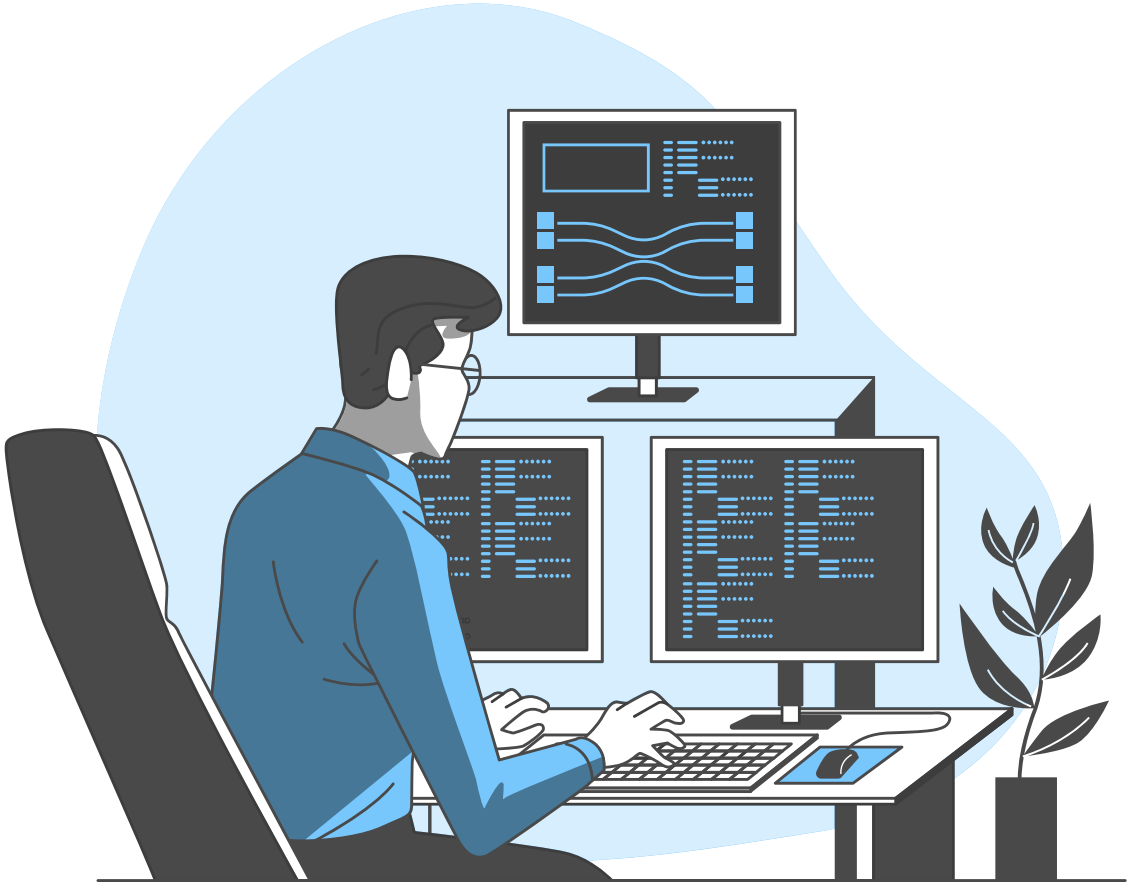
CDIP
Center for Development of IT Professionals



Professional Software Testing & Quality Assurance

Instructor
Parvez Hossain

Automation Testing

Automation software testing is a process of using automated tools and scripts to execute tests on software applications or systems. It involves automating repetitive and time-consuming tasks that are part of the software testing process, such as running regression tests, performing load testing.

Key components of automation software testing include:

1.Test Automation Tools: Specialized software tools that are used to create, execute, and manage automated tests. These tools help in simulating user interactions, data inputs, and system responses.

2.Test Scripts: Sets of instructions or code written in programming languages like Java, Python, or Selenium WebDriver that define the actions to be performed during the automated test.

3.Test Data: Input values, configurations, and other data required to execute the automated tests. This data helps in testing different scenarios and conditions.

4.Automation Frameworks: A set of guidelines, rules, and best practices that provide a structured way to create and organize automated tests. Frameworks improve maintainability, reusability, and scalability of automation scripts.

Benefits of automation software testing include:

•**Efficiency**: Automated tests can be executed quickly and repeatedly, saving time compared to manual testing.

•**Consistency**: Automated tests can ensure that the same steps are followed precisely for each test execution, improving consistency and reducing the risk of human error.

•**Increased Test Coverage**: Automated tests can cover a broader range of test cases, enabling a more thorough examination of the application.

•**Early Detection of Defects**: Automated tests can be run early in the development cycle, allowing for early detection and resolution of defects, which can significantly reduce development costs.

•**Regression Testing**: Automated tests are particularly useful for regression testing, ensuring that new changes or updates to the software do not negatively impact existing functionalities.

Selenium Webdriver

Selenium WebDriver is a popular automation tool used for controlling web browsers and automating web application testing. It provides a programming interface to interact with web elements and perform actions like clicking buttons, filling forms, navigating between pages, and validating the behavior of a web application.

Here are some key points about Selenium WebDriver:

1. **Automation Tool**: Selenium WebDriver is primarily used for automating web applications for testing purposes, but it can also be used for various other tasks involving web interactions.
2. **Cross-browser and Cross-platform**: Selenium WebDriver allows you to write scripts that can run on various web browsers (e.g., Chrome, Firefox, Safari, Internet Explorer) and operating systems (e.g., Windows, macOS, Linux).
3. **Interacting with Web Elements**: Selenium WebDriver allows users to interact with various web elements such as buttons, text fields, dropdowns, checkboxes, and radio buttons, using methods like click, sendKeys, getText, etc.
4. **Navigation and Control**: WebDriver enables navigation actions like opening a URL, navigating forward and backward in the browser history, refreshing a page, and managing browser windows and tabs.
5. **Integration with Testing Frameworks**: Selenium WebDriver can be integrated with various testing frameworks such as **TestNG**, NUnit, and more, to organize and manage test cases effectively.
6. **Community and Support**: Selenium WebDriver has a large and active community, providing support, documentation, and a rich set of resources, including tutorials, forums, and sample projects.

Selenium Locators

When using Selenium WebDriver to interact with web elements, you typically use a combination of locators and methods to find and manipulate these elements.

1. **ID**: The unique identifier for an element in the HTML document. You can use **By.id()** to locate an element using its ID.
2. **Name**: The name attribute of an element. You can use **By.name()** to locate an element using its name attribute.
3. **Class Name**: The CSS class name of an element. You can use **By.className()** to locate an element using its class name.
4. **Tag Name**: The HTML tag name of an element. You can use **By.tagName()** to locate elements based on their HTML tag name.
5. **Link Text**: The visible text of a link. You can use **By.linkText()** to locate links by their visible text.
6. **CSS Selector**: A CSS selector to identify elements. You can use **By.cssSelector()** to locate elements using CSS selectors.
7. **XPath**: An XPath expression to identify elements. You can use **By.xpath()** to locate elements using XPath.

Selenium Locator Example

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class LocatorExample {
    public static void main(String[] args) {
        // Set the path to your chromedriver executable
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");

        // Initialize the WebDriver
        WebDriver driver = new ChromeDriver();

        // Navigate to a webpage
        driver.get("https://example.com");

        // Locate an element by name attribute
        WebElement elementByName = driver.findElement(By.name("username"));
        // Perform actions on the element
        elementByName.sendKeys("exampleusername");

        // Locate an element by id attribute
        WebElement elementById = driver.findElement(By.id("password"));
        // Perform actions on the element
        elementById.sendKeys("examplepassword");

        // Close the browser
        driver.quit();
    }
}
```

TestNG

TestNG (Test Next Generation) is an open-source testing framework. It is widely used for test automation in Java programming language, specifically for unit testing, integration testing, and end-to-end testing of applications.

Key features of TestNG include:

1. **Annotations**: TestNG uses annotations to define the methods that will be run as tests. Annotations like **@Test, @BeforeSuite, @AfterSuite, @BeforeTest, @AfterTest,** etc. help in structuring and organizing the test suite.
2. **Test Configuration**: TestNG allows users to define test configurations using XML files, providing flexibility in setting up suites, test cases, and their configurations such as parallel execution, parameterization, dependency management, and more.
3. **Parameterization**: TestNG supports parameterized tests, allowing you to run the same test method with different sets of data.
4. **Dependency Management**: TestNG allows you to define dependencies between test methods, ensuring that certain methods run before or after others, providing better control over test execution flow.
5. **Parallel Execution**: TestNG allows parallel execution of test cases at the suite, class, or method level, enhancing efficiency and reducing the overall test execution time.
6. **Reporting**: TestNG generates comprehensive HTML reports, providing detailed information about test execution, test results, logs, and more. This makes it easier to analyze the test results.

TestNG

TestNG provides a set of annotations that allow users to control the test execution flow and behavior. These annotations are used to specify the methods that need to be executed as tests, configure the test environment, and handle test results.

Here are some of the commonly used annotations in TestNG and their descriptions:

- **@Test:** Marks a method as a test method that TestNG should execute.

→ Syntax: @Test

- **@BeforeSuite:** Specifies a method to run before all tests in a suite.

→ Syntax: @ **BeforeSuite**

- **@AfterSuite:** Specifies a method to run after all tests in a suite have executed.

→ Syntax: @ **AfterSuite**

- **@BeforeTest:** Specifies a method to run before any test method belonging to the specified **<test>** tag in the XML suite file.

→ Syntax: @ **BeforeTest**

- **@AfterTest:** Specifies a method to run after all the test methods belonging to the specified **<test>** tag in the XML suite file.

→ Syntax: @ **AfterTest**

TestNG

TestNG provides a set of annotations that allow users to control the test execution flow and behavior. These annotations are used to specify the methods that need to be executed as tests, configure the test environment, and handle test results.

Here are some of the commonly used annotations in TestNG and their descriptions:

- **@BeforeClass:** Specifies a method to run before the first test method in the current class.

→ Syntax: @ **BeforeClass**

- **@AfterClass:** Specifies a method to run after all the test methods in the current class have executed.

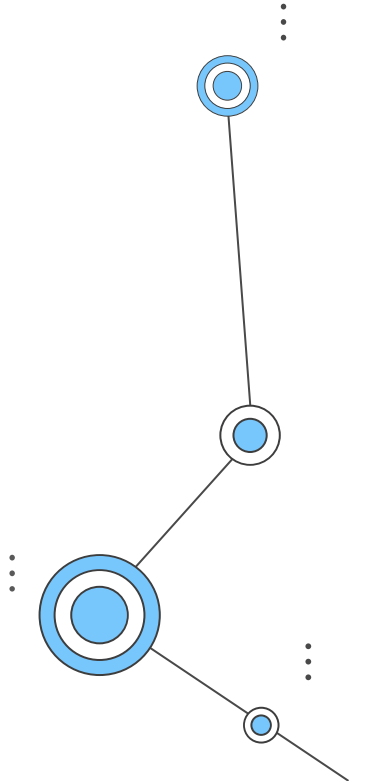
→ Syntax: @ **AfterClass**

- **@BeforeMethod:** Specifies a method to run before each test method in the class.

→ Syntax: @ **BeforeMethod**

- **@AfterMethod:** Specifies a method to run after each test method in the class.

→ Syntax: @ **AfterMethod**



Thanks!

Do you have any questions?

Email: parvez9605@gmail.com
WhatsApp: +88 01772880239