Importance of the TestNG Framework

TestNG (Test Next Generation) is a testing framework inspired by JUnit and NUnit but introduces some new functionalities that make it more powerful and easier to use. The importance of TestNG includes:

- Annotations for better control over test cases.
- Ability to run tests in parallel.
- Flexible test configuration.
- Support for parameterized and data-driven testing.
- Detailed and customizable test reports.

Why We Use TestNG in Your Framework

We use TestNG in our framework because it:

- Provides annotations that simplify the structure and readability of test cases.
- Allows for configuration through XML files, which helps manage test suites and test cases efficiently.
- Supports parallel test execution, which reduces test execution time.
- Generates detailed HTML reports by default, which can be further customized.
- Handles dependencies between test methods, making the test suite more robust.

Purpose of TestNG XML

The `testng.xml` file is used to configure and control the execution of tests. It allows us to:

- Define test suites and test cases.
- Specify the classes and methods to be executed.
- Manage the order of test execution.
- Include or exclude specific tests.
- Configure parameters for tests.

Purpose of Listeners in TestNG

Listeners in TestNG are used to modify the default behavior of TestNG. They allow you to:

- Perform actions before and after test methods, test classes, and test suites.
- Capture events like test start, test success, test failure, etc.
- Generate custom reports. Listeners are a concept specific to TestNG and not directly related to Selenium.

Running the Same Method 100 Times in TestNG with the Same Data

You can achieve this by using the `invocationCount` attribute in the `@Test` annotation:

```
@Test(invocationCount = 100)
public void testMethod() {
    // test code
}
```

Reporting Tool in Your Framework

A common reporting tool used in TestNG frameworks is the **ExtentReports**. It provides:

- Detailed and interactive HTML reports.
- Logs of test steps and results.
- Screenshots of failed test cases.
- Integration with CI/CD tools.

Questions in TestNG XML

TestNG XML is used to:

- Define test suites and test cases.
- Set parameters.
- Include/exclude tests.
- Configure listeners and reporters.

Different TestNG Annotations

- `@BeforeSuite`, `@AfterSuite`
- `@BeforeTest`, `@AfterTest`
- `@BeforeClass`, `@AfterClass`
- `@BeforeMethod`, `@AfterMethod`
- `@Test`
- `@DataProvider`
- `@Factory`

Configuring Tests in TestNG

Tests can be configured in TestNG using the `testng.xml` file and annotations within test classes.

What is @DataProvider?

`@DataProvider` is used to pass multiple sets of data to a test method:

```
@DataProvider(name = "dataProviderName")
public Object[][] dataProviderMethod() {
    return new Object[][] { {"data1"}, {"data2"} };
}

@Test(dataProvider = "dataProviderName")
public void testMethod(String data) {
```

```
    // test code
}
```

Difference Between @Factory and @DataProvider

- `@DataProvider` supplies multiple sets of data to a single test method.
- `@Factory` creates instances of test classes, allowing multiple test instances to run with different parameters.

@Factory Real-Time Example

```
public class TestClass {
    private String data;

    public TestClass(String data) {
        this.data = data;
    }

    @Test
    public void testMethod() {
        System.out.println("Data: " + data);
    }
}

public class TestFactory {
    @Factory
    public Object[] createInstances() {
        return new Object[] { new TestClass("data1"), new TestClass("data2")
};
    }
}
```

Test Order in TestNG

The order of test execution in TestNG can be controlled using the `priority` attribute in the `@Test` annotation:

```
@Test(priority = 1)
public void testOne() {
    // test code
}

@Test(priority = 2)
public void testTwo() {
    // test code
}
```

Adding/Removing Test Cases in Testng.xml

To add/remove test cases in `testng.xml`, you can include or exclude methods, classes, or packages:

```
<test name="TestName">
    <classes>
        <class name="com.example.TestClass">
            <methods>
                <include name="testMethod"/>
                <exclude name="anotherTestMethod"/>
            </methods>
        </class>
    </classes>
</test>
```

Difference Between BeforeMethod, BeforeTest, and BeforeClass

- `@BeforeMethod`: Runs before each test method.
- `@BeforeTest`: Runs before any test method in a `<test>` tag.
- `@BeforeClass`: Runs before the first method in the current class.

TestNG Annotation Hierarchy Order

1. `@BeforeSuite`
2. `@BeforeTest`
3. `@BeforeClass`
4. `@BeforeMethod`
5. `@Test`
6. `@AfterMethod`
7. `@AfterClass`
8. `@AfterTest`
9. `@AfterSuite`

Achieving Parallel Execution Using TestNG

Parallel execution can be configured in `testng.xml`:

```
<suite name="Suite" parallel="methods" thread-count="5">
    <test name="Test">
        <classes>
            <class name="com.example.TestClass"/>
        </classes>
    </test>
</suite>
```

Running Only Failed Test Cases

Failed test cases can be rerun using the `testng-failed.xml` file, which is generated after the first run.

Taking Screenshot for Failed Test Cases

You can use a listener (e.g., `ITestListener`) to capture screenshots on test failure:

```
public class ScreenshotListener implements ITestListener {
    public void onTestFailure(ITestResult result) {
        // code to take a screenshot
    }
}
```

Running the Same Tests for 10 Times

Using the `invocationCount` attribute:

```
@Test(invocationCount = 10)
public void testMethod() {
    // test code
}
```

Types of Listeners

- `ITestListener`
- `ISuiteListener`
- `IReporter`
- `IAnnotationTransformer`
- `IInvokedMethodListener`

TestNG: Parallel Executions, Grouping

- **Parallel Executions**: Configured in `testng.xml`.
- **Grouping**: Use the `groups` attribute in the `@Test` annotation.

Difference Between AfterSuite and BeforeSuite

- `@BeforeSuite`: Runs before all tests in the suite.
- `@AfterSuite`: Runs after all tests in the suite.

Use of testng.xml

Used to define and configure test suites, tests, classes, methods, listeners, parameters, and parallel execution settings.

Number of Suites in TestNG

You can have multiple suites in TestNG. Running all suites can be managed in the `testng.xml` file.

Syntax for Parallel Testing in TestNG

```
<suite name="Suite" parallel="methods" thread-count="5">
    <test name="Test">
        <classes>
            <class name="com.example.TestClass"/>
        </classes>
    </test>
</suite>
```

In `parallel="methods"`, you can specify methods, classes, or tests.

Multiple Suites in One XML File

Yes, you can have multiple suites in one XML file. To run all suites, configure them within the `suite` tags.

InvocationCount in TestNG

`invocationCount` is used to run a test method multiple times:

```
@Test(invocationCount = 5)
public void testMethod() {
    // test code
}
```

Cucumber Tags and Annotations

- **Tags**: Used to filter and run specific scenarios or features.
- **Annotations**: Define steps (`@Given`, `@When`, `@Then`, `@And`, `@But`).

Background in Cucumber

`Background` is used to define a common set of steps that should be run before each scenario in a feature file.

Difference Between Scenario and Scenario Outline

- **Scenario**: Runs a single set of steps with fixed data.
- **Scenario Outline**: Runs the same steps multiple times with different sets of data.

Skeleton of Test Runner

```
@RunWith(Cucumber.class)
@CucumberOptions(
    features = "src/test/resources/features",
    glue = {"stepDefinitions"},
    plugin = {"pretty", "html:target/cucumber-reports"}
)
public class TestRunner {
}
```

Retry Analyzer

Retry Analyzer allows re-execution of failed tests:

```
public class RetryAnalyzer implements IRetryAnalyzer {
    private int retryCount = 0;
    private static final int maxRetryCount = 3;

    public boolean retry(ITestResult result) {
        if (retryCount < maxRetryCount) {
            retryCount++;
            return true;
        }
        return false;
    }
}
```

Cucumber Tags and Running Combinations

Tags are used to run specific sets of scenarios. Use `@CucumberOptions` to specify tags:

```
@CucumberOptions(
    features = "src/test/resources/features",
    glue = {"stepDefinitions"},
    tags = "@tag1 and @tag2"
)
```

Difference Between Hooks and Tags

- **Hooks**: Special blocks of code that run before or after each scenario (`@Before`, `@After`).
- **Tags**: Used to filter which scenarios to run.