TestNG – Test Next Generation

TestNG is a popular framework used for Java applications that provide effective and organized execution of test cases.

It has features like parameterising, grouping, parallel executions and reporting.

TestNG is:

* Powerful Testing Framework
* Successor to Junit
* Easy Test Control
* Enhanced Functionality

|  |  |
| --- | --- |
| Junit | TestNG |
| @Before  @Test  @After | @BeforeTest  @Test  @AfterTest |
| Used for Unit Testing | Used for Automation Testing |
| Cannot do reporting | Can do Reporting |
| Cannot do CBT | Can do CBT |

# Annotations in TestNG:

**@BeforeSuite 🡪** This annotation will get executed before the execution of all the test cases inside a TestNG Suite.

**@AfterSuite 🡪** This annotation will get executed after the execution all the test cases inside a TestNG Suite.

**@BeforeTest 🡪** is executed before the execution of all the @test annotated methods inside a TestNG Suite

**@AfterTest 🡪** is executed after the execution of all the @test annotated methods inside a TestNG Suite

**@Test 🡪** Validation steps will be added under this annotation.

**@BeforeMethod 🡪** is executed before each test method within a test class. Suppose there are n test methods within a test class, then n times @BeforeMethod annotated method will be invoked.

**@AfterMethod 🡪** is executed after each test method within a test class. Suppose there are n test methods within a test class, then n times @BeforeMethod annotated method will be invoked.

**@BeforeClass 🡪** is executed before all the methods of the current class start their execution.

**@AfterClass 🡪** is executed after all the methods of the current class finish their execution.

# Understanding the testNG.xml file

* testng.xml file is a configuration file in TestNG.
* It is used to define test suites and tests. It is also used to pass Parameters to the test methods.
* It defines how your tests should be run.
* The elements that are added to the file in the form of xml tags have a pre-defined hierarchy that should not be altered.
* The main elements we use in this file are: suite, test, classes

Example:

**<suite** name="RegressionSuite"**>**

**<test** name="Functionality Test"**>**

**<classes>**

**<class** name="com.example.tests.LoginTest"**/>**

**<class** name="com.example.tests.SearchTest"**>**

**</ class>**

**</classes>**

**</test>**

**<test** name="UI Test"**>**

**<classes>**

**<class** name="com.example.tests.ScrollTest"**/>**

**<class** name="com.example.tests.NightModeTest" **/>**

**</classes>**

**</test>**

**</suite>**

* **Test Suite:** This is the root tag of the xml tree. Each file can have only one **Suite** tag.
* **Test Tag:**This tag is defined inside the suite tag and there can be many test tags inside the suite tags. For example: <test name=”UI Test”></test>, <test name=”Functionality Test”></test> etc.
* **Test Classes**: This tag is inside of the test tag. Again, you can have only one of the <classes /> tag defining the classes you want to have in your test.

# Running Multiple Tests

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">

<suite name=*"Suite"*>

<test thread-count=*"5"* name=*"Test"*>

<classes>

<class name=*"demo.TestNGDemo"*/>

<class name=*"demo.Test2"* />

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

Note – We can also run all the classes in a package by using the package keyword.

<packages>

<package name="\*" />

</packages>

# Assertions in TestNG

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

 An example of assertion can be logging into the website, checking the title of the webpage.

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

|  |  |
| --- | --- |
| Method | Description |
| assertEquals(boolean expected, boolean actual) | It checks whether two values are equals |
| assertNotEquals(boolean expected, boolean actual) | It checks whether two values are not equal |
| assertFalse(boolean condition) | functionality is to check that a condition is false. |
| assertNotNull | Method to check that an object is not null. |
| assertNull | Method to check that an object is null. |
| assertTrue(boolean condition) | Method to check that a condition is true. |
| assertAll() | Used in soft assert |

## Types of Asserts in TestNG

* Hard Assert
* Soft Assert

**Hard Asserts** are those asserts that stop the test execution when an assert statement fails, and the subsequent assert statements are therefore not validated.

**Soft asserts** are just the opposite of hard asserts. In soft assertions, the subsequent assertions keep on running even though one assert validation fails, i.e., the test execution does not stop. Soft assert does not include by default in TestNG. For this, you need to include the package org.testng.asserts.Softassert. So, when should we use soft asserts in TestNG? We use soft asserts when we do not care about the failure of specific validations and want the test execution to proceed and also want to see the exception errors.

SoftAssert softassert = **new** SoftAssert();

softassert.assertEquals("abc", "xyz");

softassert.assertAll();

# Parameterization

TestNG Parameters are the arguments that we pass to the test methods.

TestNG Parameters are present in the xml file.

@Test

@Parameters({"a","b"})

**public** **void** add(**int** c, **int** d)

{

**int** sum=c+d;

System.***out***.println("Sum of two numbers : "+sum);

}

@Test

@Parameters({"a","b"})

**public** **void** subtract(**int** c, **int** d)

{

**int** subtract=c-d;

System.***out***.println("Subtraction of two numbers : "+subtract);

}

@Test

@Parameters({"a","b"})

**public** **void** multiply(**int** c, **int** d)

{

**int** mul=c\*d;

System.***out***.println("Multiplication of two numbers : "+mul);

}

-------------------TestNG xml file---------------------

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">

<suite name=*"Suite"*>

<test thread-count=*"5"* name=*"Test"*>

<parameter name=*"a"* value=*"5"*/>

<parameter name=*"b"* value=*"3"*/>

<classes>

<class name=*"demo.Test2"* />

</classes>

</test> <!-- Test -->

<test thread-count=*"5"* name=*"Test2"*>

<parameter name=*"a"* value=*"1"*/>

<parameter name=*"b"* value=*"2"*/>

<classes>

<class name=*"demo.Test2"* />

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

# TestNG Data providers

* The *DataProviders* in TestNG are another way to pass the parameters in the test function
* It is a part of the inbuilt TestNG data-driven testing for which TestNG
* If no name is provided for data provide, then it takes method name as data provide name

@DataProvider (name = "name\_of\_dataprovider")

public Object[][] dpMethod() {

return new Object [][] { values}

}

Example:

**public** **class** Test2 {

@DataProvider (name = "data-provider")

**public** Object[][] dpMethod(){

**return** **new** Object[][] {{1,2,3}, {3,4,7}};

}

@Test (dataProvider = "data-provider")

**public** **void** addnumbertest(**int** num1, **int** num2, **int** result) {

Assert.*assertEquals*(num1+num2, result);

}

--------------------------------------Data provider method---------------------------------------

@DataProvider (name = "data-provider")

public Object[][] dpMethod (Method m){

switch (m.getName()) {

case "Sum":

return new Object[][] {{2, 3 , 5}, {5, 7, 9}};

case "Diff":

return new Object[][] {{2, 3, -1}, {5, 7, -2}};

}

return null;

}

@Test (dataProvider = "data-provider")

public void Sum (int a, int b, int result) {

int sum = a + b;

Assert.assertEquals(result, sum);

}

@Test (dataProvider = "data-provider")

public void Diff (int a, int b, int result) {

int diff = a - b;

Assert.assertEquals(result, diff);

}

# TestNG Factory

Sometimes we may need to run a set of tests with different data values.

One of the main advantages of using the factory methods is that we can pass parameters to test classes while initializing them. These parameters can then be used across all the test methods present in the said test classes.

A factory in TestNG is a component that is responsible for creating and managing test instances at runtime.

On the other hand, a data provider in TestNG is used to supply test data to test methods. This can be useful when you need to test a single method with different sets of input data.

**public** **class** TestNGFactory {

@Test

**public** **void** printName() {

System.***out***.println("Test is a Test");

}

}

**public** **class** TestFactory {

@Factory

**public** Object[] factoryMethod() {

**return** **new** Object[] { **new** TestNGFactory(), **new** TestNGFactory() };

}

## TestNG Factory with multiple parameters

Data to class variables can be passed through TestNG Factory

Consider two numbers num1 and num2 and three methods accessing these number, addNumbers, subNumbers and mulNumbers.

Using Factory, we can send multiple data to num1 and num2 to execute these methods multiple times.

**public** **class** TestFactory {

@Factory

**public** Object[] factoryMethod() {

**return** **new** Object[] { **new** TestNGFactory(2,1), **new** TestNGFactory(3,4), **new** TestNGFactory(2,0)};

}

}

-------------------------Test Class-------------------------------------

**public** **class** TestNGFactory {

**int** num1;

**int** num2;

**public** TestNGFactory(**int** n1, **int** n2) {

**this**.num1 = n1;

**this**.num2 = n2;

}

@Test

**public** **void** sddNumbers() {

**int** sum = num1+num2;

System.***out***.println("Sum of two numbers is "+sum);

}

@Test

**public** **void** subNumbers() {

**int** diff = num1-num2;

System.***out***.println("Difference of two numbers is "+diff);

}

@Test

**public** **void** multiplyNumbers() {

**int** product = num1\*num2;

System.***out***.println("Product two numbers is "+product);

}

}

# TestNG Groups

* TestNG Groups allow you to perform groupings of different test methods.
* Grouping of test methods is required when you want to access the test methods of different classes.
* TestNG can be asked to include a certain set of groups while excluding another set of groups.
* Groups are specified in the testng.xml file with <groups> tag.

Example, we can divide our tests into smoke test and regression test

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">

<suite name=*"Suite"*>

<groups>

<run>

<include name = *"SmokeTest"*/>

<exclude name = *"RegressionTest"*/>

</run>

</groups>

<test thread-count=*"5"* name=*"Test"*>

<classes>

<class name=*"testNGDemo.AccountCreation"*/>

<class name=*"testNGDemo.Login"*/>

<class name=*"testNGDemo.SearchProducts"*/>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

------------------------Tests-------------------------------------------

@Test(groups= {"SmokeTest"})

**public** **void** AccountCreationWithMandatoryData()

{

System.***out***.println("This is AccountCreationWithMandatoryData method");

}

@Test(groups= {"RegressionTest"})

**public** **void** AccountCreationWithoutMandatoryData()

{

System.***out***.println("This is AccountCreationWithoutMandatoryData method");

}

## Multiple Groups

@Test(groups= {"Group A","Group B"})

**public** **void** testcase2()

{

System.out.println("Test case belonging to both Group A and Group B");

}

## Using of Regular Expressions

<include name="Include.\*"/>

## Groups in Groups

@Test(groups= {"Smoke"})

**public** **void** test1()

 {

     System.out.println("test1");

 }

@Test(groups= {"Regression"})

**public** **void** test2()

 {

     System.out.println("test2");

 }

@Test

**public** **void** test3()

 {

     System.out.println("test3");

 }}

---------------------------XML file-----------------------------------

<groups>

<define name="Group 1">

<include name="Smoke"/>

<include name="Regression"/>

</define>

<run>

<include name="Group 1"/>

</run>

</groups>

# Test Dependencies

In TestNG, we often require to run the test in a specific order. Along with that, we may require that a test must run only when another test has run.

For example, we want testB to run if testA has run. testB is dependent on testA, and these are called Dependent Tests in TestNG.

The dependent tests in TestNG determine the dependency of a test on a single or group of tests.

Dependent tests are implemented using dependsOnMethods in @Test annotations

public class Dependent {

@Test (dependsOnMethods = { "OpenBrowser" })

public void SignIn() {

System.out.println("This will execute second (SignIn)");

}

@Test

public void OpenBrowser() {

System.out.println("This will execute first (Open Browser)");

}

}

## Depends on group

We can make a test method depended on a group

public class GroupDependency

{

@Test(dependsOnGroups = { "SignIn" })

public void ViewAcc() {

System.out.println("SignIn Successful");

}

@Test(groups = { "SignIn" })

public void LogIn() {

System.out.println("Logging In Success");

}

}

## Multiple Dependencies

public class DependsOnTest

{

@Test

public void OpenBrowser() {

System.out.println("Opening The Browser");

}

@Test(dependsOnMethods = { "SignIn", "OpenBrowser" })

public void LogOut() {

System.out.println("Logging Out");

}

@Test

public void SignIn() {

System.out.println("Signing In");

}

}

# Creating Dependencies in TestNG XML file

Dependencies on groups can also be created using XML file.

Example:

<test name="AccountInfo" >

<groups>

<dependencies>

<group depends-on= "openbrowser" name= "login"></group>

<group depends-on= "login" name= "viewaccount"></group>

<group depends-on= "viewaccount" name= "logout"></group>

</dependencies>

</groups>

<classes>

<class name="GroupDependency" />

</classes>

</test>

# Running tests in parallel

Parallel execution is where the tests are run parallelly by opening multiple instances of the application.

Write the below line in testing.xml file at Suite level

<suite name=*"Suite"* parallel=*"tests"*>

# TestNG Listeners

**Listener** is an interface that modifies the default TestNG's behaviour. Listeners "listen" to the event defined in the selenium script and behave accordingly. "listeners" must be listening to something in the code. TestNG listeners are the piece of code that listens to the events occurring in the TestNG.

For example, we want to print the exception error onto the reports only if the test fails. Here, we can apply a TestNG listener that will listen to the event of "failing of test case" and when it does, it will log the error.

## Methods in listeners

1. **onTestStart();** - An onTestStart() is invoked only when any test method gets started.
2. **onTestSuccess();** - An onTestSuccess() method is executed on the success of a test method.
3. **onTestFailure();** - An onTestFailure() method is invoked when test method fails.
4. **onTestSkipped();** - An onTestSkipped() run only when any test method has been skipped.
5. **onTestFailedButWithinSuccessPercentage();** - This method is invoked each time when the test method fails but within success percentage. This is used with **successPercentage** and **invocationCount** parameters for the Test methods.
6. **onStart();** - An onStart() method is executed on the start of any test method.
7. **onFinish();** - An onFinish() is invoked when any test case finishes its execution.

# Creating Listeners

There are two ways of creating listeners:

1. Using the @Listeners annotation within the class
2. Using the @Listeners annotation within the Suite

Method 1:

1. Create a listener class that will override the methods from ITestListener
2. Add custom code for overridden methods.
3. Test class will have the below annotation for the listener

@Listeners(testNGDemo.Listener.**class**)

Example:

@Listeners(testNGDemo.Listener.**class**)

**public** **class** TestNGDemo {

@Test

**public** **void** methods1()

{

**int** num1 = 10;

**int** num2 = 20;

**int** sum = num1 + num2;

Assert.*assertEquals*(sum, 30);

}

@Test

**public** **void** methods2()

{

**int** num1 = 10;

**int** num2 = 20;

**int** diff = num1 - num2;

Assert.*assertEquals*(diff, 30);

}

}

---------------------------------------------Listener class--------------------------------------------

**public** **class** Listener **implements** ITestListener{

@Override

**public** **void** onTestStart(ITestResult result) {

System.***out***.println("This is the start of the test");

}

@Override

**public** **void** onTestSuccess(ITestResult result) {

System.***out***.println("Success of test cases and its details are : "+result.getName());

}

@Override

**public** **void** onTestFailure(ITestResult result) {

System.***out***.println("Failure of test cases and its details are : "+result.getName());

}

@Override

**public** **void** onStart(ITestContext context) {

System.***out***.println("Start of TestNG testing");

}

@Override

**public** **void** onFinish(ITestContext context) {

System.***out***.println("End of TestNG testing");

}

}

Method 2:

In the second method, listener details are given in the testNG.xml file.

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">

<suite name=*"Suite"*>

<listeners>

<listener class-name=*"testNGDemo.Listener"*/>

</listeners>

<test thread-count=*"5"* name=*"Test"*>

<classes>

<class name=*"demo.TestNGDemo"*/>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

# Reporter class in TestNG

Reporter class in [TestNG](https://testng.org/doc/documentation-main.html) helps the testers log the messages on to the test execution reports. This will help with easy debugging of the code.

Syntax:

Reporter.log(“message”);

Import Required: import org.testng.Reporter;

This will add the reporter message details in both index.html and emailablereport.html files.

## Printing the messages in both the console and reports

To print log messages in both reports and the console, an additional Boolean parameter must be passed as true.

Reporter.log(“message”, true);

Custom logging can also be used with listeners. In that case, the listener class should extend TestListenerAdapter

Example:

@Override

public void onTestFailure(ITestResult tr) {

log(tr.getName()+ "--Test method failed\n");

}

## Types of logs

* Reporter.log(String s);
* Reporter.log(String s, Boolean logToStandardOut);
* Reporter.log(String s, int level); - level – The verbosity of the message to be logged
* Reporter.log(String s, int level, Boolean logToStandardOut);

Note - The Verbose Level in TestNG is used to define the amount of logging performed on the console. The verbosity level ranges from 0 to 10, where 10 is the most detailed logging level whereas 0 means minimal logging.