

TestNg

What is TestNG?

TestNG is an automation testing framework in which NG stands for "Next Generation".

TestNG, you can generate a proper report, and you can easily come to know how many test cases are passed, failed, and skipped. You can execute the failed test cases separately.

The TestNG provides an option, i.e., testng-failed.xml file in test-output folder. If you want to run only failed test cases means you run this XML file. It will execute only failed test cases. TestNG is case-sensitive.

Why Use TestNG with Selenium?

Default Selenium tests do not generate a proper format for the test results. Using TestNG we can generate test results.

Generate the report in a proper format including a number of test cases runs, the number of test cases passed, the number of test cases failed, and the number of test cases skipped.

WebDriver has no native mechanism for generating reports. TestNG can generate the report in a readable format.

There is no more need for a static main method in our tests.

Advantages of TestNG over JUnit

There are three major advantages of TestNG over JUnit:

- Annotations are easier to understand
- Test cases can be grouped more easily
- Parallel testing is possible

Generating HTML Reports

TestNG has the ability to generate reports in HTML format.

Annotations used in TestNG

Parameters

Priority

If test priority is not defined while running multiple test cases, TestNG assigns all @Test a priority as zero (0).

Demo of TestNG code without Priority in Alphabetical Order

If we don't mention any priority, TestNg will execute the @Test methods based on alphabetical order of their method names irrespective of their place of implementation in the code.

<pre> @Test public void c_method () { System.out.println("I'm in method C"); } @Test public void b_method () { System.out.println("I'm in method B"); } @Test public void a_method () { System.out.println("I'm in method A"); } </pre> <p>Output</p> <p>I'm in method A I'm in method B I'm in method C</p>	<pre> @Test(priority=-1) public void c_method () { System.out.println("I'm in method C"); } @Test(priority=0) public void b_method () { System.out.println("I'm in method B"); } @Test(priority=1) public void a_method () { System.out.println("I'm in method A"); } </pre> <p>Output</p> <p>I'm in method C I'm in method B I'm in method A</p>
--	---

Methods with Same Priority:

- There may be a chance that methods may contain same priority. In those cases, TestNg considers the alphabetical order of the method names whose priority is same.

```

@Test(priority=6) public void c_method () {}
@Test(priority=6) public void b_method () {}

```

Output: Execute b_method First because in alphabetical order.

If you want the methods to be executed in a different order, use the parameter "priority". **Parameters are keywords that modify the annotation's function**

```

@Test(priority = 0)

```

parameter value of the parameter

Parameters require you to assign a value to them. Parameters are enclosed in a pair of parentheses which are placed right after the annotation like the code snippet shown below.

Multiple Parameters

Aside from "priority," @Test has another parameter called "alwaysRun" which can only be set to either "true" or "false." **To use two or more parameters in a single annotation, separate them with a comma** such as the one shown below.

```

@Test (priority = 0, alwaysRun = true)

```

@BeforeSuite: The annotated method will be run before all tests in this suite have run.

@AfterSuite: The annotated method will be run after all tests in this suite have run.

@BeforeTest: The annotated method will be run before any test method belonging to the classes inside the tag is run. Ex: launching browser

@AfterTest: The annotated method will be run after all the test methods belonging to the classes inside the tag have run. Ex: Close/quit Browser.

@BeforeGroups: The list of groups that this configuration method will run before. This method is guaranteed to run shortly before the first test method that belongs to any of these groups is invoked.

@AfterGroups: The list of groups that this configuration method will run after. This method is guaranteed to run shortly after the last test method that belongs to any of these groups is invoked.

@BeforeClass: The annotated method will be run before the first test method in the current class is invoked.

@AfterClass: The annotated method will be run after all the test methods in the current class have been run.

@BeforeMethod: The annotated method will be run before each test method.

@AfterMethod: The annotated method will be run after each test method.

@Test: The annotated method is a part of a test case

TestNg Groups: Include, Exclude with Example

- We don't want to define test methods separately in different classes (depending upon functionality) and
- At the same time want to ignore (not to execute) some test cases as if they do not exist in the code.
- So, to carry out this we have to Group them. This is done by using "include" and "exclude" mechanism supported in TestNG.

```
@Test (priority=1, invocationCount=2, groups= {"Sanity", "Regression"})
    public void TestCase2() {
        System.out.println("Test Cases 1");
    }
@Test (priority=2, groups= {"Sanity"})
    public void TestCase3() {
        System.out.println("Test Cases 2");
    }
```

Include Groups: Include the test cases

Exclude Groups: Do not include test cases

testNg.Xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<suite name="Groups Suite">
    <test name="Grouping Test ">
        <groups>
            <run>
                <include name="Sanity" />
            </run>
        </groups>
    </test>
</suite>
```

```

        <exclude name="Regression" />
    </run>
</groups>
<classes>
    <class name="com.bank.test.TestCases" />
</classes>
</test>
</suite>

```

Multiple tags are used in a sequence to build a working TestNG xml like <suite>, <test> and <class>

- First is <suite> tag, which holds a logical name which defines full information to TestNG reported to generate execution report.
- Second is <test name=" Grouping Test ">, note it is logical name which holds the information of test execution report like pass, fail, skip test cases and other information like total time for execution and group info
- Third is <class name com.bank.test.TestCases " />, com.group.guru99 is the package used, and Test Class name is TestCases.

TestCases.java

testng.xml

TestNG Report

file:///E:/Eclipse/Eclipse-MarketPlace-photon/JavaMaven/test-output/emailable-report.html

Test	# Passed	# Skipped	# Retried	# Failed	Time (ms)	Included Groups	Excluded Groups
Groups Suite							
Grouping Test	2	0	0	0	103		Regression

Class	Method	Start	Time (ms)
Groups Suite			
Grouping Test — passed			
com.bank.test.TestCases	TestCases1	1596612427088	16
	TestCases3	1596612427113	2

Grouping Test

com.bank.test.TestCases#TestCase1

[back to summary](#)

com.bank.test.TestCases#TestCase3

Parallel Test Case Execution and How to Run Multiple Test Suites in Selenium using TestNg

As the thread count is three, three
WebDriver instances can run parallel in
three different browsers

Test cases can run
parallel

```

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="TestSuite" thread-count="3" parallel="methods" >
    <test name="testGuru">

```

Testng.xml file Below

1) thread-count: This is used for parallel execution, based on the number script. It will execute in parallel or sequential order.

```
<suite name="ParrallelTest" parallel="tests" thread-count="2"
  preserve-order="true">
  <test name="Crome Test">
    <parameter name="myBrowsers" value="chrome" />
    <classes><class name="com.bank.test.LoginTC" /></classes>
  </test>
  <test name="FireFox Test">
    <parameter name="myBrowsers" value="gecko" />
    <classes><class name="com.bank.test.LoginTC" /></classes>
  </test>
</suite>
```

Test Base Class file

```
@BeforeClass
@Parameters("myBrowsers")
public static void browserInitialization(String myBrowsers) {
    if (myBrowsers.equalsIgnoreCase("chrome")) {
        System.setProperty("webdriver.chrome.driver", path +
"/browserDrivers/chromedriver.exe");
        driver = new ChromeDriver();
    } else if (myBrowsers.equalsIgnoreCase("gecko")) {
        System.setProperty("webdriver.gecko.driver", path +
"/browserDrivers/geckodriver.exe");
        driver = new FirefoxDriver();
    }
    else if (myBrowsers.equalsIgnoreCase("chrome")) {
        System.setProperty("webdriver.ie.driver", path +
"/browserDrivers/IEDriverServer.exe");
        driver = new InternetExplorerDriver();
    }
    driver.manage().window().maximize();
    driver.get(prop.getProperty("url"));
}
```

TestNG Listeners in Selenium: ITestListener & ITestResult Example

What are Listeners in TestNG?

Listener is defined as interface that modifies the default TestNG's behavior. As the name suggests Listeners "listen" to the event defined in the selenium script and behave accordingly. It is used in selenium by implementing Listeners Interface. It allows customizing TestNG reports or logs. There are many types of TestNG listeners available.

1. IAnnotationTransformer ,
2. IAnnotationTransformer2 ,

3. IConfigurable ,
4. IConfigurationListener ,
5. IExecutionListener,
6. IHookable ,
7. IInvokedMethodListener ,
8. IInvokedMethodListener2 ,
9. IMethodInterceptor ,
10. IReporter,
11. ISuiteListener,
12. ITestListener .

ITestListener has following methods

- **OnStart**- OnStart method is called when any Test starts.
- **onTestSuccess**- onTestSuccess method is called on the success of any Test.
- **onTestFailure**- onTestFailure method is called on the failure of any Test.
- **onTestSkipped**- onTestSkipped method is called on skipped of any Test.
- **onTestFailedButWithinSuccessPercentage**- method is called each time Test fails but is within success percentage.
- **onFinish**- onFinish method is called after all Tests are executed.

```
public class CustomListenerTest implements ITestListener{
```

```
@Listeners(Listener_Demo. CustomListenerTest.class)
Public class TestLogin{}
```

```
<suite name="Suite">
  <listeners>
    <listener class-name="com.torqus.utilities. CustomListenerTest " />
  </listeners>
  <test thread-count="1" name="Test Listener ">
    <classes><class name="com.bamk.loginTest" /></classes>
  </test>
</suite>
```

How to Execute Failed Test Cases in TestNG: Selenium WebDriver

When we execute testng.xml file, after completion of test cases execution,
In console shows, how many test cases are failed or passed.

Go to the project and side click the project, refresh the project.

TestNg create one test-Output folder in project folder there is an emailable HTML report and also TestNg failed .xml file.

You can just run only TestNg failed. Xml file

Parameterization in Selenium

Parameterization in Selenium **is a process to parameterize the test scripts in order to pass multiple data to the application at runtime. It is a strategy of execution which automatically runs test cases multiple times using different values. The concept achieved by parameterizing the test scripts is called Data Driven Testing.**

There are **two ways** by which we can achieve parameterization in TestNG

1. With the help of **Parameters annotation** and **TestNG XML** file.

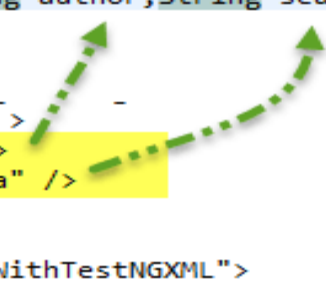
```
@Parameters({"name", "searchKey"})
```

2. With the help of **Data Provider** annotation.

```
@DataProvider(name="SearchProvider")
```

Parameters Annotation in TestNG is a method used to pass values to the test methods as arguments using .xml file. Users may be required to pass the values to the test methods during run time. The @Parameters annotation method can be used in any method having @Test, @Before, @After or @Factory annotation.

```
@Test
@Parameters({"author", "searchKey"})
public void testParameterWithXML(String author, String searchKey)
```



```
<suite name="TestSuite" thread-count="1">
  <parameter name="author" value="demo" />
  <parameter name="searchKey" value="India" />
  <test name="testGuru">
    <classes>
      <class name="parameters.ParameterWithTestNGXML">
        </class>
      </classes>
    </test>
  </suite>
```

Here 'author' parameter from xml will map with the 'author' parameter in test method

Data Provider in TestNG

Data Provider in TestNG is a method used when a user needs to pass complex parameters. Complex Parameters need to be created from Java such as complex objects, objects from property files or from a database can be passed by the data provider method. The method is annotated by @DataProvider and it returns an array of objects.

@Parameters annotation is easy but to test with multiple sets of data we need to use Data Provider. To fill thousands of web forms using our testing framework we need a different methodology which can give us a very large dataset in a single execution flow.

This data driven concept is achieved by **@DataProvider** annotation in TestNG.

Data provider returns a **two-dimensional JAVA object** to the test method and the test method, will invoke M times in a M*N type of object array. For example, if the Data Provider returns an array of 2*3 objects, the corresponding testcase will be invoked 2 times with 3 parameters each time.

Return Type of data provider is 2D Array object

```
@Test(dataProvider="SearchProvider")
public void testMethod(String author,String searchKey) {}

@DataProvider(name="SearchProvider")
public Object [ ] [ ] getDataFromDataProvider(){
    return new Object [ ] [ ] {{ "Guru99", "India" }, {"Krishna", "UK" }, {"Bhupesh", "USA" }};
}
```

Invoke DataProvider from different class

By default, DataProvider resides in the same class where test method is or its base class.

To put it in some other class we need to make data provider method as static and in test method we need to add an attribute **dataProviderClass** in **@Test** annotation.

```
@Test(dataProvider="SearchProvider",dataProviderClass=DataProviderClass.class)
public void testMethod(String author,String searchKey) {}
```

InvocationCount:

Run single test case in multiple times by using invocation count.

Below test case are execute in 4 times because we are set the values of invocation count is 4

```
@Test(priority=1,invocationCount=4)
    public void TestCase2() {
        System.out.println("Test Cases 2"); }
}
```

Skip Exception OR Ignore Test in TestNG

In TestNG, **@Test(enabled=false)** annotation is used to skip a test case if it is not ready to test. We don't need to import any additional statements.

```
@Test(enabled=false) //Not skip test cases @Test(enabled=true)
    public void testCaseEnabling(){sop("I'm Not Ready, please don't execute me");}
```

This is exception of actual Skip test case. We can Skip a test by using TestNG Skip Exception if we want to Skip a particular Test.

```
@Test
    public void testCaseSkipException(){
        System.out.println("I am in skip exception");
        throw new SkipException("Skipping this exception");
    }
```


How to Handle Exception in TestNg

In TestNG we use `expectedException` with `@Test` annotation and we need to specify the type of exceptions that are expected to be thrown when executing the test methods.

```
@Test (expectedExceptions= ArithmeticException.class)
public void dividedByZeroExample1() {
    int e = 1/0; }
```

```
Test(expectedExceptions= {ArithmeticException.class, IOException.class})
public void dividedByZeroExample1(){
    int e = 1/0; }
```

DependsOnMethod

Dependency is a feature in TestNG that allows a test method to depend on a single or a group of test methods. This will help in **executing a set of tests to be executed before a test method**.

Tests dependency only works if the `depend-on-method` is part of the same class or any of the inherited base class (i.e. while extending a class).

Test dependency only works with other tests that belong to the same class or in one of the inherited classes but not across different classes

Depends on Method

```
@Test(dependsOnMethods = { "testTwo" })
    public void testOne() {
        sop("Test method one"); }
@Test
    public void testTwo() {
        sop ("Test method two"); }

Output: Test method two
          Test method one
@Test(dependsOnMethods={ "testTwo", "testThree" })
    public void testOne() { sop("Test method one"); }
@Test
    public void testTwo() { sop ("Test method two"); }
@Test
    public void testThree() { sop ("Test method three"); }
Output:
    Test method three
    Test method two
    Test method one
```

Depends on Groups

```
@Test(dependsOnGroups = { "test-group" })
    public void groupTestOne() {
        sop ("Group Test method one");
    }
@Test(groups = { "test-group" })
    public void groupTestTwo() {
        sop ("Group test method two");
    }
@Test(groups = { "test-group" })
    public void groupTestThree() {
        sop ("Group Test method three");
    }

Output: Group Test method three
          Group test method two
          Group Test method one
```

What Is the Difference Between Assert and Verify in Selenium?

Both Assert and Verify commands are used to find whether a given input is present or not on the webpage.

There are some difference between Assert and Verify in Selenium.

Assert (**Hard Assertions**):

If the assert condition is true then the program control will execute the next test step but if the condition is false, the execution will stop and further test step will not be executed. When an “assert” command fails, the test execution will be aborted. So when the Assertion fails, all the test steps after that line of code are skipped. The solution to overcoming this issue is to use a try-catch block.

In simple words, if the assert condition is true then the program control will execute the next test step but if the condition is false, the execution will stop and further test step will not be executed.

```
Assert.assertTrue(false);  
Assert.assertEquals("Tutorialspoint", "Tutorial");
```

Verify (**Soft Assertions**):

There won't be any halt in the test execution even though the verify condition is true or false.

Verify is used in less critical things. Cases where we can move forward even if the other test cases fail.

Soft Assertions are the type of assertions that do not throw an exception automatically when an assertion fails unless it is asked for. This is useful if you are doing multiple validations in a form, out of which only a few validations directly have an impact on deciding the test case status.

Here, we use a class called SoftAssert and the method **assertAll()** is called to throw all exceptions caught during execution. When softAssert is used, it performs assertion and if an exception is found, it's not thrown immediately, rather it continues until we call the method **assertAll()** to throw all exceptions caught.

```
SoftAssert softly = new SoftAssert();  
Softly.assertTrue(false);  
Softly.assertEquals("Tutorialspoint", "Tutorial");  
Softly.assertAll();
```