МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ В. Н. КАРАЗІНА
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ КОМП'ЮТЕРНОЇ ФІЗИКИ ТА ЕНЕРГЕТИКИ
Кафедра інформаційних технологій в фізико-енергетичних системах

ПОЯСНЮВАЛЬНА ЗАПИСКА

до дипломної роботи
за освітньо-кваліфікаційним рівнем магістр

«Розробка автоматизованих тестів для перевірки WEB UI системи наповнення БД методичних матеріалів КІТФЕС»

Виконав: студент 2 курсу, групи НФ-63
Перерва О.С.
спеціальність:
105 Прикладна фізика та наноматеріали
освітня програма: Прикладна фізика енергетичних систем
Керівник: ст. викладач Кофман О. Й.

Рецензент:

Харків – 2020

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імени В. Н. Каразіна

Навчально-науковий інститут комп'ютерної фізики та енергетики
Кафедра інформаційних технологій у фізико-енергетичних системах
Рівень вищої освіти (освітньо-кваліфікаційний рівень) магістр
Спеціальність 105 Прикладна фізика та наноматеріали

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ доц. Сухов Р.В.
підпис
"_____"_____2020 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ (ПРОЕКТ)

Перерва Олександр Сергійович

1. Тема роботи " Розробка автоматизованих тестів для перевірки WEB UI системи наповнення БД методичних матеріалів КІТФЕС"
   керівник роботи: Кофман Олег Йосипович, старший викладач.
   затверджені наказом по університету від "23" жовтня 2020 року №0210-05/1907
   Строк подання студентом роботи 27.11.2020
2. Перелік питань, які потрібно розробити:
   1) Огляд літератури та формулювання задачі про розробку автоматизованих тестів
   2) Методи використання WEB UI елементів для тестуваня
   3) Огляд систем розгортування програмного забеспечення
   4) Огляд систем підтримки середовища

5) Використання системи генерації звітів

3. План роботи

№ з/п	Назви етапів роботи
1	Вибір теми, узгодження, а також затвердження її на кафедрі
2	Робота над літературою та іншими джерелами
3	Затвердження теми на кафедрі
4	Написання вступу до роботи та першого розділу
5	Написання другого розділу
6	Написання третього розділу
7	Завершення написання дипломної роботи(висновки та література)
8	Подача науковому керівнику
9	Відгук, рецензія, підготовк до захисту
10	Попередній захист
11	Захист

4. Дата видачі завдання 26.10.2020

Студент _____ Олександр ПЕРЕРВА

підпис

Керівник роботи _____ Олег Кофман

підпис

РЕФЕРАТ

Перерва О.С. Розробка автоматизованих тестів для перевірки WEB UI системи наповнення БД методичних матеріалів КІТФЕС

Дипломна робота представлена на 42 сторінках; складається з вступу, 3 розділів та висновків; містить 16 рисунків, список використаної літератури з 25 найменувань.

Мета даної роботи – розробка і програмна реалізація системи автоматизованих тестів для перевірки WEB UI системи наповнення БД, призначеної для розв'язку задачі автоматичної генерації науково-методичної бази документів.

Об'єкт дослідження – WEB UI елементи, які використовуються для внесення інформації до БД.

Предмет дослідження – система автоматизованих тестів для перевірки WEB UI елементів.

Розроблена система система автоматизованих тестів, система розгортування автоматизованого тестування та система генерації звітів.

Розв'язок задачі розробки автоматизованих тестів реалізується на базі існуючих OpenSource рішень та сучасних технологій використання та запровадженя систем тестування.

У системі автоматизованих тестів використовуєтся незалежна реалізація хмарного запуску тестів та генерації звітів. Завдяки чому, використання системи є можливим незалежно від технічних ресурсів.

Система автоматизованих тестів значно спрощує перехід від реалізації до використання кінцевими користувачами.

КЛЮЧОВІ СЛОВА: АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ, ТЕСТУВАННЯ WEB UI, ГЕНЕРАЦІЯ ЗВІТІВ ТЕСТУВАННЯ, СИСТЕМИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ.

РЕФЕРАТ

Перерва А.С. Разработка автоматизированных тестов для проверки WEB UI системы наполнения БД методических материалов КИТФЕС.

Дипломная работа представлена на 42 страницах; состоит из введения, 3 глав и выводов; содержит 16 рисунков, список использованной литературы из 25 наименований.

Цель данной работы - разработка и программная реализация системы автоматизированных тестов для проверки WEB UI системы наполнения БД, предназначенной для решения задачи автоматической генерации научно-методической базы документов.

Объект исследования - WEB UI элементы, которые використовуются для внесения информации в БД.

Предмет исследования - система автоматизированных тестов для проверки WEB UI элементов.

Разработана система автоматизированных тестов, система розгортування автоматизированного тестирования и система генерации отчетов.

Решение задачи разработки автоматизированных тестов реализуется на базе существующих OpenSource решений и современных технологий использования и внедрения систем тестирования.

В системе автоматизированных тестов використовуется независимая реализация облачного запуска тестов и генерации отчетов. Благодаря чему, использование системы возможно независимо от технических ресурсов.

Система автоматизированных тестов значительно упрощает переход от реализации к использованию конечными пользователями.

КЛЮЧЕВЫЕ СЛОВА: АВТОМАТИЗАЦИЯ ТЕСТИРОВАНИЯ, ТЕСТИРОВАНИЕ WEB UI, ГЕНЕРАЦИЯ ОТЧЕТОВ ТЕСТИРОВАНИЯ, СИСТЕМЫ АВТОМАТИЗИРОВАННОГО ТЕСТИРОВАНИЯ.

ABSTRACT

Pererva O.S. Development of automated tests for testing WEB UI of the KITFES methodological materials database filling system..

The diploma project is presented on 42 pages; consists of introduction, 3 chapters and conclusions; contains 16 figures and a list of used literature of 25 items.

The purpose of this work is the development and software implementation of of a system of automated tests for checking the WEB UI of the database filling system, designed to solve the problem of automatically generating a scientific and methodological base of documents.

The object of study is WEB UI elements that are selected for entering information into the database.

The subject of the study is a system of automated tests for checking WEB UI elements.

A system of automated tests, an automated testing system and a reporting system have been developed.

The solution to the problem of developing automated tests is implemented on the basis of existing OpenSource solutions and modern technologies for using and implementing test systems.

In the automated test system, an independent implementation of cloud-based test launch and report generation is used. Due to this, the use of the system is possible regardless of technical resources.

The automated test system greatly simplifies the transition from implementation to use by end users.

KEYWORDS: AUTOMATION OF TESTING, TESTING WEB UI, GENERATION OF TEST REPORTS, AUTOMATED TESTING SYSTEMS.

ЗМІСТ

ВСТУП 8
1 АРХІТЕКТУРА СИСТЕМИ 11
1.1 Оновлення WEB UI елементів 12
1.2 Сервіс реалізації системи 15
1.3 Мова реалізації системи 16
1.4 Система генерації звітів 19
1.5 Сервіс підтримки середовища 19
2 СТРУКТУРА СИСТЕМИ 24
2.1 Реалізація WEB UI елементів 24
2.2 Побудова сервісу Jenkins 25
2.3 Спосіб розробки тестів 26
2.4 Інтеграція Allure 34
3 ОПИС СИСТЕМИ 39
3.1 Загальні відомості 39
3.2 Функціональне призначення 40
ВИСНОВКИ 41
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 42

ВСТУП

Зі швидким розвитком комп'ютерних технологій, та швидкістю розробки програмного забеспечення в будь-яких сферах діяльності, виникає необхідність забеспечення якісного та швидкого тестування. Питання тестування програмного продукту виникає на перших підступах до розробки. А саме в моменти, коли вже на початковому етапі потрібно визначити якість створюємого продукту. Завдяки цій необхідності й виник процес тестування.

Але ручне тестування всіх необхідних компонентів навіть на перших етапах розробки займае дуже багато часу та ресурсів. Мета цієї роботи – створити програмне забеспечення для швидкого та інформативного тестування, базуючусь на основних вимогах до програмного продукту. Це дозволить безпечно вносити будь-які зміни в програмний код та завдяки інфроструктурі автоматизованого тестування визначати якість продукта після кожної зміни. В свою чергу це забеспечить зниження необхідних ресурсів на тестування. Проблема ресурсів є багатогранною, це може бути звичайний час, персонал або навіть наявність стендових машин для проведення тестуваня. Новітні технології здатні забеспечити користувачів всим необхідним та завдяки створенню архітектури тестування на базі новітніх технологічних сервісів, проблема ресурсів для відтворення необхідної кількості тестів та швидкості їх відтворення стає розв'язуваною. Для створення необхіодною інфроструктури необхідно створити відповідальне забеспечення програмних елементів та компонентів, тому для цього потрібно відредагувати існуючі елементи та компоненти.

Створення автоматизованого тестуваня відбувается завдяки наявності існуючих тестових документів створенних на базі вимог до програмного продукту. Автоматизовані тести імітують дії звичайних користувачів або навіт професійних тестувальників для перевірки продукту.

Враховуючи те, що тестування проходе у віртуальному середовищі слід використовувати сервіси для розгортування віртуальних середовищ з можливістю використання їх у декілька потоків одночасно для покращення швидкості проходження тестових випадків. Тестування на має значення, якщо воно не приносить результатів. Тому також необхідно інтегрувати процесс тестування в сервіс для відображення результатів, який автоматично генерує звіти після кожного запуску автоматизованих тестів. Також для забеспечення безпечної можливості внесення змін до прогрманого коду, є необхідність інтегрувати створену систему автоматизованих тестів у процесс безпреривного розгортуваня. Фінальним результатом вважается автоматизована система експлуатації тестів після кожної зміни програмного коду, яка автоматично генерує звіти, що дозволить відобразити динаміку покращення або погіршення якості програмного продукту. Система автоматизованих тестів дозволить покращити швидкість розробки прогрманого забеспечення завдяки автономному запуску тестування та створення автоматитичних звітів. Також важливою перевагою автоматизованого тестування є програмне виконання тестів, що в свою чергу не допускає можливості людської похибки в перевірці будь-якого функціоналу. Саме тому використання саме системи автоматизованих тестів дає змогу бути впевненим в результатах тестування не зважаючи на різні фактори та ресурси, на базі яких, і відбувається тестування. Адже кожен запуск системи автоматизованих тестів починається з розгортування середовища виконання тестів, що забеспечує ідентичні умови прогону тестового набору перевірок. Тобто, кожен запуск відтворюється не на фізичній машині, яка є вразлива до зовнішніх факторів, а у хмарному середовищі при використанні новітніх технологій керівництва цим середовищем. Саме завдяки контейнерізацій процесів є можливість паралельного запуску тестових наборів та відтворення функціонального процесу системи незалежно від зовнішніх факторів будь-яких ресурсів.

Недоліками системи є необхідність завжди оновлювати набір тестових випадків після будь-яких змін функціоналу програмного забезпечення, які впливають на результати запуску системи автоматизованих тестів, а також на кінцеві звіти. Саме оновлення кодової бази системи автоматизованих тестів може забезпечити стабільність та актуальність результатів тестування, що в свою чергу, забезпечить команду розробки можливістю використання якісних результатів тестування будь-якої зміни функціоналу, що стосується частин функціоналу під тестуванням.

1 АРХІТЕКТУРА СИСТЕМИ

Архітектура системи охоплює рішення щодо того, де розмістити конкретні програмні компоненти. Чи слід розміщувати певні компоненти на одному сервері або на різних машинах? Підприємства можуть мати певний високошвидкісний сервер обробки або висококласний надійний сховище, який вони хочуть використовувати для певних компонентів. Ці рішення призведуть до різних типів архітектурних організацій. Ми можемо широко класифікувати їх як централізовані та децентралізовані організації.

1. Централізовані організації: Клієнт-серверні системиСервер - це процес, що реалізує певну послугу (наприклад, службу баз даних). Клієнт - це процес запиту цієї послуги з сервера. Клієнт надсилає запит і чекає відповіді (поведінка запит-відповідь). Там, де ці процеси фізично розподілені, це призведе до різних багатошарових архітектурних стилів.

Хоча ми розрізняємо три логічні рівні, існує кілька способів фізично розподілити клієнт-серверну програму між машинами.

Найпростіша організація - використання машин двох типів: 1) клієнтська машина з користувальницьким інтерфейсом (перший рівень) та 2) серверний інтерфейс, що містить програму, що реалізує процес, і дані (два та три рівні). У цьому випадку всім обробляє сервер, тоді як клієнт - не що інше, як тупий термінал. Це називається (фізично) дворівневою архітектурою. Однак існують інші способи розділити ці три шари на двох машинах. Можливості варіюються від поділу, як було згадано вище, до того, щоб все, крім збережених даних, працювало на клієнтській машині.

У (фізично) трирівневих архітектурах один із серверів також повинен виконувати функції клієнта. Додаток розподілено на трьох машинах, одному клієнті та двох серверах. Одному із серверів може знадобитися ввід іншого сервера для обробки запиту клієнта, виконуючи функції клієнта.

Трирівнева архітектура: один клієнт і два сервери, при цьому сервер додатків також виконує функції клієнта.

Трирівнева архітектура: один клієнт та два сервери, при цьому сервер

додатків також виконує функції клієнта.

Багаторівневі архітектури клієнт-сервер є прямим наслідком розподілу розподілених програм на користувальницький інтерфейс, компоненти обробки та компоненти управління даними. Різні рівні безпосередньо відповідають логічній організації додатків (архітектура програмного забезпечення відповідає архітектурі системи). Це називається вертикальним розподілом, при якому логічно різні компоненти розміщуються на різних машинах. Це дозволяє пристосувати кожну машину до певної функції, наприклад, розміщення шару обробки на високоякісному сервері обробки.

2. Децентралізована організація: однорангові системи

У сучасних архітектурах розподіляються самі клієнти та сервери, у так званому горизонтальному розподілі. Клієнт або сервер можуть бути фізично розділені, але кожна частина працює на власній частці повних даних, таким чином балансуючи навантаження. Отже, замість того, щоб мати одне сховище даних, усі компоненти мають доступ, що потенційно може перетворитися на вузьке місце, кожен компонент має власний набір даних. Вони називаються одноранговими системами. Кожен процес виступає одночасно як клієнт і сервер, часто його називають службовим.

Однорангові архітектури розвиваються навколо питання про те, як організувати процеси в накладній мережі - мережі, в якій вузли утворені процесами, а посилання представляють можливі канали зв'язку.

Як ми вже бачили, існує кілька способів організації додатків у логічні компоненти (міркування щодо архітектури програмного забезпечення). Чи буде шаруватим, орієнтованим на об'єкт / послугу, REST чи pub / subscribe найкращим підходом, багато в чому залежить від програми. Потім ці компоненти розміщуються на різних фізичних машинах (міркування щодо архітектури системи), або клієнт-сервер, або рівноправний.

1.1 Оновлення WEB UI елементів

Для покращення способу використання WEB UI елементів було

прийнято рішення оновити елементи и інтерфейс користувача цілком. Обрано нову мову програмування та технологію реалізації інтерфейсу.

Python - це проста в освоєнні, потужна мова програмування. Він має ефективні структури даних високого рівня та простий, але ефективний підхід до об’єктно-орієнтованого програмування. Елегантний синтаксис та динамічне введення тексту Python, а також його інтерпретований характер роблять його ідеальною мовою для сценаріїв та швидкої розробки додатків у багатьох областях на більшості платформ.

Інтерпретатор Python та велика стандартна бібліотека є у вільному доступі у вихідній або двійковій формі для всіх основних платформ із веб-сайту Python і можуть вільно розповсюджуватися.

Інтерпретатор Python легко розширити за допомогою нових функцій та типів даних, реалізованих на C або C ++ (або інших мовах, що викликаються з C).

Python - мова програмування з багатьма парадигмами. Об'єктно-орієнтоване програмування та структуроване програмування повністю підтримуються, і багато його функцій підтримують функціональне програмування та аспектно-орієнтоване програмування (в тому числі за допомогою метапрограмування та метаоб'єктів (магічні методи)). Багато інших парадигм підтримуються за допомогою розширень, включаючи проектування за контрактом та логічне програмування.

Python використовує динамічне введення тексту та комбінацію підрахунку посилань та збирача сміття для виявлення циклів для управління пам’яттю. Він також має динамічну роздільну здатність імен (пізнє прив'язування), яка зв'язує імена методів та змінних під час виконання програми.

Дизайн Python пропонує певну підтримку функціонального програмування за традицією Lisp. Він має функції фільтрування, картографування та зменшення; перелічити розуміння, словники, набори та вирази генератора. Стандартна бібліотека має два модулі (itertools та functools), що реалізують функціональні інструменти, запозичені у Haskell та

Standard ML.

Замість того, щоб усі його функціональні можливості були вбудовані в його ядро, Python був розроблений таким чином, щоб бути дуже розширюваним. Ця компактна модульність зробила його особливо популярним як засіб додавання програмованих інтерфейсів до існуючих програм. Бачення Ван Россума малої основної мови з великою стандартною бібліотекою та легко розширюваним перекладачем випливало з його розчарувань у роботі з ABC, яка підтримувала протилежний підхід.

Python прагне до більш простого, менш захаваного синтаксису та граматики, одночасно надаючи розробникам можливість вибору методології кодування. На відміну від девізу Perl "існує більше ніж один спосіб", Python охоплює "філософію дизайну" повинен бути один - і бажано лише один - очевидний спосіб зробити це.

Розробники Python прагнуть уникнути передчасної оптимізації та відкидають виправлення для некритичних частин еталонної реалізації CPython, які пропонують незначне збільшення швидкості ціною ясності. Коли швидкість важлива, програміст Python може переміщати критично важливі для часу функції до модулів розширення, написаних мовами, такими як C, або використовувати PyPy, своєчасний компілятор. Також доступний Cython, який перекладає сценарій Python на C і здійснює прямі виклики API рівня C в інтерпретатор Python.

Django - це безкоштовний веб-фреймворк на базі Python, який відповідає архітектурному зразку model-template-views (MTV). Він підтримується Фондом програмного забезпечення Django (DSF), американською незалежною організацією.

Основна мета Django - полегшити створення складних веб-сайтів, керованих базами даних. Структура підкреслює багаторазовість та "підключеність" компонентів, менше коду, низьку зв'язок, швидкий розвиток та принцип "не повторюватися". Python використовується всюди, навіть для налаштувань, файлів та моделей даних. Django також надає додатковий

адміністративний інтерфейс створення, читання, оновлення та видалення, який генерується динамічно за допомогою самоаналізу та налаштовується за допомогою моделей адміністратора.

1.2 Сервіс реалізації системи

Jenkins - це автономний відкритий сервер автоматизації, який можна використовувати для автоматизації різноманітних завдань, пов’язаних зі створенням, тестуванням та доставкою або розгортанням програмного забезпечення.

Jenkins можна встановити через власні системні пакети, Docker або навіть запустити автономно на будь-якій машині з встановленим середовищем виконання Java (JRE).

Jenkins - це засіб автоматизації з відкритим кодом, написаний на Java, з додатками, створеними для цілей безперервної інтеграції. Jenkins використовується для постійного створення та тестування ваших програмних проектів, що полегшує розробникам інтеграцію змін до проекту та полегшує користувачам отримання нової збірки. Це також дозволяє постійно поставляти програмне забезпечення, інтегруючись із великою кількістю технологій тестування та розгортання. За допомогою Дженкінса організації можуть пришвидшити процес розробки програмного забезпечення за допомогою автоматизації. Дженкінс інтегрує процеси життєвого циклу всіх видів, включаючи побудову, документування, тестування, пакет, етап, розгортання, статичний аналіз та багато іншого. Дженкінс досягає постійної інтеграції за допомогою плагінів. Плагіни дозволяють інтегрувати різні етапи DevOps. Якщо ви хочете інтегрувати певний інструмент, вам потрібно встановити плагіни для цього інструменту. Наприклад, Git, проект Maven 2, Amazon EC2, видавець HTML тощо. Безперервна інтеграція - це практика розробки, при якій розробники повинні вносити зміни у вихідний код у спільному сховищі кілька разів на день або частіше. Потім будується кожен коміт, зроблений у сховищі. Це дозволяє командам рано виявляти проблеми.

Окрім цього, залежно від інструменту Безперервна інтеграція, існує ще кілька функцій, таких як розгортання програми збірки на тестовому сервері, надання відповідним командам результатів збірки та тестування тощо. Давайте зрозуміємо його важливість на прикладі використання. Приклад безперервної інтеграції: Nokia.

Я майже впевнений, що всі ви колись користувалися телефонами Nokia. У проекті розробки програмного продукту в Nokia був процес, що називається Nightly builds. Нічні збірки можна сприймати як попередника Безперервної інтеграції. Це означає, що щовечора автоматизована система витягує доданий до спільного сховища код протягом дня і створює цей код. Ідея досить схожа на Безперервну інтеграцію, але оскільки код, який був створений вночі, був досить великий, пошук та виправлення помилок було справжньою проблемою. Завдяки цьому Nokia прийняла безперервну інтеграцію (CI). В результаті було створено кожне комітування, зроблене з вихідним кодом у сховищі. Якщо результат збірки показує, що в коді є помилка, розробникам потрібно лише перевірити цей конкретний коміт. Це значно скоротило час, необхідний для випуску нового програмного забезпечення. Давайте уявимо сценарій, коли повний вихідний код програми був побудований, а потім розміщений на тестовому сервері для тестування. Це звучить як ідеальний спосіб розробки програмного забезпечення, але цей процес має багато недоліків.

З вищезазначених проблем видно, що не тільки процес доставки програмного забезпечення став повільним, але і якість програмного забезпечення також знизилася. Це призводить до невдоволення клієнтів. Тож для подолання такого хаосу була гостра потреба в існуванні системи, де розробники можуть постійно запускати збірку та перевіряти кожну зміну, внесену у вихідний код.

1.3 Мова реалізації системи

Автоматизоване тестування за допомогою Selenium з Java полегшило життя як розробникам, так і тестувальникам. Будучи інструментом з

відкритим вихідним кодом, він надає можливість пришвидшити час виконання та усунути ручне резервування та людські помилки. Java є надзвичайно популярною мовою програмування, враховуючи широку підтримку, яку вона отримує від спільноти розробників. Selenium із Java виявився корисним та забезпечив оптимізований спосіб тестування, особливо з точки зору регресійного та крос-браузерного тестування. Завдяки тому, що кілька плагінів пробиваються в Selenium, це призвело до того, що тестування було легким та менш трудомістким. Постійна інтеграція з такими інструментами, як Jenkins та Maven, призвела до безперервного тестування моделей.

Мова програмування Java була розроблена Sun Microsystems на початку 1990-х. Хоча вона в першу чергу використовується для програм на базі Інтернету, Java є простою, ефективною мовою загального призначення. Java спочатку була розроблена для вбудованих мережевих додатків, що працюють на декількох платформах. Це портативна, об’єктно-орієнтована, інтерпретована мова.

Java надзвичайно портативна. Той самий додаток Java буде однаково працювати на будь-якому комп'ютері, незалежно від апаратних можливостей або операційної системи, якщо у нього є інтерпретатор Java. Окрім портативності, ще однією з ключових переваг Java є набір функцій безпеки, які захищають ПК, на якому запущена програма Java, не тільки від проблем, спричинених помилковим кодом, але і від шкідливих програм (таких як віруси). Ви можете сміливо запускати аплет Java, завантажений з Інтернету, оскільки функції безпеки Java заважають цим типам аплетів отримувати доступ до жорсткого диска ПК або мережевих з'єднань. Аплет - це, як правило, невелика програма Java, яка вбудована в HTML-сторінку.

Java можна вважати компільованою та інтерпретованою мовою, оскільки її вихідний код спочатку компілюється у двійковий байт-код. Цей байт-код працює на віртуальній машині Java (JVM), яка, як правило, є програмним інтерпретатором. Використання скомпільованого байт-коду

дозволяє інтерпретатору (віртуальній машині) бути невеликим та ефективним (і майже таким же швидким, як центральний процесор, що працює зі скомпільованим кодом). Крім того, цей байт-код надає Java портативність: він буде працювати на будь-якій JVM, яка правильно реалізована, незалежно від конфігурації комп'ютера або програмного забезпечення. Більшість веб-браузерів (таких як Microsoft Internet Explorer або Netscape Communicator) містять JVM для запуску аплетів Java.

Порівняно з C ++ (іншою об'єктно-орієнтованою мовою), код Java працює трохи повільніше (через JVM), але він є більш портативним і має набагато кращі функції безпеки. Віртуальна машина забезпечує ізоляцію між ненадійною програмою Java та ПК, на якому запущено програмне забезпечення. Синтаксис Java схожий на C ++, але мови досить різні. Наприклад, Java не дозволяє програмістам реалізовувати перевантаження оператора, тоді як це робить C ++. Крім того, Java - це динамічна мова, де ви можете безпечно модифікувати програму під час її роботи, тоді як C ++ не дозволяє цього. Це особливо важливо для мережевих додатків, які не можуть дозволити собі простою. Крім того, всі базові типи даних Java визначені заздалегідь і не залежать від платформи, тоді як деякі типи даних можуть змінюватися в залежності від платформи, що використовується на C або C ++ (наприклад, типу int).

Програми Java більш структуровані, ніж еквіваленти C ++. Усі функції (або методи Java) та виконувані оператори в Java повинні знаходитись у класі, тоді як C ++ дозволяє визначенням функцій та рядкам коду існувати поза класами (як у програмах у стилі C). Глобальні дані та методи не можуть знаходитись поза класом на Java, тоді як C ++ це дозволяє. Ці обмеження, хоча часом і громіздкі, допомагають підтримувати цілісність і безпеку програм Java і змушують їх бути повністю об’єктно-орієнтованими.

Selenium - це парасольковий проект для ряду інструментів та бібліотек, що дозволяють та підтримують автоматизацію веб-браузерів.

Він надає розширення для імітації взаємодії користувача з браузерами,

сервер розповсюдження для масштабування розподілу браузера та інфраструктуру для реалізації специфікації W3C WebDriver, яка дозволяє писати взаємозамінний код для всіх основних веб-браузерів.

Цей проект став можливим завдяки волонтерським співавторам, які витратили тисячі годин власного часу та зробили вихідний код у вільному доступі для будь-кого для використання, користування та вдосконалення.

Selenium об'єднує постачальників браузерів, інженерів та ентузіастів для подальшого відкритого обговорення питань автоматизації веб-платформи. Проект організовує щорічну конференцію з метою навчання та виховання громади. В основі Selenium лежить WebDriver, інтерфейс для написання наборів команд, який може бути взаємозамінним у багатьох браузерах.

1.4 Система генерації звітів

Allure Framework - це гнучкий легкий багатомовний інструмент звіту про тестування, який не тільки демонструє дуже стисле уявлення про те, що було перевірено в охайній формі веб-звіту, але і дозволяє кожному, хто бере участь у процесі розробки, отримати максимум корисної інформації із щоденного виконання тестів.

З точки зору розробки та тестування звіти Allure скорочують загальний життєвий цикл дефектів: невдачі тестів можна розділити на помилки та непрацюючі тести, а також журнали, етапи, пристосування, вкладення, таймінги, історію та інтеграцію з системами відстеження помилок, тому відповідальні розробники та тестувальники матимуть всю інформацію під рукою.

1.5 Сервіс підтримки середовища

Контейнер - це стандартна одиниця програмного забезпечення, яке пакує код та всі його залежності, завдяки чому програма працює швидко та надійно від одного обчислювального середовища до іншого. Предмет

контейнера Docker - це легкий, автономний, виконуваний пакет програмного забезпечення, що включає все необхідне для запуску програми: код, час роботи, системні інструменти, системні бібліотеки та налаштування.

Предмети контейнерів стають контейнерами під час виконання, а у випадку контейнерів Docker - предмети стають контейнерами, коли вони працюють на Docker Engine. Контейнерне програмне забеспечення, доступне як для Linux, так і для Windows, завжди працюватиме однаково, незалежно від інфраструктури. Контейнери ізолюють програмне забезпечення від навколишнього середовища та забезпечують його рівномірну роботу, незважаючи на різницю, наприклад, між розробкою та інсталяцією. Контейнери Docker, що працюють на Docker Engine. Docker створив галузевий стандарт для контейнерів, щоб їх можна було переносити де завгодно. Контейнери використовують спільне ядро ОС машини, тому не потрібно ОС для кожного додатка, що сприяє підвищенню ефективності роботи сервера та зменшенню витрат на ліцензування та сервер. Додатки безпечніші в контейнерах, і Docker забезпечує найсильніші можливості ізоляції за замовчуванням у галузі.

Контейнерна технологія Docker була випущена в 2013 році як Docker Engine з відкритим кодом. Він використовував існуючі обчислювальні концепції навколо контейнерів, зокрема у світі Linux, примітивів, відомих як cgroups та простори імен. Технологія Docker унікальна, оскільки вона орієнтована на вимоги розробників та системних операторів, щоб відокремити залежності додатків від інфраструктури. Успіх у світі Linux зумовив партнерство з Microsoft, яке перенесло контейнери Docker та їх функціональність на Windows Server (іноді їх називають контейнерами Docker Windows). Технологія, доступна від Docker та її проекту з відкритим кодом, Moby була використана усіма основними постачальниками центрів обробки даних та хмарними провайдерами. Багато з цих постачальників використовують Docker для своїх власних пропозицій IaaS для контейнерів. Крім того, провідні безсерверні платформи з відкритим кодом

використовують технологію контейнерів Docker.

Контейнери - це абстракція на рівні програми, яка пакує код і залежності разом. Кілька контейнерів можуть працювати на одній машині та спільно використовувати ядро ОС з іншими контейнерами, кожен з яких виконується як ізольовані процеси в просторі користувача. Контейнери займають менше місця, ніж віртуальні машини (розміри зображень контейнерів, як правило, складають десятки МБ), можуть обробляти більше додатків і вимагати меншої кількості віртуальних машин та операційних систем.

Віртуальні машини (ВМ) - це абстракція фізичного обладнання, що перетворює один сервер на багато серверів. Гіпервізор дозволяє запускати кілька віртуальних машин на одній машині. Кожна віртуальна машина включає повну копію операційної системи, програми, необхідних бінарних файлів та бібліотек - займає десятки ГБ. Віртуальні машини також можуть повільно завантажуватися.

Запуск Docker у 2013 році розпочав революцію у розробці додатків - шляхом демократизації програмних контейнерів. Docker розробив технологію контейнерів Linux - портативну, гнучку та просту в розгортанні. Docker з відкритим кодом libcontainer та співпрацює зі світовим співтовариством для подальшого його розвитку. У червні 2015 року Docker передав специфікацію зображення контейнера та код виконання, який тепер відомий як runc, Ініціативі відкритих контейнерів (OCI), щоб допомогти встановити стандартизацію в міру зростання та дозрівання екосистеми контейнера. Після цієї еволюції Docker продовжує віддавати проект containerd, який Docker подарував Cloud Native Computing Foundation (CNCF) у 2017 році. Containerd - це стандартний у галузі виконання контейнер, який використовує runc і був створений з акцентом на простоті, надійності і портативність. containerd - основний час роботи контейнера Docker Engine.

Раніше організації запускали додатки на фізичних серверах. Не було ніякого способу визначити межі ресурсів для додатків на фізичному сервері, і

це викликало проблеми з розподілом ресурсів. Наприклад, якщо кілька додатків виконуються на фізичному сервері, можуть бути випадки, коли один додаток буде займати більшу частину ресурсів, і в результаті чого інші додатки будуть працювати гірше. Рішенням цього було запустити кожен додаток на іншому фізичному сервері. Але це не масштабувати, оскільки ресурси використовувалися не повністю, через що організаціям було накладно підтримувати безліч фізичних серверів. В якості вирішення була представлена віртуалізація. Вона дозволила запускати кілька віртуальних машин (ВМ) на одному фізичному сервері. Віртуалізація ізолює додатки між віртуальними машинами і забезпечує певний рівень безпеки, оскільки інформація одного додатка не може бути вільно доступна іншому додатку. Віртуалізація дозволяє краще використовувати ресурси на фізичному сервері і забезпечує кращу масштабованість, оскільки додаток можна легко додати або оновити, крім цього знижуються витрати на обладнання і багато іншого. За допомогою віртуалізації можна перетворити набір фізичних ресурсів в кластер одноразових віртуальних машин.

Кожна віртуальна машина являє собою повноцінну машину, на якій виконуються всі компоненти, включаючи власну операційну систему, поверх віртуалізованого обладнання. Контейнери схожі на віртуальні машини, але у них є властивості ізоляції для спільного використання операційної системи (ОС) між додатками. Тому контейнери вважаються легкими. Подібно віртуальній машині, контейнер має свою власну файлову систему, процесор, пам'ять, простір процесу і багато іншого. Оскільки вони не пов'язані з базовою інфраструктурою, вони переносяться між хмарами і збірками ОС. Контейнери - відмінний спосіб зв'язати і запустити ваші програми. У виробничому середовищі необхідно управляти контейнерами, які запускають додатки, і гарантувати відсутність простоїв. Наприклад, якщо контейнер виходить з ладу, необхідно запустити інший контейнер. Не було б простіше, якби така поведінка оброблялося системою?

Ось тут Kubernetes приходить на допомогу! Kubernetes дає вам

фреймворк для гнучкої роботи розподілених систем. Він займається масштабуванням і обробкою помилок в додатку, надає шаблони розгортання і багато іншого. Наприклад, Kubernetes може легко управляти канарковим розгортанням вашої системи. Kubernetes - це не традиційна комплексна система PaaS (платформа як послуга). Оскільки Kubernetes працює на рівні контейнерів, а не на рівні обладнання, у нього є певні загальноприйнятних можливості, характерні для PaaS, такі як розгортання, масштабування, балансування навантаження, ведення журналів і моніторинг. Проте, Kubernetes це не монолітне рішення, тому зазначені можливості за замовчуванням є додатковими і підключаються. У Kubernetes є компоненти для створення платформи розробки, але він зберігає право вибору за користувачем і гнучкість там, де це важливо.

2 СТРУКТУРА СИСТЕМИ

Структура системи залежить від потреб виконання операцій. При всіх обраних системах та сервісах необхідно поєднати їх в одну велику систему, що виконує багато операцій з однією метою. Запуск сервісів підтримки середовища виконує функцію забезпечення працездатності платформи під час виконання тестування. Другим етапом є автоматизований запуск сервісу реалізації системи та виконання автоматизованих тестів. Після завершення другого етапу, а саме виконання автоматизованих тестів, починається третій етап на якому автоматично відбувається автоматична генерація звітів.

Кожен з цих етапів важливо поєднати в одну систему. Система має бути побудована без будь-яких конфліктів між мікросервісами, тому важливим завданням є використання офіційної документації для інтеграції кожного сервіса, мікросервіса та систем в одну єдину системау автоматизованого тестування. Важливо розуміти, що для проходження автоматизованих тестів також потрібно підтримувати працездатність платформи де і буде виконуватись тестування.

2.1 Реалізація WEB UI елементів

Управління формами в Django використовує ті ж самі техніки: відображення отримує запит, виконує необхідні дії, що включають в себе читання даних з моделей, генерацію і повернення сторінки HTML (з шаблону, в який передається контекст, у якому дані, які і будуть показані). Що робить цей процес більш складним, так це те, що серверної частини треба додатково обробити дані, надані користувачем і, в разі виникнення помилок, знову перемалювати сторінку. Показ форми за замовчуванням при першому запиті з боку користувача.

Форма може містити порожні поля (наприклад, якщо ви створюєте новий запис в базі даних), або вони (поля) можуть мати початкові значення (наприклад, якщо ви змінюєте запис, або хочете заповнити її будь-яким початковим значенням).

Форма в даний момент є незв'язаної, тому що вона не асоціюється з будь-якими введеними користувачем даними (хоча і може мати початкові значення).

Отримання даних з форми (з HTML-форми) з боку клієнта і зв'язування їх з формою (класом форми) на стороні сервера.

Зв'язування даних з формою означає, що дані, введені користувачем, а також можливі помилки, при переотрісовке надалі, будуть ставитися саме до цієї форми, а не до будь-якої ще.

Очищення та валідація даних. Очищення даних - це їх перевірка на наявність можливих значень, або вставок в поля введення (тобто очищення - це видалення неправильних символів, які потенційно можуть використовуватися для відправки шкідливого вмісту на сервер), з подальшою конвертацією очищених даних в підходящі типи даних Python.

Валідація перевіряє, значення полів (наприклад, правильність введених дат, їх діапазон і так далі). Якщо будь-які дані є невірними, то виконання перемальовування форми, але на цей раз, з уже введеними користувачем даними і повідомленнями про помилки, що описують виникли проблеми.

Якщо всі дані вірні, то виконання необхідних дій (наприклад, збереження даних, відправка листів, повернення результату пошуку, завантаження файлу і так далі)

Коли всі дії були успішно завершені, то перенаправлення користувача на іншу сторінку.

2.2 Побудова сервісу Jenkins

Jenkins Pipeline - це набір плагінів, який підтримує впровадження та інтеграцію конвеєрів безперервної доставки в Jenkins. Конвеєр безперервної доставки - це автоматизований вираз вашого процесу для отримання програмного забезпечення від контролю версій до ваших користувачів та клієнтів. Jenkins Pipeline забезпечує розширюваний набір інструментів для

моделювання простих до складних конвеєрів доставки Визначення Jenkins Pipeline зазвичай записується у текстовий файл (званий Jenkinsfile), який, у свою чергу, перевіряється у сховищі джерел керування проектом.

Інтеграція сервісу відбувається на будь-якому рівні та дозволяє обрати будь-яку технологію та мову програмування. За умови, що наша система розроблена на Python, тому і побудова Jenkins відбувається на цій мові програмування, Рис. 2.1.

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent { docker { image 'php' } }
    stages {
        stage('build') {
            steps {
                sh 'php --version'
            }
        }
    }
}
```

Рис. 2.1. Інтеграція Jenkins на Python

2.3 Спосіб розробки тестів

Після створення проекту на Java, среда розробки автоматично генерує декілька файлів - Groupid, Artefiactid, Version. Pom.xml - це файл який описує проект. Pom-файл зберігає список всіх бібліотек (залежностей), які використовуються в проекті. Після підключення додатковиъ бібліотек Selenium Java та Junit мы отрумуємо готовий Pom-файл, Рис. 2.2.

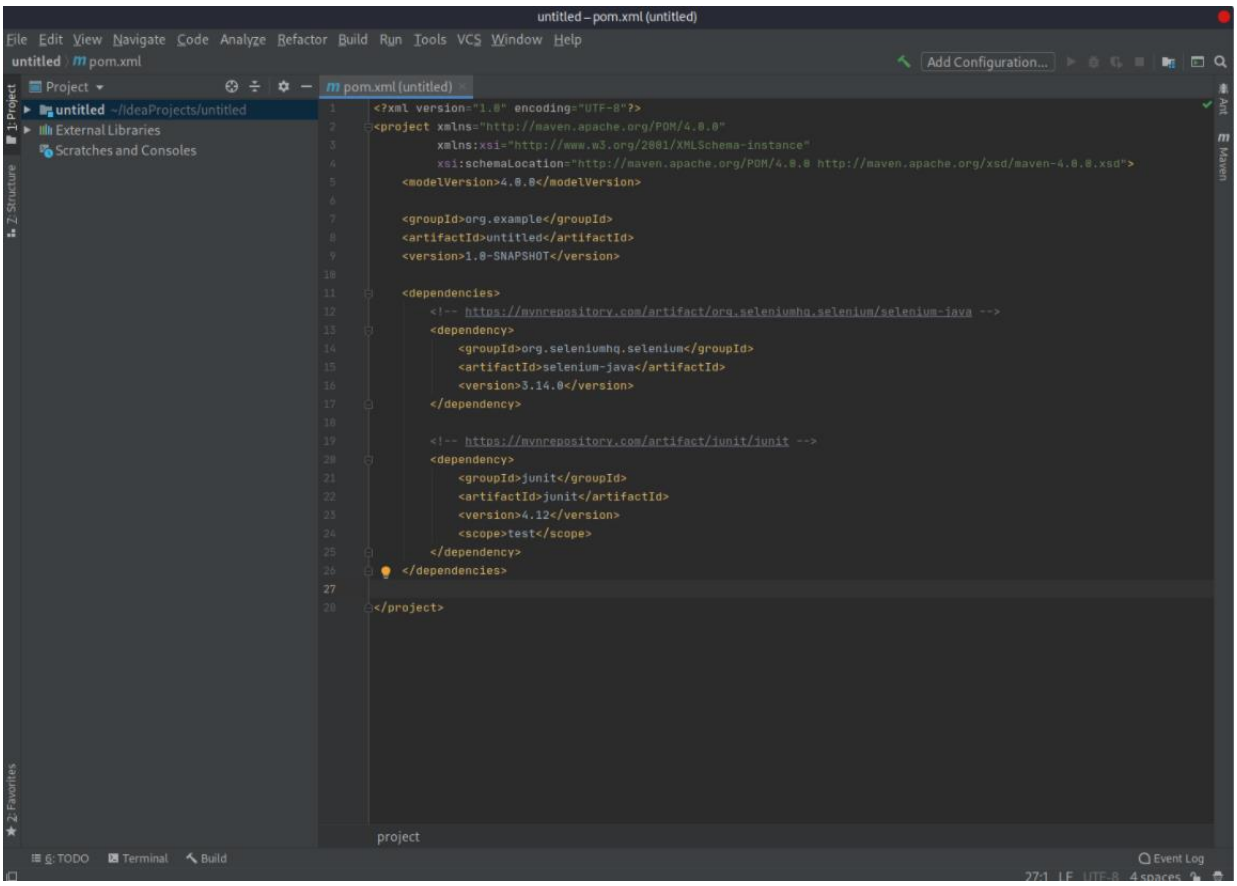


Рис. 2.2. Готовий Pom.xml файл

Розкриємо структуру проекту. Директорія src містить в собі дві директорії: «main» і «test». Для тестів використовується, відповідно, директорія «test». Відкриємо директорію «test», кліком правої кнопки миші по директорії «java» виберемо пункт «New», а потім пункт «Package». У діалоговому вікні необхідно ввести назву пакета. Ім'я базового пакета повинно носити теж ім'я, що і Groupid - «org.example».

Наступний крок - створення класу Java, в якому пишеться код автотеста. Кліком правої кнопки миші за пакунком виберемо пункт «New», а потім пункт «Java Class», Рис. 2.3.

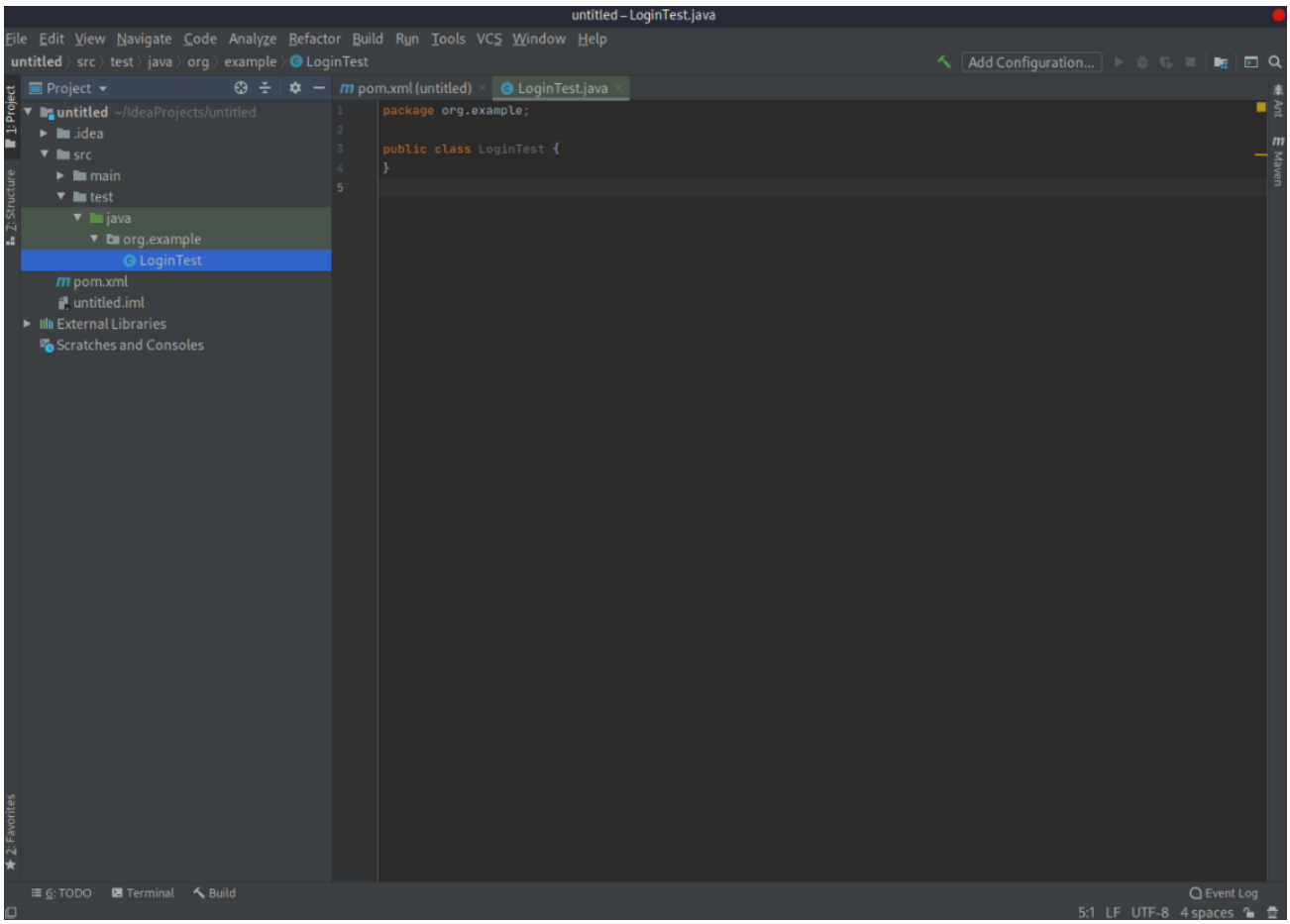


Рис. 2.3. Створення класу Java

Перш ніж почати, необхідно налаштувати IDE. Кліком правої кнопки миші по назві проекту виберемо пункт «Open Module Settings». У вікні, у вкладці «Sources» поле «Language level» за замовчуванням має значення 5. Необхідно змінити значення поля на 8 (для використання всіх можливостей, присутніх в цій версії Java) і зберегти зміни, Рис. 2.4.

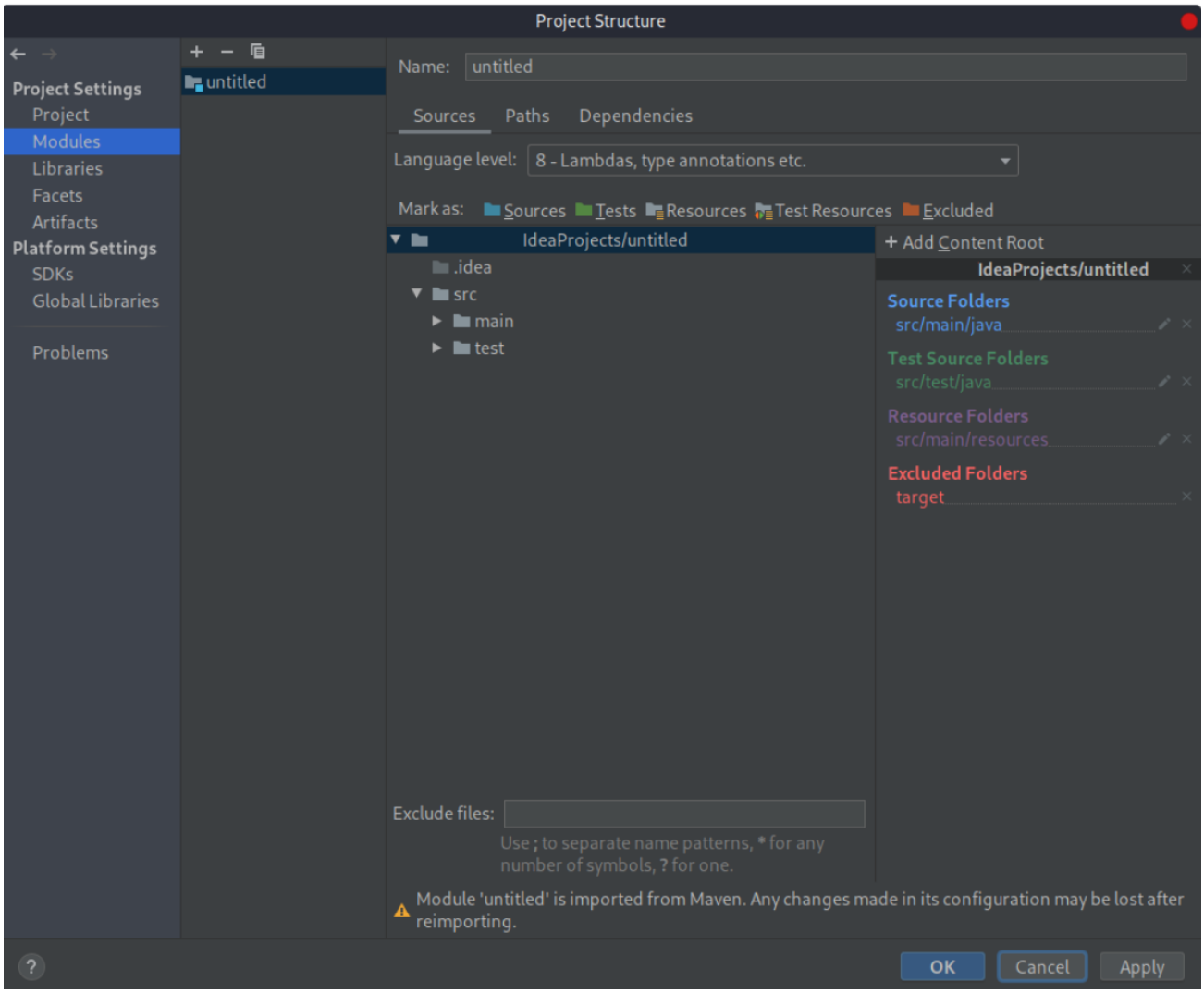


Рис. 2.4. Налаштування IDE

Далі необхідно змінити версію компілятора Java: натиснемо меню «File», а потім виберемо пункт Settings. У відкритому перейдемо «Build, Execution, Deployment» -> «Compiler» -> «Java Compiler». За умовчанням встановлена версія 1.5. Змінимо версію на 8 і збережемо зміни, Рис. 2.5.

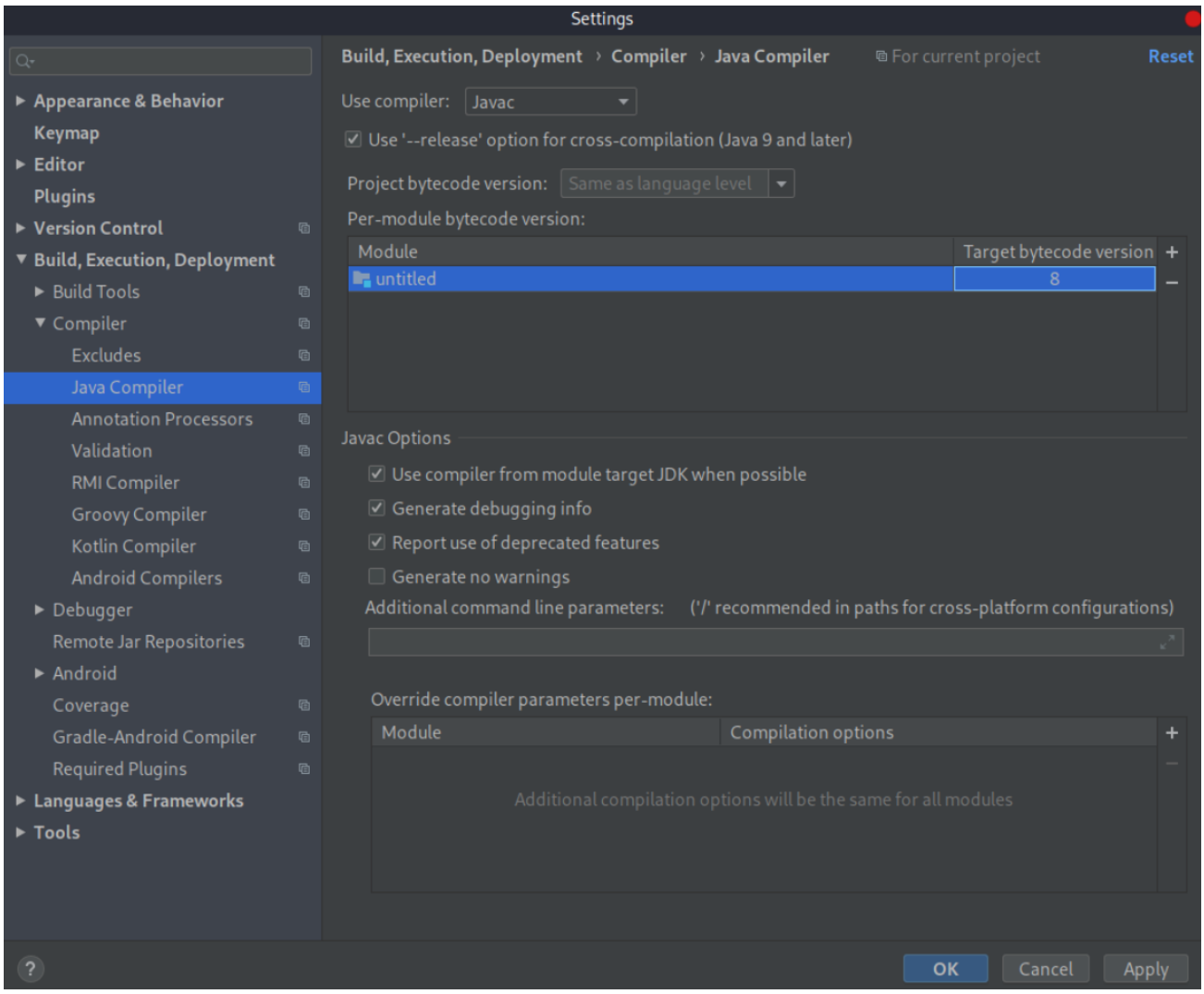


Рис. 2.5. Зміна компілятора

У класі LoginTest буде описана логіка тесту. Створимо в цьому класі метод «setup ()», в якому будуть описані попередні налаштування. Отже, для запуску браузера необхідно створити об'єкт драйвера, Рис. 2.6.

```
WebDriver driver = new ChromeDriver();
```

Рис. 2.6. Створення метода

Перед створенням об'єкта WebDriver слід встановити залежність, що визначає шлях до chromedriver, Рис. 2.7.

```
System.setProperty("webdriver.chrome.driver", "/usr/bin/chromedriver");
```

Рис. 2.7. Залежність шляху до chromedriver

Трапляється, що елементи на сторінках доступні не відразу, і необхідно дочекатися появи елемента. Для цього існують очікування. Вони бувають двох видів: явні і неявні. У прикладі буде використано неявне очікування Implicitly Wait, яке задається спочатку тесту і буде працювати при кожному виклику методу пошуку елемента, Рис. 2.8.

```
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
```

Рис. 2.8. Реалізація неявного очікування

Таким чином, якщо елемент не знайдений, то драйвер буде чекати його появи в перебігу заданого часу (10 секунд) і кроком в 500 мс. Як тільки елемент буде знайдений, драйвер продовжить роботу, однак, в іншому випадку тест впадемо по закінченню часу.

У пакеті «org.example» створимо ще один клас «ConfProperties», який буде читати записані в файл «conf.properties» значення, Рис. 2.9.

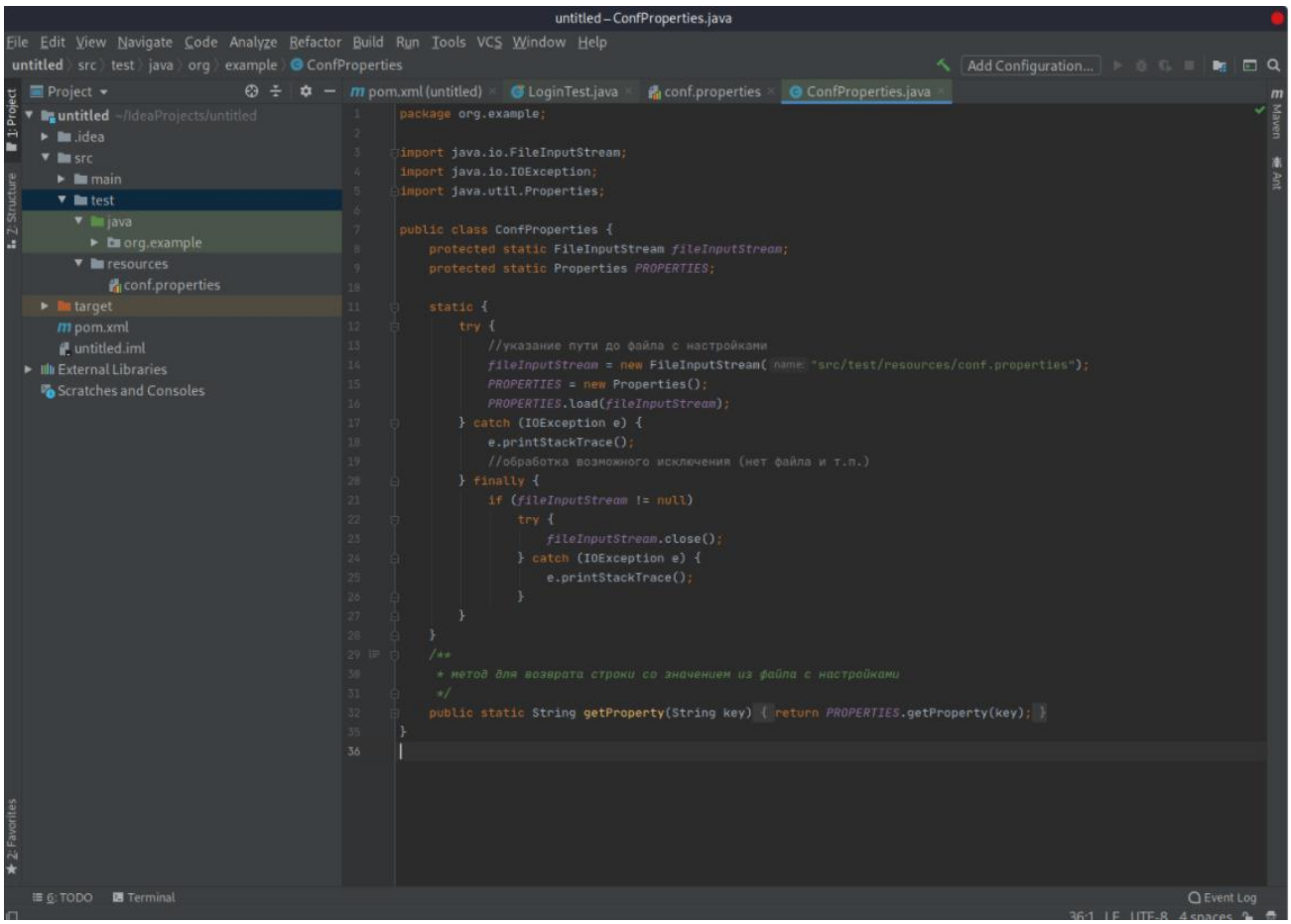


Рис. 2.9. Створення нового класу

При використанні Page Object елементи сторінок, а також методи безпосередньої взаємодії з ними, виносяться в окремий клас. Створимо в пакеті «org.example» клас LoginPage, який буде містити локацію елементів сторінки логіна і методи для взаємодії з цими елементами. Відкриємо сторінку авторизації в браузері Chrome. Для визначення локаторів елементів сторінки, з якими буде взаємодіяти автотест, скористаємося інструментами розробника. Кліком правої кнопки миші викличемо меню «Переглянути код». У панелі, що з'явилася натиснемо на значок курсора (лівий верхній кут панелі розробника) і наведемо курсор на цікавий для нас елемент - поле введення логіна. В результаті ми побачимо цей елемент серед безлічі інших. Тепер ми можемо скопіювати його локацію. Для цього натискаємо правою кнопкою миші по виділеному в панелі розробника елементу, вибираємо меню «Copy» -> «Copy XPath». Таким чином необхідно знайти всі елементи інтерфейсу з

якими потрібно взаємодіяти під час виконання автоматизованих тестів, Рис. 2.10.

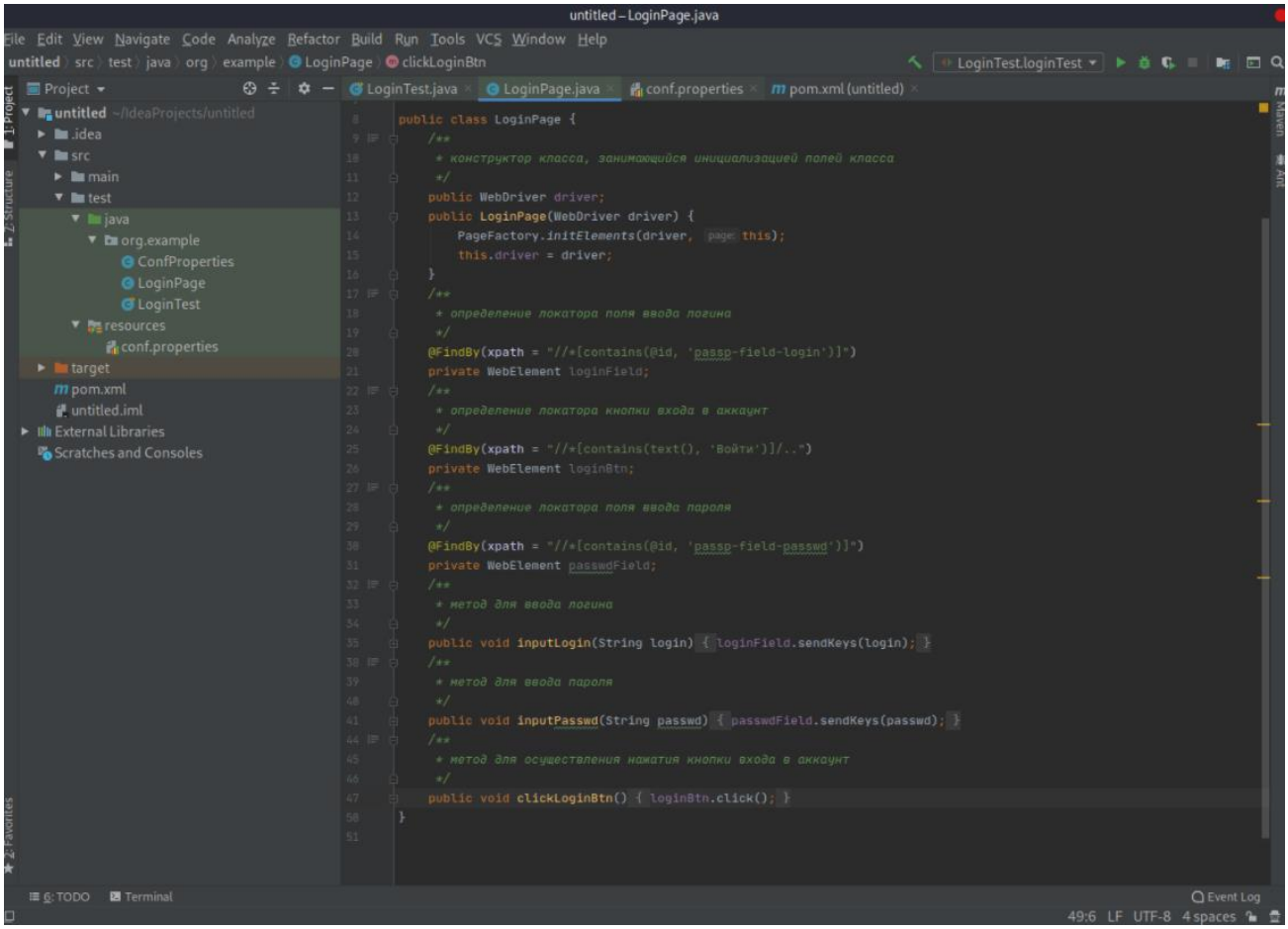


Рис. 2.10. Опис елементів інтерфейсу

Тепер можна перейти безпосередньо до написання логіки тесту. Створимо метод loginTest () і позначимо його відповідної анотацією, Рис. 2.11.

```
@Test
    public void loginTest() {
        //значение login/password берутся из файла настроек по аналогии с chromedriver
//и Loginpage
//вводим логин
        loginPage.inputLogin(ConfProperties.getProperty("login"));
    //нажимаем кнопку входа
        loginPage.clickLoginBtn();
    //вводим пароль
        loginPage.inputPasswd(ConfProperties.getProperty("password"));
    //нажимаем кнопку входа
        loginPage.clickLoginBtn();
        //получаем отображаемый логин
        String user = profilePage.getUserName();
    //и сравниваем его с логином из файла настроек
        Assert.assertEquals(ConfProperties.getProperty("login"), user); }
```

Рис. 2.11. Створення автотесту авторизації

Таким чином потрібно описати всі необхідні тести базуючись на основних вимогах до програмного забеспечення. Після створення всіх необхідних тестів потрібно інтегрувати систему автоматичної генерації звітів.

2.4 Інтеграція Allure

Для того щоб отримати інформацію про виконання автотеста, з якої потім буде сформований інформативний Allure-звіт, необхідно до проекту з автотестами підключити адаптер. Під час виконання тестів адаптер збере всю необхідну інформацію: час початку тесту, час його закінчення, з якою помилкою тест впав і т.д. Після проходження кожного тесту адаптер збере всі дані і агрегує їх в .json файли і файли ресурсів. З безлічі таких файлів в подальшому і буде сформований сам звіт.

Даний адаптер дозволяє збирати інформацію з тестів, розроблених з використанням фреймворку Junit4. Для підключення адаптера необхідно модифікувати pom.xml вашого проекту, Рис. 2.12.

```
<dependencies>
    ...
    <dependency>
        <groupId>io.qameta.allure</groupId>
        <artifactId>allure-junit4</artifactId>
        <version>${allure-junit4.version}</version>
    </dependency>
    ...
</dependencies>
```

Рис. 2.12 Підключення адаптера

Також необхідно модифікувати (або створити) секцію build, Рис. 2.13.

```
<build>
   <plugins>
      ...
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-surefire-plugin</artifactId>
         <version>${maven-surefire-plugin.version}</version>
         <configuration>
            <testFailureIgnore>false</testFailureIgnore>
            <argLine>-javaagent:"${settings.localRepository}/org/aspectj/aspectjweaver/${aspect
j.version}/aspectjweaver-${aspectj.version}.jar"</argLine>
            <properties>
               <property>
                  <name>listener</name>
                  <value>io.qameta.allure.junit4.AllureJunit4</value>
               </property>
            </properties>
            <systemProperties>
               <property>
                  <name>allure.results.directory</name>
                  <value>${project.build.directory}/allure-results</value>
               </property>
            </systemProperties>
         </configuration>
         <dependencies>
            <dependency>
               <groupId>org.aspectj</groupId>
               <artifactId>aspectjweaver</artifactId>
               <version>${aspectj.version}</version>
            </dependency>
         </dependencies>
      </plugin>
      ...
   </plugins>
</build>
```

Рис. 2.13 Модифікація секції build

Вихідні файли являють собою безліч JSON-файлів з результатами тестів, а також файлів деяких інших типів. Дані файли будуть автоматично генеруватися після прогону автотестів (в нашому прикладі в директорії target / allure-results). Потім з цих файлів необхідно зібрати зручний для читання html-звіт. Для цього потрібно додати в pom.xml проекту allure-maven плагін. Для цього модифікуємо секцію build наступним чином, Рис. 2.14.

```
<build>
   <plugins>
      ...
      <plugin>
         <groupId>io.qameta.allure</groupId>
         <artifactId>allure-maven</artifactId>
         <version>2.9</version>
         <configuration>
            <reportVersion>2.6.0</reportVersion>
         </configuration>
      </plugin>
      ...
   </plugins>
</build>
```

Рис. 2.14 Підключення плагіну

Після виконання автотестів в блоці Build History навпроти номера збірки з'явиться значок Allure, клікнувши по якому, ви побачите сформований html-звіт, Рис. 2.15.

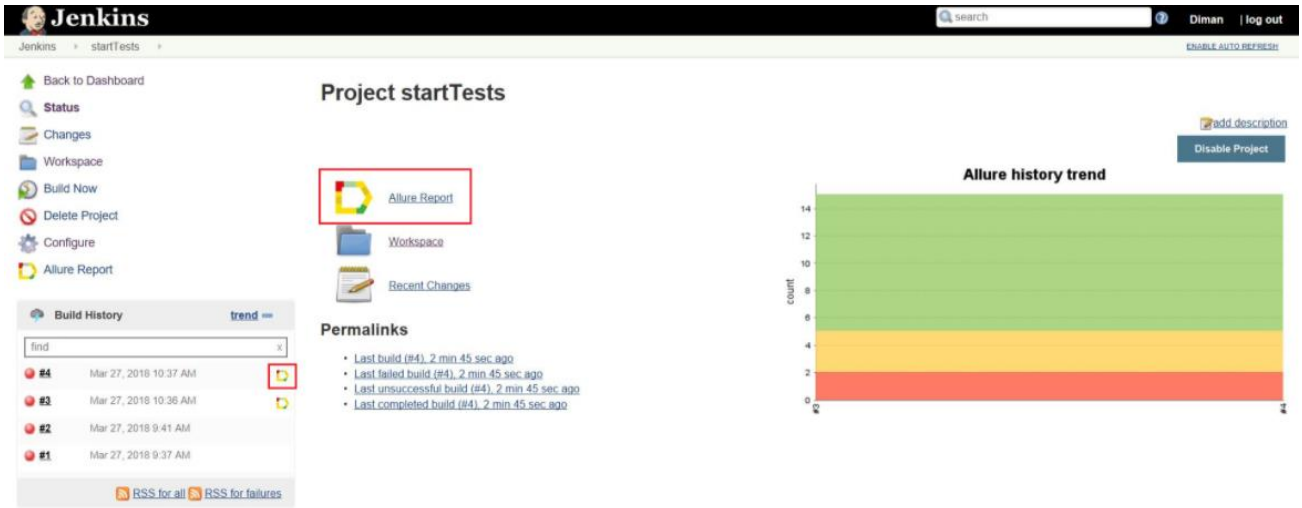


Рис. 2.15. Формування звіту

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

1. Блок ALLURE REPORT. Включає в себе дату і час проходження тесту, загальна кількість прогнати кейсів, а також діаграму з зазначенням відсотка та кількості успішних, що впали і зламалися в процесі виконання тестів.
2. Блок TREND. Показує тренд проходження тестів від збірки до збірки.
3. Блок SUITES. Показує розподіл результатів тестів за тестовими наборам. В даному випадку тести були розподілені по пакетам.
4. Блок ENVIRONMENT. Показує тестове оточення, на якому запускалися тести. Дана інформація потрапляє до звіту зі спеціального файлу, розташованого в проекті.
5. Блок CATEGORIES. Показує розподіл неуспішно минулих тестів за видами дефектів.
6. Блок FEATURES BY STORIES. Показує розподіл тестів по функціоналу, який вони перевіряють.
7. Блок EXECUTORS. Показує виконавця поточної збірки. Якщо виконання вироблялося на інструменті CI (наприклад, на Jenkins), то буде надана інформація про Джоб і номері збірки, Рис 2.16.

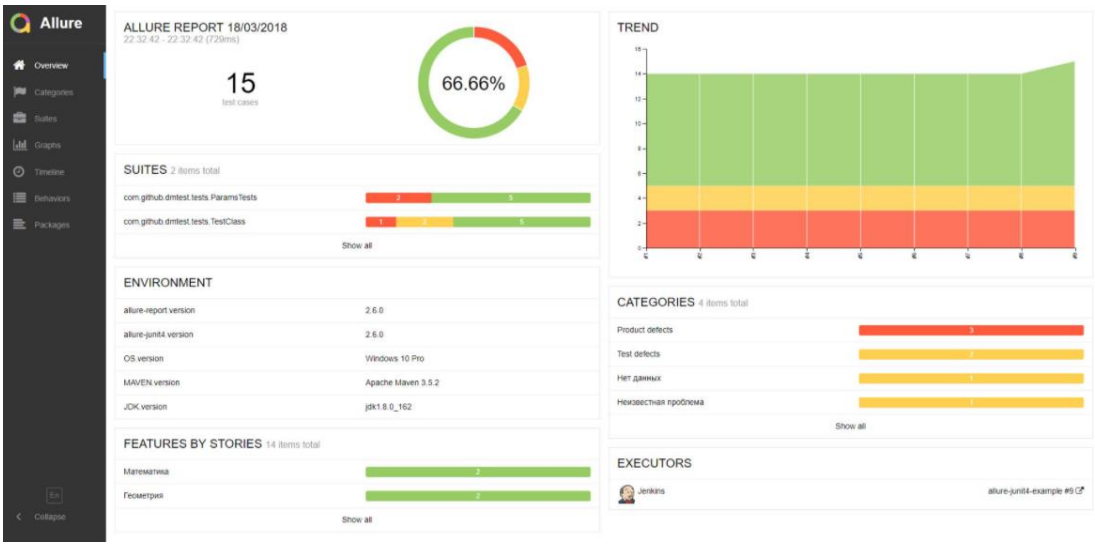


Рис. 2.16. Загальний вигляд звіту

3 ОПИС СИСТЕМИ

Створена система автоматизованого тестування здатна забезпечити процес оновлення або подальшої розробки програмного забезпечення необхідними перевірками базуючись на відповідних вимогах до продукту. Інтеграція системи тестування має перевагу легкого вбудовування в будь-яку систему розгортування кодової бази та реалізації запуску продукту. Завдяки вибору новітніх технологій та використання актуальних сервісів система має здатність оновлюватись та мігрувати із одного сервіса реалізації до іншого. Головне значення системи – забезпечити команду інформативною інформацією про якість продукту. Інтеграція до системи збору платформи (продукту) дає змогу автоматизувати процесс запуску тестів після кожної зборки.

3.1 Загальні відомості

Система створена на базі автономного та хмарного програмного забезпечення тому необхідні ресурси для використання системи можуть бути обрані без будь-яких обмежень. В ході розробки системи автоматизованого тестування були використані новітні інструменти та актуальні мови програмування. Кожна з використаних сервісів знаходиться у вільному доступі та може бути використана для будь-яких цілей, тому робить цю систему абсолютного безкоштовную для використання. До незаперечних переваг автоматичного тестування відносяться:

Циклічність - гарантія того, що створені Автотест завжди будуть дотримуватися один алгоритм перевірок, який не пропустить передбаченого тесту при одному з випадків застосування.

Швидкий результат - відпадає необхідність в часі, яке потрібно людині для звірки проміжних підсумків, підтвердження безпомилковості при виконанні вимог.

Дешевизна - одноразово створений софт для тестування вимагає менше зусиль для аналізу отриманих даних, в результаті, замінюючи ті ж обсяги ручного тестування без втрат в якості.

Простір в звітності - готові результати легко обробляти, а самі звіти неважко поширити по зацікавленим особам.

Вільні руки - людина-тестер, під час роботи програми, може виконувати іншу корисну діяльність, яка автоматизації не схильна до. Дозволено проводити тестування в той час, коли навантаження на числові ресурси знижена (в неробочий час).

3.2 Функціональне призначення

При створенні програмного забеспечення помилки обходяться дорого, це особливо помітно в медицині, де від якості ПО залежать людські життя, або в сфері банкінгу, де можливі великі фінансові втрати. Автоматизація тестування дозволяє організувати постійну перевірку якості продукту. Давайте розберемося, в яких випадках вона необхідна.

Одні компанії помилково вважають, що автоматизація - марна трата часу і коштів, інші - що це крутий тренд і «таблетка» від усіх хвороб. Розглянемо, де золота середина і в чому сенс автоматизації.

Найменша помилка в програмному забезпеченні загрожує величезними витратами. Чим краще збудовані процеси розробки, тим менший ризик. Однак, якщо звернутися до фактів, ми бачимо, що недоліки допускають навіть такі гіганти, як Google, Microsoft або провідні банки.

Очевидно, що три перших затвердження надмірно категоричні, а тому, як це часто буває, помилкові: для забезпечення якості IT-продукту важливо розглядати комплекс з безлічі параметрів, а не тільки автоматизацію тестування.

У свою чергу, два останніх затвердження дійсно справедливі: грамотна автоматизація тестування допомагає і в прискоренні релізів, і в збільшенні покриття кейсів. При цьому ручне забезпечення якості та автоматизація, як правило, однаково важливі і використовуються в комплексі. Крім того, кожен проект індивідуальний, і іноді в автоматизації немає необхідності.

ВИСНОВКИ

Мета та задачі, поставлені в цій роботі, були виконані, а саме:

1. Розроблена і програмно реалізована система автоматизованих тестів для переверіки WEB UI системи наповненя БД.
2. Оглянуто сервіси та системи для реалізації програмної підтримки.
3. Обрано новітні та актуальні інструменти реалізації.
4. Інтегровано систему генерації звітів.
5. Реалізовано систему підтримки середовища програмною працездатності.

Розроблена система автоматизованого тестуваня призначена для розв’язку задачи підтримки якості програмного забеспечення для наповненя БД необхідною інформацією.

Розв’язок задачі реалізується на основі використаня WEB UI елементів та керування ними за допомогою автоматичних тестів.

Розв’язок задачі візуалізується у вигляди автоматично згенерованих звітів які відображують якість програмного забеспечення після будь-яких змін в кодовій базі.

Завдяки використанню автоматизованої системи тестування, розробка та зміна програмного забеспечення значно спрощується та не потребую значних ресурсів для відтворення результатів тестування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. https://java.com/ru/download/help/whatis_java.html
2. https://ru.wikipedia.org/wiki/Java
3. https://ru.wikipedia.org/wiki/Java_Runtime_Environment
4. https://www.oracle.com/java/technologies/
5. https://www.python.org/
6. https://ru.wikipedia.org/wiki/Python
7. https://habr.com/ru/hub/python/
8. https://www.djangoproject.com/
9. https://ru.wikipedia.org/wiki/Django
10. https://developer.mozilla.org/ru/docs/Learn/Server-side/Django
11. https://habr.com/ru/hub/django/
12. https://www.jenkins.io/
13. https://ru.wikipedia.org/wiki/Jenkins
14. https://livetyping.com/ru/blog/ios-ci
15. https://habr.com/ru/company/selectel/blog/339390/
16. https://docs.qameta.io/allure/
17. https://habr.com/ru/company/sberbank/blog/359302/
18. http://allure.qatools.ru/
19. https://github.com/allure-framework
20. https://github.com/allure-framework/allure2
21. https://ru.selenide.org/
22. https://habr.com/ru/post/274071/
23. https://ru.selenide.org/documentation/selenide-vs-selenium.html
24. https://github.com/selenide/selenide
25. https://ru.wikipedia.org/wiki/Apache_Maven