Selenium webdriver

With the growing need for efficient software products, every software development group need to carry out a series of tests before launching the final product into the market. Test engineers strive to catch the faults or bugs before the software product is released, yet delivered software always has defects. Even with the best manual testing processes, there's always a possibility that the final software product is left with a defect or is unable to meet the end user requirement. Automation testing is the best way to increase the effectiveness, efficiency and coverage of your software testing.

Selenium webdriver is the popular and open source tool and framework for automating the browser interactions.It enables testers and developers to write the code in various programming languages to automate the actions performed by a normal user.The actions can be anything such as clicking buttons, entering login details, performing transactions etc.

Key aspects/ features:

1. Selenium web driver supports multiple web browsers such as chrome, edge, firefox etc
2. Selenium web driver allows testers and developers to write the test scripts in various programming languages.
3. The web driver also allows to access and manipulate various web elements present in a web site such as buttons,link texts etc.
4. This driver is only used to automate the tests on the web based applications to check whether they are working fine or not.
5. The driver can be integrated with various testing frameworks such as TestNG and Junit in order to get additional features and better testing reports.

Selenium web driver architecture

Selenium web driver provides communication between programming languages and browsers

There are four basic components in selenium architecture :

- Selenium language bindings
- JSON wire protocols
- Browser drivers
- Real Browsers

1. Selenium language bindings : The language bindings allows the testers and developers to write the test scripts in their favourite programming languages such as java, python, C, Ruby.
2. JSON wire protocol :
   The JSON Wire Protocol, often simply referred to as the WebDriver protocol, was a communication protocol used by the Selenium WebDriver framework to interact with web browsers. It defined a standard way for test scripts to communicate with browser drivers, enabling automation of web browser actions.
3. Browser drivers : Selenium uses drivers, specific to each browser in order to establish a secure connection with the browser without revealing the internal logic of browser's functionality. The browser driver is also specific to the language used for automation such as Java, C#, etc.

When we execute a test script using WebDriver, the following operations are performed internally.

- HTTP request is generated and sent to the browser driver for each Selenium command.
- The driver receives the HTTP request through HTTP server.
- HTTP Server decides all the steps to perform instructions which are executed on browser.
- Execution status is sent back to HTTP Server which is subsequently sent back to automation script.

4.Real browsers : Browsers supported by Selenium WebDriver:

- Internet Explorer
- Mozilla Firefox
- Google Chrome
- Safari
- Microsoft Edge

Automation Testing

Automation testing, often referred to as automated testing, is a software testing technique where automated tools and scripts are used to perform tests on software applications or systems. The primary goal of automation testing is to automate repetitive and time-consuming manual testing tasks, thereby improving testing efficiency, repeatability, and accuracy.

Features

1. Automation testing relies on specialized software tools and frameworks designed for creating, executing, and managing test scripts. Some popular automation testing tools include Selenium, Appium, TestNG, JUnit, and more.
2. Automation tests are typically scripted, meaning that test scenarios and test cases are written as code or scripts.
3. Automated tests can be run repeatedly without fatigue or variation, ensuring consistent test results.
4. Automated tests can execute a large number of test cases in a relatively short amount of time compared to manual testing.
5. Automation is commonly used for regression testing, where previously tested functionalities are automatically retested after code changes to identify potential regressions or unintended side effects.
6. Automation tools typically provide detailed test execution reports, making it easier to track and analyze test results.

Automation testing life cycle

The Automation Testing Life Cycle (ATLC) is a series of steps and processes followed to effectively plan, design, implement, and maintain automated tests. It helps ensure that automated testing is organized, efficient, and produces reliable results. Here's a simple explanation of the key stages in the Automation Testing Life Cycle:

1. **Test Planning:**
    - In this initial stage, the automation testing team identifies what needs to be automated. They analyze the application, its requirements, and prioritize which test cases should be automated based on factors like frequency, complexity, and criticality.

2. **Test Design:**
    - Once the test cases are selected for automation, the team designs the automated test scripts. This involves defining the steps that the automated script will follow, including interactions with the application's user interface and the expected results.
3. **Script Development:**
    - During this stage, testers write the actual automation scripts. These scripts are typically written in a programming language and use automation testing tools like Selenium or Appium to interact with the application. Test data and parameters may also be defined in this phase.
4. **Script Execution:**
    - The automated scripts are executed against the application under test. This involves running the scripts using automation tools and capturing the results. Automated tests can be executed on different browsers, devices, or environments to ensure cross-compatibility.
5. **Result Analysis:**
    - After the automated tests are executed, the results are analyzed to identify any defects or issues in the application. Test reports are generated to provide insights into test coverage and the pass/fail status of each test case.
6. **Defect Reporting:**
    - If defects or issues are identified during automated testing, they are reported to the development team for resolution. Detailed information about the defects, including steps to reproduce them, is typically provided.
7. **Regression Testing:**
    - Automated tests are often used for regression testing, where previously tested functionalities are retested to ensure they still work after code changes. This helps identify any unintended side effects of recent code modifications.
8. **Maintenance and Enhancement:**

- As the application evolves, automated test scripts may need to be updated or enhanced to reflect changes in the application's functionality or user interface. Maintenance ensures that the automated tests remain relevant and effective.

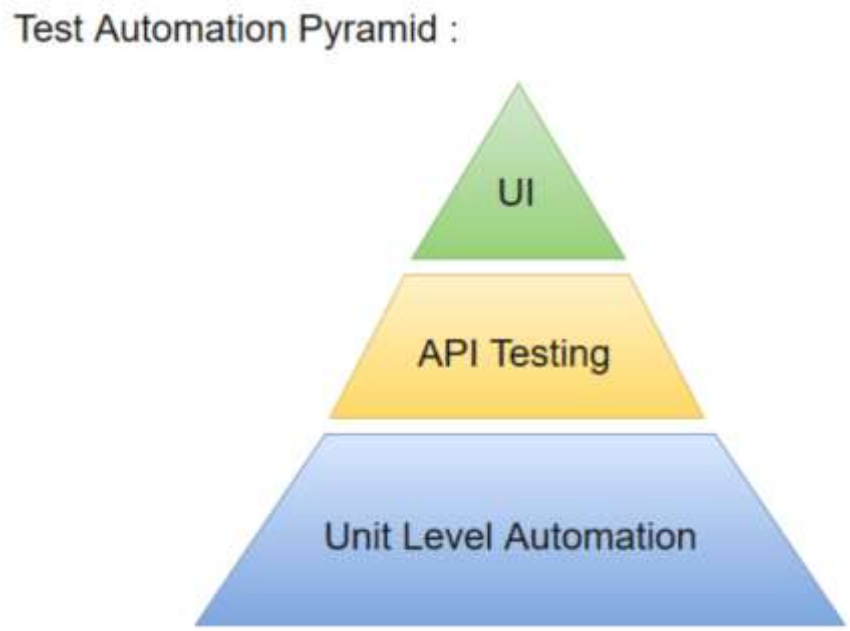
9. **Continuous Integration (CI)/Continuous Delivery (CD) Integration:**
    - Automated tests are integrated into the CI/CD pipeline, where they are executed automatically whenever there are code changes. This ensures that tests are continuously run to catch issues early in the development process.
10. **Monitoring and Reporting:**
    - Automated tests can be monitored and scheduled to run at specific intervals. Comprehensive reports are generated, and test metrics are collected to track the overall quality of the application.

Why choose automated testing

1. Automated tests can be executed much faster than manual tests. They are especially useful for repetitive and time-consuming tasks, allowing testers to focus on more creative and complex aspects of testing.
2. Automated tests ensure that the same steps are followed precisely in every test run. This consistency is crucial for regression testing, where previously tested functionalities are retested to check for regressions after code changes.
3. Automated tests eliminate human errors that can occur during manual testing, such as typos, oversight, or variations in test execution. This leads to more reliable and consistent test results.
4. Automation tools can be used for data-driven testing, where test cases are executed with different sets of test data to validate application behavior under various conditions.
5. Automated tests can be configured to run consistently across different platforms, browsers, and operating systems, ensuring cross-compatibility.
6. Automated tests serve as documentation of test cases and can be valuable for knowledge transfer, especially when new team members join a project.

Test automation for web applications

The most effective manner to carry out test automation for web application is to adopt a pyramid testing strategy.This pyramid testing strategy includes automation tests at three different levels. Unit testing represents the base and biggest percentage of this test automation pyramid. Next comes, service layer, or API testing. And finally, GUI tests sit at the top. The pyramid looks something like this:



The Test Automation Pyramid is a concept in software testing that represents the ideal distribution of different types of tests in an automated testing strategy. The pyramid is often used as a visual model to guide testing efforts and prioritize the types of tests to create for better test coverage and efficiency. The pyramid consists of three layers, each representing a different level of testing:

1. **Unit Tests (Bottom Layer):**
    - The base of the pyramid consists of unit tests, which are the smallest and most focused tests. Unit tests validate the behaviour of individual code units, such as functions or methods, in isolation. They typically do not interact with external dependencies like databases or web services.
    - Key characteristics of unit tests:
        - They are fast to execute.

- They are written by developers for the code they are responsible for.
- They help ensure that each component of the software functions correctly at the code level.

2. **Integration Tests (Middle Layer):**
    - The middle layer of the pyramid is made up of integration tests. These tests focus on verifying that different components or modules of the software work together as expected when integrated. Integration tests often involve testing interactions with external systems, databases, APIs, or services.
    - Key characteristics of integration tests:
        - They are broader in scope than unit tests but narrower than end-to-end tests.
        - They may involve setting up test environments and data.
        - They help detect issues related to the integration of various software components.
3. **End-to-End Tests (Top Layer):**
    - The top layer of the pyramid includes end-to-end tests, also known as UI tests or acceptance tests. These tests simulate user interactions with the entire application, from the user interface to the back-end systems. End-to-end tests validate that the entire application functions correctly from the user's perspective.
    - Key characteristics of end-to-end tests:
        - They are typically slower and more complex to create than unit and integration tests.
        - They interact with the application's user interface and external dependencies.
        - They are often written by QA engineers to validate user workflows and business requirements.

Test automation pyramid by taking the live project as an example

let's explore the Test Automation Pyramid in the context of a real-time project scenario. Suppose we're working on an e-commerce web application like Amazon, and we want to design our test automation strategy using the pyramid model.

1. Unit Tests:

In our e-commerce application, unit tests would focus on testing individual code units, such as functions or methods, that handle specific functionalities. For example:

- **Unit Test 1:** We have a function that calculates the total price of items in a shopping cart. A unit test can verify that this function correctly calculates the total amount based on the items' prices and quantities.
- **Unit Test 2:** There's a function responsible for validating user addresses during the checkout process. A unit test can check if this function correctly identifies valid and invalid addresses.

2. Integration Tests:

Integration tests come into play when we want to ensure that different parts of our application work together smoothly. In our e-commerce app:

- **Integration Test 1:** We have a module that handles user authentication. An integration test can verify that the login process correctly authenticates users, integrates with the user database, and redirects to the user's profile page upon successful login.
- **Integration Test 2:** Our application interacts with a payment gateway to process payments. An integration test can confirm that the application communicates effectively with the payment gateway and handles successful and failed transactions appropriately.

3. End-to-End Tests:

End-to-end tests simulate real user interactions with our e-commerce website. These tests ensure that the entire application, from the user interface to the back-end systems, functions correctly:

- **End-to-End Test 1 (UI Test):** This test can mimic a user's journey through the application, from searching for a product, adding it to the cart,

proceeding to checkout, and completing the purchase. It verifies that the UI elements and workflows are functioning as expected.

- **End-to-End Test 2 (Scenario Test):** We may want to test a specific user scenario, such as a user returning a product. This test can involve logging in, navigating to the order history, initiating a return request, and tracking the return status. It validates that the entire process works seamlessly.

Limitations of Selenium

1. Selenium does not support test automation for desktop applications.
2. This requires more expert level skills in order to automate the test more effectively.
3. We should know atleast one programming language in order to write the test scripts using selenium webdriver.
4. Selenium does not implicitly delivers test reports.It should explicitly rely on other frame works such as TestNG and Junit.
5. Selenium does not provide any test tool integration for test management.

Differences between selenium and QTP

Selenium	QTP
1.It is open source tool.	1.It is not open source tool.
2.Supports automation only for web based applications.	2.Supports test on both web based and desktop applications.
3.Low resource consumption.	3.High resource consumption.
4.Supports Java,C,Ruby,Python languages.	4.Supports VB script
5.Supports Android,IOS,Windows,Linux,Mac environments.	5.Supports only for windows.
6. It relies on external tool for generating test reports.	6. Built-in test report generation within the tool.

Selenium IDE

Selenium IDE is like a helpful robot that records and plays back your actions when you use a web browser. It's a simple tool for creating and running automated tests on websites without needing to write a lot of code.

Here's a simple breakdown:

1. **Recording Your Actions:** Imagine you want to test a website by clicking buttons, filling out forms, and checking if certain things work correctly. With Selenium IDE, you click a button, type text, or do anything you'd normally do on a website, and it remembers what you did.
2. **Replaying Your Actions:** After you've recorded your actions once, you can tell Selenium IDE to do the same things again automatically. It plays back the actions, like a video of you using the website, so you can check if everything still works as expected.
3. **No Coding Required:** The great thing about Selenium IDE is that you don't need to be a programmer to use it. It's designed to be user-friendly, so anyone can create automated tests for websites without writing a lot of complex code.

Selenium RC

Selenium RC, or Selenium Remote Control, was an older version of the Selenium testing framework. It allowed you to control web browsers, like Chrome or Firefox, from a remote computer. Here's a simple explanation:

Imagine you have two computers: one for controlling and running your tests (let's call it "Computer A"), and another where you want to open a web browser and test a website (let's call it "Computer B").

Selenium RC acted as a bridge between these two computers. It allowed you to write test instructions on Computer A, like "Open the browser, go to this website, click this button," and then it sent these instructions to Computer B. Computer B followed these instructions, just like a robot following your commands, and performed the actions in a web browser.

Selenium Grid

Selenium Grid is like a traffic cop for your automated tests. It helps you run your tests on different web browsers and computers at the same time, making web testing faster and more efficient. Here's a simple explanation:

Imagine you have a lot of tests to run, and you want to make sure they work on different web browsers (like Chrome, Firefox, and Safari) and on different computers (say, Windows and Mac). Doing this one by one would take a lot of time.

Selenium Grid comes to the rescue. It's like a central hub that manages all your tests. You send your tests to Selenium Grid, and it figures out which computer and browser to use for each test. It then runs multiple tests simultaneously on different setups.

So, instead of running tests one after the other, Selenium Grid lets you run several tests at once on various browsers and computers, saving you time and helping ensure your website works well everywhere. It's a helpful tool for testing your web applications across different environments.

Selenium IDE commands

Selenium commands are basically classified in three categories:

1. Actions
2. Accessors
3. Assertions

Actions : Actions are the selenium commands that generally manipulate the state of the application. Execution of Actions generates events like click this link, select that option, type this box, etc. If an Action fails, or has a bug, the execution of current test is stopped.

Command/Syntax	Description
open (url)	It launches the desired URL in the specified browser and it accepts both relative and absolute URLs.
type (locator,value)	It sets the value of an input field, similar to user typing action.
typeKeys (locator,value)	This command simulates keystroke events on the specified element.
click (locator)	This command enables clicks on a link, button, checkbox or radio button.
clickAt (locator,coordString)	This command enables clicks on an element with the help of locator and co-ordinates
doubleClick (locator)	This command enables double clicks on a webelement based on the specified element.
focus (locator)	It moves the focus to the specified element

Accessors : Accessors are the selenium commands that examine the state of the application and store the results in variables. They are also used to automatically generate Assertions.

Command/Syntax	Description
storeTitle (variableName)	This command gets the title of the current page.
storeText (locator, variableName)	This command gets the text of an element..
storeValue (locator,variableName)	This command gets the (whitespace-trimmed) value of an input field.
storeTable (tableCellAddress, variableName)	This command gets the text from a cell of a table.
storeLocation (variableName)	This command gets the absolute URL of the current page.
storeElementIndex (locator, variableName)	This command gets the relative index of an element to its parent (starting from 0).
storeBodyText (variableName)	This command gets the entire text of the page.
storeAllButtons (variableName)	It returns the IDs of all buttons on the page.
storeAllFields (variableName)	It returns the IDs of all input fields on the page.
storeAllLinks (variableName)	It returns the IDs of all links on the page.

Assertions : Assertions are the commands that enable testers to verify the state of the application. Assertions are generally used in three modes assert, verify and waitfor.

Command/Syntax	Description
verifySelected(selectLocator, optionLocator)	This command verifies that the selected option of a drop-down satisfies the optionSpecifier.
verifyAlert (pattern)	This command verifies the alert text; used with accessorstoreAlert.
verifyAllButtons (pattern)	This command verifies the button which is used withaccessorstoreAllButtons.
verifyAllLinks (pattern)	This command verifies all links; used with the accessorstoreAllLinks.
verifyBodyText(pattern)	This command verifies the body text; used with the accessorstoreBodyText.
verifyAttribute(attributeLocator, pattern)	This command verifies an attribute of an element; used with the accessorstoreAttribute.
waitForErrorOnNext (message)	This command enables Waits for error; used with the accessorassertErrorOnNext.
waitForAlert (pattern)	This command enables waits for the alert; used with the accessorstoreAlert.
verifyAllWindowIds (pattern)	This command verifies the window id; used with the accessorstoreAllWindowIds.

Difference between RC and driver

Aspect	Selenium WebDriver	Selenium RC (Remote Control)
Architecture	Direct communication with browsers, no need for a separate server.	Requires a Selenium Server as a mediator between the test scripts and browsers.
Programming Language Support	Supports multiple programming languages with language-specific bindings.	Supports multiple programming languages but with less extensive language-specific support.
Ease of Setup	Easier to set up and configure.	More complex setup due to the need for the Selenium Server.
Stability	Generally more stable and reliable.	Less stable due to the additional server layer.
Speed and Efficiency	Faster and more efficient due to direct browser communication.	Slower and less efficient due to server-based communication.
Browser Support	Supports a wide range of browsers.	Limited browser support, especially for newer browser versions.
Maintainability	Easier to maintain due to simpler architecture.	More challenging to maintain due to the complexity of the server setup.
Community Support	Strong and active community support.	Less active community support as it is an older technology.

How to setup selenium Grid

1. Download the Selenium Standalone Server to run Remote Selenium webdriver. It is available in a single jar file.
2. Store the jar file at any of the drive.
3. Open the cmd.
4. Register the hub through cmd. Enter the command java -jar selenium-server-standalone-3.8.1.jar -role hub. This command will treat the machine as a hub.
5. Open the link, i.e., http://192.168.1.12:4444/grid/console where the server resides. Hub is the server only.
6. Log in to another machine and register it as a node for a hub. I will remotely connect my machine to another machine through Teamviewer. To register the node with your hub, you can do it from the node machine only not from your machine, so I connect my machine to another machine. In a node machine, run the command "java -jar selenium-server-standalone-3.141.59.jar role webdriver -hub >ipaddress>/grid/register -port 5566".
7. Now if we want to run the test cases in a Google chrome or Firefox browser, then we need to download the chrome driver or geekodriver in a node machine. In order to achieve this, we need to run the following command in a node machine: "java -Dwebdriver.chrome.driver="D:\chromedriver.exe" -jar selenium-server-standalone-3.141.59.jar role webdriver -hub >ipaddress>/grid/register -port 5566".

Refer to interview questions :

https://www.javatpoint.com/selenium-interview-questions

Junit

JUnit is like a friendly assistant that helps ensure your software works correctly. It's a popular tool in the world of automated testing, specifically designed for Java applications. Here's a simple explanation:

Imagine you've built a complex machine, and you want to make sure it runs smoothly. You need a reliable way to check if all the parts are working as expected. This is where JUnit comes in:

1. **Test Instructions:** You write simple sets of instructions (tests) to check different parts of your machine (code). For example, you might test if the engine starts, if the wheels turn, and if the lights work.
2. **Running Tests:** JUnit is like your testing manager. You hand over your instructions (tests) to JUnit, and it makes sure to run them one by one, just like a checklist.
3. **Checking Results:** JUnit keeps an eye on each test and tells you if any of them fail. If a test fails, it means a part of your machine isn't working correctly, and you need to fix it.
4. **Reporting:** JUnit provides a report card, showing which tests passed and which ones failed. This helps you quickly identify and fix any problems in your machine

Annotations of Junit

JUnit uses annotations to define and control the behavior of test methods. These annotations provide instructions to JUnit about how to run the tests. Here are some common JUnit annotations, explained with simple examples:

@Test:

- The **@Test** annotation marks a method as a test method. JUnit will execute any method annotated with **@Test** when running tests.
- Example:

```
import org.junit.Test;

public class MyTestClass {

   @Test
```

```
    public void testAddition() {
        int result = 1 + 2;
        assertEquals(3, result);
    }
}
```

@Before and @After:

- **@Before** and **@After** annotations are used to specify methods that should run before and after each test method, respectively. These are often used for setting up and cleaning up test data.
- Example:

```
import org.junit.Before;
import org.junit.After;
public class MyTestClass {
    @Before
    public void setUp() {
        // Set up test data or resources
    }
    @Test
    public void testSomething() {
        // Test logic
    }
    @After
    public void tearDown() {
        // Clean up after the test
    }
}
```

@BeforeClass and @AfterClass:

- **@BeforeClass** and **@AfterClass** annotations are used for methods that should run once before and after all the test methods in a class, typically for expensive setup or teardown operations.
- Example:

```
import org.junit.BeforeClass;
import org.junit.AfterClass;
public class MyTestClass {
    @BeforeClass
    public static void setUpClass() {
        // One-time setup for the entire test class
    }
    @Test
    public void testSomething() {
        // Test logic
    }
    @AfterClass
    public static void tearDownClass() {
        // One-time cleanup for the entire test class
    }
}
```

@Ignore:

- The **@Ignore** annotation marks a test method to be ignored or skipped during test execution. This can be useful when you temporarily want to exclude a test from running.
- Example:

```
import org.junit.Ignore;
import org.junit.Test;
public class MyTestClass {
    @Ignore
    @Test
    public void ignoredTest() {
        // This test will be skipped
    }
    @Test
    public void testSomething() {
        // Test logic
    }
}
```

@Test(expected = Exception.class):

- You can use the **@Test** annotation with the **expected** attribute to specify that a particular test method is expected to throw a specific exception.
- Example:

```
import org.junit.Test;
public class MyTestClass {
    @Test(expected = ArithmeticException.class)
    public void testDivisionByZero() {
        int result = 1 / 0; // This should throw ArithmeticException
    }
}
```

@RunWIth(Parametrized.class) and @Parameters : These are some advanced annotations in Junit.

@RunWith(Parameterized.class): This annotation tells JUnit to use the Parameterized test runner, which is designed for running parameterized tests. It allows you to run the same test method with different sets of parameters.

@Parameters gives the list of inputs along with the expected output as a 2D array.

Ex :

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import java.util.Arrays;
import java.util.Collection;
@RunWith(Parameterized.class)
public class StringConcatenationTest {
   private String str1;
   private String str2;
   private String expected;
   public StringConcatenationTest(String str1, String str2, String expected) {
      this.str1 = str1;
      this.str2 = str2;
      this.expected = expected;
   }
   @Parameters
   public static Collection<Object[]> data() {
      return Arrays.asList(new Object[][] {
```

```
            { "Hello, ", "world!", "Hello, world!" },
            { "Good ", "morning!", "Good morning!" },
            { "JUnit is ", "awesome!", "JUnit is awesome!" },
        });
    }
    @Test
    public void testConcatenation() {
        String result = concatenateStrings(str1, str2);
        assertEquals(expected, result);
    }
    // The function to be tested
    private String concatenateStrings(String str1, String str2) {
        return str1 + str2;
    }
}
```

Features of Junit

JUnit is a widely used testing framework for Java that provides several features to facilitate unit testing and test-driven development. Here are the key features of JUnit:

1. Annotations: JUnit uses annotations to define test methods, setup and teardown methods, and custom testing behavior. Annotations make it easy to mark and organize test methods within test classes.
2. Assertions: JUnit provides a set of assertion methods (e.g., assertEquals, assertTrue, assertNotNull) that allow you to check if the actual results of your tests match the expected outcomes. These assertions help identify and report test failures.

3. Parameterized Tests: JUnit supports parameterized tests, allowing you to run the same test method with different sets of input data. This feature is useful for testing a method or function with multiple test cases efficiently.
4. **Exception Testing:** JUnit provides annotations like **@Test(expected = Exception.class)** and methods like **expectThrows** to test that specific exceptions are thrown under specific conditions.
5. **Timeouts:** You can set a timeout for test methods to ensure that they complete within a specified time limit. This is useful for detecting slow or potentially hanging tests.
6. **Test Reporting:** JUnit generates detailed test reports, including information about test successes and failures. Popular build tools like Maven and Gradle integrate with JUnit for generating test reports.

TestNG

TestNG is a popular open source testing framework which is used in automated testing. It stands for Test Next Generation and is made to make the testing more easier and convenient.

Features

1. TestNG lets you organize the tests in a meaningful sequence or order and makes it easier to run them for the testers.
2. TestNG also uses annotations similar to Junit to mark which methods are test methods, setup methods, teardown methods etc.
3. TestNG allows you to pass parameters for a test case making it run at different scenarios to understand its accuracy.
4. TestNG generates detailed test reports of all the test cases, which lets us know which tests passed and failed and also why they have failed.
5. TestNG also allows us to run multiple test cases simultaneously to save time for us and speed up the testing process.

Difference between TestNG and Junit

TestNG	Junit
1.Supports testing using XML configuration and annotations.	1.Primarily uses annotations for testing.
2.Built in support for parallel execution of test methods.	2.Limited support for parallel execution using third party libraries.
3.Supports defining dependencies between test methods using “dependsOnMethods” and “dependsOnGroups”	3.Lacks built in support of defining dependencies but possible using custom code.
4.Allows parameterizing the test methods using @Parameters.	4.Allows parameterizing using @Parameterized tests but it is less flexible than TestNG.
5.Generates detailed HTML reports after testing.	5.Generates XML reports after testing.
6.Integrates well with IDE’s such as Intellij, eclipse	6.This also integrates well but not very much when compared to TestNG.

Annotations in TestNG

1.**@Test**: Marks a method as a test case.

Example :

```
import org.testng.annotations.Test;

public class ExampleTest {
   @Test
   public void testAddition() {
      int result = 2 + 3;
      assert result == 5;
   }
}
```

2.**@BeforeMethod** and **@AfterMethod**: Used to run setup and teardown code before and after each test method.

Example :

```
import org.testng.annotations.*;

public class ExampleTest {
   @BeforeMethod
   public void setup() {
      // Setup code (e.g., initializing resources)
   }
   @AfterMethod
   public void teardown() {
      // Teardown code (e.g., releasing resources)
   }
   @Test
   public void testSomething() {
      // Test code
```

```
    }
}
```

3.**@BeforeClass** and **@AfterClass**: Used to run setup and teardown code once before and after all test methods in a class.

Example :

```
import org.testng.annotations.*;

public class ExampleTest {
    @BeforeClass
    public void setupClass() {
        // Setup code for the entire class
    }
    @AfterClass
    public void teardownClass() {
        // Teardown code for the entire class
    }
    @Test
    public void test1() {
        // Test code
    }
    @Test
    public void test2() {
        // Test code
    }
}
```

4.**@BeforeSuite** and **@AfterSuite**: Used to run setup and teardown code once before and after all test suites.

Example :

```
import org.testng.annotations.*;

public class ExampleTest {
    @BeforeSuite
    public void setupSuite() {
        // Setup code for the entire test suite
    }
    @AfterSuite
    public void teardownSuite() {
        // Teardown code for the entire test suite
    }
    @Test
    public void test1() {
        // Test code
    }
    @Test
    public void test2() {
        // Test code
    }
}
```

5.**@DataProvider**: Provides data for parameterized tests.

Example :

```
import org.testng.annotations.*;
```

```
public class ExampleTest {
    @DataProvider(name = "data")
    public Object[][] testData() {
        return new Object[][] {
            { 2, 3, 5 },
            { 0, 0, 0 },
            { -1, 1, 0 }
        };
    }
    @Test(dataProvider = "data")
    public void testAddition(int a, int b, int expected) {
        int result = a + b;
        assert result == expected;
    }
}
```

6.**@Test(dependsOnMethods = {"methodName"})**: Specifies test method dependencies. A test method will only run if the specified dependent methods pass.

Example :

```
import org.testng.annotations.*;
public class ExampleTest {
    @Test
    public void login() {
        // Login test
    }
```

```
    @Test(dependsOnMethods = "login")
    public void performActionAfterLogin() {
        // This method will run only if the "login" method passes.
    }
}
```

7.**@Test(enabled = false)**: Disables a test method temporarily.

Example :

```
import org.testng.annotations.*;
public class ExampleTest {
    @Test(enabled = false)
    public void temporarilyDisabledTest() {
        // This test will not be executed until enabled is set to true.
    }
}
```

8.**@Test(priority = n)**: Specifies the priority order for test methods within a class. Tests with lower priority values will run first.

Example :

```
import org.testng.annotations.*;
public class ExampleTest {
    @Test(priority = 2)
    public void highPriorityTest() {
        // Test code with high priority
    }
    @Test(priority = 1)
    public void lowPriorityTest() {
```

```
    // Test code with low priority
  }
}
```

9.**@Test(timeOut = n)**: Sets a maximum time limit (in milliseconds) for a test method to execute. If the test takes longer, it will be marked as a failure.

Example :

```
import org.testng.annotations.*;

public class ExampleTest {
  @Test(timeOut = 2000)
  public void testWithTimeout() throws InterruptedException {
    // This test should complete within 2 seconds; otherwise, it will fail.
    Thread.sleep(3000);
  }
}
```

10.**@Test(expectedExceptions = Exception.class)**: Specifies that a test method is expected to throw a particular exception. If the expected exception is not thrown, the test will fail.

Example :

```
import org.testng.annotations.*;

public class ExampleTest {
  @Test(expectedExceptions = ArithmeticException.class)
  public void testExceptionHandling() {
    int result = 1 / 0; // This should throw an ArithmeticException.
  }
}
```

Grouping in TestNG

Grouping in TestNG allows you to categorize your test methods into logical groups, making it easier to run specific subsets of tests based on their characteristics. This is useful for organizing and executing tests efficiently.

Suppose you have a test suite for an e-commerce application, and you want to categorize your tests into groups like "checkout," "product search," and "user management."

1. **Define Groups in TestNG XML Configuration**:

First, define groups in your TestNG XML configuration file (**testng.xml**). Here's an example:

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="Ecommerce Suite">
  <test name="Ecommerce Test">
    <groups>
      <run>
        <include name="checkout" />
      </run>
    </groups>
    <classes>
      <class name="com.example.CheckoutTests" />
    </classes>
  </test>
</suite>
```

In this example, we have a suite named "Ecommerce Suite" with a test named "Ecommerce Test." We've included the "checkout" group, which means that only test methods belonging to the "checkout" group will be executed in this test.

2. **Group Your Test Methods**:

In your Java test classes, use the **@Test** annotation with the **groups** attribute to specify which group(s) a test method belongs to. Here's an example:

```
import org.testng.annotations.Test;

public class CheckoutTests {
    @Test(groups = { "checkout" })
    public void testAddToCart() {
        // Test code for adding items to the cart
    }
    @Test(groups = { "checkout" })
    public void testCheckoutProcess() {
        // Test code for the checkout process
    }
    @Test(groups = { "product search" })
    public void testSearchProduct() {
        // Test code for searching a product
    }
}
```

In this example, we have three test methods. The first two methods belong to the "checkout" group, and the third method belongs to the "product search" group.

3.**Run the Test Suite**:

When you run your TestNG suite using the **testng.xml** configuration file, TestNG will execute only the test methods that belong to the specified groups. In this case, it will run the "checkout" group of tests.

4.**Results**:

After running the suite, TestNG will generate reports indicating the status of each test method within the selected group(s).

invocationCount in TestNG

In TestNG, the **invocationCount** attribute is used to specify how many times a test method should be invoked (executed) during a single test run. This attribute allows you to run the same test method multiple times with different sets of data or under different conditions.

Example :

```
import org.testng.annotations.Test;

public class ExampleTest {
    @Test(invocationCount = 3)
    public void testMethod() {
        // This test method will be executed three times.
        // You can use it for repetitive testing or data-driven testing.
    }
}
```

Interview questions :

https://www.javatpoint.com/testng-interview-questions

Cucumber

Cucumber is an open source tool used for automation testing in software development.

It is often associated with BDD(behaviour driven development) and allows testers to develop various test cases in a human readable format.

Features:

1. Human readable tests : Cucumber lets you write the test scenarios in such a way that could be even understood by the non technical people. It uses gherkin language to write the scenarios ie a natural language which can be read and understood by any person.
2. Gherkin language : Gherkin is the language used by the cucumber to write the test scenarios. It uses simple keywords such as Given, when, then etc to write the test scenarios.
3. Step Definitions : When a scenario is written in natural language, each step should be mapped to a corresponding method or function which shows the steps execution. This code is written by the testers and is known as Step definitions.
4. Automation : Cucumber lets you to automate the test scenarios by matching the scenarios which are written in a ".features" extension file with the code which is written in "Stepdefinitions.java" file.When you run the tests, cucumber reads the scenarios matches them with the corresponding code and executes it accordingly.
5. Integration : Cucumber can be mapped with various testing frameworks such as TestNG, Junit in order to handle the test cases efficiently.
6. Reporting : Cucumber gives you a detailed reports of which steps have passed and failed which could be helpful to trace out the errors and resolve the issues in the code.

Behaviour driven development

BDD (Behavioral Driven Development) is a **software development** approach that was developed from **Test Driven Development (TDD)**.

All test cases are written in the form of simple English statements inside a feature file, which is human-generated.

BDD improves communication between technical and non-technical teams and stakeholders.

Key features of BDD in testing :

1. Natural language scenarios : BDD uses gherkin language to write the test scenarios.It uses human understandable language so that it would be easy for the technical and also the non technical stake holders to understand what is actually happening.
2. Given-When-Then Structure : This is a syntax called Gherkin language. This is written in a feature file.
   Example :
   Given a user is on the login page
   When they enter valid credentials
   Then they should be logged in successfully
3. Automation : BDD scenarios are not only documentation, each line written in gherkin language can be mapped to the some code and make it executed using many automation tools such as cucumber.
4. **Collaboration**: BDD encourages collaboration among cross-functional team members, including developers, testers, product owners, and business analysts. This collaboration ensures that everyone has a shared understanding of the desired behavior and requirements.
5. **Clear and Traceable Test Results**: BDD tools generate detailed test reports, making it easy to track the status of scenarios and identify which steps passed or failed. These reports enhance visibility into the testing process.

Behavior-Driven Development (BDD) is an approach to software development that emphasizes understanding and describing how software should behave from a user's perspective.

It uses plain language to express the behavior of a software feature or system, making it easier for both technical and non-technical team members to collaborate and ensure that the software meets user expectations.

In simple terms, BDD is like telling a story about what you want the software to do, step by step, in a way that anyone can understand. Here's a simple example:

Suppose you're building a calculator app, and you want to describe how the addition feature should work using BDD:

Traditional Requirement (Technical):

- "The addition function should take two numbers as input and return their sum."

BDD Scenario (User-Focused):

- **Scenario:** Adding two numbers
    - **Given** the user is on the calculator app
    - **When** the user enters "5" and "3" into the calculator
    - **Then** the calculator should display "8" as the result

In this BDD scenario, you describe the behavior step by step, starting with the initial conditions ("Given"), specifying the action ("When"), and stating the expected outcome ("Then"). This makes it clear how the addition feature should work from the user's perspective.

Difference between TDD and BDD

TDD	BDD
1.Testing of individual codes.	1.Testing of external behaviour from user perspective.
2.Typically written in a programming language.	2.written in natural language scenarios.
3.Emphasizes collaboration between testers and developers.	3.Promotes cross-functional team collaboration, including non-technical stakeholders.
4.Utilizes testing frameworks such as TestNG, Junit.	4.Uses BDD tools such as cucumber.
5.Focuses on testing small codes.	5.Focuses on testing large behavioural aspects of the software.

Difference between Cucumber and QTP(Quick Test Professional)

Cucumber	QTP
1.It is a BDD tool	1.It is an automated testing too.
2.It is open source tool.	2.It is an expensive paid software.
3.Supports many languages such as Java,Scala,Groovy etc	3.QTP supports only VB script.
4.It is used to test only web applications.	4.It is used to test web applications,desktop applications and any client based applications.

Feature file in cucumber

In Cucumber, a feature file is a plain-text file that serves as a starting point for defining and describing the behavior of a software feature or functionality in a human-readable and structured format. Feature files are a fundamental part of Behavior-Driven Development (BDD) and are written using the Gherkin language, which is designed to be easy to read and understand by both technical and non-technical stakeholders.

Key Components of a Feature File:

1. **Feature:** The feature file begins with the **Feature** keyword, followed by a concise title that describes the feature or functionality being tested.
2. **Description:** Below the **Feature** line, you can provide a more detailed description of the feature, providing context and background information.
3. **Scenarios:** The main body of the feature file consists of one or more scenarios, each described using the **Scenario** keyword. Scenarios represent specific use cases or test cases related to the feature.
4. **Steps:** Inside each scenario, you define steps using keywords like **Given**, **When**, **Then**, **And**, and **But**. These steps describe the sequence of actions, events, or conditions that lead to the expected behavior of the feature.

Tags in cucumber

In Cucumber, tags are labels or markers that you can assign to scenarios, features, or even individual scenario outlines within your feature files. Tags allow you to categorize and filter scenarios, making it easier to control which tests should be executed during test runs. Tags are often used for various purposes, including regression testing, test prioritization, and running subsets of tests.

Here's how you can use tags in Cucumber:

1. **Tagging Scenarios and Features:**

You can add tags to scenarios and features in your feature files using the **@** symbol followed by a tag name. Tags are typically placed just above the **Feature**, **Scenario**, or **Scenario Outline** keywords.

For example:

@smoke

Feature: User Registration

@positive

Scenario: Successful Registration

Given a user is on the registration page

When they fill in valid registration details

Then they should be registered successfully

@negative

Scenario: Registration with Duplicate Email

Given a user is on the registration page

When they use an email that already exists

Then they should see an error message

In this example, scenarios and features are tagged with **@smoke**, **@positive**, and **@negative** tags, allowing you to categorize and filter them.

Running Scenarios by Tags:

To run scenarios with specific tags, you can use the **--tags** or **-t** option when executing Cucumber tests from the command line.

For example: cucumber --tags @smoke

This command will run only the scenarios and features tagged with **@smoke**.

Combining Tags:

You can also combine multiple tags using logical operators **and**, **or**, and **not**. For example:

- **--tags @smoke and @positive**: Runs scenarios with both **@smoke** and **@positive** tags.
- **--tags @smoke or @regression**: Runs scenarios with either **@smoke** or **@regression** tags.
- **--tags not @wip**: Runs scenarios that do not have the **@wip** tag.

Gherkin language :

Gherkin is a simple, human-readable, and structured language that is used primarily for writing behavioral specifications of software features. It is often associated with Behavior-Driven Development (BDD) and is supported by tools like Cucumber, SpecFlow, and Behave. Gherkin acts as a bridge between non-technical stakeholders (such as business analysts and product owners) and technical team members (such as developers and testers) by providing a

common language to describe and understand the behavior of a software system.

Key features :

1.Natural Language: Gherkin is designed to be easy to read and write by using plain, human-readable language. It avoids technical jargon and complexities, making it accessible to non-technical team members.

2.Structured Syntax: Gherkin scenarios are organized in a structured format using specific keywords. The core keywords include:

- Feature: Describes the high-level feature or functionality being tested.
- Scenario: Represents a specific test case or use case.
- Given: Describes the initial context or preconditions for the scenario.
- When: Specifies the action or event that triggers the scenario.
- Then: Defines the expected outcome or result of the scenario.
- And and But: Used for additional steps within a scenario to maintain clarity.

3.**Comments**: Gherkin supports comments using the **#** symbol, allowing you to add explanations or notes to your scenarios.

Gherkin scenarios like this one serve as both documentation and the basis for automated tests. Developers write code (known as step definitions) that maps each step to the corresponding actions to automate the testing process. This approach ensures that software development and testing align closely with the intended behavior of the software from a user's perspective, promoting collaboration and clarity in the development process.

Scenario outlines

Scenario Outlines in Gherkin are a way to write more concise and reusable Gherkin scenarios, especially when you have multiple test cases with similar steps but different sets of data. They allow you to create a template for a scenario and then specify multiple examples, each with its own set of data. Scenario Outlines are particularly useful for data-driven testing. Here's an explanation with an example:

Without scenario outline :

```
Feature: Calculator Addition
  Scenario: Add Two Numbers (5 + 3)
    Given the calculator is open
    When I enter "5" into the calculator
    And I press the "+" button
    And I enter "3" into the calculator
    And I press the "=" button
    Then the result should be "8" on the screen
  Scenario: Add Two Numbers (2 + 7)
    Given the calculator is open
    When I enter "2" into the calculator
    And I press the "+" button
    And I enter "7" into the calculator
    And I press the "=" button
    Then the result should be "9" on the screen
  Scenario: Add Two Numbers (10 + 6)
    Given the calculator is open
    When I enter "10" into the calculator
    And I press the "+" button
    And I enter "6" into the calculator
    And I press the "=" button
    Then the result should be "16" on the screen
```

With scenario outline :

```
Feature: Calculator Addition

 Scenario Outline: Add Two Numbers

   Given the calculator is open

   When I enter "<number1>" into the calculator

   And I press the "+" button

   And I enter "<number2>" into the calculator

   And I press the "=" button

   Then the result should be "<result>" on the screen

   Examples:

    | number1 | number2 | result |

    | 5       | 3       | 8      |

    | 2       | 7       | 9      |

    | 10      | 6       | 16     |
```

In this Scenario Outline:

- The scenario is defined once with placeholders like **<number1>**, **<number2>**, and **<result>**.
- The **Examples** section provides a table of data, where each row represents a set of values for the placeholders.
- During test execution, the Scenario Outline is run for each row in the **Examples** table, replacing placeholders with actual data.

Pom.xml file

The **pom.xml** file is a critical component of Apache Maven, a popular build automation and project management tool used primarily in Java development. It stands for "Project Object Model" and serves several essential purposes in software development.

XML stands for "eXtensible Markup Language." It is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable

Here's a clear explanation of what a **pom.xml** file is and why it's used:

Project Configuration: The **pom.xml** file is used to define the configuration and settings for a software project. It contains essential information about the project, such as its name, version, description, and the group or organization that is responsible for it.

Dependencies Management: One of the most crucial functions of a **pom.xml** file is to manage project dependencies. Dependencies are external libraries and frameworks that your project relies on. In the **pom.xml**, you specify the dependencies your project needs, along with their versions. Maven will automatically download these dependencies from remote repositories and include them in your project's build process.

Reporting: The **pom.xml** file can also configure reporting plugins, which generate project reports such as code quality metrics, test results, and documentation. These reports can be valuable for project management and monitoring.

Build Configuration: The **pom.xml** file specifies how your project should be built. It includes information on which source code files to include, how to compile them, and how to package them into a distributable format (e.g., JAR, WAR). It also defines the build lifecycle phases and goals, allowing you to run various tasks like compiling, testing, packaging, and deploying your project.

Maven

Maven is a widely used build automation and project management tool primarily used in Java development,It is an open-source tool that simplifies and streamlines the process of building, managing, and distributing software projects.

Build Automation: Maven automates the build process of a software project. This includes compiling source code, running tests, packaging the application into executable formats (e.g., JAR, WAR), and performing other tasks necessary for creating a deployable artifact. Developers can initiate these tasks through

simple command-line commands or integrate them into a Continuous Integration (CI) system.

Dependency Management: Maven provides a centralized and standardized way to manage project dependencies. You specify the dependencies your project requires in the **pom.xml** file (Project Object Model), including their versions. Maven then retrieves these dependencies from remote repositories and ensures that the correct versions are used, simplifying the management of external libraries and frameworks.

Project Structure: Maven enforces a consistent project structure, which makes it easier for developers to understand and collaborate on a project.

Reporting and Documentation: Maven can generate various project reports, including code quality reports, test results, and project documentation.

Config.properties file

In software testing, a **config.properties** (or similar) file typically refers to a configuration file that stores various settings and parameters related to the testing environment or test execution.

This file is commonly used in test automation frameworks and scripts to manage configuration options, making it easier to modify settings without changing the code.

It's a file that holds important information and settings that a program needs to run correctly.

In software testing, this file might contain things like website addresses, usernames, and passwords that your testing program needs to know. Having these details in a separate file makes it easy to change them without having to modify the program's code. It's like changing your game settings without having to go into the game's code.

1. **Configuration Parameters**: It stores key-value pairs representing various configuration parameters. These parameters can include things like:
    - URLs: Addresses of web applications or APIs under test.
    - Credentials: Username and password for authentication.

- Environment-specific settings: Such as database connection details for different test environments (e.g., development, staging, production).
- Test data file paths: Paths to files containing test data.
- Timeout values: Time limits for waiting in tests.

2. **Environment Management**: Test environments can vary (e.g., development, testing, production), and a **config.properties** file allows testers to switch between them by changing the configuration settings. This ensures that the same test script can be used in different environments without code modification.
3. **Ease of Maintenance**: Separating configuration details from the test code improves maintainability. Testers and developers can modify settings in the **config.properties** file without needing to dig into the test code. This separation of concerns simplifies maintenance and reduces the risk of introducing errors when tweaking configuration options.
4. **Dynamic Test Execution**: Test scripts can read values from the **config.properties** file at runtime, allowing for dynamic behavior. For example, if the test script needs to interact with different URLs, it can read the URL from the configuration file, making the script adaptable to different scenarios.