Selenium Automation Testing

An internship report submitted
for completion of the
Bachelor of Technology in
Computer Science & Engineering (Cyber Security)

By

Krishnashis Das (Roll no: 10331720029)
Siddhartha Chakraborty (Roll no: 10331720050)

Under the supervisor of
Wipro Certified Faculty Member of Haldia Institute of Technology

HALDIA INSTITUTE OF TECHNOLOGY
DEPARTMENT OF CSE(CYBER SECURITY)
HALDIA, PURBA MEDINIPUR, WEST BENGAL, INDIA 2024

DECLARATION

We hereby declare that this project work titled "**Selenium Automation Testing**" is our original work and no part of it has been submitted for any other degree purpose or published in any other form till date.

Signature of students

…………………………….
Krishnashis Das (Roll no: 10331720029)

…………………………….
Siddhartha Chakraborty (Roll no: 10331720050)

ACKNOWLEDGEMENT

We use this opportunity to express our gratitude to God Almighty for his gracious blessings and for bringing our endeavour to a fruitful conclusion. We sincerely thank our respected mentor, Dr. Arpita Mazumdar, WIPRO for teaching us the proper path and for her wise counsel and direction at this critical juncture.

We owe equal gratitude to the diverse team members whose varied skills and unwavering dedication drove the project's progress. Collaboratively, we confronted challenges, leveraging our strengths and fostering an environment of mutual support and motivation. The Haldia Institute of Technology deserves recognition for enabling our vision by providing crucial resources and steadfast support, without which our ambitions would have remained unrealized.

The engagement of faculty members, extending beyond our mentor, played a pivotal role in enhancing our research and findings. Their openness to sharing experiences and insights added depth and practicality to our project. Our deepest appreciation also goes to the authors, researchers, and scholars whose foundational work laid the groundwork for our exploration and discoveries. Their contributions formed the bedrock upon which our project stood.

In the end, there are no enough words to describe how grateful we are to our loved ones for their support and encouragement; without them, we could not have progressed to this point.

INDEX

List of Figures

List of Tables

*Chapter **1***

INTRODUCTION

In the dynamic and ever-evolving world of software development, ensuring that software behaves as intended is paramount to achieving success. This assurance is realized through a rigorous and systematic process known as software testing. Software testing is an indispensable phase in the software development lifecycle, where applications or systems are meticulously evaluated to identify defects, errors, or discrepancies between the expected and actual outcomes. As a vital component of quality assurance, software testing is designed to verify that software meets specified requirements, functions correctly, and delivers a positive user experience, thereby safeguarding the software's integrity and reliability.

The practice of software testing encompasses a diverse array of techniques, methodologies, and tools, each tailored to address different facets of the software’s functionality, performance, security, and usability. From the granular examination of individual components in unit testing to the comprehensive evaluation of the entire system in system testing, the testing process is meticulously designed to uncover vulnerabilities, mitigate risks, and ensure the robustness and reliability of the software. Additional layers of testing, such as integration testing, acceptance testing, and regression testing, further contribute to a holistic assessment of the software, ensuring that each component interacts seamlessly and that new updates do not introduce new issues.

Beyond its technical dimensions, software testing is a collaborative endeavour that requires close coordination between developers, testers, and stakeholders. This collaboration is essential for defining test objectives, designing test cases, executing tests, and interpreting results. Effective communication and feedback loops within the testing process facilitate continuous improvement, driving the refinement and enhancement of software products over time. The involvement of diverse perspectives ensures that the software is scrutinized from multiple angles, promoting a thorough and comprehensive evaluation.

In today’s fast-paced and competitive software landscape, the importance of software testing cannot be overstated. It serves as a cornerstone of quality assurance, instilling confidence in the software’s performance, stability, and security. By investing in thorough and systematic testing practices, organizations can mitigate risks, accelerate time-to-market, and deliver superior products that meet the evolving needs and expectations of users. The benefits of software testing extend beyond the immediate detection of defects; it also fosters a culture of quality within the development team, encouraging best practices and a proactive approach to problem-solving.

Moreover, the adoption of automated testing tools and practices has revolutionized the testing landscape, enabling more efficient and repeatable testing processes. Automation helps in executing large test suites consistently and quickly, freeing up human testers to focus on more complex and exploratory testing activities. This integration of automated and manual testing strategies ensures a balanced approach, leveraging the strengths of both methods to achieve comprehensive test coverage.

In conclusion, software testing is an essential practice that underpins the success of software development projects. It ensures that software products are reliable, secure, and capable of meeting user expectations. As software continues to play a critical role in various industries and aspects of daily life, the role of software testing in delivering high-quality software is more crucial than ever. By prioritizing software testing, organizations can not only enhance the quality and reliability of their products but also gain a competitive edge in the market.

1.1Abstract

Selenium automation testing has emerged as a cornerstone in modern software development, offering a robust framework for efficiently and effectively testing web applications. Selenium, an open-source tool suite, provides a powerful set of tools and libraries for automating web browsers and conducting functional testing across various platforms and browsers. Its versatility and scalability have made it an essential tool for developers and testers aiming to ensure the quality and reliability of web applications. Selenium supports a wide range of programming languages, including Java, C#, Python, Ruby, and JavaScript, allowing testers to write scripts in the language they are most comfortable with. This flexibility makes Selenium an attractive choice for diverse development teams.

One of the key advantages of Selenium is its ability to perform cross-browser testing. Selenium WebDriver, a core component of the Selenium suite, interacts directly with the browser, mimicking the actions of a real user. This enables the execution of tests across different browsers such as Chrome, Firefox, Safari, and Edge, ensuring that the web application performs consistently and correctly across all supported environments. By automating this process, Selenium significantly

reduces the manual effort required for testing and helps in identifying browser-specific issues early in the development cycle, leading to more reliable and user-friendly applications.

Moreover, Selenium Grid enhances the testing process by allowing the concurrent execution of tests across multiple machines and environments. This parallel testing capability dramatically reduces the time needed to execute comprehensive test suites, which is particularly beneficial for continuous integration and continuous deployment (CI/CD) workflows. Selenium Grid's ability to distribute tests across a wide array of configurations ensures thorough coverage and quick feedback, enabling rapid iteration and improvement of the application. Additionally, the extensive community support and rich ecosystem of plugins and integrations further bolster Selenium's utility, making it a comprehensive solution for modern web application testing needs. Overall, Selenium automation testing is indispensable for maintaining high standards of software quality and ensuring a seamless user experience.

1.1.1 Overview of Selenium Automation Testing

These abstract aims to provide a comprehensive overview of Selenium automation testing, highlighting its key features, benefits, and applications. We delve into the core components of Selenium, including Selenium WebDriver, Selenium IDE, and Selenium Grid, outlining their roles in automating web interactions, creating test scripts, and executing tests in parallel across multiple environments.

Selenium WebDriver is the most widely used component of the Selenium suite. It provides a programming interface to create and execute test scripts that can interact with web elements just like a real user would. WebDriver supports multiple programming languages, including Java, C#, Python, Ruby, and JavaScript, allowing testers to write tests in their preferred language. It also supports various browsers such as Chrome, Firefox, Safari, and Edge, ensuring cross-browser compatibility.

Selenium IDE is a browser extension that provides a user-friendly interface for creating and recording test cases without needing to write code. It is particularly useful for quick prototyping and for testers who may not have strong programming skills. Selenium IDE allows for easy playback of recorded tests, making it a valuable tool for regression testing.

Selenium Grid enables the parallel execution of test cases across different machines and browsers. This component is crucial for scaling tests and reducing the overall test execution time. By distributing tests across a grid of nodes, Selenium Grid helps ensure that applications work consistently across various environments, improving the reliability and coverage of tests.

1.1.2 Best Practices for Implementing Selenium Automation Testing

To maximize the benefits of Selenium automation testing, it is essential to follow best practices and strategies:

1. Test Case Design: Well-designed test cases are critical for effective automation. Test cases should be modular, reusable, and maintainable. Writing clear and concise test scripts helps ensure that tests are easy to understand and modify.
2. Test Data Management: Proper management of test data is crucial for accurate and consistent test results. Using data-driven testing techniques, where test data is separated from test scripts, allows for more flexible and comprehensive testing.
3. Integration with CI/CD Pipelines: Integrating Selenium tests with continuous integration/continuous deployment (CI/CD) pipelines enhances the overall efficiency of the software development process. Automated tests can be triggered with each code commit, providing immediate feedback, and facilitating continuous quality assurance.
4. Regular Maintenance: Automation scripts require regular updates and maintenance to remain effective. As web applications evolve, test scripts must be revised to reflect changes in the application. Regular maintenance ensures that automated tests continue to provide accurate and relevant results.
5. Parallel Testing: Leveraging Selenium Grid for parallel testing can drastically reduce test execution times. By running tests concurrently across multiple nodes, teams can achieve faster feedback and more efficient use of resources.

Selenium automation testing is a foundational framework for ensuring the quality and performance of web applications in today's digital landscape. By harnessing the power of Selenium, organizations can streamline their testing processes, enhance the quality of their web applications, and deliver superior user experiences. Through effective planning, execution, and maintenance of Selenium test suites, teams can achieve greater efficiency, reliability, and scalability in their testing efforts.

*Chapter **2***

HISTORY OF SELENIUM

Selenium's story begins in 2004 with Jason Huggins, a developer at ThoughtWorks. Huggins was tasked with testing an internal application, and he quickly realized the inefficiency of manually repeating tests every time there was a change. To streamline this process, he developed a JavaScript library that could automate interactions with a web page, enabling automated testing across multiple browsers. This innovation led to the creation of Selenium Core, which formed the backbone of Selenium Remote Control (RC) and Selenium IDE.

Selenium RC was revolutionary at the time because it allowed testers to control a browser from their preferred programming language, a capability unmatched by other tools. Despite its advantages, Selenium RC had limitations. The JavaScript-based automation engine faced security restrictions imposed by browsers, making certain actions impossible. This issue became more pronounced as web applications grew more complex and started utilizing advanced browser features.

In 2006, Simon Stewart, an engineer at Google, recognized these limitations while using Selenium. He envisioned a tool that could communicate directly with the browser using native methods, bypassing the constraints of the JavaScript environment. This led to the inception of the WebDriver project. WebDriver aimed to address the shortcomings of Selenium by providing a more robust and flexible automation framework.

Fast forward to 2008, a year marked by significant global events like the Beijing Olympics and the financial crisis triggered by mortgage defaults in the United States. Amid these events, the testing community witnessed a pivotal development: the merging of Selenium and WebDriver. Selenium had garnered substantial community and commercial support, while WebDriver represented the future of browser automation with its innovative approach.

The merger of the two projects was formally announced in a joint email from Simon Stewart on August 6, 2009. Stewart explained the rationale behind the merger: WebDriver addressed several limitations of Selenium, such as bypassing the JavaScript sandbox, and boasted a refined API. Conversely, Selenium supported a wider range of browsers and had a strong user base. By combining their strengths, the unified project aimed to provide the best possible testing framework for users.

The integration of Selenium and WebDriver brought together the best features of both tools, leveraging the expertise of leading minds in test automation to create a comprehensive and powerful testing framework. This collaboration has since become a cornerstone in the field of automated browser testing, continuing to evolve and adapt to the needs of developers and testers worldwide.

Chapter ***3***

SELENIUM AUTOMATION TESTING

Selenium is one of the most widely used open-source Web UI (User Interface) automation testing suite. It was originally developed by Jason Huggins in 2004 as an internal tool at Thought Works. Selenium supports automation across different browsers, platforms, and programming languages.

Selenium can be easily deployed on platforms such as Windows, Linux, Solaris and Macintosh. Moreover, it supports OS (Operating System) for mobile applications like iOS, windows mobile and android.

Selenium supports a variety of programming languages through the use of drivers specific to each language. Languages supported by Selenium include C#, Java, Perl, PHP, Python and Ruby. Currently, Selenium Web driver is most popular with Java and C#. Selenium test scripts can be coded in any of the supported programming languages and can be run directly in most modern web browsers. Browsers supported by Selenium include Internet Explorer, Mozilla Firefox, Google Chrome, and Safari.

3.1 Selenium Tool Suite

Selenium Software is not just a single tool but a suite of software, each piece catering to different Selenium QA testing needs of an organization. Here is the list of tools

1. Selenium Integrated Development Environment (IDE)
2. Selenium Remote Control (RC)
3. WebDriver
4. Selenium Grid

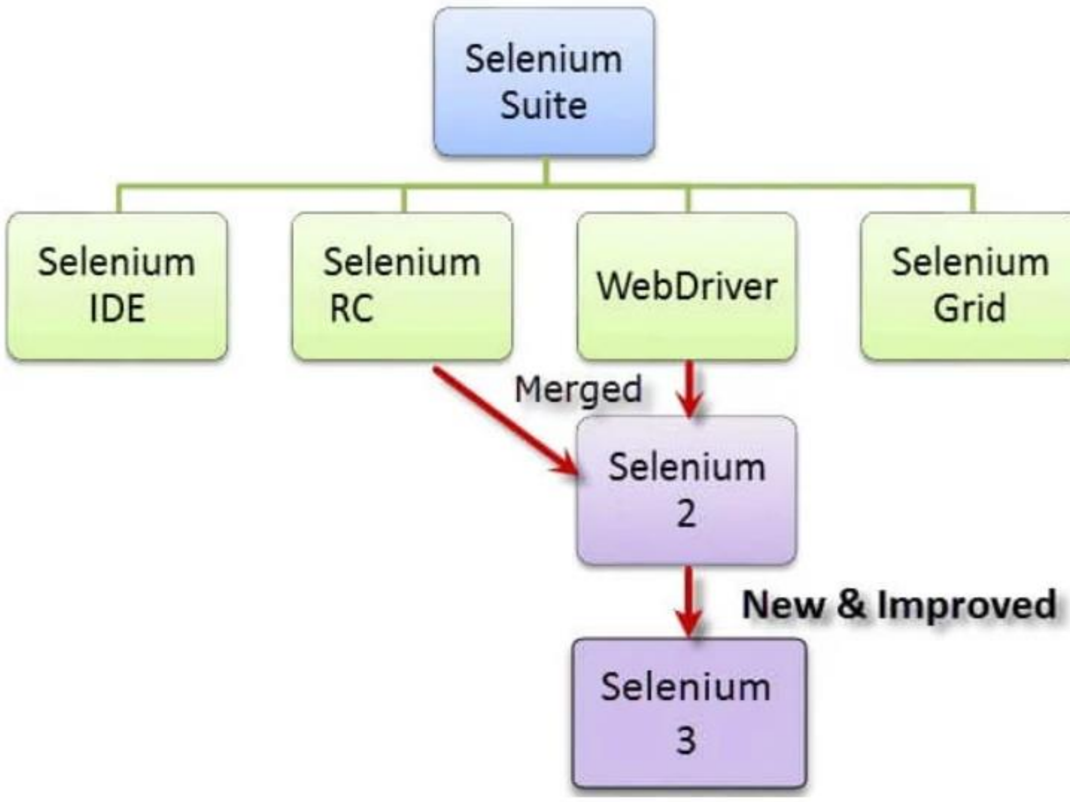


Figure 3.1 Components of Selenium

3.2 Selenium IDE

Selenium IDE (Integrated Development Environment) is a user-friendly record and playback tool designed to streamline the process of creating automated tests for web applications. Available as an add-on or extension for both Firefox and Chrome, Selenium IDE allows testers to generate tests quickly without the need for extensive programming knowledge. This accessibility makes it an ideal tool for those who are new to test automation or who have limited coding experience. By providing a straightforward interface, Selenium IDE empowers users to create functional tests with ease, thereby accelerating the testing process and reducing the learning curve associated with more complex testing frameworks.

The core functionality of Selenium IDE revolves around recording user interactions with the web application, which it then converts into reusable test scripts. These scripts capture the sequence of actions performed by the user, such as clicking buttons, entering text, and navigating between pages. Once recorded, these scripts can be played back to validate the application's behavior, ensuring that it responds correctly to various user inputs. This record and playback feature simplifies the test creation process, allowing testers to quickly build comprehensive test suites without needing to manually write test code. Additionally, Selenium IDE supports the export of recorded scripts into various programming languages, providing a pathway for more advanced users to integrate these scripts into larger test automation frameworks

Selenium IDE simplifies the creation of functional tests by eliminating the need to learn a test scripting language, thereby enabling testers to focus on ensuring the quality and reliability of the application through intuitive and accessible means. The tool also includes features such as test case management, debugging, and reporting, which further enhance its usability and effectiveness. By offering a low barrier to entry and powerful capabilities for test creation and execution, Selenium IDE serves as a valuable resource for both novice and experienced testers. It facilitates rapid development of automated tests, promotes consistent testing practices, and ultimately contributes to the delivery of robust and reliable web applications.

3.3 Selenium RC

When working with Selenium RC (Remote Control), a solid understanding of at least one programming language is essential. Selenium RC allows testers to develop responsive design tests using any scripting language they prefer, providing significant flexibility in test automation. The tool is composed of two main components: the server and the client libraries. The server acts as a mediator between the browser and the Selenium commands, translating the commands into browser actions and returning the results to the client. Despite its versatility, Selenium RC's architecture is complex, involving multiple layers of communication between the test scripts and the browser, which can lead to slower execution times and increased maintenance overhead. Additionally, Selenium RC has certain limitations, such as the need to start the server before running tests and its dependence on JavaScript for driving browser interactions, which can make it less efficient and more challenging to work with compared to its successor, Selenium WebDriver.

3.4 Selenium WebDriver

Selenium WebDriver is an enhanced and more efficient version of Selenium RC, designed to address the limitations encountered with Selenium RC. While it serves as an advanced successor to RC, Selenium WebDriver features a fundamentally different architecture. Unlike Selenium RC, which relies on a server to act as an intermediary between the test scripts and the browser, Selenium

WebDriver communicates directly with the web browser using browser-specific drivers. This direct interaction leads to faster and more reliable test execution. Selenium WebDriver also supports multiple programming languages, including Java, Python, C#, Ruby, and more, providing testers with the flexibility to write their test scripts in a language they are comfortable with. This direct and simplified communication model not only enhances performance but also reduces the complexity and potential for errors, making Selenium WebDriver a robust and preferred choice for automating web application testing.

3.5 Selenium Grid

Selenium Grid is a powerful tool designed for the concurrent execution of test cases across various browsers, machines, and operating systems simultaneously, significantly simplifying cross-browser compatibility testing. The primary advantage of Selenium Grid lies in its ability to distribute test execution across multiple environments, allowing testers to run tests in parallel. This parallel execution capability drastically reduces the time required for comprehensive testing, which is especially beneficial in large-scale projects with extensive test suites. By leveraging Selenium Grid, testers can ensure that web applications function correctly across a wide array of browser configurations and operating systems, providing a robust solution for cross-browser compatibility testing.

There are two versions of Selenium Grid: the older version, known as Grid 1, and the more recent version, Grid 2. Grid 2 introduced several significant improvements over its predecessor. One of the most notable enhancements is the support for parallel execution, which allows for multiple tests to be run concurrently on different browsers and operating systems. This capability is further augmented by Grid 2's improved scalability, enabling the addition of more nodes to the grid to handle increased testing demands. Additionally, Grid 2 offers a more straightforward configuration and management process for test nodes, making it easier for testers to set up and maintain their testing infrastructure. These advancements make Grid 2 a more efficient and effective tool for managing large-scale automated testing efforts.

Selenium Grid's ability to execute tests across diverse environments makes it an indispensable tool for thorough validation of web applications. By running tests in parallel across various browser and OS configurations, Selenium Grid helps in early detection of compatibility issues, ensuring that any problems are identified and resolved quickly. This not only accelerates the testing process but also enhances the reliability and stability of the application. The comprehensive coverage achieved through Selenium Grid's concurrent testing ensures that web applications deliver a consistent user experience, regardless of the environment in which they are accessed. As a result, Selenium Grid plays a crucial role in maintaining high-quality standards in web application development, making it an essential component of modern automated testing strategies.

3.6 Architecture of Selenium WebDriver

Selenium WebDriver Architecture is made up of four major components:

I. **Selenium Client library**

Selenium provides support to multiple libraries such as Ruby, Python, Java, etc as language bindings.

II. **JSON wire protocol over HTTP**

JSON is an acronym for JavaScript Object Notation. It is an open standard that provides a transport mechanism for transferring data between client and server on the web.

III. **Browser Drivers**

Selenium browser drivers are native to each browser, interacting with the browser by establishing a secure connection. Selenium supports different browser drivers such as Chrome Driver, Gecko Driver, Microsoft Edge WebDriver, Safari Driver, and Internet Explorer Driver.

IV. **Browsers:**

Selenium provides support for multiple browsers like Chrome, Firefox, Safari, Internet Explorer etc.

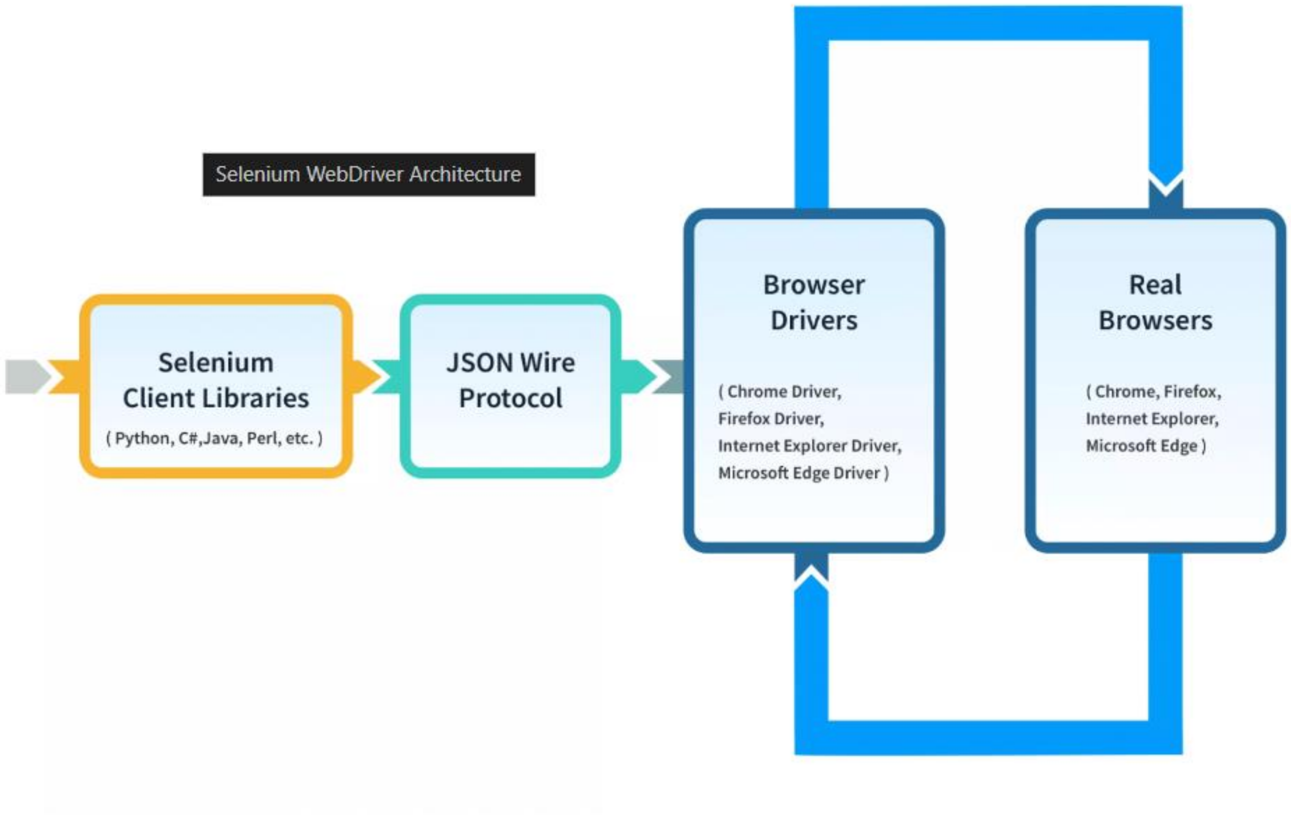

Selenium WebDriver Architecture
Selenium Client Libraries
(Python, C#,Java, Perl, etc.)
JSON Wire Protocol
Browser Drivers
(Chrome Driver, Firefox Driver, Internet Explorer Driver, Microsoft Edge Driver)
Real Browsers
(Chrome, Firefox, Internet Explorer, Microsoft Edge)

Figure 3.2 Selenium WebDriver Architecture

Chapter 4

USE OF SELENIUM

Selenium is a powerful testing tool primarily used for automating web applications for testing purposes. Here is how it's commonly utilized.

4.1 Automated Testing

Selenium is a widely adopted tool for automating the testing of web applications across multiple browsers and platforms, enabling comprehensive end-to-end testing. Leveraging Selenium's robust suite of APIs, such as Selenium WebDriver, testers can write automation scripts in various programming languages including Java, Python, C#, and more. These scripts can interact with web elements to simulate user actions such as clicking buttons, entering text, navigating through web pages, and verifying expected behaviours. Selenium WebDriver provides a programming interface to create and execute browser-based test cases, allowing for the precise control of web browsers. This includes support for different browser drivers like ChromeDriver for Google Chrome, GeckoDriver for Mozilla Firefox, and others, ensuring compatibility and coverage across different environments. Additionally, Selenium supports parallel test execution, cross-browser testing, and integration with other tools and frameworks, making it an indispensable tool for continuous integration and delivery pipelines in modern software development practices.

4.2 Cross-Browser Testing

One of Selenium's primary strengths is its robust capability for cross-browser testing, which is essential given the varying behaviours web applications can exhibit across different browsers such as Chrome, Firefox, Safari, and Edge. Different browsers can render web pages differently, interpret JavaScript in unique ways, and have varying levels of support for web standards. Selenium addresses this challenge by enabling testers to run the same set of test scripts across multiple browsers, ensuring that the application performs consistently and reliably regardless of the browser being used. This ensures that users have a seamless and uniform experience, no matter their browser choice, which is critical for web application success in a diverse user environment.

This cross-browser functionality is facilitated by browser-specific drivers, such as ChromeDriver for Chrome and GeckoDriver for Firefox, which act as intermediaries between Selenium and the browser. These drivers translate the Selenium commands into actions that the browser can perform, allowing the same test script to be executed across different browser environments. The use of browser-specific drivers ensures that tests are not only consistent but also accurate, reflecting the true behaviour of the application in each browser. This approach allows testers to catch and address browser-specific issues early in the development cycle, preventing potential user-facing problems that could arise from browser inconsistencies.

By automating the execution of test cases across various browsers, Selenium helps identify browser-specific issues early in the development cycle. This early detection is crucial for prompt resolution of compatibility problems, leading to a smoother and more uniform user experience. Additionally, the ability to run tests in parallel across multiple browsers significantly enhances the efficiency of the testing process. Testers can execute comprehensive test suites in a fraction of the time it would take to perform the same tests manually across different browsers. This not only accelerates the development cycle but also contributes to the overall quality and reliability of the web application. Selenium's robust cross-browser testing capabilities make it an indispensable tool for developers and testers aiming to deliver high-quality, consistent web experiences.

4.3 Regression Testing

Selenium is invaluable for regression testing, a crucial phase in the software development lifecycle where the goal is to ensure that recent code changes do not adversely affect existing functionality. Regression testing involves re-running previous test cases on new code to detect defects that may have been inadvertently introduced. By automating these repetitive test cases, Selenium allows testers to efficiently re-run tests across various environments with minimal manual effort. This automation capability is especially beneficial for regression testing, as it guarantees comprehensive and consistent test coverage with every code update. This ensures that new features or bug fixes do not introduce regressions, maintaining the integrity and performance of the software over time.

Moreover, Selenium's ability to execute tests across different browsers and platforms is a significant advantage in the current software landscape, where applications are accessed through diverse environments. Cross-browser and cross-platform testing help in the early detection of potential issues, ensuring that new changes do not introduce bugs or degrade performance on any supported environment. This capability saves significant time and effort compared to manual testing, enabling faster identification and resolution of defects. Selenium’s robust support for parallel test execution further enhances efficiency, allowing multiple tests to run simultaneously across various configurations. This not only accelerates the testing process but also ensures that any inconsistencies or issues are promptly addressed.

By automating regression tests, Selenium enhances the reliability and stability of the application, making it an essential tool for maintaining high-quality software. Automation ensures that repetitive tests are executed consistently without human error, providing reliable results. This consistency is critical for maintaining user trust and ensuring that the user experience remains unaffected by ongoing development. Additionally, Selenium's detailed reporting and logging capabilities aid in diagnosing issues quickly, facilitating continuous integration and continuous deployment (CI/CD) processes. Overall, Selenium's comprehensive test automation capabilities make it indispensable for regression testing, contributing significantly to the delivery of robust and reliable software.

4.4 Parallel Testing

Selenium Grid, a vital component of the Selenium suite, revolutionizes test automation by facilitating parallel testing. By enabling the simultaneous execution of multiple test cases across a variety of machines and browsers, Selenium Grid significantly reduces test execution time. Instead of running tests sequentially, which can be time-consuming, it distributes the test workload across different environments, including various operating systems, browser versions, and both physical and virtual machines. This distribution not only speeds up the testing process but also ensures comprehensive coverage, making it easier to identify and address environment-specific issues.

The parallel testing capability of Selenium Grid enhances the scalability and robustness of the test suite, making it an essential tool for large-scale and continuous integration testing environments. By efficiently managing multiple tests running in parallel, Selenium Grid supports faster feedback cycles and more frequent releases, crucial for agile development practices. This ability to run extensive test suites quickly and effectively ensures that the software is thoroughly vetted across different configurations, leading to more reliable and high-quality applications.

4.5 Integration with Testing Frameworks

Selenium's integration with various testing frameworks such as JUnit, TestNG, and NUnit significantly enhances its functionality by providing robust test management capabilities. These frameworks enable testers to efficiently manage test cases, generate detailed test reports, and handle dependencies. With features like parameterization, testers can run the same test with different sets of input data, thereby increasing test coverage and flexibility.

Grouping test cases into logical collections allows for more organized test execution, making it easier to run related tests together and streamline test management. Additionally, these frameworks support the setup of pre-conditions and post-conditions for tests, ensuring that the necessary environment is prepared before test execution and cleaned up afterward. This integration simplifies the test development process and improves the maintainability and scalability of the test suite, enhancing Selenium's effectiveness as a tool for automated testing.

4.6 Continuous Integration (CI) and Continuous Deployment (CD)

Selenium plays a crucial role in Continuous Integration and Continuous Deployment (CI/CD) pipelines by automating the testing phase, ensuring that every new code change is thoroughly evaluated before being merged into the main codebase. When integrated with CI/CD tools like Jenkins, Travis CI, or GitLab CI, Selenium tests can be triggered automatically with every code change. This automated testing process ensures that new features or bug fixes do not introduce unforeseen issues, thus maintaining the application's stability and reliability.

Incorporating Selenium tests into the CI/CD pipeline enables development teams to quickly detect and resolve defects, significantly reducing the need for manual testing. This integration accelerates the release cycle, allowing for faster and more frequent software updates. By streamlining the testing process, Selenium enhances the overall efficiency of the development workflow, ensuring that high-quality software is delivered consistently with each iteration. This seamless integration supports agile development practices, fostering a more reliable and responsive software development environment.

4.7 UI Testing

Selenium is primarily utilized for UI testing, allowing testers to validate that the user interface of a web application functions correctly according to specifications. It ensures that all elements are displayed properly, forms can be submitted without issues, and user interactions produce the expected outcomes. By automating these tests, Selenium helps detect and resolve UI inconsistencies and functional errors early in the development process. Its ability to simulate user actions such as clicking buttons, entering text, and navigating through pages makes it an invaluable tool for comprehensive UI testing.

Overall, Selenium is a versatile and powerful tool that significantly enhances the efficiency and effectiveness of web application testing. By automating repetitive and complex test scenarios, it not only saves time and effort but also contributes to improved software quality. This leads to faster release cycles and more reliable applications, as developers can confidently make changes and enhancements knowing that the user interface will continue to perform as expected across different browsers and environments.

*Chapter **5***

TRAIN TICKET BOOKING AUTOMATION

Selenium is a powerful tool used for automating web applications for testing purposes. It can also be leveraged for tasks such as booking train tickets by simulating user actions on a web page. Here's a general outline of how Selenium can be used for booking train tickets:

Steps to Automate Train Ticket Booking Using Selenium

1. Set Up Selenium Environment:
    - Install Selenium WebDriver for your preferred programming language (e.g., Python).
    - Install a browser driver (e.g., ChromeDriver for Google Chrome).

```
pip install selenium
```

2. Import Necessary Libraries:
    - Import Selenium libraries and any other required libraries (e.g., time for delays).

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import time
```

3. Initialize WebDriver:
    - Initialize the WebDriver and open the train ticket booking website.

```
driver = webdriver.Chrome()
driver.get('https://www.irctc.co.in/nget/train-search')  # Example for IRCTC
```

4. Log In:

- Locate the login elements, input username and password, and submit the login form.

```
username = driver.find_element(By.ID, 'userId')
password = driver.find_element(By.ID, 'pwd')
username.send_keys('your_username')
password.send_keys('your_password')
driver.find_element(By.ID, 'loginSubmit').click()
```

5. Search for Trains:

- Input journey details such as source, destination, date, and class.

```
source = driver.find_element(By.ID, 'origin')
destination = driver.find_element(By.ID, 'destination')
journey_date = driver.find_element(By.ID, 'journeyDate')

source.send_keys('Source Station')
destination.send_keys('Destination Station')
journey_date.send_keys('DD-MM-YYYY')
driver.find_element(By.ID, 'searchTrains').click()
```

6. Select Train and Book Tickets:

- Choose the desired train and class, then proceed with booking.

```
WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.XPATH, "//button[contains(text(), 'Check availability & fare')]")))
driver.find_element(By.XPATH, "//button[contains(text(), 'Check availability & fare')]").click()

WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.XPATH, "//button[contains(text(), 'Book Now')]")))
driver.find_element(By.XPATH, "//button[contains(text(), 'Book Now')]").click()
```

7. Enter Passenger Details:

- Fill in passenger details, contact information, and proceed to payment.

```
WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.ID, 'passengerName')))
passenger_name = driver.find_element(By.ID, 'passengerName')
passenger_age = driver.find_element(By.ID, 'passengerAge')
passenger_gender = driver.find_element(By.ID, 'passengerGender')

passenger_name.send_keys('John Doe')
passenger_age.send_keys('30')
passenger_gender.send_keys('Male')

driver.find_element(By.ID, 'contactInfo').send_keys('your_contact_number')
driver.find_element(By.ID, 'continueBooking').click()
```

8. Complete Payment:

- Select a payment method and complete the payment process.

```
WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.XPATH, "//button[contains(text(), 'Make Payment')]")))
driver.find_element(By.XPATH, "//button[contains(text(), 'Make Payment')]").click()

# Follow the steps to complete the payment process, which might vary depending on the payment method.
```

Important Considerations

I. Dynamic Content: Handle dynamic content and elements that load asynchronously using WebDriverWait.

II. Captcha and 2FA: Many train booking websites use captchas and two-factor authentication (2FA) to prevent automated bookings. These can be difficult to bypass and might require manual intervention.

III. Legal and Ethical Considerations: Ensure that automating such processes complies with the website’s terms of service and relevant laws and regulations.

Example Code

Here is a more complete example: -

```python
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import time

# Initialize the Chrome driver
driver = webdriver.Chrome()

# Open the train ticket booking website
driver.get('https://www.irctc.co.in/nget/train-search')

# Log in
username = driver.find_element(By.ID, 'userId')
password = driver.find_element(By.ID, 'pwd')
username.send_keys('your_username')
password.send_keys('your_password')
driver.find_element(By.ID, 'loginSubmit').click()

# Search for trains
source = driver.find_element(By.ID, 'origin')
destination = driver.find_element(By.ID, 'destination')
journey_date = driver.find_element(By.ID, 'journeyDate')

source.send_keys('Source Station')
destination.send_keys('Destination Station')
journey_date.send_keys('DD-MM-YYYY')
driver.find_element(By.ID, 'searchTrains').click()

# Select train and class
WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.XPATH, "//button[contains(text(), 'Check availability & fare')]")))
driver.find_element(By.XPATH, "//button[contains(text(), 'Check availability & fare')]").click()

WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.XPATH, "//button[contains(text(), 'Book Now')]")))
driver.find_element(By.XPATH, "//button[contains(text(), 'Book Now')]").click()

# Enter passenger details
WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.ID, 'passengerName')))
passenger_name = driver.find_element(By.ID, 'passengerName')
passenger_age = driver.find_element(By.ID, 'passengerAge')
passenger_gender = driver.find_element(By.ID, 'passengerGender')

passenger_name.send_keys('John Doe')
passenger_age.send_keys('30')
passenger_gender.send_keys('Male')

driver.find_element(By.ID, 'contactInfo').send_keys('your_contact_number')
driver.find_element(By.ID, 'continueBooking').click()

# Complete payment
WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.XPATH, "//button[contains(text(), 'Make Payment')]")))
driver.find_element(By.XPATH, "//button[contains(text(), 'Make Payment')]").click()

# Add additional payment steps as required

# Close the browser after booking
driver.quit()
```

Chapter ***6***

LIMITATIONS OF SELENIUM

While Selenium is a powerful and widely used tool for automated web testing, it does have some limitations:

6.1 Limited Support for Non-Web Applications

Selenium is primarily designed for automating the testing of web applications, making it an excellent choice for validating the functionality, performance, and user interface of websites and web-based applications across different browsers and platforms. However, Selenium does not offer robust support for testing non-web applications, such as desktop or mobile apps. This limitation is due to its architecture, which is specifically built to interact with web elements through browser drivers.

For testing desktop applications, other specialized tools and frameworks are required, such as Appium for mobile apps and tools like WinAppDriver, TestComplete, or AutoIt for desktop applications. Appium, for example, extends Selenium's WebDriver protocol to support mobile testing on iOS and Android platforms, providing a similar automation experience for mobile apps. Meanwhile, tools like TestComplete and WinAppDriver are designed to handle the unique challenges of desktop application testing, such as interacting with various UI elements, handling system dialogs, and performing actions at the operating system level.

6.2 No Built-in Reporting

Selenium itself does not offer built-in reporting capabilities, limiting its ability to produce detailed test reports and analytics. While Selenium can generate basic logs that track the execution flow and any encountered errors, these logs are often insufficient for comprehensive reporting needs. To overcome this limitation, testers typically integrate Selenium with additional libraries or frameworks that provide robust reporting functionalities.

Frameworks like TestNG and JUnit, which are commonly used for managing test cases in Selenium, include features for generating detailed test reports. These reports can include information on test execution status, time taken, and any failures or exceptions that occurred. Furthermore, tools like ExtentReports offer even more advanced reporting capabilities, enabling the creation of visually appealing and informative reports with features like step-wise logging, screenshots, and customizable templates.

By leveraging these additional tools, testers can gain deeper insights into test execution results, easily identify issues, and communicate findings effectively to stakeholders. This integration not only enhances the overall testing process but also ensures that all aspects of test execution and outcomes are thoroughly documented and analysed.

6.3 Handling Dynamic Elements

Selenium can face challenges when dealing with dynamically changing elements on a web page. Modern web applications often feature dynamic content that updates without a full page reload, such as elements appearing or disappearing, changing positions, or modifying attributes in response to user actions or data changes. These frequent updates can cause Selenium test scripts to break, as they may rely on specific identifiers or element locations that change unexpectedly.

As a result, test scripts often need to be updated frequently to accommodate changes in the application's structure or behaviour. This can lead to significant maintenance overhead, as testers must continually adjust their scripts to ensure they accurately interact with the application's current state. Additionally, dynamically generated elements may require more sophisticated strategies to locate and interact with, such as using explicit waits to handle timing issues or leveraging more flexible locators like CSS selectors and XPath expressions.

To mitigate these issues, testers can adopt best practices such as using robust locators, implementing page object models, and incorporating wait mechanisms to handle dynamic content. Despite these strategies, the inherently dynamic nature of modern web applications means that some level of ongoing maintenance will always be necessary to keep Selenium test scripts up to date and reliable.

6.4 Complexity for Beginners

Using Selenium for automated web testing requires proficiency in programming languages such as Java, Python, C#, and others. Testers need to write and maintain test scripts in these languages, which can be challenging for beginners or those without a strong programming background. In addition to programming skills, knowledge of HTML and CSS is essential for understanding the structure of web pages and identifying elements to interact with.

Testers must also be familiar with XPath or CSS selectors, which are used to locate web elements within the HTML document. Crafting accurate and efficient selectors can be complex, especially when dealing with dynamic or nested elements. The need to combine these technical skills—programming, web technologies, and selector strategies—can make Selenium challenging to learn and use effectively for those new to test automation or lacking a solid foundation in coding. This learning curve often necessitates additional training and practice to become proficient in creating reliable and maintainable Selenium test scripts.

6.5 Browser Compatibility

In cross-browser testing with Selenium, ensuring consistency across different browsers involves navigating through nuances in browser behaviors and addressing potential compatibility issues. Browsers can interpret and render web elements differently, leading to discrepancies in how Selenium interacts with them. Test scripts may need fine-tuning or modifications to accommodate these discrepancies, ensuring that tests produce consistent outcomes across all target browsers.

Compatibility issues may also arise due to differences in browser versions or underlying technologies. For instance, a feature that works seamlessly in one browser version may behave differently or even fail in another. Selenium users may need to employ version-specific workarounds or utilize Selenium's built-in capabilities to handle browser version discrepancies.

Furthermore, the ever-evolving nature of browsers and web technologies means that Selenium users must stay vigilant and adaptable. New browser updates may introduce changes that affect test behaviour, necessitating regular review and maintenance of test scripts to ensure continued compatibility.

Addressing these challenges requires a combination of technical expertise, thorough testing strategies, and proactive monitoring of browser updates and changes. Despite the complexities involved, Selenium's cross-browser testing capabilities remain invaluable for validating the consistency and reliability of web applications across diverse browser environments.

6.6 Speed of Execution

Selenium test scripts, while highly beneficial for automating repetitive tasks and ensuring consistent test execution, can occasionally exhibit slower performance compared to manual testing. This delay is particularly noticeable in scenarios involving complex test cases or extensive test suites. Factors such as the initialization of browser instances, navigation through web pages, and sequential execution of test steps can contribute to increased execution times. Additionally, running tests in parallel using Selenium Grid, while effective for reducing overall test execution time, may require considerable setup and infrastructure resources. Configuring and maintaining a Selenium Grid infrastructure involves provisioning multiple machines or virtual instances, installing and configuring browser drivers, and managing test distribution across nodes. Ensuring the scalability, reliability, and synchronization of test execution across parallel environments demands careful planning and ongoing management.

Despite these challenges, the advantages of Selenium automation, such as increased test coverage, repeatability, and reliability, often outweigh the drawbacks of slower execution times. Furthermore, optimization strategies such as implementing efficient test design patterns, using intelligent wait mechanisms, and parallelizing test execution can help mitigate performance issues and maximize the efficiency of Selenium test automation. Overall, while Selenium may require initial investment in setup and optimization, its ability to streamline testing processes and improve software quality makes it an indispensable tool in modern software development workflows.

6.7 Dependency on Browser Automation

Selenium's dependency on browser automation exposes it to the risk of encountering compatibility issues due to changes in browser versions or updates. Since Selenium WebDriver interacts directly with web browsers to execute test scripts, any modifications or updates to the browser's underlying technologies can potentially impact Selenium's functionality. These changes may manifest as unexpected behaviours, broken test scripts, or failure to interact with web elements as intended. To mitigate these challenges, Selenium users must remain vigilant about browser updates and regularly update their WebDriver bindings to align with the latest browser versions. Additionally, maintaining a proactive approach to test script maintenance and thorough regression testing after browser updates can help identify and address compatibility issues promptly, ensuring the continued reliability and effectiveness of Selenium-based test automation.

6.8 No Support for Captcha and Barcode Recognition

Selenium's dependency on browser automation exposes it to the risk of encountering compatibility issues due to changes in browser versions or updates. Since Selenium WebDriver interacts directly with web browsers to execute test scripts, modifications to the browser's underlying technologies can lead to unexpected behaviours, broken test scripts, or failure to interact with web elements as intended.

To mitigate these challenges, Selenium users must stay vigilant about browser updates and regularly update their WebDriver bindings to align with the latest browser versions. Proactive test script maintenance and thorough regression testing after browser updates are crucial to identify and address compatibility issues promptly. Leveraging automated testing tools and continuous integration (CI) systems can provide timely feedback and highlight areas needing attention.

Engaging with the Selenium community provides valuable insights and solutions to emerging compatibility challenges. Communities and forums discuss recent issues and share patches or workarounds, allowing users to benefit from collective knowledge and experience. By staying connected and contributing to these discussions, Selenium users can navigate the complexities of browser automation more effectively, ensuring the resilience and reliability of their test automation efforts.

Chapter 7

REQUIREMENT & ANALYSIS

7.1 Program Reliability

The given code exemplifies automated testing for an e-commerce platform using Selenium WebDriver and TestNG. The program is designed to navigate through the website, perform various actions, and validate the functionalities of an online purchase process. This includes crucial steps such as searching for products, selecting items, adding them to the cart, and completing the checkout process. By automating these tasks, the program ensures that the critical paths in the user journey are tested consistently and thoroughly. This not only helps in identifying bugs early in the development cycle but also ensures that the core functionalities of the e-commerce platform work seamlessly.

For program reliability, it is crucial to ensure that these critical paths are tested under various conditions and scenarios to mimic real user interactions. However, the current implementation's reliability can be compromised by the extensive use of Thread.sleep() for synchronization. Thread.sleep() introduces arbitrary pauses in the test execution, which can lead to either unnecessary delays or insufficient wait times, causing flaky tests that pass or fail unpredictably based on timing issues. This approach is not only inefficient but also fails to handle dynamic web elements that may load at different speeds under different conditions.

To enhance the program's reliability, it is advisable to replace Thread.sleep() with more robust synchronization mechanisms provided by WebDriver, such as WebDriver Wait. WebDriver Wait allows the test to wait for specific conditions to be met before proceeding, such as the presence of an element, its visibility, or its ability to be clicked. This dynamic waiting mechanism ensures that the test execution is synchronized with the actual behavior of the web application, leading to more stable and reliable tests. Implementing WebDriver Wait can significantly reduce test flakiness, improve execution speed, and enhance the overall robustness of the automated testing suite, thereby ensuring a more dependable validation of the e-commerce platform's functionalities.

7.2 Problem of Accuracy

The accuracy of the test scripts depends on the proper identification and interaction with web elements. Using hard-coded Thread.sleep() may lead to inaccurate results due to timing issues, such as elements not being fully loaded. Instead, explicit, or implicit waits should be used to ensure elements are interactable. The accuracy of the test cases can be further enhanced by validating the expected outcomes at each step, such as checking if the correct page is loaded or if the cart contains the correct items.

7.3 Requirements & Specifications

7.3.1 Functional Requirements

I. **Browser Automation**: The code should launch a Chrome browser, navigate to the specified URL, and perform automated tests.

II. **UI Interactions**: Perform actions such as clicking links, selecting options from dropdowns, and filling out forms.

III. **Order Processing**: Automate the process of adding a product to the cart, proceeding to checkout, and completing the order.

IV. **Test Case Execution**: Run multiple test cases to cover different functionalities like navigation, form submission, and finalizing orders.

V. **Test Initialization and Cleanup**: Initialize the browser before tests and close it after all tests are executed.

7.4 Constraints

I. **Browser Dependency**: The tests are dependent on ChromeDriver and Chrome browser, limiting the compatibility with other browsers unless adapted.

II. **Hardcoded Values**: The script contains hardcoded paths, data, and sleeps, which may need adjustments for different environments or test data.

III. **Thread.sleep Usage**: Relies on Thread.sleep(), which is not a reliable synchronization method and can lead to flaky tests.

7.5 Assumptions and Dependencies

I. **Website Stability**: The e-commerce website (https://tutorialsninja.com/demo/index.php) remains stable and its structure does not change frequently.

II. **Browser and Driver Compatibility**: Assumes that the version of ChromeDriver used is compatible with the installed version of Chrome.

III. **Network Reliability**: Assumes a stable internet connection for accessing the website.

7.6 Input/Output Specifications

7.6.1 Input Specifications

I. **URL**: https://tutorialsninja.com/demo/index.php

II. **Test Data**: User details such as name, email, telephone, address, etc.

III. **Browser Driver Path**: Path to ChromeDriver executable

7.6.2 Output Specifications

I. **Browser Actions**: Navigation and interactions on the website

II. **Console Logs**: Outputs from TestNG regarding the execution status of each test case

III. **Visual Verification**: User can visually verify the steps performed by the automated tests.

7.6.3 Error Handling

I. **Element Not Found**: Use try-catch blocks to handle scenarios where elements are not found.

II. **Timeouts**: Implement WebDriverWait to handle elements that take time to appear or become clickable.

III. **Invalid Inputs**: Validate input data before using it in form submissions to avoid errors during test execution.

IV. **Browser Failures**: Capture screenshots and logs for analysis in case of test failures.

7.7 SOFTWARE & HARDWARE REQUIREMENTS

7.7.1 Hardware

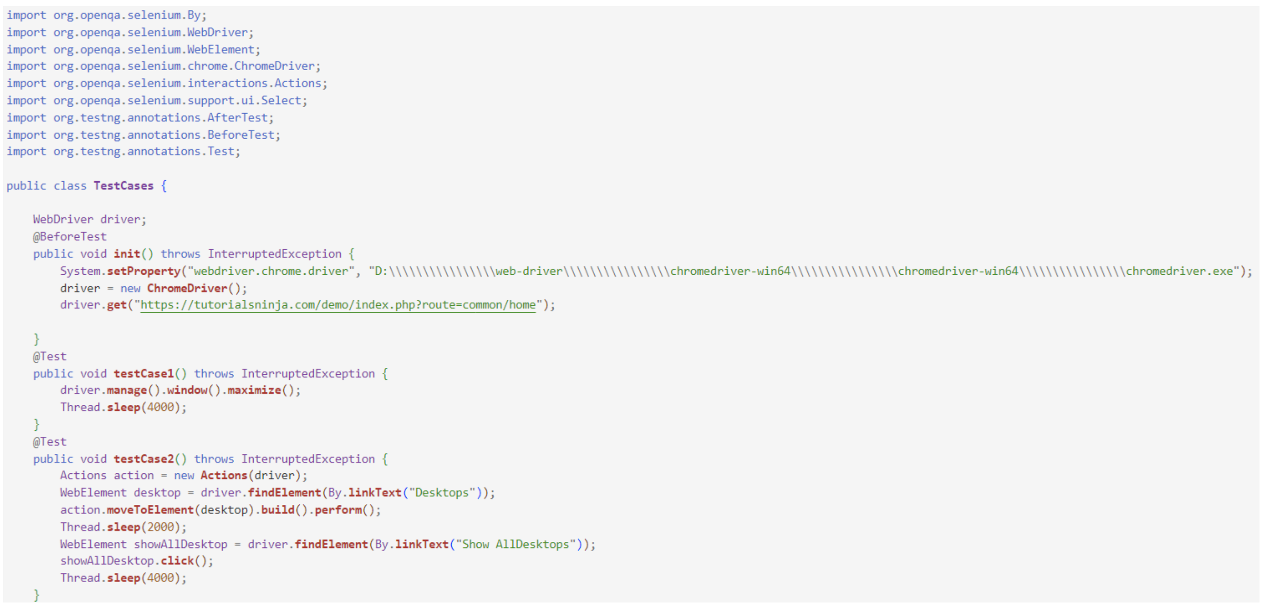
Processor	Inter core 2 Duo or above
Memory	2GB RAM or above
Cache Memory	128 KB or above
Hard Disk	30 GB or above [at least 10 MB free space required]

7.7.2 Software

Operating System	Windows, Mac OS, Linux
Coding Environment	Eclipse
Google Chrome	97 or above

Chapter 8

<u>CODE</u>

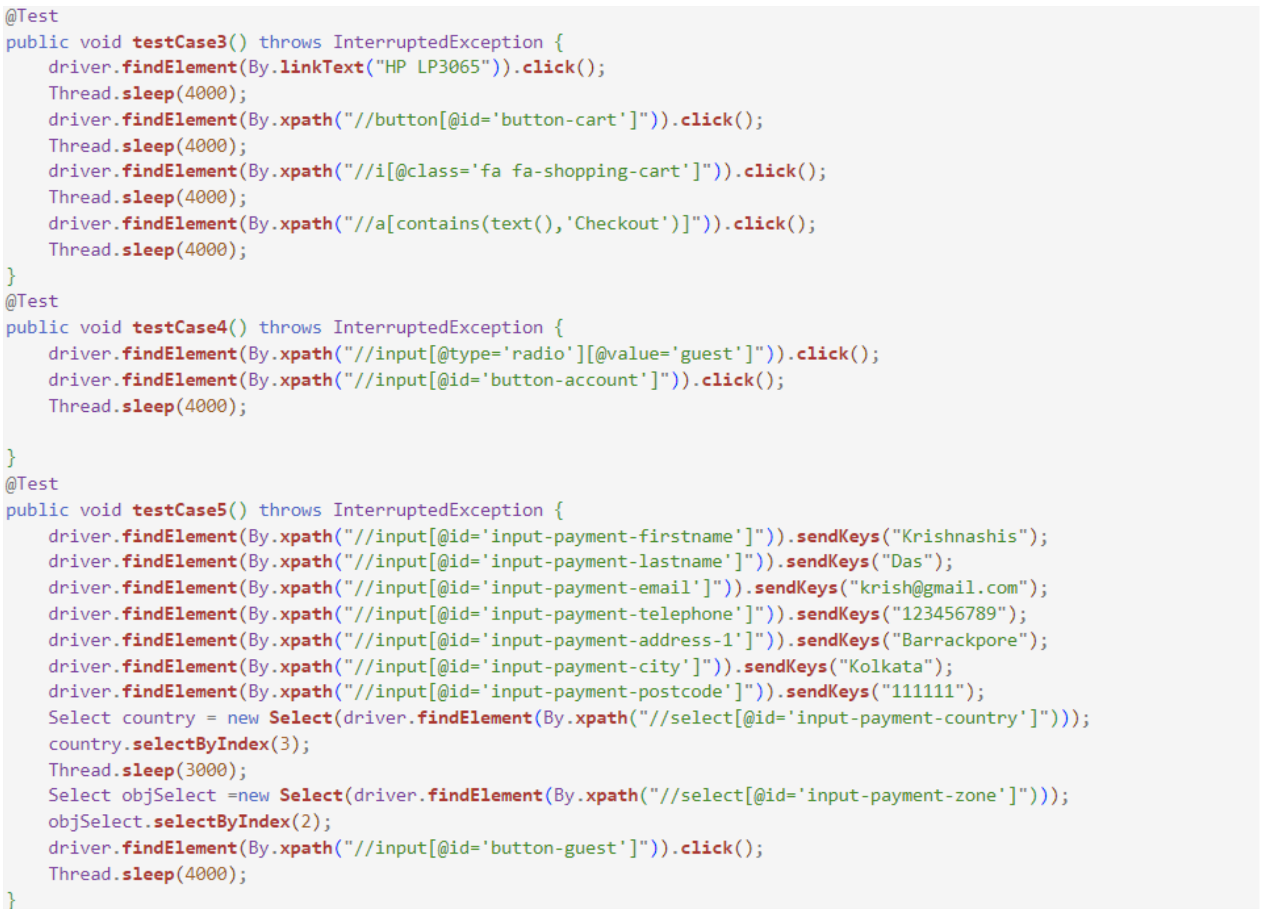


```java
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.interactions.Actions;
import org.openqa.selenium.support.ui.Select;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;

public class TestCases {

    WebDriver driver;
    @BeforeTest
    public void init() throws InterruptedException {
        System.setProperty("webdriver.chrome.driver", "D:\\\\\\\\\\\\\\\\web-driver\\\\\\\\\\\\\\\\chromedriver-win64\\\\\\\\\\\\\\\\chromedriver-win64\\\\\\\\\\\\\\\\chromedriver.exe");
        driver = new ChromeDriver();
        driver.get("https://tutorialsninja.com/demo/index.php?route=common/home");

    }
    @Test
    public void testCase1() throws InterruptedException {
        driver.manage().window().maximize();
        Thread.sleep(4000);
    }
    @Test
    public void testCase2() throws InterruptedException {
        Actions action = new Actions(driver);
        WebElement desktop = driver.findElement(By.linkText("Desktops"));
        action.moveToElement(desktop).build().perform();
        Thread.sleep(2000);
        WebElement showAllDesktop = driver.findElement(By.linkText("Show AllDesktops"));
        showAllDesktop.click();
        Thread.sleep(4000);
    }
```



```java
@Test
public void testCase3() throws InterruptedException {
    driver.findElement(By.linkText("HP LP3065")).click();
    Thread.sleep(4000);
    driver.findElement(By.xpath("//button[@id='button-cart']")).click();
    Thread.sleep(4000);
    driver.findElement(By.xpath("//i[@class='fa fa-shopping-cart']")).click();
    Thread.sleep(4000);
    driver.findElement(By.xpath("//a[contains(text(),'Checkout')]")).click();
    Thread.sleep(4000);
}
@Test
public void testCase4() throws InterruptedException {
    driver.findElement(By.xpath("//input[@type='radio'][@value='guest']")).click();
    driver.findElement(By.xpath("//input[@id='button-account']")).click();
    Thread.sleep(4000);

}
@Test
public void testCase5() throws InterruptedException {
    driver.findElement(By.xpath("//input[@id='input-payment-firstname']")).sendKeys("Krishnashis");
    driver.findElement(By.xpath("//input[@id='input-payment-lastname']")).sendKeys("Das");
    driver.findElement(By.xpath("//input[@id='input-payment-email']")).sendKeys("krish@gmail.com");
    driver.findElement(By.xpath("//input[@id='input-payment-telephone']")).sendKeys("123456789");
    driver.findElement(By.xpath("//input[@id='input-payment-address-1']")).sendKeys("Barrackpore");
    driver.findElement(By.xpath("//input[@id='input-payment-city']")).sendKeys("Kolkata");
    driver.findElement(By.xpath("//input[@id='input-payment-postcode']")).sendKeys("111111");
    Select country = new Select(driver.findElement(By.xpath("//select[@id='input-payment-country']")));
    country.selectByIndex(3);
    Thread.sleep(3000);
    Select objSelect =new Select(driver.findElement(By.xpath("//select[@id='input-payment-zone']")));
    objSelect.selectByIndex(2);
    driver.findElement(By.xpath("//input[@id='button-guest']")).click();
    Thread.sleep(4000);
}
```

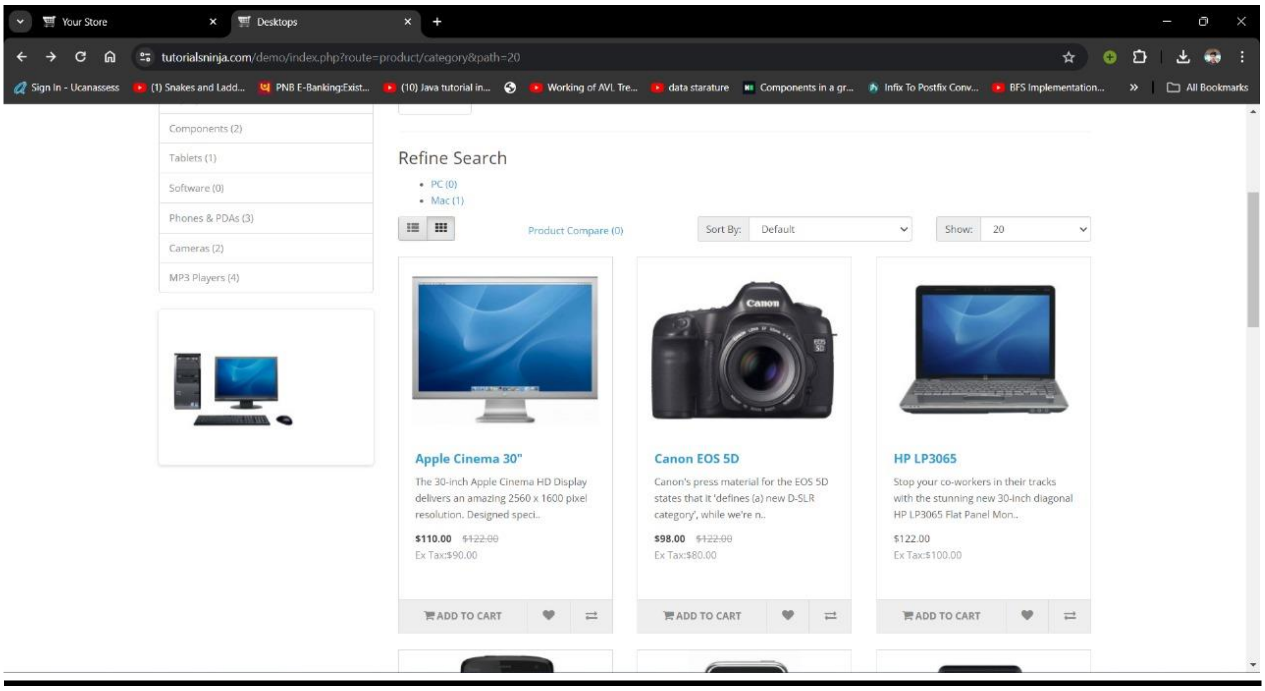
```
    @Test
    public void testCase6() throws InterruptedException {

        driver.findElement(By.xpath("//input[@id='button-shipping-method']")).click();
        Thread.sleep(4000);
    }
    @Test
    public void testCase7() throws InterruptedException {
        driver.findElement(By.xpath("//input[@name='agree']")).click();
        Thread.sleep(2000);
        driver.findElement(By.xpath("//input[@id='button-payment-method']")).click();
        Thread.sleep(4000);
    }
    @Test
    public void testCase8() throws InterruptedException {
        driver.findElement(By.xpath("//input[@id='button-confirm']")).click();
        Thread.sleep(4000);
    }
    @Test
    public void testCase9() {
        driver.findElement(By.xpath("//a[contains(text(),'Continue')]")).click();
    }
    @AfterTest
    public void destroy() {
        //driver.quit();
    }

}
```

Chapter 9

OUTPUT



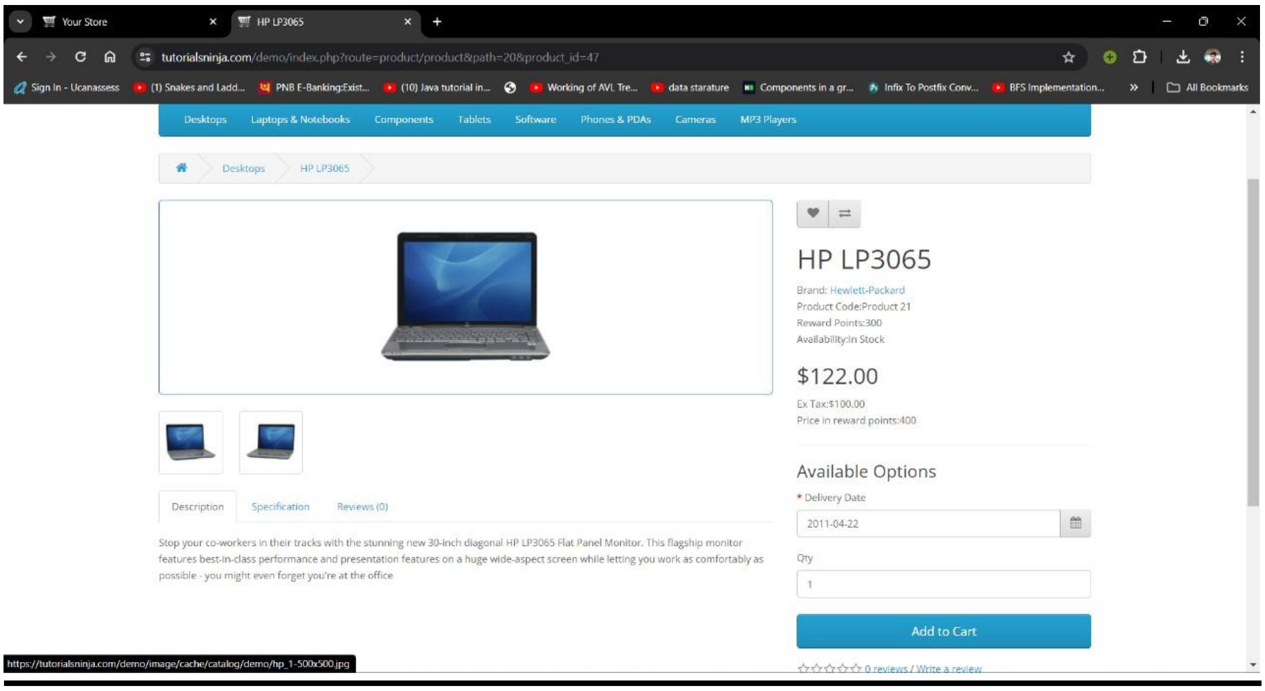
Refine Search

- PC (0)
- Mac (1)

Apple Cinema 30"
The 30-inch Apple Cinema HD Display delivers an amazing 2560 x 1600 pixel resolution. Designed speci..
$110.00 ~~$122.00~~
Ex Tax:$90.00

Canon EOS 5D
Canon's press material for the EOS 5D states that it 'defines (a) new D-SLR category', while we're n..
$98.00 ~~$122.00~~
Ex Tax:$80.00

HP LP3065
Stop your co-workers in their tracks with the stunning new 30-inch diagonal HP LP3065 Flat Panel Mon..
$122.00
Ex Tax:$100.00

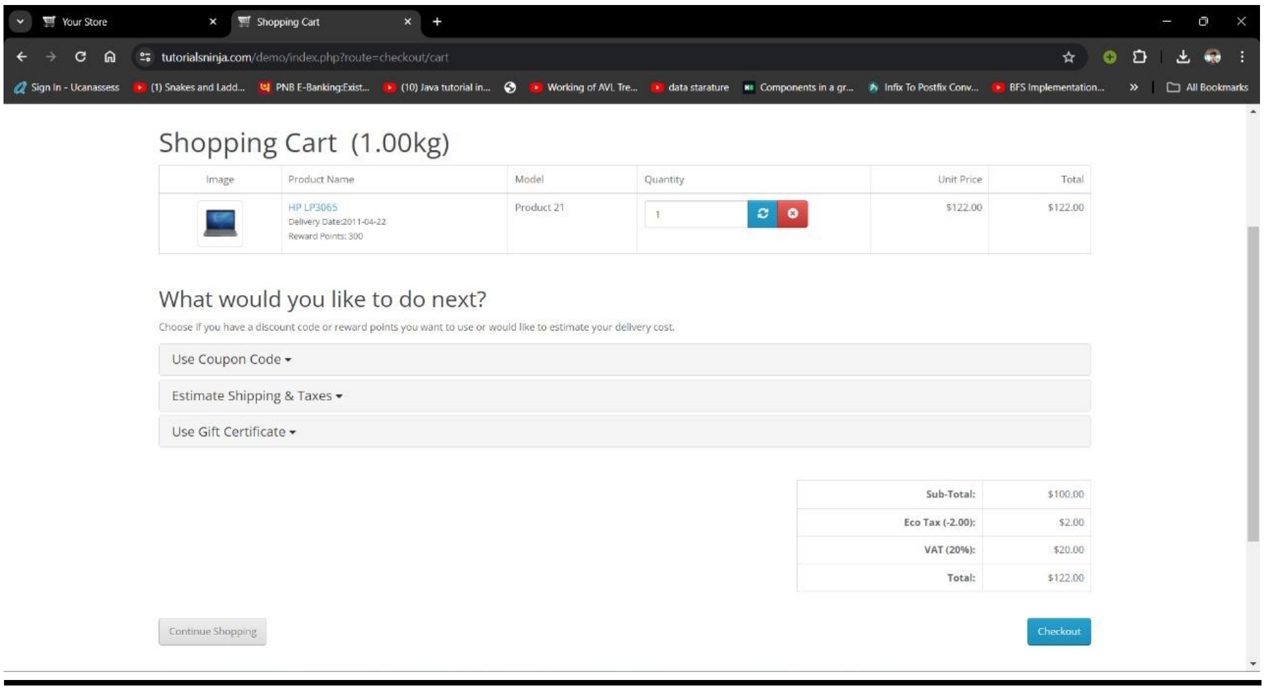

Your Store
HP LP3065
tutorialsninja.com/demo/index.php?route=product/product&path=20&product_id=47
Desktops
Laptops & Notebooks
Components
Tablets
Software
Phones & PDAs
Cameras
MP3 Players
Desktops
HP LP3065
Description
Specification
Reviews (0)
Add to Cart
0 reviews / Write a review
https://tutorialsninja.com/demo/image/cache/catalog/demo/hp_1-500x500.jpg

HP LP3065
Brand: Hewlett-Packard
Product Code:Product 21
Reward Points:300
Availability:In Stock

$122.00
Ex Tax:$100.00
Price in reward points:400

Available Options
* Delivery Date
2011-04-22
Qty
1

Stop your co-workers in their tracks with the stunning new 30-inch diagonal HP LP3065 Flat Panel Monitor. This flagship monitor features best-in-class performance and presentation features on a huge wide-aspect screen while letting you work as comfortably as possible - you might even forget you're at the office


tutorialsninja.com/demo/index.php?route=checkout/cart

Shopping Cart (1.00kg)

Image	Product Name	Model	Quantity	Unit Price	Total
	HP LP3065 Delivery Date:2011-04-22 Reward Points: 300	Product 21	1	$122.00	$122.00

What would you like to do next?

Choose if you have a discount code or reward points you want to use or would like to estimate your delivery cost.

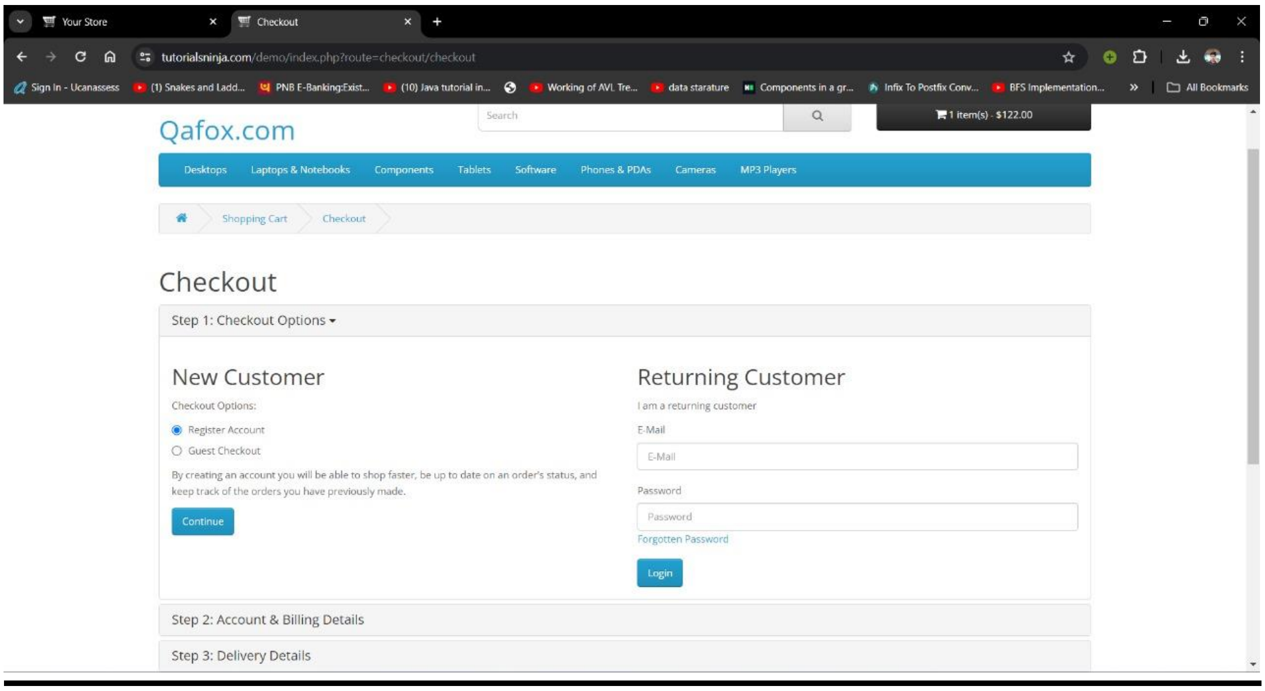
Use Coupon Code ▾

Estimate Shipping & Taxes ▾

Use Gift Certificate ▾

Sub-Total:	$100.00
Eco Tax (-2.00):	$2.00
VAT (20%):	$20.00
Total:	$122.00

Continue Shopping   Checkout


tutorialsninja.com/demo/index.php?route=checkout/checkout

Qafox.com   Search   1 item(s) - $122.00

Desktops Laptops & Notebooks Components Tablets Software Phones & PDAs Cameras MP3 Players

Shopping Cart > Checkout

Checkout

Step 1: Checkout Options ▾

New Customer

Checkout Options:

- (●) Register Account
- ( ) Guest Checkout

By creating an account you will be able to shop faster, be up to date on an order's status, and keep track of the orders you have previously made.

Continue

Returning Customer

I am a returning customer

E-Mail: E-Mail

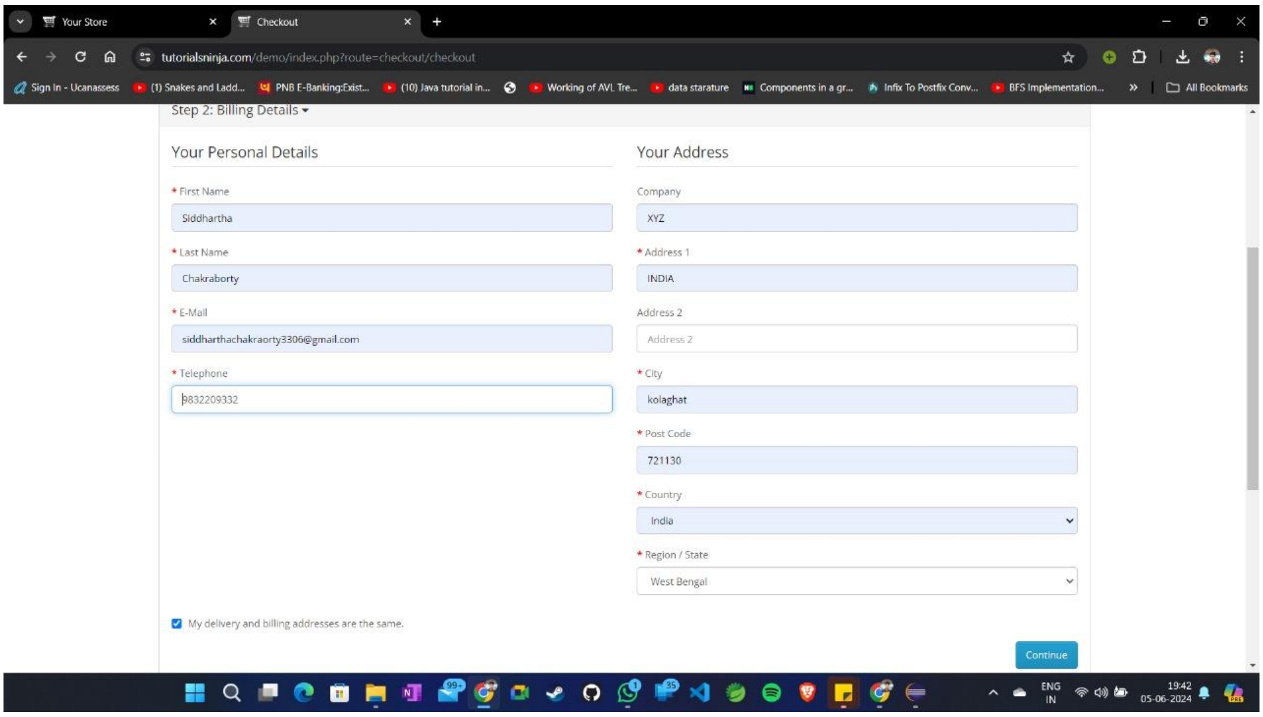
Password: Password

Forgotten Password

Login

Step 2: Account & Billing Details

Step 3: Delivery Details


tutorialsninja.com/demo/index.php?route=checkout/checkout

Step 2: Billing Details

Your Personal Details

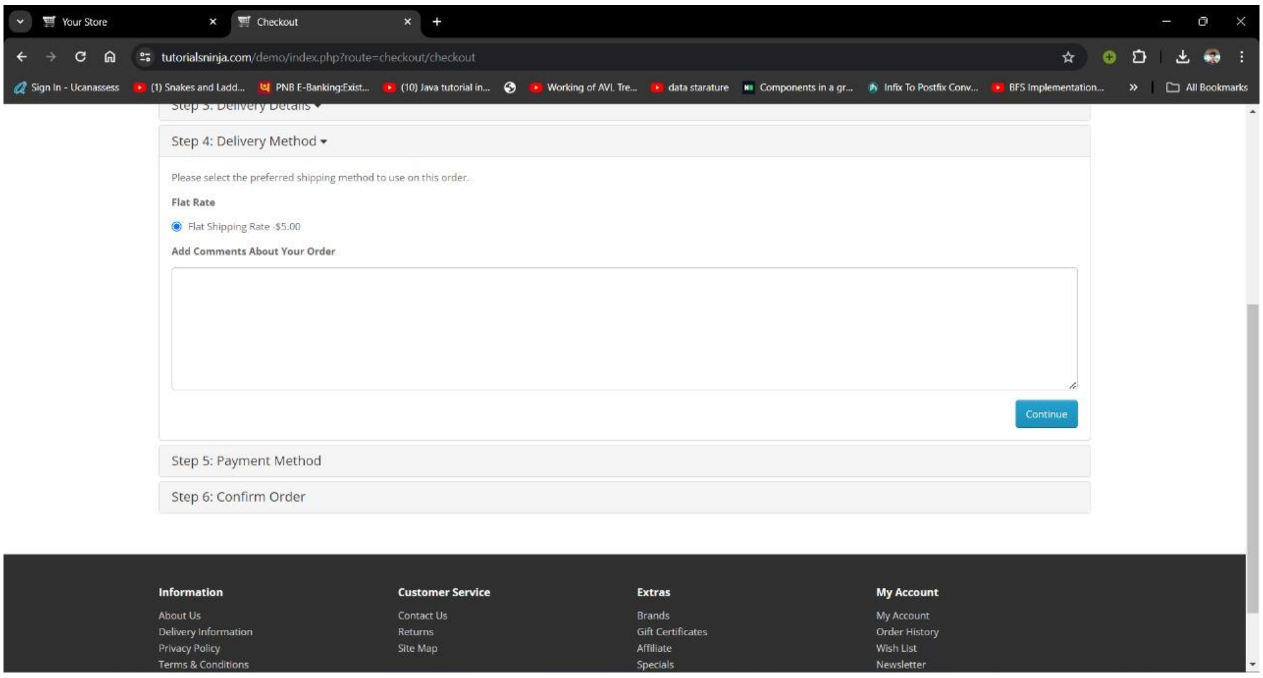
* First Name	Siddhartha
* Last Name	Chakraborty
* E-Mail	siddharthachakraorty3306@gmail.com
* Telephone	9832209332

Your Address

Company	XYZ
* Address 1	INDIA
Address 2	
* City	kolaghat
* Post Code	721130
* Country	India
* Region / State	West Bengal

[✓] My delivery and billing addresses are the same.

Continue


tutorialsninja.com/demo/index.php?route=checkout/checkout

Step 4: Delivery Method

Please select the preferred shipping method to use on this order.

Flat Rate

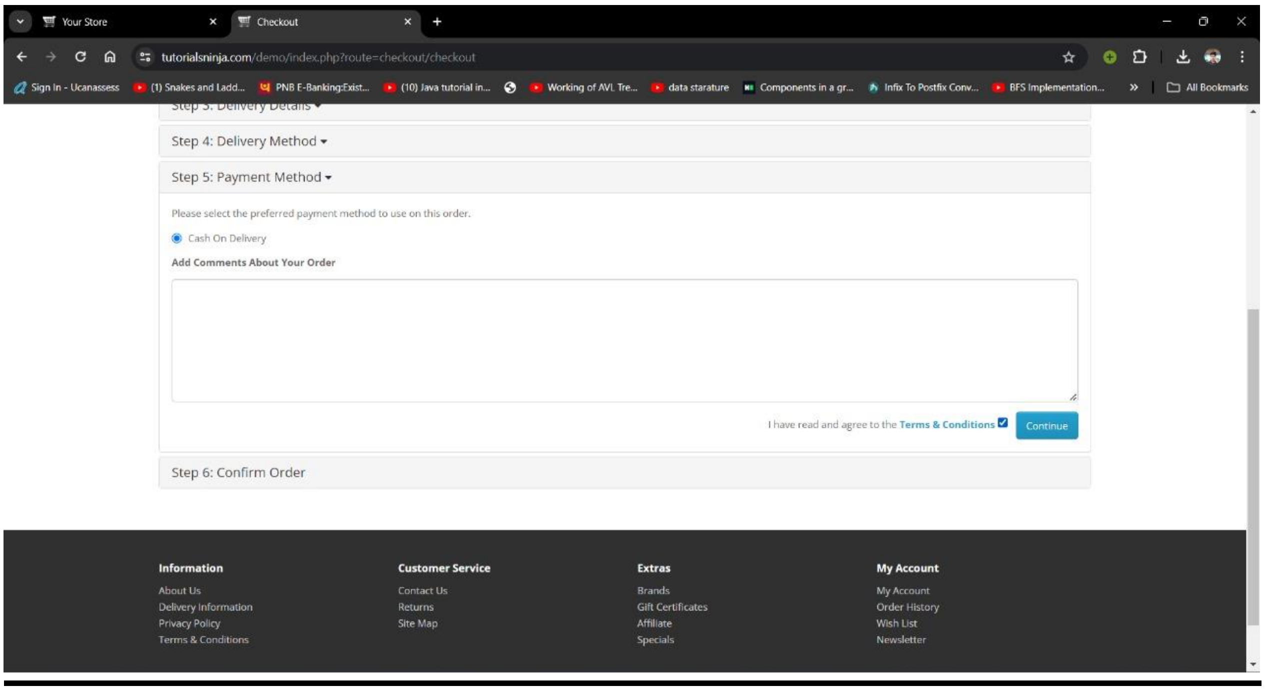
[●] Flat Shipping Rate - $5.00

Add Comments About Your Order

Continue

Step 5: Payment Method

Step 6: Confirm Order



Step 4: Delivery Method ▾

Step 5: Payment Method ▾

Please select the preferred payment method to use on this order.

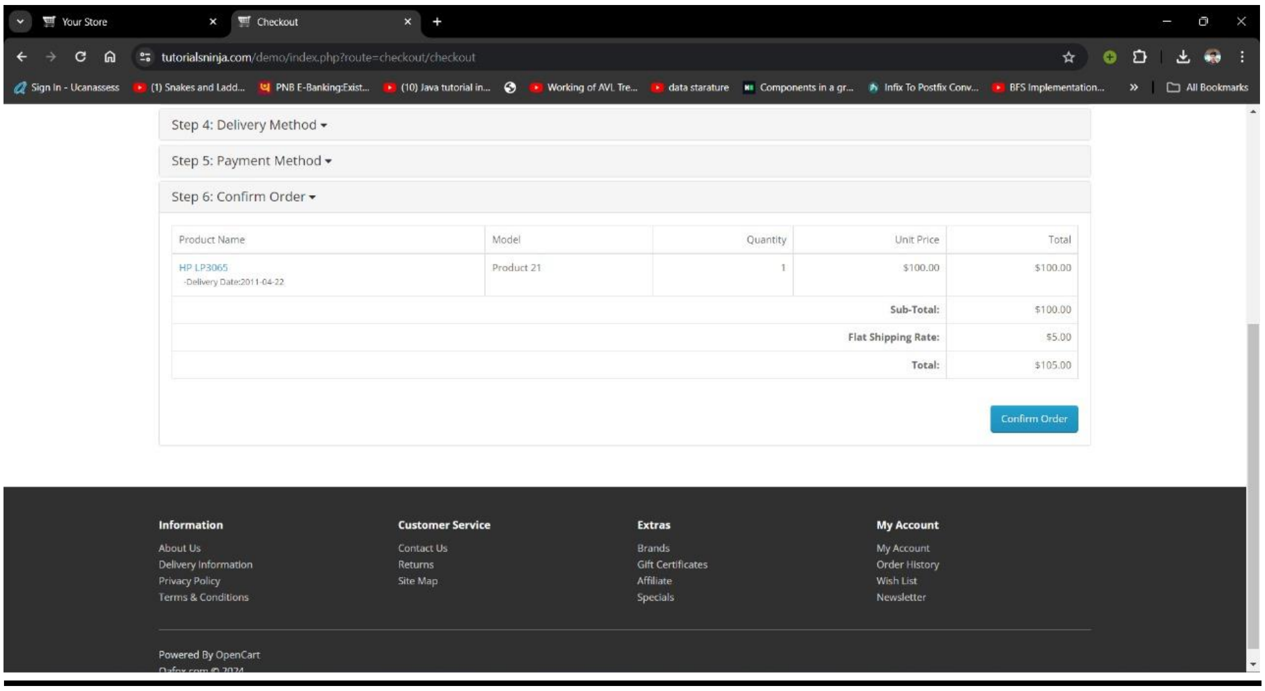
[✓ Cash On Delivery]

Add Comments About Your Order

I have read and agree to the **Terms & Conditions** ✓ Continue

Step 6: Confirm Order

Information	**Customer Service**	**Extras**	**My Account**
About Us	Contact Us	Brands	My Account
Delivery Information	Returns	Gift Certificates	Order History
Privacy Policy	Site Map	Affiliate	Wish List
Terms & Conditions		Specials	Newsletter



Step 4: Delivery Method ▾

Step 5: Payment Method ▾

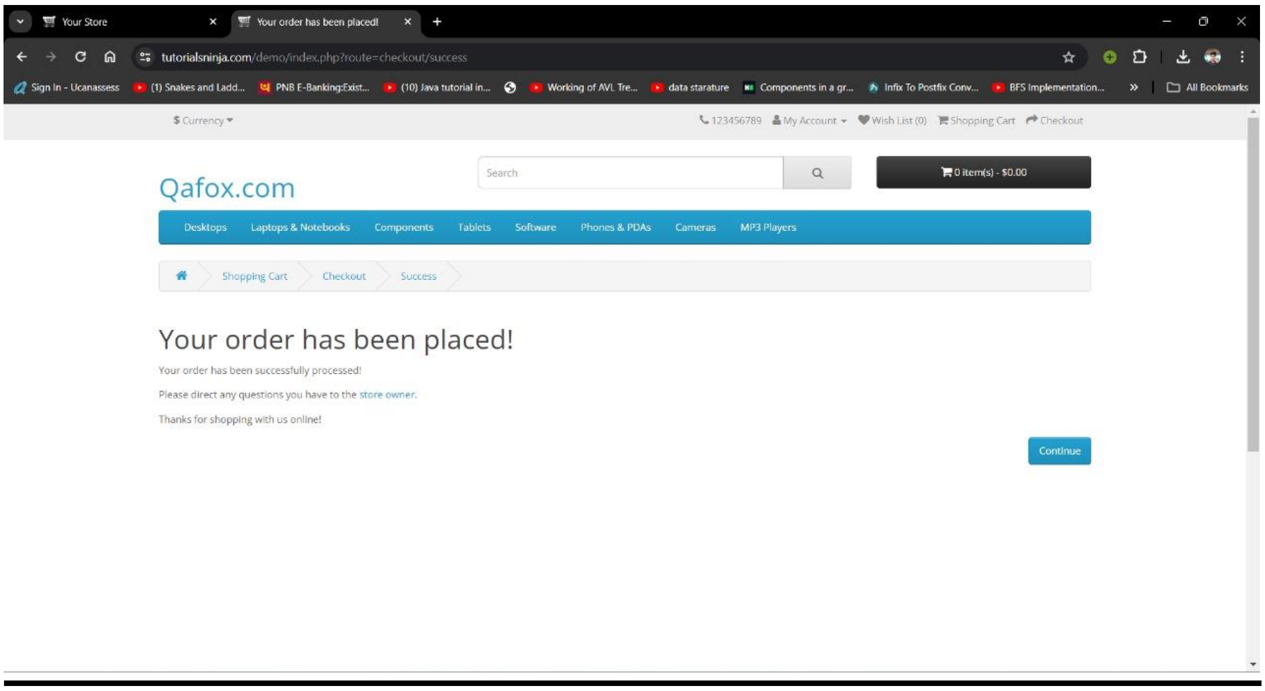
Step 6: Confirm Order ▾

Product Name	Model	Quantity	Unit Price	Total
HP LP3065 -Delivery Date:2011-04-22	Product 21	1	$100.00	$100.00
			Sub-Total:	$100.00
			Flat Shipping Rate:	$5.00
			Total:	$105.00

Confirm Order

Information	**Customer Service**	**Extras**	**My Account**
About Us	Contact Us	Brands	My Account
Delivery Information	Returns	Gift Certificates	Order History
Privacy Policy	Site Map	Affiliate	Wish List
Terms & Conditions		Specials	Newsletter

Powered By OpenCart



Your order has been placed!

Your order has been successfully processed!

Please direct any questions you have to the store owner.

Thanks for shopping with us online!

Chapter ***10***

CONCLUSION

This project exemplifies the power and flexibility of Selenium WebDriver combined with TestNG for automated testing. By automating a complete user journey on an e-commerce platform, the script ensures comprehensive coverage of critical functionalities. This thorough validation enhances the reliability and stability of the application, as it simulates real user interactions from browsing products to completing a purchase. The script's ability to handle complex user actions and form interactions showcases the robustness of Selenium WebDriver in providing precise and consistent test execution.

The integration of TestNG in this project adds significant value through efficient test management capabilities. TestNG allows for organized execution of test cases, parameterization, and detailed reporting, which simplifies the test development and maintenance process. The modular structure of the test cases facilitates easy scalability, enabling the addition of new tests and modification of existing ones without disrupting the overall framework. This structured approach ensures that the testing process remains organized and manageable, even as the test suite expands.

Ultimately, this project not only improves testing efficiency but also contributes to faster development cycles and higher quality software delivery. Automating repetitive tasks reduces manual effort, increases test coverage, and ensures consistency across different environments. By catching potential issues early in the development cycle, the automated tests help in maintaining a high standard of software quality. This leads to more reliable releases and a better user experience, showcasing the critical role of automated testing in modern software development practices.

Chapter ***11***

BIBLIOGRAPHY

1. www.selenium.dev/documentation/webdriver/
2. www.selenium.dev/documentation/en/support_packages/testng/
3. sites.google.com/a/chromium.org/chromedriver/
4. tutorialsninja.com/demo/index.php?route=common/home
5. www.selenium.dev/documentation/en/support_packages/mouse_and_keyboard_actions_in_detail/
6. www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/WebElement.html
7. www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/By.html
8. www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/support/ui/Select.html
9. testng.org/doc/documentation-main.html#annotations
10. testng.org/doc/documentation-main.html#test-suite
11. www.selenium.dev/documentation/en/getting_started_with_webdriver/locating_elements/
12. www.w3.org/TR/webdriver/
13. www.selenium.dev/documentation/en/webdriver/browser_manipulation/#handling-pop-ups-and-alerts
14. www.selenium.dev/documentation/en/webdriver/browser_manipulation/#handling-forms
15. www.selenium.dev/documentation/en/webdriver/waits/
16. https://maven.apache.org/guides/
17. https://www.selenium.dev/documentation/webdriver/
18. https://scholar.harvard.edu/files/tcheng2/files/selenium_documentation_0.pdf
19. https://www.browserstack.com/selenium
20. https://www.educba.com/uses-of-selenium/