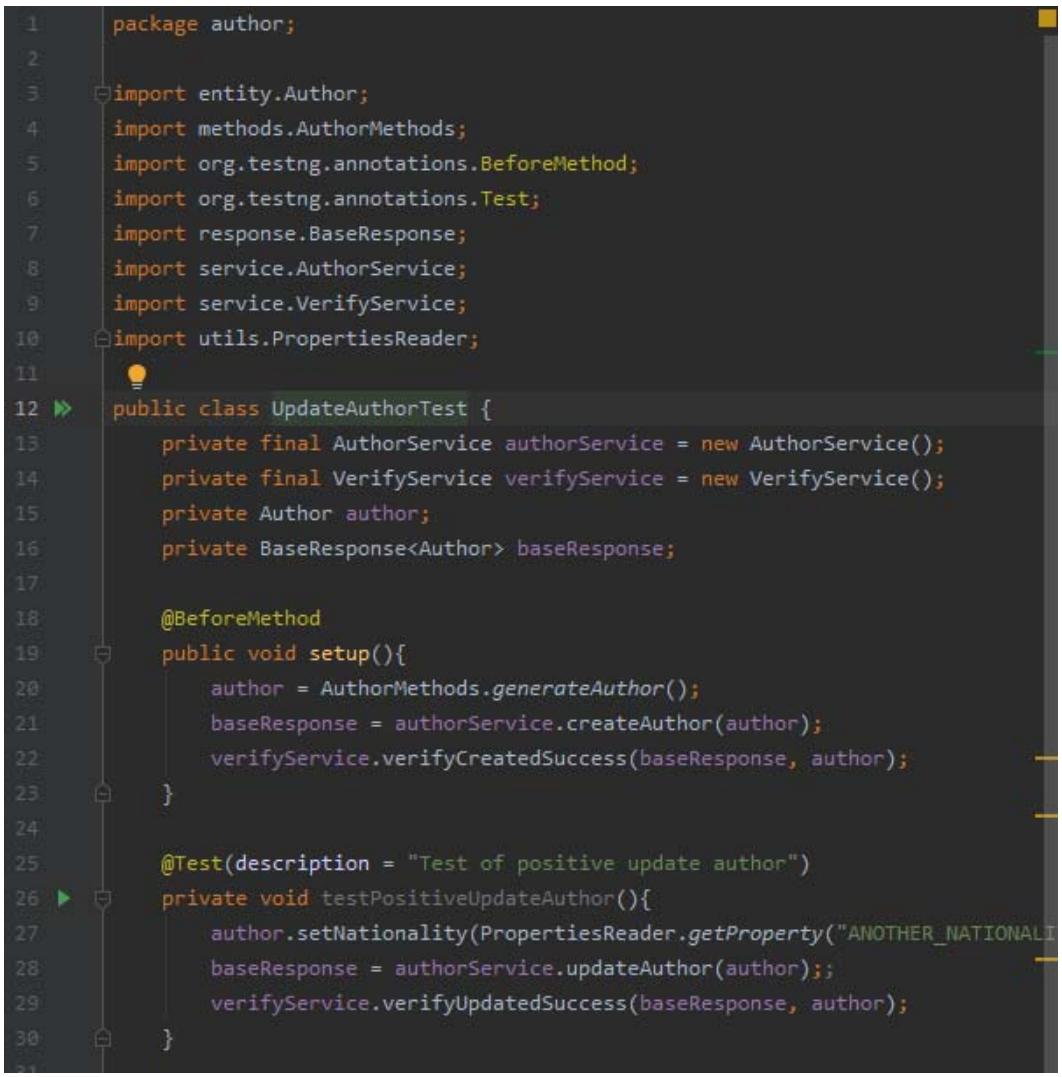
Він складається з наступних блоків:

- package – назва пакету в якому знаходиться клас;
- import – список імпортованих бібліотек які використовуються в класі;
- public class ClassName – блок який і являє собою клас;

TestNG є гнучкішим завдяки своїм анотаціям.

Існує 10 керованих анотацій TestNG:

- @BeforeSuite – вказує, що цей метод запускатиметься перед будь-яким методом тестового класу;
- @BeforeGroups – анотує методи, які будуть виконуватись перед першим методом у будь-якій із зазначених груп;
- @BeforeClass – вказує, що метод буде виконано до всіх тестових методів тестового класу;
- @BeforeTest – анотований метод запускатиметься до всіх тестових методів;
- @AfterTest – анотований метод буде запущений після всіх тестових методів, що належать класам усередині тегу <test>;



```java
package author;

import entity.Author;
import methods.AuthorMethods;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;
import response.BaseResponse;
import service.AuthorService;
import service.VerifyService;
import utils.PropertiesReader;

public class UpdateAuthorTest {
    private final AuthorService authorService = new AuthorService();
    private final VerifyService verifyService = new VerifyService();
    private Author author;
    private BaseResponse<Author> baseResponse;

    @BeforeMethod
    public void setup(){
        author = AuthorMethods.generateAuthor();
        baseResponse = authorService.createAuthor(author);
        verifyService.verifyCreatedSuccess(baseResponse, author);
    }

    @Test(description = "Test of positive update author")
    private void testPositiveUpdateAuthor(){
        author.setNationality(PropertiesReader.getProperty("ANOTHER_NATIONALI
        baseResponse = authorService.updateAuthor(author);;
        verifyService.verifyUpdatedSuccess(baseResponse, author);
    }
```

Рис. 3.15 Приклад написаного тестового класу

- @BeforeMethod – анотований метод виконуватиметься перед кожним тестовим методом;
- @AfterMethod – анотований метод запускатиметься після кожного тестового методу;
- @AfterClass – анотований метод буде запущений після всіх тестових методів у поточному класі;
- @AfterGroups – анотується методи, які будуть виконуватись після всіх методом у будь-якій із зазначених груп;
- @AfterSuite – вказує, що цей метод буде запускатися після всіх методів тестового класу;

У вище показаному прикладі використовується тільки одна анотація, так як не має необхідності в використанні інших.[13]

Відловити очікувану помилку можна за допомогою анотації @Test через параметр expectedExceptions(див. рис. 3.16):

```
@Test(expectedExceptions = NullPointerException.class)
public void testNullPointerException() {
    List list = null;
    int size = list.size();
}
```

Рис. 3.16 Приклад відлову очікуваної помилки

Іноді потрібно проігнорувати тестовий метод, якщо вам, наприклад, в черговому тестуванні не потрібно виконувати даний тестований метод.

Щоб проігнорувати тестовий метод, вам достатньо вказати в анотації @Test параметрі enabled = false.

Якщо потрібно обмежити час проведення тесту для певного тестового методу, то вам на допомогу приходить параметр timeOut анотації @Test(див. рис. 3.17).

```
public class TestNGTimeTest {
    @Test(timeOut = 1000)
    public void waitLongTime() throws Exception {
        Thread.sleep(1001);
    }
}
```

Рис. 3.17 Приклад використання timeOut

В даному випадку тестовий метод буде завалений, тому що ліміт часу на його виконання перевищує заданий ми змушуємо потік чекати 1001 мілісекунду.[14]

3.2.6 Підключення репортів Allure

Allure Framework – популярний інструмент побудови звітів автотестів, що спрощує їх аналіз. Це гнучкий та легкий інструмент, який дозволяє отримати не лише коротку інформацію про хід виконання тестів, але й надає всім учасникам виробничого процесу максимум корисної інформації із повсякденного виконання автоматизованих тестів.

Розробникам та тестувальникам використання звітів Allure дозволяє скоротити життєвий цикл дефекту: падіння тестів можуть бути поділені на

дефекти продукту та дефекти самого тесту, що скорочує витрати часу на аналіз дефекту та його усунення. Також до звіту можуть бути прикріплені логи, позначені тестові кроки, додані вкладення з різноманітним контентом, отримана інформація про таймінги та час виконання тестів. Крім того, Allure-звіти підтримують взаємодію з системами безперервної інтеграції та баг-трекінговими системами, що дозволяє завжди тримати під рукою потрібну інформацію про проходження тестів та дефекти.

Тест-менеджерам Allure дає загальне уявлення про працездатність проекту, дозволяє зрозуміти, які фічі проекту покриті тестами, як згруповані дефекти, яким є загальний тренд якості проекту.

Після формування Allure-звіт є легкочитаний інтерактивний HTML-документ, який може бути збережений або відправлений поштою.

Щоб отримати інформацію про виконання автотесту, з якої потім буде сформовано інформативний Allure-звіт, необхідно до проекту з автотестами підключити адаптер. Під час виконання тестів адаптер збере всю необхідну інформацію: час початку тесту, його закінчення, з якою помилкою тест впав тощо. Після проходження кожного тесту адаптер збере всі дані та агрегує їх у json файли та файли ресурсів. З множини таких файлів надалі і буде сформовано сам звіт.

Для різних тестових фреймворків потрібні різні адаптери

Allure-TestNG

Даний адаптер дозволяє збирати інформацію з тестів, розроблених на TestNG. Для підключення адаптера необхідно модифікувати pom.xml проекту. До секції properties додаємо (див. рис. 3.18):

```
<properties>
    <maven.compiler.source>8</maven.compiler.source>
    <maven.compiler.target>8</maven.compiler.target>
    <aspectj.version>1.8.10</aspectj.version>
</properties>
```

Рис. 3.18 Секція properties

До секції dependencies додаємо (див. рис. 3.19):

Рис. 3.19 Секція dependencies

Також необхідно модифікувати (або створити) секцію build наступним чином(див. рис. 3.20):

Рис. 3.20 Секція build

Для складання html-звіту необхідно додати в pom.xml проекту allure-maven плагін. Для цього модифікуємо секцію build наступним чином (див. рис. 3.21):

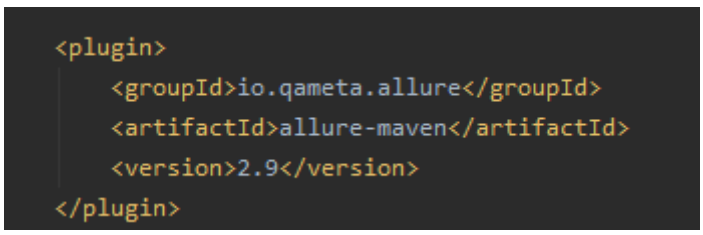


Рис. 3.21 Секція build.plugin

Тепер необхідно просто виконати команду mvn allure:serve в терміналі і репорт автоматично згенерується та відкриється у браузері.

Огляд сторінок звіту

- Сторінка "Overview" (див. рис. 3.21).

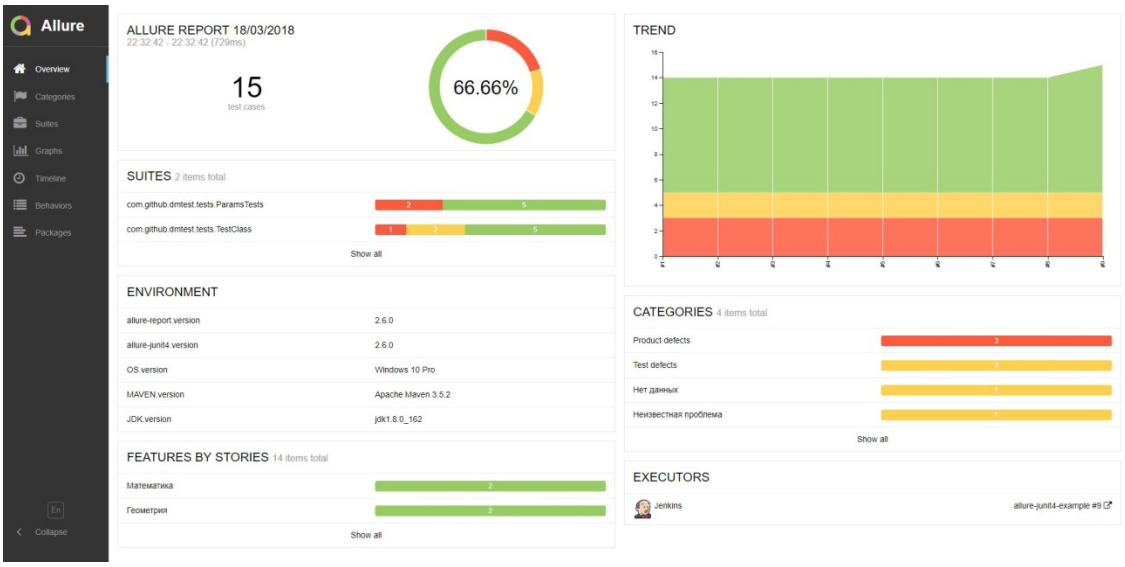


Рис. 3.21 Сторінка "Overview"

Overview є головною сторінкою Allure-звіту. Вона має блокову структуру. Розглянемо блоки, які є на головній сторінці за замовчуванням:

- блок ALLURE REPORT. Включає в себе дату та час проходження тесту, загальну кількість прогнаних кейсів, а також діаграму із зазначенням відсотка та кількості успішних, що впали та зламалися в процесі виконання тестів;
- блок TREND. Показує тренд проходження тестів від складання до складання;
- блок SUITES. Показує розподіл результатів тестів за тестовими наборами. У цьому випадку тести було розподілено за пакетами;
- блок ENVIRONMENT. Показує тестове оточення, у якому запускалися тести. Ця інформація потрапляє до звіту зі спеціального файлу, розташованого в проекті;
- блок CATEGORIES. Показує розподіл неуспішних тестів за видами дефектів;

- блок FEATURES BY STORIES. Показує розподіл тестів за функціоналом, який вони перевіряють;
- блок EXECUTORS. Показує виконавця поточного складання. Якщо виконання здійснювалося на інструменті CI (наприклад, на Jenkins), то буде надана інформація про джоб і номер складання;

- сторінка "Categories"

Дана сторінка надає інформацію про розподіл дефектів за їхніми видами (див. рис. 3.22);

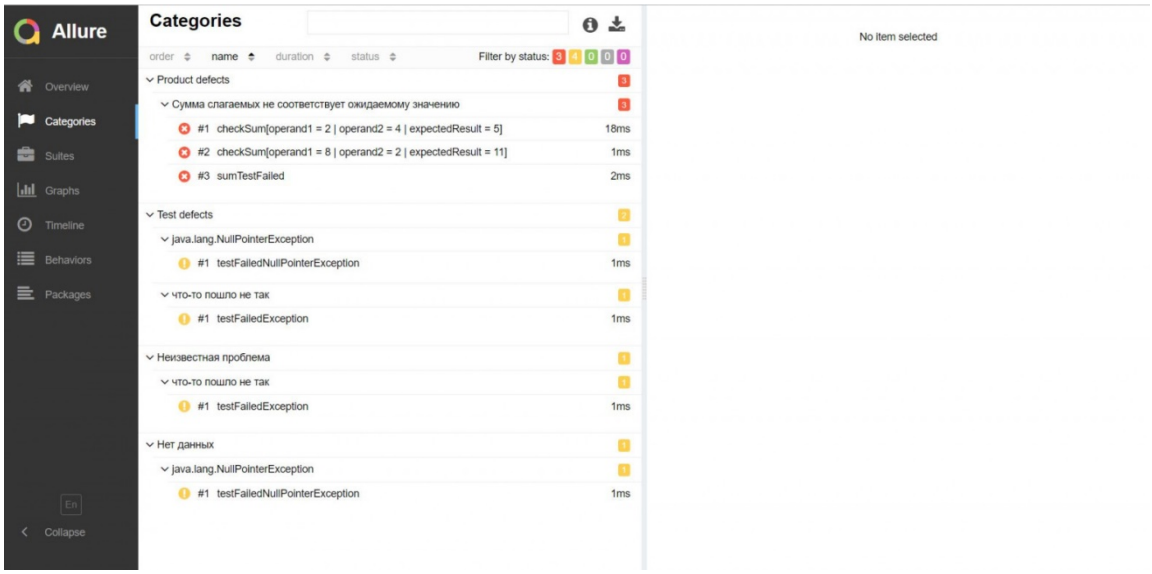


Рис. 3.22 Сторінка " Categories "

- сторінка "Suites"

На цій сторінці подається стандартний розподіл тестів, що виконувались за тестовими наборами або класами, в яких знаходяться тестові методи (див. рис. 3.23);

- Сторінка "Graphs"

На цій сторінці можна отримати інформацію про тестовий прогін у графічному вигляді: статус прогону, розподіл тестів за їх критичністю, тривалість проходження, перезапуски, категорії дефектів і так далі (див. рис. 3.24);

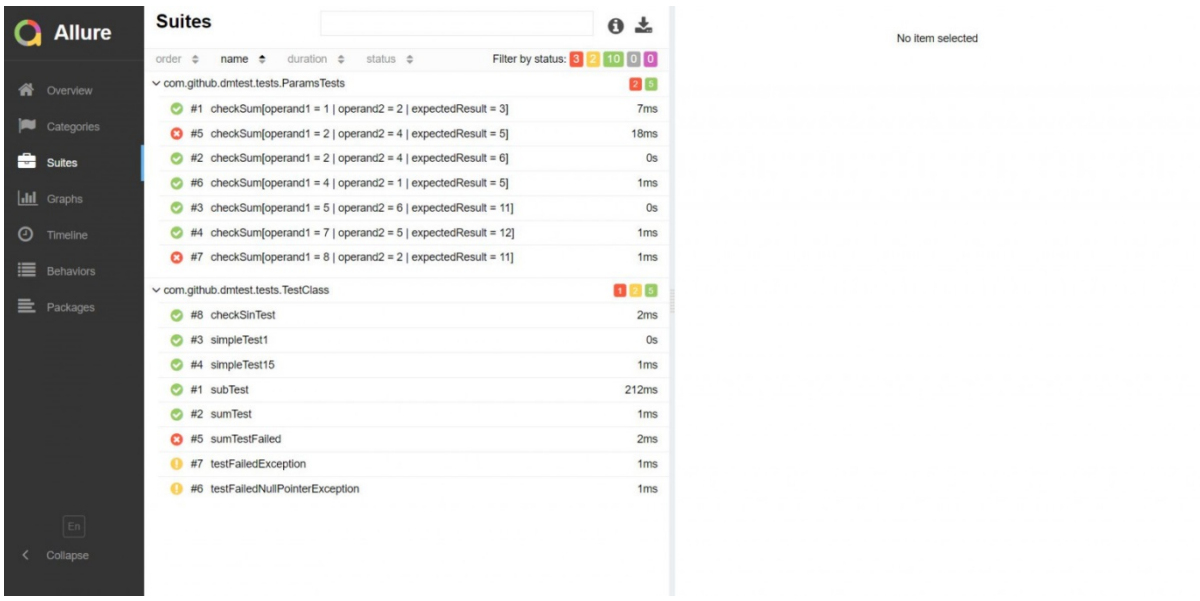


Рис. 3.23 Сторінка " Suites "

Рис. 3.24 Сторінка " Graphs "

- сторінка "Timeline"

У разі запуску тестів у паралельному режимі дана сторінка візуалізує тимчасові рамки проходження кожного тесту. (див. рис. 3.25 та 3.26 відповідно);

- сторінка "Behaviors"

На даній сторінці тести згруповані за функціоналом, що перевіряється (Epic, Feature, Story) (див. рис. 3.27);

Рис. 3.25 Сторінка " Timeline "

Рис. 3.26 Сторінка " Timeline " у паралельному режимі.

- Сторінка "Packages"

На цій сторінці тести згруповані за пакетами, де лежать тестові класи (див. рис. 3.28);

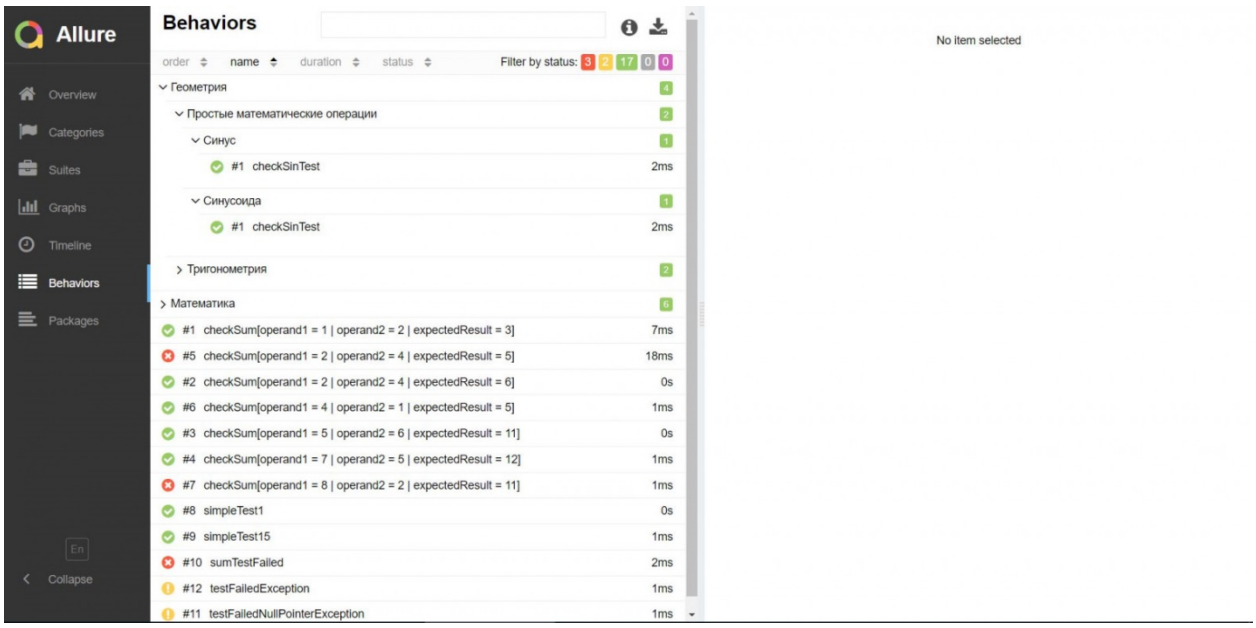


Рис. 3.27 Сторінка " Behaviors "

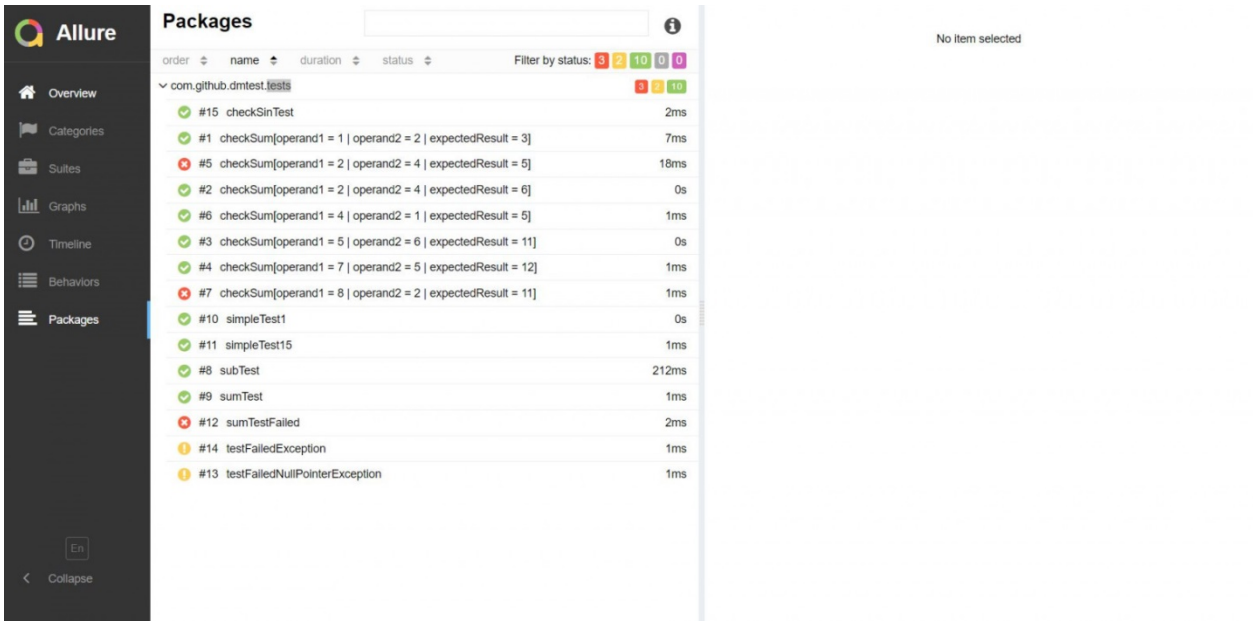


Рис. 3.28 Сторінка " Packages "

3.3 Аналіз отриманих результатів роботи готового фреймворку для автоматизації веб-сервісів

Фреймворк використовує Maven для збірки та описання структури в файлах, TestNG для гнучкого конфігурування тестів, log4j для логування, rest assured для комунікації фреймворку з веб-сервісом, та allure для генерування репортів.

Структура проекту має наступний вигляд (див. рис. 3.29):

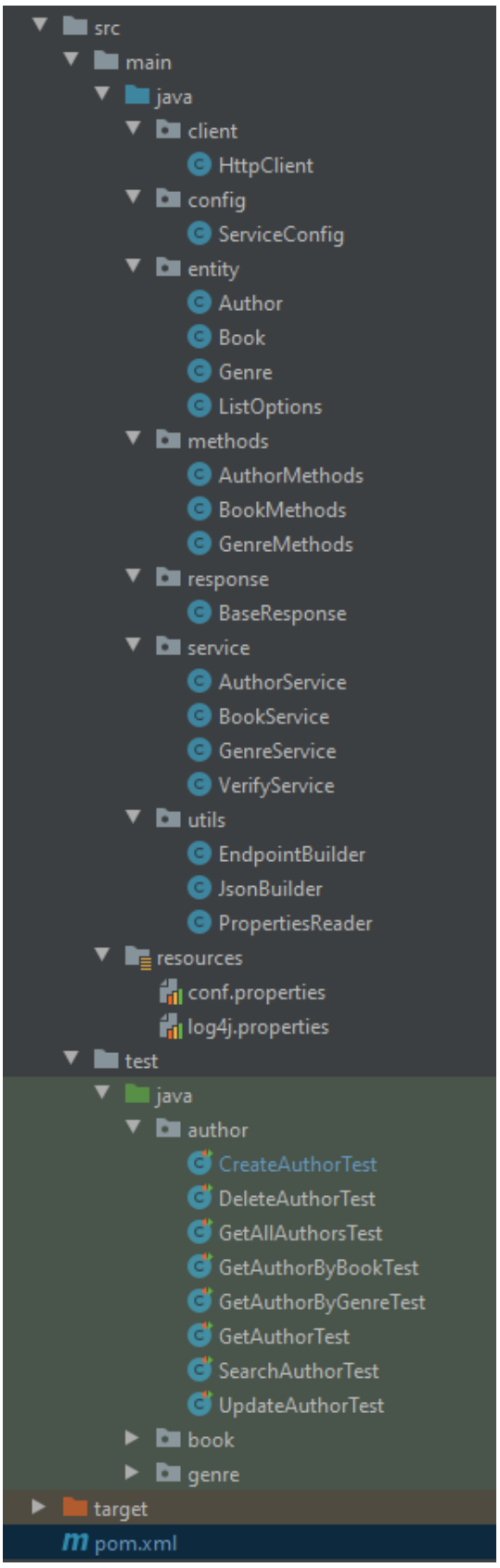


Рис. 3.29 Структура проекту

- client – клас який зберігає в собі основні методи для комунікації з сервісом (див. рис. 3.30);

```java
public class HttpClient {
    private static final Logger LOG = Logger.getLogger(HttpClient.class);

    @Step("Get {endpoint}")
    public static Response get(String endpoint, String contentType) {
        return HttpClient.sendRequest(Method.GET, endpoint, contentType);
    }

    @Step("Get {endpoint}")
    public static Response get(String endpoint, String contentType, String test) {
        return HttpClient.sendRequest(Method.GET, endpoint, contentType);
    }

    @Step("Post {endpoint} with body - {body}")
    public static Response post(String endpoint, String body, String contentType) {
        return HttpClient.sendRequest(Method.POST, endpoint, body, contentType);
    }

    @Step("Put {endpoint} with body - {body}")
    public static Response put(String endpoint, String body, String contentType) {
        return HttpClient.sendRequest(Method.PUT, endpoint, body, contentType);
    }

    @Step("Delete {endpoint}")
    public static Response delete(String endpoint, String contentType) {
        return HttpClient.sendRequest(Method.DELETE, endpoint, contentType);
    }
```

Рис. 3.30 Клас з методами для комунікації з клієнтом

- config – класс для створення хосту з сервісом(див. рис. 3.31);

```java
package config;

import utils.PropertiesReader;

public class ServiceConfig {
    public static final String HOST = ServiceConfig.getHost();

    private static String getHost() { return PropertiesReader.getProperty("HOST"); }
}
```

Рис. 3.31 Клас для створення хосту з сервісом

- entity – пакет який містить в собі класи для створення об’єктів(див. рис. 3.32);

```java
public class Genre {
    private Integer genreId;
    private String genreName;
    private String genreDescription;

    public Integer getGenreId() { return genreId; }

    public Genre setGenreId(Integer genreId) {
        this.genreId = genreId;
        return this;
    }

    public String getGenreName() { return genreName; }

    public Genre setGenreName(String genreName) {
        this.genreName = genreName;
        return this;
    }

    public String getGenreDescription() { return genreDescription; }

    public Genre setGenreDescription(String genreDescription) {
        this.genreDescription = genreDescription;
        return this;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Genre genre = (Genre) o;
        return Objects.equals(genreId, genre.genreId) && Objects.equals(genreName, genre.genreName) && Objects.equals(genreDescription, genre.genreDescription);
    }

    @Override
    public int hashCode() { return Objects.hash(genreId, genreName, genreDescription); }

    @Override
    public String toString() {
        return "Genre{" +
                "genreId=" + genreId +
                ", genreName='" + genreName + '\'' +
                ", genreDescription='" + genreDescription + '\'' +
                '}';
```

Рис. 3.32 Приклад одного з класів для створення об’єктів

- methods - пакет який містить в собі класи з методами які використовуються для того щоб не повторювати великі шматки коду (див. рис. 3.33);

```java
public class GenreMethods {
    private static final Logger LOG = Logger.getLogger(AuthorMethods.class);

    @Step("Generating author object")
    public static Genre generateGenre(){
        boolean isRight = false;
        int id;
        do {
            id = (int) (Math.random() * 100000);
            LOG.info(String.format("Get id for genre: '%s'", id));
            BaseResponse<Genre> baseResponse = new GenreService().getGenre(id);
            if(id != Integer.parseInt(PropertiesReader.getProperty("NOT_FOUND_ID")) && baseResponse.getStatusCode() == 404){
                LOG.info("ID is created successful!");
                isRight = true;
            }
        } while (!isRight);

        return new Genre().setGenreId(id)
                .setGenreName(PropertiesReader.getProperty("GENRE_NAME") + id)
                .setGenreDescription(PropertiesReader.getProperty("GENRE_DESCRIPTION"));
    }
}
```

Рис. 3.33 Приклад одного з таких класів

- response – пакет який містить клас для обробки відповідей сервісу (див. рис. 3.34) ;

```java
public class BaseResponse<T> {
    protected Response response;
    private Class<T> responseClass;
    private static final Logger LOG = Logger.getLogger(BaseResponse.class);

    public BaseResponse(Response response, Class<T> responseClass) {
        this.response = response;
        this.responseClass = responseClass;
    }

    @Attachment
    public int getStatusCode() {
        int code = this.response.getStatusCode();
        LOG.info(String.format("Get status code - %s", code));
        return code;
    }

    @Attachment
    public String getErrorMessage(){
        String message = this.response.getBody().path( s: "errorMessage");
        LOG.info(String.format("Get error message - %s", message));
        return message;
    }

    @Attachment
    public String getHeader(String header) {
        String headerResult = this.response.getHeader(header);
        LOG.info(String.format("Get header - %s", headerResult));
        return headerResult;
    }
```

Рис. 3.34 Приклад методів які використовуються для обробки відповіді

- service – пакет з класами які збирають в собі усі попередні класи і методи для виконання певних дій(див. рис. 3.35);

```java
public class BookService {
    @Step("Get book with id {bookId}")
    public BaseResponse<Book> getBook(Integer bookId) {
        String endpoint = new EndpointBuilder().pathParameter("book").pathParameter(bookId).get();
        return new BaseResponse<>(HttpClient.get(endpoint,  contentType: null), Book.class);
    }

    @Step("Get books with options {options}")
    public BaseResponse<Book> getBooks(ListOptions options) {
        EndpointBuilder endpoint = new EndpointBuilder().pathParameter("books");
        if (options.orderType != null) endpoint.queryParam( param: "orderType", options.orderType);
        endpoint
                .queryParam( param: "page", options.page)
                .queryParam( param: "pagination", options.pagination)
                .queryParam( param: "size", options.size);
        if (options.sortBy != null) endpoint.queryParam( param: "sortBy", options.sortBy);
        return new BaseResponse<>(HttpClient.get(endpoint.get(),  contentType: null), Book.class);
    }
```

Рис. 3.35 Приклад одного з таких класів

- utils – пакет з класами різних інструментів які використовувались для роботи з сервісами (див. рис. 3.36)

```
public class JsonBuilder {
    public static String parseToJson(Object object){
        GsonBuilder builder = new GsonBuilder();
        Gson gson = builder.create();
        return gson.toJson(object);
    }
}
```

Рис. 3.36 Приклад одного такого інструменту

- resources – папка яка зберігає в собі файл conf.properties (набір даних які використовує фреймворк) та log4j.properties (файл конфігурації логера) (див. рис. 3.37);

```
GENRE_NAME=genreName
GENRE_DESCRIPTION=goodGenre
GENRE_ANOTHER_DESCRIPTION=badGenre

BOOK_NAME=BookName
BOOK_LANGUAGE=Ukrainian
BOOK_DESCRIPTION=goodDescription
BOOK_ANOTHER_DESCRIPTION=badDescription
BOOK_PAGE_COUNT=756
BOOK_HEIGHT=10
BOOK_WIDTH=20
BOOK_LENGTH=30
```

Рис. 3.38 Приклад даних з файлу conf.properties

- test.java – Пакет з файлами класів, для виконання тестів (див. рис. 3.39);

```
public class CreateAuthorTest {

    private final AuthorService authorService = new AuthorService();
    private final VerifyService verifyService = new VerifyService();
    private Author author;
    private BaseResponse<Author> baseResponse;

    @BeforeMethod
    public void setup() {
        author = AuthorMethods.generateAuthor();
        baseResponse = authorService.createAuthor(author);
    }

    @Test(description = "Test of positive creating author")
    private void testPositiveScenario() {
        verifyService.verifyCreatedSuccess(baseResponse, author);
    }

    @Test(description = "Test of Author with such id already exists")
    private void testCreateAuthorWithExistsId() {
        baseResponse = authorService.createAuthor(author);
        verifyService.verifyEntityAlreadyExist(baseResponse, PropertiesReader.getProperty("ENTITY_AUTHOR"));
    }

    @Test(description = "Test of Author without body")
    private void testCreateAuthorWithoutBody() {
        baseResponse = authorService.createAuthor(null);
        verifyService.verifyRequestWithoutBody(baseResponse);
    }
}
```

Рис. 3.39 Приклад класу з тестами

- pom.xml – файл який зберігає в собі усі налаштування для maven;

Усього під час роботи над проектом було автоматизовано 76 тест кейсів. (див. рис. 3.40 та 3.41 відповідно)

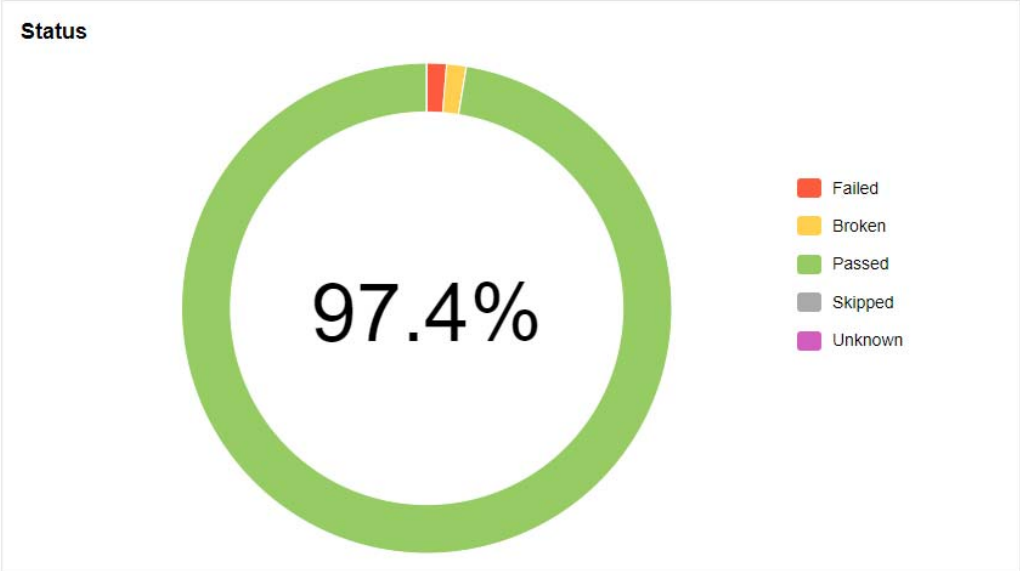


Рис. 3.40 Результат виконання тестів

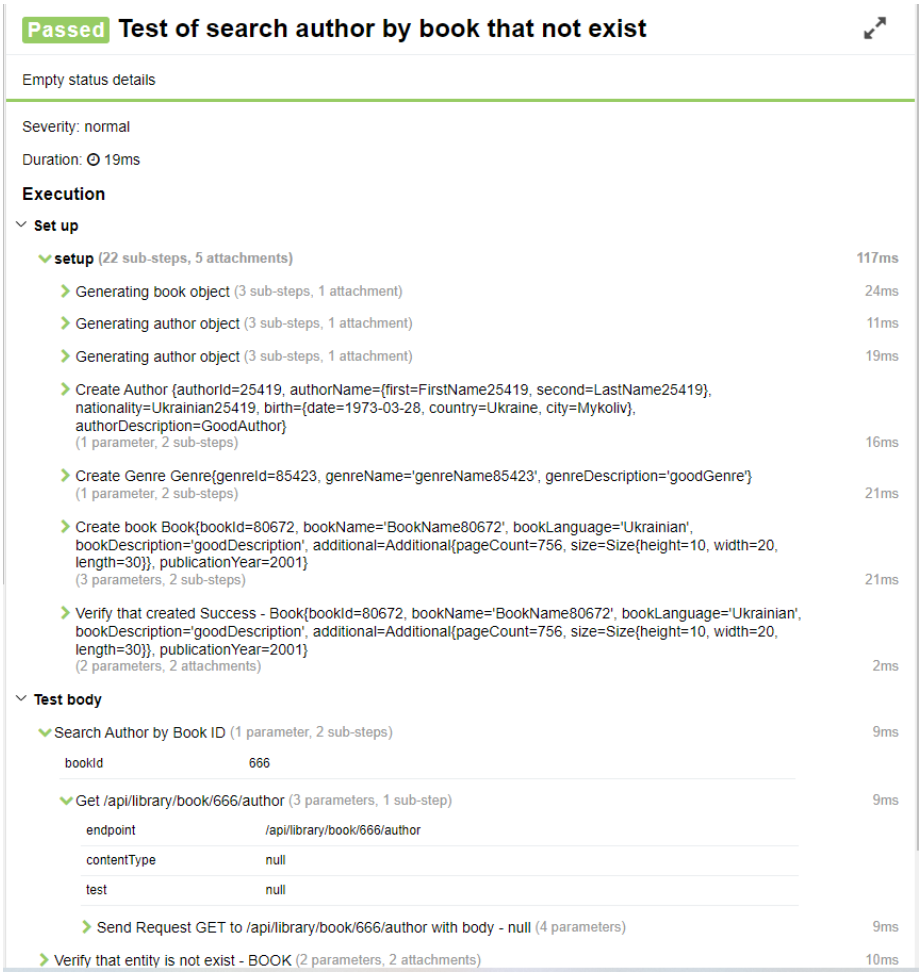


Рис. 3.41 Приклад результатів тест кейсу в репорті

Для того щоб виконати усі тести необхідно в середньому всього 13 секунд, при тому що на виконання усіх тестів вручну знадобилось би приблизно 1 година часу (див. рис. 3.42).

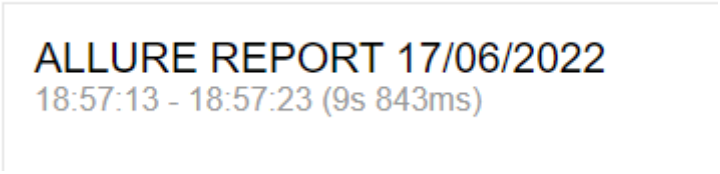


Рис. 3.42 Дата і час виконання тестів

Висновки за розділом 3

У розділі розроблена структура фреймворку для автоматичного тестування веб-сервісу, а також описані подробиці реалізованих тестів.

Окремо описано архітектурну складову проекту, яка включає використовувані патерни, методики, а також засоби, інструменти та мови програмування, використані у проекті. Це дозволяє відображати архітектуру програми в цілому та оцінювати застосування розробленого інструменту для певних проектів.

Крім того, докладно описані всі основні функціональні класи, як з основним функціоналом, так і зі службовими функціями. В результаті отримали специфікацію класів за функціональністю та підкреслена універсальність розробленого рішення для широкого кола завдань з автоматизації тестування.

Отже, отриманий фреймворк є готовим до використання програмним засобом, причому за рахунок відкритого вихідного коду, готовим до подальшого нарощування функціональності та можливостей.

					151.4341.КР.ПЗ.Р4		
Змн	*Арк*	*№ докум.*	*Підпис*	*Дата*			
Розроб.		*Бойчук А. М,*			ОХОРОНА ПРАЦІ	*Літ.*	*Арк*	*Аркушів*
							71	
Норм. контр		*Вінниченко І.Л.*				НУК ім. адм. Макарова		
Керівник		*Герасін О.С.*						
Зав. Каф.		*Черно О.О.*						

РОЗДІЛ 4

Охорона праці

4.1 Базові поняття про охорону праці.

Охорона праці — це система правових, соціально-економічних, організаційно-технічних, санітарно-гігієнічних та лікувально-профілактичних заходів та засобів, спрямованих на збереження життя, здоров'я та працездатності людини у процесі трудової діяльності.

Роботодавець — власник підприємства, установи, організації або уповноважений ним орган, незалежно від форм власності, виду діяльності, господарювання, та фізична особа, яка використовує найману працю.

Працівник - особа, яка працює на підприємстві, в організації, установі та виконує обов'язки або функції згідно з трудовим договором (контрактом).

Дія цього Закону поширюється на всіх юридичних та фізичних осіб, які відповідно до законодавства використовує найману працю та на всіх працюючих.

Законодавство про охорону праці складається з цього Закону, Кодексу законів про працю України, Закону України «Про загальнообов'язкове державне соціальне страхування від нещасного випадку на виробництві та професійного захворювання, що спричинили втрату працездатності» та прийнятих відповідно до них нормативно-правових актів.

Якщо міжнародним договором, згода на обов'язковість якого надана Верховною Радою України, встановлено інші норми, ніж ті, що передбачені законодавством України про охорону праці, застосовуються норми міжнародного договору.

Державна політика в галузі охорони праці визначається відповідно до Конституції України Верховною Радою України та спрямована на створення належних, безпечних та здорових умов праці, запобігання нещасним випадкам та професійним захворюванням.

Державна політика у сфері охорони праці виходить з принципах:

- пріоритету життя та здоров'я працівників, повної відповідальності роботодавця за створення належних, безпечних та здорових умов праці;
- підвищення рівня промислової безпеки шляхом забезпечення суцільного технічного контролю за станом виробництв, технологій та продукції, а також сприяння підприємствам у створенні безпечних та нешкідливих умов праці;
- комплексного вирішення завдань охорони праці на основі загальнодержавної, галузевих, регіональних програм з цього питання та з урахуванням інших напрямів економічної та соціальної політики, досягнень у галузі науки та техніки та охорони навколишнього середовища;
- соціального захисту працівників, повного відшкодування збитків особам, які потерпіли від нещасних випадків на виробництві та професійних захворювань;
- встановлення єдиних вимог щодо охорони праці для всіх підприємств та суб'єктів підприємницької діяльності незалежно від форм власності та видів діяльності;
- адаптації трудових процесів до можливостей працівника з урахуванням його здоров'я та психологічного стану;
- використання економічних методів управління охороною праці, участі держави у фінансуванні заходів з охорони праці, залучення добровільних внесків та інших надходжень з цією метою, отримання яких не суперечить законодавству;
- інформування населення, проведення навчання, професійної підготовки та підвищення кваліфікації працівників з питань охорони праці;
- забезпечення координації діяльності органів державної влади, установ, організацій, об'єднань громадян, які вирішують проблеми охорони здоров'я, гігієни та безпеки праці, а також співробітництва та проведення консультацій між роботодавцями та працівниками (їх представниками), між

усіма соціальними групами при прийнятті рішень з охорони праці на місцевому та державному рівні;

- використання світового досвіду організації роботи з покращення умов та підвищення безпеки праці на основі міжнародного співробітництва;[15]

4.2 Аналіз небезпечних та шкідливих факторів, що впливають на людину при розробці засобів автоматизованого тестування веб-сервісу

Небезпечні та шкідливі фактори, що впливають на людину при розробці засобів автоматизованого тестування веб-сервісу насправді теж саме що і при роботі за комп’ютером, тому буде проведено аналіз факторів під час роботи за комп’ютером.

Робота з комп'ютером характеризується значною розумовою напругою та нервово-емоційним навантаженням операторів, високою напруженістю зорової роботи та досить великим навантаженням на м'язи рук при роботі з клавіатурою ЕОМ. Велике значення має раціональна конструкція та розташування елементів робочого місця, що важливо для підтримки оптимальної робочої пози людини-оператора.

Основним фактором, що впливає на продуктивність праці людей, що працюють з ПЕОМ та ВДТ, є комфортні та безпечні умови праці.

Умови праці користувача, що працює з персональним комп'ютером, визначаються:

- особливостями організації робочого місця;
- умовами виробничого середовища (освітленням, мікрокліматом, шумом, електромагнітними та електростатичними полями, візуальними ергономічними параметрами дисплея тощо);
- характеристиками інформаційної взаємодії людини та персональних електронно-обчислювальних машинж

При виконанні робіт на персональному комп'ютері (ПК) згідно з ГОСТом 12.0.003-74 "ССБТ. Небезпечні та шкідливі виробничі фактори. Класифікація" можуть мати такі фактори:

- підвищена температура поверхонь ПК;
- підвищена або знижена температура повітря робочої зони;
- виділення повітря робочої зони низки хімічних речовин;
- підвищена чи знижена вологість повітря;
- підвищений або знижений рівень негативних та позитивних аероіонів;
- підвищене значення напруги в електричному ланцюзі, замикання;
- підвищений рівень статичної електрики;
- підвищений рівень електромагнітних випромінювань;
- підвищена напруженість електричного поля;
- відсутність чи нестача природного світла;
- недостатня штучна освітленість робочої зони;
- підвищена яскравість світла;
- підвищена контрастність;
- пряма та відбита блискітність;
- зорова напруга;
- монотонність трудового процесу;
- нервово-емоційне навантаження;

Робота на ПК супроводжується постійною і значною напругою функцій зорового аналізатора. Однією з основних особливостей є інший принцип читання інформації, ніж за звичайного читання. При звичайному читанні текст на папері, розташований горизонтально на столі, зчитується працівником із нахиленою головою під час падіння світлового потоку на текст. Працюючи на ПК оператор зчитує текст, майже нахиляючи голову, очі дивляться прямо чи майже вперед. Текст формується з іншого боку екрану, тому користувач не зчитує відбитий текст, а дивиться безпосередньо джерело світла, що змушує очі й орган зору цілому працювати у невластивому йому стресовому режимі тривалий час.

Розлад органів зору різко збільшується під час роботи понад чотири години на день. Всесвітня організація охорони здоров'я (ВООЗ) ввела поняття "комп'ютерний зоровий синдром" (КЗС), типовими симптомами якого є печіння в очах, почервоніння повік і кон'юнктиви, почуття стороннього тіла або піску під віками, болі в області очниць і чола, затуманювання зору, сповільнене перефокусування із ближніх об'єктів на дальні.

Основним небезпечним виробничим чинником під час роботи з ПК є електричний струм певної сили. Впливаючи на людину, він призводить до травм:

- судомне скорочення м'язів, без непритомності;
- судомне скорочення м'язів, із втратою свідомості;
- втрата свідомості з порушенням роботи органів дихання та кровообігу;
- стан клінічної смерті;
- місцеві травми;
- електричні опіки;
- електричний знак;
- електроавтольмія;

Проходячи через тіло людини, електричний струм має такі впливи:

- термічне - нагрівання тканин та біологічного середовища;
- електролітичне - розкладання крові та плазми;
- біологічне - здатність струму збуджувати та дратувати живі тканини організму;
- механічне – виникає небезпека механічного травмування внаслідок судомного скорочення м'язів;

Тяжкість ураження електричним струмом залежить від:

- величини струму;
- часу протікання;

- шляхи протікання;
- роду та частоти струму;
- опору людини;
- довкілля;
- стан людини;
- статі та віку людини;

Найбільш небезпечним змінним струмом є струм 20-100Гц. Оскільки комп'ютер живиться від мережі змінного струму частотою 50Гц, цей струм є небезпечним людини.

4.3 Заходи з техніки безпеки, що необхідно провести для зменшення або усунення впливу шкідливих виробничих факторів при розробці засобів автоматизованого тестування.

Аналогічно попередньому пункту заходи будуть розглянуті для роботи за комп’ютером.

Робота за комп'ютером - це, як правило, тривала сидяча робота, яка може призвести до різних проблем зі здоров'ям, таких як зниження зору, біль у спині та м'язах кистей рук. Щоб значно зменшити ризик зіткнутися з подібними неприємностями, достатньо засвоїти кілька простих правил та прийомів та періодично застосовувати їх на практиці.

Швидке погіршення зору властиве всім, хто змушений проводити у роботі за комп'ютером 6-8 годин на день. Розгляд дрібних літер, що світяться, на екрані монітора — це справді серйозне навантаження на зорову систему людини. Якщо ви відчуваєте такі симптоми, як зниження гостроти зору, сльозогінність очей, головний біль, то, швидше за все, вас спіткав комп'ютерний зоровий синдром. Для того щоб уникнути цього, необхідно скористатися такими рекомендаціями. Освітлення в кімнаті, де працюєте на комп'ютері, не повинно бути надто яскравим, але, в той же час, і надто тьмяним. Не варто розташовувати екран напроти вікна, так як це може призвести до появи відблисків на моніторі, найкраще розташувати його під прямим кутом до вікна. Якщо кімната дуже світла, то на вікнах мають бути

жалюзі або штори. Крім загального освітлення кімнати, необхідна лампа для робочого простору, з яскравістю від 60 Вт, з щільним абажуром, що висвітлює текст на папері, з яким доводиться працювати.

Щоб при читанні тексту з екрана уникнути надмірної напруги м'язів очей і, як наслідок, розвитку короткозорості, відстань від монітора до очей повинна становити щонайменше 60 див (відстань витягнутої руки). Через кожні 60-80 хвилин роботи на комп'ютері необхідно робити перерву на 5-10 хвилин, через кожні 2-3 години - на 15-20 хвилин. Під час перерв корисно зробити вправи для очей та кистей рук. Через кожні 20-25 хвилин обов'язково необхідно переводити погляд на якийсь далекий предмет, найкраще подивитися у вікно. Щоб позбавитися відчуття сухості очей, треба частіше кліпати і прикривати очі на 5-6 секунд. При наборі тексту треба якнайменше дивитися на екран комп'ютера, необхідно зосередити свій погляд на клавіатурі. Якщо після тривалої роботи на комп'ютері є відчуття оніміння або болі у зап'ястях тієї руки, якою тримають мишку, це відчуття симптомів такого захворювання, як синдром зап'ястного каналу. Причина виникнення цієї хвороби - одноманітна робота пальцями та кистями рук, наприклад рух мишки або друк на клавіатурі протягом тривалого часу. Ще одна неприємна проблема, з якою постійно стикається користувач ПК, - це болі в спині, які є наслідком сколіозу (викривлення хребта) та остеохондрозу (ущемлення нервових закінчень). Працюючи за комп'ютером досить тривалий час доводиться проводити у одному й тому ж позі, що призводить до тривалого навантаження однією групи м'язів, отже, і до перенапруження і виникнення болю у спині. При цьому інші групи м'язів позбавлені цього навантаження, що, у свою чергу, призводить до деградації цих м'язів.

Щоб не допустити появи та розвитку таких неприємностей, як хвороби хребта та синдром зап'ястного каналу, достатньо слідувати нескладним рекомендаціям. Необхідно піднімати по висоті комп'ютерне крісло так, щоб погляд упирався трохи нижче верхньої точки монітора. Сам монітор слід нахилити трохи назад. Під час роботи кисті рук краще тримати

горизонтально, лікті повинні бути зігнуті під прямим кутом і спиратися на підлокітники крісла. Періодично необхідно робити найпростішу розминку для рук та пальців, тобто просто згинати та розгинайте їх. При тривалій роботі за комп'ютером кожні 25-30 хвилин треба змінювати позу на більш зручну. Спина має бути прямою, але не напруженою. Якщо з'являється відчуття втоми або болю в спині, відкиньтесь на спинку крісла, заплющте очі і повністю розслабтеся на 2-3 хвилини. Після цього трохи розімніться, походіть по кімнаті, потягніться. При сильній втомі також можна трохи сполоснути прохолодною водою обличчя, руки, потилицю - це допоможе вам підбадьоритися.

При появі болю в спині і руках її можна спробувати зняти фізичними вправами, такими як потягування всім тілом від шкарпеток і до верхівки з піднятими вгору руками, вигинання тіла назад і нахили вперед, повороти голови, розведення рук у сторони, стискання-розтискання кистей рук, струшування розслабленими кистями та виконання ними обертальних рухів. Після розминки зробіть собі легкий масаж шиї та плечей, тобто просто добре розімніть їх руками.[16]

Висновки за розділом 4.

У розділі розглянуті базові поняття, такі як: охорона праці, роботодавець, працівник та закони про охорону праці в Україні.

Проведений аналіз небезпечних та шкідливих факторів, що впливають на людину при розробці засобів автоматизованого тестування веб-сервісу, а також розглянуті заходи з техніки безпеки, та практики для зменшення або усунення впливу шкідливих виробничих факторів.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи була проведена робота по аналізу сучасного стану проблеми автоматизації процесів тестування веб-сервісів, а також різноманітних інструментів для створення фреймворку. Використання цих інструментів значно полегшує задачу по написанню автоматизованих тестів веб-сервісів. Фреймворк значно заощаджує грошові ресурси та час на тестування вже протестованих сервісів. В рамках цього проекту побудований фреймворк для автоматизації тестування. Для побудови фреймворку використано Maven для управління залежностями, багатомодульної побудови, узгодженої моделі побудови та механізму модулів, що підключаються. До проекту додано систему логування за допомогою log4j, Rest Assured для роботи з сервісами, TestNG а також систему Allure для генерування репортів.

В рамках роботи складено універсальний фреймворк, за допомогою якого час, необхідний на тестування веб-сервісів, зменшиться з годин до хвилин, крім того, передбачено можливість доповнення фреймворку новими тестами. Результат роботи фреймворку генерується у спеціальний репорт, який можна переглянути через будь-який браузер. Зокрема, в роботі отримано зменшення часу на проведення тестів з однієї години до 13 секунд. Основна вимога автоматизованого тестування запропонованими програмними засобами - стабільне підключення до Інтернету, та комп'ютер, на якому будуть запускатись тести. Для багатьох компаній такі фреймворки є невід'ємною частиною розробки, адже вони зберігають дуже багато часу мануальних тестувальників, який вони можуть використати для того, щоб більш якісно перевірити новий функціонал, а також уникнути людського фактору під час тестування вже протестованих сервісів.

Крім того, в роботі розглянуті питання охорони праці людини при розробці засобів автоматизованого тестування веб-сервісу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Що таке QA і як стати тестувальником [Електронний ресурс]. Режим доступу: https://blog.ithillel.ua/ru/articles/chto-takoe-qa-i-kak-poluchit-professiyu-testirovshchika
2. Автоматизація тестування в геймдеві - Як роблять ігри. 297 [Електронний ресурс]. Режим доступу: https://dtf.ru/kdi/645741-avtomatizaciya-testirovaniya-v-geymdeve-kak-delayut-igry-297
3. Автоматизація тестування: вчимося заощаджувати [Електронний ресурс]. Режим доступу: https://software-testing.ru/library/around-testing/processes/437-learn-to-save
4. Посібник із тестування автоматизації REST API [Електронний ресурс]. Режим доступу: https://www.loadview-testing.com/ru/blog/
5. Автоматизація тестування: що треба, а що не треба [Електронний ресурс]. Режим доступу: https://cleverics.ru/digital/2021/01/sw-testing-automation/
6. Що таке тест кейс: приклад і чек-лист тест кейсів для тестувальників-початківців, які підійдуть кожному [Електронний ресурс]. Режим доступу: https://solvery.io/blog/ru/interesting/chto-takoe-test-case-primer-i-check-list-dlya-nachinayushhih-testirovshhikov/
7. Чек-листи у тестуванні: що потрібно знати тестувальнику! [Електронний ресурс]. Режим доступу: https://qualitica.ru/blog/chek-list/
8. Тестова документація та аналіз вимог [Електронний ресурс]. Режим доступу: https://habr.com/ru/company/otus/blog/588923/
9. Введення у фреймворки (Частина 1) [Електронний ресурс]. Режим доступу: https://habr.com/ru/post/150508/
10. Автоматизація тестування: Java чи Python? [Електронний ресурс]. Режим доступу: https://habr.com/ru/company/otus/blog/492200/
11. Різниця між Maven та Gradle [Електронний ресурс]. Режим доступу: https://russianblogs.com/article/98301680394/

12. Вчимося вести логування за допомогою Log4j [Електронний ресурс]. Режим доступу: https://devcolibri.com/log4j/

13. Тестування Rest API || Короткий посібник з RestAssured tutorial 2020 [Електронний ресурс]. Режим доступу: https://ru.lambdageeks.com/api-testing-automation-restassured/

14. Стратегія тестування REST API: що вам потрібно тестувати? [Електронний ресурс]. Режим доступу: https://habr.com/ru/post/568360/

15. ЗАКОН УКРАЇНИ Про охорону праці [Електронний ресурс]. Режим доступу: https://nvkvector.com/ru/zakon-ukrainy-ob-okhrane-truda/.

16. Аналіз небезпечних та шкідливих факторів при роботі з ПК [Електронний ресурс]. Режим доступу: https://studbooks.net/1269114/informatika/analiz_opasnyh_vrednyh_faktorov_rabote