**Cucumber & TestNG**

**Cucumber**

**1.Cucumber framework explanation / Have you used Cucumber in your projects?**

Cucumber is a popular open-source testing tool used for Behavior-Driven Development (BDD). It allows developers, testers, and business stakeholders to collaborate and define application behavior in plain text scenarios written in a language called Gherkin. These scenarios are written in a human-readable format and can be understood by non-technical stakeholders.

Here's a basic rundown of how Cucumber works:

1.Feature Files: Scenarios are written in feature files using Gherkin syntax. Each scenario outlines a particular behavior or feature of the application.

2.Step Definitions: Step definitions are the actual code implementations of the steps outlined in the feature files. They map each step in the feature files to executable code.

3.Glue Code: Glue code connects the steps defined in feature files to the corresponding step definitions.

4.Test Execution: Cucumber reads the feature files, matches each step to its definition, and executes the associated code. It reports the results in a readable format, often generating detailed reports for easy analysis.

5.Integration: Cucumber can be integrated with various testing frameworks and programming languages, such as Java, Ruby, Python, etc.

I have used Cucumber in project, I'm familiar with its concepts and how it's commonly used in software development. If you have any specific questions about Cucumber or its usage, feel free to ask!

**2.What is the difference between feature file and step definition file?**

In Cucumber, a feature file and a step definition file serve different purposes in the context of Behavior-Driven Development (BDD):

Feature File:

1.A feature file is where you define the behavior of your application in a human-readable format.

It's written using Gherkin syntax, which includes keywords like Feature, Scenario, Given, When, Then, etc.

Feature files describe the behavior of a particular feature or aspect of your application from the perspective of an end user or stakeholder.

Each scenario in a feature file represents a specific test case or example of how the feature should behave under certain conditions.

Feature files are typically created collaboratively by developers, testers, and business stakeholders to ensure clarity and alignment on requirements.

2.Step Definition File:

A step definition file contains the actual code implementation for the steps defined in the feature file.

It's written in the programming language of choice (e.g., Java, Ruby, Python) and defines the logic to execute when each step in the feature file is encountered during test execution.

Step definition files provide the automation behind the scenarios described in the feature files.

Each step in the feature file is mapped to a corresponding step definition in the step definition file.

Step definition files are where you interact with your application, perform actions, and make assertions to validate the behavior described in the feature files.

In summary, the feature file defines the behavior of your application in a human-readable format using Gherkin syntax, while the step definition file contains the code implementations that execute the steps described in the feature file. Together, they form the basis of Behavior-Driven Development in Cucumber.

**3.Cucumber framework Annotations.**

Here are some common annotations/hooks used in Cucumber:

1.Before: This hook runs before each scenario. It's typically used for setup tasks such as initializing resources, opening a browser, etc.

java

Example:

@Before

public void setUp() {

// Setup tasks

}

2.After: This hook runs after each scenario. It's commonly used for teardown tasks such as closing resources, quitting the browser, etc.

java

Example:

@After

public void tearDown() {

// Teardown tasks

}

3.BeforeStep: This hook runs before each step within a scenario. It's used for actions that need to be performed before each step.

java

Example:

@BeforeStep

public void beforeStep() {

// Actions before each step

}

4.AfterStep: This hook runs after each step within a scenario. It's used for actions that need to be performed after each step.

java

Example:

@AfterStep

public void afterStep() {

// Actions after each step

}

These annotations/hooks are typically used in conjunction with step definition methods to provide setup, teardown, and additional behavior during the execution of Cucumber scenarios. They allow for greater control and customization of the testing process.

**4.Explain keywords in cucumber.**

In Cucumber, keywords are predefined terms used within feature files written in Gherkin syntax to define the structure and behavior of the software being tested. These keywords provide a standardized way to describe scenarios and steps in a human-readable format. Here are the main keywords in Gherkin:

1.Feature: It represents a high-level description of a software feature or functionality being tested. Each feature typically contains one or more scenarios.

2.Scenario: It describes a specific test scenario or example that illustrates a particular behavior of the software being tested. Each scenario consists of a series of steps.

'

3.Scenario Outline: It allows for parameterization of scenarios by using placeholders (e.g., '<username>', '<password>') that are replaced with actual values during scenario execution. It's particularly useful for testing multiple variations of the same scenario.

4.Given, When, Then, And, But: These are step keywords used to describe the actions or conditions in each step of a scenario. They help to structure the scenario and make it more readable.

Given: It represents the initial context or setup for the scenario.

When: It represents the action or event that triggers a behavior in the system.

Then: It represents the expected outcome or result of the scenario.

And, But: These keywords are used to add additional steps to the scenario and improve readability.

**5.Explain about hooks and tags / What is hooks? what is the purpose for this? \ What are tags? \ How to group and execute testcases?**

Hooks and tags are two important features in Cucumber that help in organizing and controlling the execution of test cases.

Hooks:

Hooks are special methods in Cucumber that allow you to perform actions at various points in the test execution lifecycle, such as before or after scenarios. Hooks are defined in the step definition files and are annotated with @Before, @After, @BeforeStep, or @AfterStep. Here's a brief explanation of each:

1.Before: This hook runs before each scenario and is typically used for setup tasks such as initializing resources, opening a browser, etc.

2.After: This hook runs after each scenario and is commonly used for teardown tasks such as closing resources, quitting the browser, etc.

3.BeforeStep: This hook runs before each step within a scenario and is used for actions that need to be performed before each step.

4.AfterStep: This hook runs after each step within a scenario and is used for actions that need to be performed after each step.

Hooks are useful for setting up the environment before running tests and cleaning up afterward. They can also be used for logging, reporting, or any other custom actions needed during the test execution lifecycle.

Tags:

Tags are labels that you can assign to scenarios and features in Cucumber to categorize them or group them together. Tags start with the @ symbol and can be added anywhere in the feature file, such as before the Feature or Scenario keyword. Tags can then be used to selectively execute certain scenarios or features during test execution.

For example, you can tag scenarios related to login functionality with @login, scenarios related to registration with @registration, and so on.

gherkin

Example:

@login

Scenario: Successful Login

Given I am on the login page

When I enter valid credentials

And I click the login button

Then I should be redirected to the dashboard

gherkin

Example:

@registration

Scenario: User Registration

Given I am on the registration page

When I fill in the registration form

And I submit the form

Then I should receive a confirmation email

Grouping and Executing Test Cases:

To group and execute test cases based on tags, you can use the Cucumber command-line interface (CLI) and specify the tags you want to include or exclude. For example:

Example

cucumber --tags @login

This command will execute only the scenarios tagged with @login. Similarly, you can exclude certain tags:

Example:

cucumber --tags "not @registration"

This command will execute all scenarios except those tagged with '@registration'.

By using hooks and tags effectively, you can control the test execution flow, manage test setup and teardown tasks, and organize test cases into logical groups for better maintainability and flexibility.

**6.Difference between scenario and scenario outline \ How to execute same testcase for 100 times with different data?**

The difference between a scenario and a scenario outline lies in their usage and flexibility, particularly when it comes to parameterization and reusability.

Scenario:

A scenario in Cucumber represents a single test case or example.

It is written out explicitly with fixed values for each step.

Scenarios are useful when you have specific test cases that don't require variation or parameterization.

Example of a Scenario:

Scenario: Login with Valid Credentials

Given I am on the login page

When I enter valid username and password

And I click the login button

Then I should be redirected to the dashboard

Scenario Outline:

A scenario outline in Cucumber is used for parameterization, allowing the same scenario to be executed multiple times with different sets of data.

It uses placeholders, typically written between < and >, which are replaced with actual values during test execution.

The scenario outline is accompanied by an examples table that provides different sets of data for each execution of the scenario.

Example of a Scenario Outline:

Scenario Outline: Login with Different Credentials

Given I am on the login page

When I enter "<username>" and "<password>"

And I click the login button

Then I should see "<expected\_message>"

Examples:

| username | password | expected\_message |

| user1 | pass123 | Welcome, user1! |

| user2 | pass456 | Incorrect username or pass |

Executing the Same Test Case Multiple Times with Different Data:

To execute the same test case for 100 times with different data, you can use a scenario outline along with an examples table. However, if you want to execute the scenario multiple times with different data dynamically, you might need to use a data-driven approach.

Here's a general approach using a scenario outline for executing the same test case multiple times with different data:

Example:

Scenario Outline: Login with Different Credentials

Given I am on the login page

When I enter "<username>" and "<password>"

And I click the login button

Then I should see "<expected\_message>"

Examples:

| username | password | expected\_message |

| <username\_1> | <password\_1> | <expected\_message\_1> |

| <username\_2> | <password\_2> | <expected\_message\_2> |

| ... | ... | ... |

| <username\_100> | <password\_100> | <expected\_message\_100> |

In this approach, you would need to generate or provide the data for 100 different sets of usernames, passwords, and expected messages in the examples table. Each row in the examples table represents one execution of the scenario with a specific set of data.

Alternatively, if you're working with dynamic data sources such as databases or external files, you might need to implement a data-driven testing approach where you fetch data dynamically during test execution.

**7.What are three major components in cucumber?**

The three major components in Cucumber are:

1.Feature Files: Feature files are written in Gherkin syntax and serve as the specification for the behavior of the software being tested. They contain scenarios that describe various aspects of the application's functionality from an end-user perspective. Feature files are typically created collaboratively by developers, testers, and business stakeholders to ensure clarity and alignment on requirements.

2.Step Definitions: Step definitions are the glue that connects the scenarios described in feature files to the actual implementation code. They define the automation logic for each step in the feature files. Step definitions are written in the programming language of choice (e.g., Java, Ruby, Python) and typically use regular expressions to match steps in feature files with corresponding code implementations.

3.Test Runner: The test runner is responsible for executing the tests defined in feature files using the step definitions. It reads the feature files, matches each step to its corresponding step definition, and executes the associated code. The test runner may also generate reports and provide feedback on the test results. Cucumber supports integration with various test runners and frameworks, allowing you to execute tests in different environments and configurations.

These components work together to facilitate Behavior-Driven Development (BDD) in Cucumber, where requirements are expressed in a human-readable format (feature files), automated tests are written to validate those requirements (step definitions), and tests are executed to verify the behavior of the software.

**8.How do write a step function?**

To write a step definition function, you typically follow these steps:

1.Identify the Step: Determine which step in your feature file needs a corresponding step definition. Steps typically start with keywords like `Given`, `When`, `Then`, `And`, or `But`.

2.Write the Step Definition: Write a method in your step definition file that corresponds to the step in the feature file. This method should be annotated with the appropriate Cucumber annotation (`@Given`, `@When`, `@Then`, `@And`, or `@But`).

3.Implement the Step Logic: Implement the logic for the step inside the method. This logic will depend on what action or verification the step represents. You can interact with your application, perform actions, and make assertions within the step definition method.

4.Parameterize if Necessary: If the step in your feature file contains placeholders (e.g., `<username>`, `<password>`), you can parameterize your step definition method to accept values dynamically.

Here's an example of how to write a step definition function in Java:

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

public class MyStepDefinitions {

@Given("^I am on the login page$")

public void goToLoginPage() {

// Implementation to navigate to the login page

}

@When("^I enter valid credentials$")

public void enterValidCredentials() {

// Implementation to enter valid credentials

}

@When("^I click the login button$")

public void clickLoginButton() {

// Implementation to click the login button

}

@Then("^I should be redirected to the dashboard$")

public void verifyDashboardRedirect() {

// Implementation to verify redirection to the dashboard

}

}

In this example, we have step definition methods annotated with `@Given`, `@When`, and `@Then` to correspond to the steps in the feature file. Each method contains the implementation logic for the step it represents.

Remember that the step definition method names must match the text in the feature file exactly, and you can use regular expressions to match dynamic text or placeholders.

**9.Write the feature file for login functionality (exact syntax)?**

Here's an example of a feature file for login functionality written in Gherkin syntax:

Feature: Login Functionality

As a user

I want to be able to login to my account

So that I can access my personalized content

Scenario: Successful Login

Given I am on the login page

When I enter valid credentials

And I click the login button

Then I should be redirected to the dashboard

Scenario Outline: Invalid Login Attempts

Given I am on the login page

When I enter invalid credentials "<username>" and "<password>"

And I click the login button

Then I should see an error message "<error\_message>"

Examples:

| username | password | error\_message |

| user1 | wrong123 | Incorrect username or pass|

| user2 | pass456 | Incorrect username or pass|

| | pass456 | Please enter username |

| user3 | | Please enter password |

In this feature file:

- The `Feature` keyword is used to define the overall feature being tested.

- A description of the feature is provided after the `Feature` keyword.

- Two scenarios are defined:

- The first scenario, "Successful Login", describes a successful login attempt.

- The second scenario outline, "Invalid Login Attempts", describes multiple attempts with invalid credentials. It uses a scenario outline to parameterize the test with different sets of data.

- Steps are written using keywords like `Given`, `When`, `Then`, and `And` to describe the behavior of the system under test.

- The scenario outline includes an examples table with different sets of data for the invalid login attempts.

This feature file describes the behavior of the login functionality from the perspective of a user, including both successful and unsuccessful login attempts.

**10.How to achieve data driven testing in Cucumber?**

Data-driven testing in Cucumber can be achieved using scenario outlines combined with examples tables or by integrating with external data sources such as spreadsheets, databases, or CSV files. Here's how you can implement data-driven testing in Cucumber:

1. Using Scenario Outlines with Examples Tables:

- Define a scenario outline in your feature file with placeholders for dynamic data.

- Use an examples table to provide multiple sets of data for the scenario outline.

- Each row in the examples table represents a separate test iteration with different data.

Example:

Scenario Outline: Login with Different Credentials

Given I am on the login page

When I enter "<username>" and "<password>"

And I click the login button

Then I should see "<expected\_message>"

Examples:

| username | password | expected\_message |

| user1 | pass123 | Welcome, user1! |

| user2 | pass456 | Incorrect username or pass |

2. Using External Data Sources:

- Integrate Cucumber with external data sources such as spreadsheets, databases, or CSV files.

- Write custom code to read data from the external source and pass it to your step definitions.

- Loop through the data and execute the scenario for each set of data.

Example (reading data from a CSV file):

import cucumber.api.java.en.When;

import java.io.FileReader;

import java.io.IOException;

import java.util.List;

import com.opencsv.CSVReader;

public class MyStepDefinitions {

@When("^I enter credentials from CSV file$")

public void enterCredentialsFromCSV() throws IOException {

String csvFile = "credentials.csv";

CSVReader reader = new CSVReader(new FileReader(csvFile));

List<String[]> data = reader.readAll();

reader.close();

for (String[] row : data) {

String username = row[0];

String password = row[1];

// Enter username and password

}

}

}

3.Using Cucumber Data Tables:

- Use Cucumber data tables to pass structured data directly within the feature file.

- Parse the data table in your step definition and use it to perform data-driven testing.

Example:

Scenario: Register with Multiple Users

Given I am on the registration page

When I register with the following users:

| Name | Email | Password |

| User1 | user1@example.com | pass123 |

| User2 | user2@example.com | pass456 |

Then I should receive a confirmation email

import cucumber.api.DataTable;

import cucumber.api.java.en.When;

import java.util.List;

public class MyStepDefinitions {

@When("^I register with the following users:$")

public void registerWithMultipleUsers(DataTable userData) {

List<List<String>> data = userData.asLists(String.class);

for (List<String> row : data) {

String name = row.get(0);

String email = row.get(1);

String password = row.get(2);

// Register user with name, email, and password

}

}

}

These are some ways to achieve data-driven testing in Cucumber. Choose the method that best fits your requirements and integrates seamlessly with your testing environment.

**TestNg**

**1.What is TestNG? What are the advantages and features of TestNG?**

A)TestNG, or Test Next Generation, is a testing framework for Java that is inspired by JUnit and NUnit. It is designed to cover a wide range of test categories: unit, functional, end-to-end, integration, etc. TestNG provides more flexibility and power compared to its predecessors and has become one of the most widely used testing frameworks in the Java ecosystem.

Advantages of TestNG:

Annotations: TestNG uses annotations to define test methods, which makes it easy to organize and customize test cases. Annotations such as @Test, @BeforeMethod, @AfterMethod, @BeforeClass, @AfterClass, etc., provide control over test execution flow.

Parameterization: TestNG supports parameterized tests, allowing you to run the same test method with different sets of data. This reduces code duplication and makes tests more maintainable.

Dependent Test Execution: TestNG allows you to specify dependencies between test methods, ensuring that certain tests run only after others have passed. This helps in maintaining the test execution order and handling complex test scenarios.

Grouping: TestNG allows you to categorize tests into groups, making it easy to run subsets of tests based on categories like smoke tests, regression tests, etc. This enables selective test execution and facilitates better organization of test suites.

Parallel Execution: TestNG supports parallel execution of tests across multiple threads or processes, which can significantly reduce the overall test execution time, especially in large test suites.

Reporting: TestNG generates detailed HTML reports that provide insights into test execution results, including passed, failed, and skipped tests, along with stack traces for failed tests. These reports help in identifying issues quickly and debugging failures effectively.

Listeners: TestNG allows the implementation of custom listeners to hook into various test lifecycle events, such as test start, test success, test failure, etc. This feature can be used for custom reporting, logging, or other automation tasks.

Data Providers: TestNG supports the use of data providers, which are methods that supply data to test methods. This is particularly useful for data-driven testing, where the same test logic is applied with different input data sets.

Integration with IDEs and Build Tools: TestNG integrates seamlessly with popular IDEs like Eclipse and IntelliJ IDEA, as well as build tools like Maven and Gradle, making it easy to incorporate testing into the development workflow.

**2.How to provide data in TestNG?**

A)In TestNG, you can provide data to test methods using data providers. Data providers are methods that supply data to test methods, allowing you to run the same test logic with different sets of input data. Here's how you can provide data in TestNG:

Create a Data Provider Method: First, define a method that will act as the data provider. This method should return a two-dimensional array (Object[][]), where each row represents a set of input parameters for a test method.

java

Copy code

@DataProvider(name = "testdata")

public Object[][] provideTestData() {

return new Object[][] {

{"username1", "password1"},

{"username2", "password2"},

// Add more sets of data as needed

};

}

Use the Data Provider in Test Method: Annotate the test method with @Test and specify the data provider's name using the dataProvider attribute.

java

Copy code

@Test(dataProvider = "testdata")

public void loginTest(String username, String password) {

// Test logic using provided username and password

System.out.println("Username: " + username + ", Password: " + password);

}

Access Data in Test Method: In the test method, you can access the provided data as method parameters. TestNG will automatically inject the data provided by the data provider into the corresponding test method parameters.

Run TestNG Suite: When running your TestNG suite, TestNG will execute the test method for each set of data provided by the data provider.

Here's a complete example:

java

Copy code

import org.testng.annotations.DataProvider;

import org.testng.annotations.Test;

public class LoginTest {

@DataProvider(name = "testdata")

public Object[][] provideTestData() {

return new Object[][] {

{"username1", "password1"},

{"username2", "password2"}

};

}

@Test(dataProvider = "testdata")

public void loginTest(String username, String password) {

// Test logic using provided username and password

System.out.println("Username: " + username + ", Password: " + password);

}

}

In this example, the loginTest method will be executed twice, once for each set of data provided by the testdata data provider.

**3.How to customize execution without changing the execution order of scripts?**

A)To customize the execution of TestNG tests without changing the execution order of scripts, you can utilize TestNG's features such as groups, dependencies, and priority.

Here's how you can customize the execution:

Using Groups: Group your test methods based on their functionality or purpose using the groups attribute in the @Test annotation.

java

Copy code

@Test(groups = "smoke")

public void smokeTest1() {

// Test logic

}

@Test(groups = "regression")

public void regressionTest1() {

// Test logic

}

Then, when running tests, you can specify which groups to include or exclude using the TestNG XML file or programmatically.

Using Dependencies: Specify dependencies between test methods using the dependsOnMethods attribute in the @Test annotation. This ensures that a test method only runs after the specified methods have passed.

java

Copy code

@Test

public void setup() {

// Setup logic

}

@Test(dependsOnMethods = "setup")

public void test1() {

// Test logic

}

Using Priority: Set priorities for test methods using the priority attribute in the @Test annotation. TestNG executes test methods in ascending order of priority.

java

Copy code

@Test(priority = 1)

public void test1() {

// Test logic

}

@Test(priority = 2)

public void test2() {

// Test logic

}

By utilizing these features, you can customize the execution of your TestNG tests without altering their execution order. This allows you to run specific groups of tests, control test dependencies, and prioritize test execution based on your requirements.

**4.How to execute test cases in parallel?**

A)To execute TestNG test cases in parallel, you can leverage TestNG's built-in support for parallel execution. TestNG provides several options for parallel execution, including parallel execution at the test level, method level, or even suite level. Here's how you can execute test cases in parallel:

Parallel Execution at Test Level:Specify parallel execution at the test level by setting the parallel attribute in the <test> tag of your TestNG XML file to values such as tests, classes, or methods. Additionally, you can specify the number of threads using the thread-count attribute.

xml

Copy code

<suite name="TestSuite" parallel="tests" thread-count="2">

<test name="Test1">

<!-- Test1 configurations and classes -->

</test>

<test name="Test2">

<!-- Test2 configurations and classes -->

</test>

</suite>

Parallel Execution at Method Level:Annotate your test class or test methods with @Test and specify the parallel execution mode using the @Test annotation's parallel attribute.

java

Copy code

import org.testng.annotations.Test;

@Test(parallel = true)

public class ParallelExecutionTest {

// Test methods

}

Parallel Execution Using Data Providers:If you're using data providers, you can run test methods in parallel for each set of data by specifying parallel execution at the method level and enabling parallel execution for the data provider.

java

Copy code

import org.testng.annotations.Test;

import org.testng.annotations.DataProvider;

public class DataProviderTest {

@DataProvider(name = "testData", parallel = true)

public Object[][] provideTestData() {

// Data provider logic

}

@Test(dataProvider = "testData")

public void testMethod(String data) {

// Test method logic

}

}

Parallel Execution Using Annotations:If you want to execute specific groups of tests in parallel, you can annotate test methods or classes with @Test and set the parallel attribute accordingly.

java

Copy code

import org.testng.annotations.Test;

@Test(groups = "smoke", parallel = true)

public class SmokeTest {

// Smoke test methods

}

By utilizing these methods, you can execute TestNG test cases in parallel, which can significantly reduce the overall test execution time, especially for large test suites. Choose the appropriate parallel execution strategy based on your requirements and the structure of your test suite.

**5.How did I handle failed execution?**

A)To handle failed execution in TestNG, you can implement various mechanisms to capture and manage test failures effectively. Here's how you can handle failed execution:

Assertions: Use assertions within your test methods to validate expected outcomes. TestNG provides assertion methods such as assertEquals, assertTrue, assertFalse, etc. These assertions compare actual results with expected values and automatically mark the test as failed if the assertion condition is not met.

java

Copy code

@Test

public void testMethod() {

int result = performOperation();

Assert.assertEquals(result, expectedValue);

}

Exception Handling: Surround your test logic with try-catch blocks to handle exceptions gracefully. In the catch block, you can log the exception details and mark the test as failed using TestNG's Assert.fail() method or by throwing a custom exception.

java

Copy code

@Test

public void testMethod() {

try {

// Test logic that may throw an exception

} catch (Exception e) {

// Log exception details

Assert.fail("Test failed due to exception: " + e.getMessage());

}

}

Retry Mechanism: Implement a retry mechanism to rerun failed tests automatically. TestNG allows you to specify the maximum number of retry attempts for failed tests using the retryAnalyzer attribute in the @Test annotation. You can create a custom retry analyzer by implementing the IRetryAnalyzer interface.

java

Copy code

@Test(retryAnalyzer = RetryAnalyzer.class)

public void testMethod() {

// Test logic

}

java

Copy code

import org.testng.IRetryAnalyzer;

import org.testng.ITestResult;

public class RetryAnalyzer implements IRetryAnalyzer {

private int retryCount = 0;

private static final int maxRetryCount = 3;

@Override

public boolean retry(ITestResult result) {

if (retryCount < maxRetryCount) {

retryCount++;

return true;

}

return false;

}

}

TestNG Listeners: Implement TestNG listeners to perform custom actions upon test failure. You can create a listener class by implementing interfaces like ITestListener or IInvokedMethodListener and override methods such as onTestFailure to handle failed test cases.

java

Copy code

import org.testng.ITestResult;

import org.testng.TestListenerAdapter;

public class CustomTestListener extends TestListenerAdapter {

@Override

public void onTestFailure(ITestResult tr) {

// Custom actions to be performed on test failure

}

}

xml

Copy code

<suite name="TestSuite">

<listeners>

<listener class-name="path.to.CustomTestListener"/>

</listeners>

<!-- Test configurations and classes -->

</suite>

By employing these strategies, you can effectively handle failed execution in TestNG, ensuring robust test automation and improved test reliability.

**6.How to execute test cases in local while also performing the other QA activities?**

A)To execute test cases locally while also performing other QA activities, you can follow a few strategies to manage your time effectively and ensure productivity. Here are some tips:

Prioritize Tasks: Identify the most critical QA activities that need immediate attention, such as writing test cases, reviewing requirements, or analyzing defects. Prioritize these tasks based on their urgency and impact on the project.

Time Management: Allocate specific time slots for different QA activities, including test execution. Schedule dedicated time for executing test cases locally, ensuring that it doesn't conflict with other tasks. Use techniques like time blocking to organize your day effectively.

Use Automation: Automate repetitive test cases or regression suites to reduce the time spent on manual execution. Automation allows you to run tests in the background while focusing on other QA activities. Tools like Selenium WebDriver, Appium, or TestNG can help automate test execution.

Parallel Execution: If possible, leverage parallel execution to speed up the test execution process. Run tests in parallel on multiple machines or environments to maximize efficiency. TestNG and other testing frameworks offer built-in support for parallel execution.

Continuous Integration/Continuous Deployment (CI/CD): Integrate your testing process with CI/CD pipelines to automate the execution of tests whenever there's a code change. This ensures that tests are run automatically without manual intervention, allowing you to focus on other QA tasks.

Collaboration Tools: Use collaboration tools like Jira, Trello, or Asana to manage QA tasks, track progress, and communicate with team members. Collaborative platforms facilitate efficient task assignment, progress monitoring, and communication within the QA team.

Test Management Tools: Utilize test management tools like TestRail, Zephyr, or Xray to organize test cases, manage test execution, and track test results. These tools provide a centralized repository for test assets and offer features for scheduling and executing test runs.

Time Boxing: Allocate specific time periods for executing test cases locally, such as dedicated testing sessions during the day. Set aside uninterrupted time for testing to maintain focus and productivity.

Breaks and Rest: Take regular breaks between testing sessions to avoid burnout and maintain mental freshness. Short breaks can help rejuvenate your focus and improve productivity during testing.

**7.Write the script for dataprovider and priority \ How to execute same testcase for 100 times with different data?**

A)

Certainly! Below is an example script in Java using TestNG for parameterized testing with data provider and priority. This script demonstrates how to execute the same test case multiple times with different data using a data provider.

java

Copy code

import org.testng.annotations.Test;

import org.testng.annotations.DataProvider;

public class ParameterizedTest {

@DataProvider(name = "testData")

public Object[][] provideTestData() {

// Define test data here

return new Object[][] {

{"data1"},

{"data2"},

{"data3"},

// Add more data sets as needed

};

}

@Test(dataProvider = "testData", priority = 1)

public void testCase(String testData) {

// Test case logic using provided data

System.out.println("Test case with data: " + testData);

}

}

In this script:

We have a class named ParameterizedTest.

The @DataProvider annotation is used to specify a method provideTestData() that supplies test data.

Inside the provideTestData() method, we define the test data as a two-dimensional array of objects. Each row represents a set of data for one execution of the test case.

The @Test annotation is applied to the testCase() method, which is the test case we want to execute multiple times with different data.

We specify the data provider testData for the test method using the dataProvider attribute.

The priority attribute is set to prioritize the test execution order. In this case, it's set to 1, but you can adjust priorities based on your requirements.

To execute the testCase() method multiple times with different data, TestNG will automatically invoke it for each set of data provided by the data provider testData.

You can adjust the test data in the provideTestData() method to include as many sets of data as needed. TestNG will execute the testCase() method for each set of data, ensuring thorough testing with various input values.

**8.How to execute same testcase for 100 times without test data?**

A)

If you want to execute the same test case 100 times without using test data, you can achieve this by using a loop within your test method. Here's an example of how to execute the same test case 100 times:

java

Copy code

import org.testng.annotations.Test;

public class RepeatedTest {

@Test(priority = 1)

public void repeatedTestCase() {

for (int i = 1; i <= 100; i++) {

// Test case logic

System.out.println("Test iteration: " + i);

// Add your test logic here

}

}

}

In this script:

We have a class named RepeatedTest.

The @Test annotation is applied to the repeatedTestCase() method, which is the test case we want to execute multiple times.

Inside the repeatedTestCase() method, we use a loop to repeat the test case logic 100 times. Adjust the loop condition (i <= 100) as per your requirement.

Within the loop, you can include your test case logic. In this example, we've added a simple print statement to indicate the test iteration.

When you run this test class, TestNG will execute the repeatedTestCase() method 100 times, resulting in the test case being executed repeatedly without using test data.

**9.What are the testNg annotations and how you implemented them in your framework ?**

A)TestNG provides a variety of annotations that allow you to control the behavior of your tests, setup, and teardown methods. Here are some of the commonly used TestNG annotations and how you might implement them in a testing framework:

@Test: Marks a method as a test method.

java

Copy code

import org.testng.annotations.Test;

public class MyTest {

@Test

public void myTestMethod() {

// Test logic here

}

}

@BeforeMethod: Executes before each test method.

java

Copy code

import org.testng.annotations.BeforeMethod;

public class MyTest {

@BeforeMethod

public void setUp() {

// Setup logic here

}

@Test

public void myTestMethod() {

// Test logic here

}

}

@AfterMethod: Executes after each test method.

java

Copy code

import org.testng.annotations.AfterMethod;

public class MyTest {

@AfterMethod

public void tearDown() {

// Teardown logic here

}

@Test

public void myTestMethod() {

// Test logic here

}

}

@BeforeClass: Executes once before any test method in the class.

java

Copy code

import org.testng.annotations.BeforeClass;

public class MyTest {

@BeforeClass

public void setUpClass() {

// Class-level setup logic here

}

@Test

public void myTestMethod() {

// Test logic here

}

}

@AfterClass: Executes once after all test methods in the class.

java

Copy code

import org.testng.annotations.AfterClass;

public class MyTest {

@AfterClass

public void tearDownClass() {

// Class-level teardown logic here

}

@Test

public void myTestMethod() {

// Test logic here

}

}

@DataProvider: Provides test data to the test method.

java

Copy code

import org.testng.annotations.DataProvider;

public class MyTest {

@DataProvider(name = "testData")

public Object[][] provideTestData() {

// Provide test data here

}

@Test(dataProvider = "testData")

public void myTestMethod(String data) {

// Test logic here

}

}

**10.Advantages of testng framework?**

A)The TestNG framework offers several advantages for automated testing in Java, making it a popular choice among developers and QA professionals. Here are some of the key advantages of using TestNG:

Annotations: TestNG provides a wide range of annotations that allow you to define the behavior of your test methods, such as @Test, @BeforeMethod, @AfterMethod, @BeforeClass, @AfterClass, etc. These annotations offer flexibility in organizing and customizing test cases.

Parameterization: TestNG supports parameterized tests, allowing you to run the same test method with different sets of input data. This reduces code duplication and makes tests more maintainable.

Dependent Test Execution: TestNG allows you to specify dependencies between test methods, ensuring that certain tests run only after others have passed. This helps in maintaining the test execution order and handling complex test scenarios.

Grouping: TestNG allows you to categorize tests into groups, making it easy to run subsets of tests based on categories like smoke tests, regression tests, etc. This enables selective test execution and facilitates better organization of test suites.

Parallel Execution: TestNG supports parallel execution of tests across multiple threads or processes, which can significantly reduce the overall test execution time, especially in large test suites.

Reporting: TestNG generates detailed HTML reports that provide insights into test execution results, including passed, failed, and skipped tests, along with stack traces for failed tests. These reports help in identifying issues quickly and debugging failures effectively.

Listeners: TestNG allows the implementation of custom listeners to hook into various test lifecycle events, such as test start, test success, test failure, etc. This feature can be used for custom reporting, logging, or other automation tasks.

Data Providers: TestNG supports the use of data providers, which are methods that supply data to test methods. This is particularly useful for data-driven testing, where the same test logic is applied with different input data sets.

Integration with IDEs and Build Tools: TestNG integrates seamlessly with popular IDEs like Eclipse and IntelliJ IDEA, as well as build tools like Maven and Gradle, making it easy to incorporate testing into the development workflow.

Support for Dependency Injection: TestNG supports dependency injection using the @Parameters annotation, allowing you to inject parameters into test methods or constructors. This promotes modularity and reusability in test code.

Overall, TestNG offers a comprehensive testing framework with features that cater to a wide range of testing requirements, from basic unit testing to complex end-to-end testing scenarios. Its versatility, ease of use, and extensive documentation make it a preferred choice for Java developers and QA teams alike.

**11.What is priority in TestNg?**

A)In TestNG, the priority attribute is used to specify the priority or order in which test methods should be executed within a test class. By default, TestNG does not enforce any particular execution order for test methods within a class. However, you can use the priority attribute to assign a priority to each test method, indicating their relative importance or execution order.

Test methods with lower priority values are executed first, followed by test methods with higher priority values. If multiple test methods have the same priority, TestNG executes them in the order they are defined within the test class.

Here's how you can use the priority attribute in TestNG:

java

Copy code

import org.testng.annotations.Test;

public class MyTest {

@Test(priority = 1)

public void testMethod1() {

// Test method logic

}

@Test(priority = 2)

public void testMethod2() {

// Test method logic

}

@Test(priority = 3)

public void testMethod3() {

// Test method logic

}

}

In this example:

testMethod1 has a priority of 1.

testMethod2 has a priority of 2.

testMethod3 has a priority of 3.

When you run this test class, TestNG executes testMethod1 first, followed by testMethod2, and then testMethod3. If you don't specify the priority attribute for a test method, TestNG assigns a default priority of 0.

The priority attribute is useful for scenarios where you need to control the order of test execution or prioritize certain test methods over others. However, relying too heavily on priorities can make your test suite less maintainable and harder to understand. It's generally recommended to keep test dependencies explicit rather than relying solely on priorities for test execution order.

How to run failed test cases?

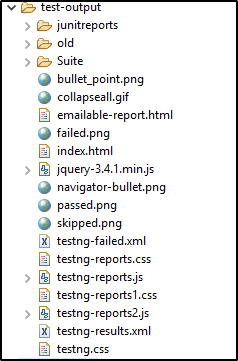
**Ans**: There are two ways to perform a failed test in TestNG:

1. Using the testng-failed.xml file.
2. Using the IRetryAnalyzer interface.

**Explanation:**

1. *Using the testng-failed.xml file*

This test-output folder comprises various files that include failed test cases as well. See below.



This Testng-failed.xml file displays the failed test. (Here in this case test 3 is failed as per xml file)

Go back to Test 3 file, ***change the assert value to true and rerun the testng-failed.xml file.*** This will execute only the failed test case with the correct output. In order to cross-check, execute the original **testng.xml** file and check if all the test cases are working as expected.

*2.Using the IRetryAnalyzer interface.*

* Create a class to implement TestNG IRetryAnalyzer interface:
* By adding Retry analyser during runtime by implementation of the listener interface.
* In Test Method specify Retry Class

Parameterization in TestNg?

Parameterization in TestNG means ***we want to pass a parameter into the test with multiple values***.

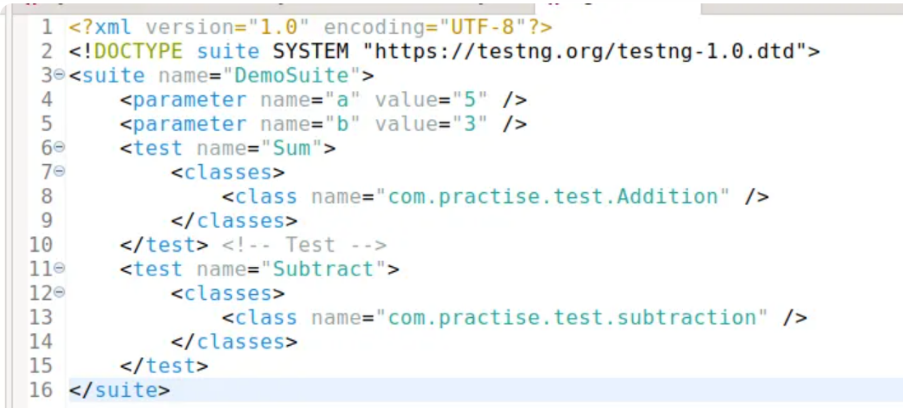
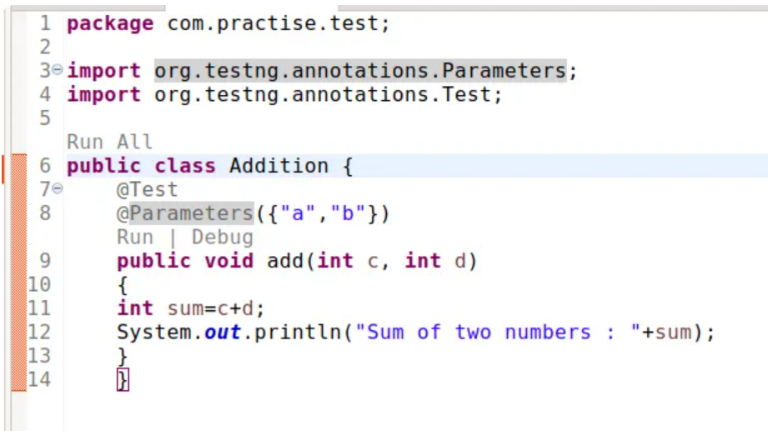
Parameterization is used in testing because it allows you to run the same test with varied parameters numerous times.

So primarily we can achieve parameterization in TestNG in two ways.

1. Using “Parameters annotation” and TestNG XML file.
2. Using “DataProvider annotation”.

**Explanation:**

Example: Parameters annotation

****

Example: DataProvider annotation

Assertion’s concept?

**Assertions in TestNG are a way to verify that the expected result and the actual result matched or not**.

**Syntax for TestNG Assertions:**

Although there are many methods for assertions, the generic syntax is:

**Assert.Method(actual, expected)**

The parameter as you see contains three values:

* **Actual**:  if the tester's assertion is on the title of the page, then what was the actual title of the page goes here.
* **Expected**:  if the tester's assertion is on the title of the page, then what value of title do you expect goes here.

**Assert.Method(actual, expected, message)**

* **Message**: A string message to display only in case of an error when the assert fails.

Different types of Asserts in TestNG.

There are two types of TestNg Assert:

* **Hard Assert**
* **Soft Assert**

Hard Asserts are the type of assets that stop the execution of the program when the condition becomes fail and mark the Test as Failed.

Soft asserts are the asserts which continue the execution even after the Assert condition fails.

What is the approach if you want to execute the test case with multiple sets of data in TestNG?

To execute a test case with multiple sets of data in TestNG, you can use parameterization. Parameterization allows you to run the same test method with different input values or scenarios.

Data provider is one of the way of parameterization.

**Explanation:**

DataProvider is a feature of the TestNG library that allows a developer to run the same suite of test cases with different data sets. It helps the developer to[test](https://coderpad.io/features/) the application with different values. You can use this to test the business logic of any application.

Usage of priority in TestNG framework and which circumstances it is used in project

Priority is an attribute that tells TestNG which order the tests need to follow. When we have multiple test cases and want to execute them in a particular order, the TestNG priority attribute helps in executing the test cases in that order.

**Explanation:**

* The test cases get executed in ascending order of the priority list. Thus, test cases with lower priority get executed first.
* One test method is allowed to have only one test priority in TestNG.

***The syntax for test priority***

**@Test (priority = x)**, where x can be any integer – **negative**, **zero**, or **positive**.

For Example,

@Test (priority = 1)

public void function(){

//test code

} Here, the test method function has a **test priority of 1**.

What is the return type of Dataprovider?

DataProvider returns a 2-dimensional array object.

**Explanation:**

In TestNG, the return type of a DataProvider method must be either **Object[][]** or **Iterator<Object[]>**. These return types allow you to provide multiple sets of data to be used as test parameters for your test methods.

What is the approach to execute a specific group of test cases in TestNG?

To execute a specific group of test cases in TestNG, you can use the concept of **Groups** and include or exclude them from the test suite configuration.

TestNG Groups allow a way to group multiple test cases written in Selenium to be grouped under a common name. Using groups in TestNG you can run multiple test cases which are tagged using a group name together, thereby separating the test cases as per their tagged groups.

**Syntax for Groups:**

@Test(groups ="group\_name")

public void testMethod() {

//Test code here

}

In TestNG, the **<include>** and **<exclude>** tags are used within the **<groups>** tag in the testng.xml file to specify which test groups should be included or excluded from execution. These tags allow you to control which test methods are executed based on their group membership.

How can we prioritize the test cases in TestNG?

In TestNG, you can prioritize, test cases using the **priority** attribute of the **@Test** annotation. This allows you to control the order in which test methods are executed.

* **Negative Priority** can be assigned to any test method when you want to have higher precedence over the test methods with default priority.
* The test cases get executed in ascending order of the priority list. Lower the priority number; higher is the priority of the test case method.
* One method is allowed to have only one priority in TestNG.
* **if two or more methods have the same priorities in TestNG, then their running test sequence is alphabetic**.
* **the test methods with no priority assigned have a default priority equal to 0. It means if we define no priorities, all the test methods will be assigned priority 0, and a similar priority case will apply**.

If we provide priority values as -1, 0 & 1 then what is the sequence of the test cases execution?

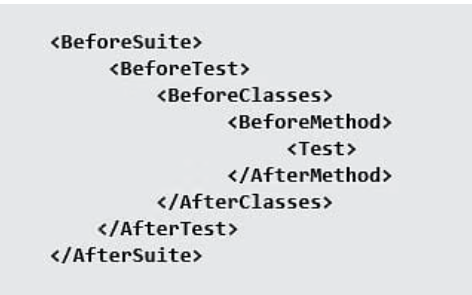
In TestNG, when priority values are provided as -1, 0, and 1, the sequence of test case execution follows this order:

1. Test methods with the lowest priority value (-1) are executed first.
2. Test methods with priority value 0 are executed next.
3. Test methods with the highest priority value (1) are executed last.

Write some TestNG annotations \ Tell the sequence of testng annotations.

The following are TestNG annotations, with explanations

* @BeforeSuite
* @BeforeTest
* @BeforeClass
* @BeforeMethod
* @Test
* @AfterMethod
* @AfterClass
* @AfterTest
* @AfterSuite

The Workflow/ Sequence of TestNG:

How to group and execute testcases?

To group tests in the source code, you have to use the @groups attribute of the @Test annotation. TestNG provides the option to structure the test such that the entire test class belongs to a particular TestNG group or a couple of methods belong to the TestNG group.

Groups in TestNG are specified in testng.xml under the <suite> or <test> tag. Groups under the <suite> tag apply to all the tests included under the <test> tag in that particular <suite>.

To group tests in the source code, you have to use the @groups attribute of the @Test annotation.



