TestNg tutorial

TestNG is a testing framework like Junit and NUnit, but it has lot more powerful feature than Junit and Nunit such as

- Annotations
- Multithreading support
- Support for Parameters
- Supported by a Variety of tools and plug-ins
- Dependent methods for application server testing.
- E-mailable HTML report

TestNG is designed to cover all categories of tests: unit, functional, end-to-end, integration, etc...

POM.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>

	<groupId>com</groupId>
	<artifactId>testNG</artifactId>
	<version>0.0.1-SNAPSHOT</version>


	<name>testNG</name>
	<url>http://maven.apache.org</url>

	<properties>
		<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
	</properties>

	<dependencies>

		<!-- https://mvnrepository.com/artifact/org.testng/testng -->
		<dependency>
			<groupId>org.testng</groupId>
			<artifactId>testng</artifactId>
			<version>6.13</version>
			<scope>test</scope>
		</dependency>

	</dependencies>
```

```
</project>
```

Annotations:

@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 <test> tag is run.
@AfterTest: The annotated method will be run after all the test methods belonging to the classes inside the <test> tag have run.
@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.

```java
package com.testNG;

import org.testng.annotations.AfterClass;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.AfterSuite;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.BeforeSuite;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;

public class BasicAnnotationsInTestNG {

	@BeforeSuite
	public void test1() {
    System.out.println("@BeforeSuite");
	}

	@BeforeTest
	public void test2() {
		System.out.println("@BeforeTest");
	}
	@BeforeClass
	public void test3() {
		System.out.println("@BeforeClass");
	}

	@BeforeMethod
	public void test4() {
		System.out.println("@BeforeMethod");
	}

	@Test
	public void test5() {
		System.out.println("@Test");
	}

	@Test
	public void test10() {
		System.out.println("@Test1");
	}

	@AfterSuite
	public void test6() {
		System.out.println("@AfterSuite");
	}

	@AfterTest
	public void test7() {
		System.out.println("@AfterTest");
	}

	@AfterClass
	public void test8() {
		System.out.println("@AfterClass");
	}

	@AfterMethod
	public void test9() {
		System.out.println("@AfterMethod");
	}

}
```

Output:

```
@BeforeSuite
@BeforeTest
```

```
@BeforeClass
@BeforeMethod
@Test1
@AfterMethod
@BeforeMethod
@Test
@AfterMethod
@AfterClass
@AfterTest
PASSED: test10
PASSED: test5

===============================================
    Default test
    Tests run: 2, Failures: 0, Skips: 0
===============================================

@AfterSuite

===============================================
Default suite
Total tests run: 2, Failures: 0, Skips: 0
===============================================
```

Assertion in TestNG

Assertions are used to perform various kinds of validations in the tests and help us to decide whether the test has passed or failed.

```java
package com.testNG;

import org.testng.Assert;
import org.testng.annotations.Test;

public class AssertionsInTestNg {

	@Test
	public void test() {
      Assert.assertTrue(true);
	}

	@Test
	public void test1() {
      Assert.assertTrue(false);
	}
```

```
        @Test
        public void test2() {
    Assert.assertEquals("Test", "Test");
        }

        @Test
        public void test3() {
    Assert.assertEquals("Test", "Test2");
        }

        @Test
        public void test4() {
    Assert.assertTrue(false, "not matching the expected condition");
        }

        @Test
        public void test5() {
    Assert.assertFalse(false, "This is expected");
        }

        @Test
        public void test6() {
                Assert.assertNotEquals("test", "test1");
        }
}
```

Output:

```
PASSED: test
PASSED: test2
PASSED: test5
PASSED: test6
FAILED: test1
java.lang.AssertionError: expected [true] but found [false]
        at org.testng.Assert.fail(Assert.java:96)
        at org.testng.Assert.failNotEquals(Assert.java:776)
        at org.testng.Assert.assertTrue(Assert.java:44)
        at org.testng.Assert.assertTrue(Assert.java:54)
        at com.testNG.AssertionsInTestNg.test1(AssertionsInTestNg.java:15)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at org.testng.internal.Invoker.invokeTestMethod(Invoker.java:707)


FAILED: test3
```

```
java.lang.AssertionError: expected [Test2] but found [Test]
	at org.testng.Assert.fail(Assert.java:96)
	at org.testng.Assert.failNotEquals(Assert.java:776)
	at org.testng.Assert.assertEqualsImpl(Assert.java:137)
	at org.testng.Assert.assertEquals(Assert.java:118)
	at 
org.testng.internal.MethodInvocationHelper.invokeMethod(MethodInvocationHelper.java:1
24)

FAILED: test4
java.lang.AssertionError: not matching the expected condition expected [true] but found 
[false]
	at org.testng.Assert.fail(Assert.java:96)
	at org.testng.Assert.failNotEquals(Assert.java:776)
	at org.testng.Assert.assertTrue(Assert.java:44)
	at com.testNG.AssertionsInTestNg.test4(AssertionsInTestNg.java:30)
	at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
	at org.testng.remote.AbstractRemoteTestNG.run(AbstractRemoteTestNG.java:132)
	at org.testng.remote.RemoteTestNG.initAndRun(RemoteTestNG.java:230)
	at org.testng.remote.RemoteTestNG.main(RemoteTestNG.java:76)


===============================================
    Default test
    Tests run: 7, Failures: 3, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 7, Failures: 3, Skips: 0
===============================================
```

Dependents On Methods in TestNG

Depends on method is used to run the test based on certain order , Or if you want your test input from other test. if dependent method fails then calling method will get skipped.

```
package com.testNG;

import org.testng.Assert
import org.testng.annotations.Test;

public class DependentsOnMethodsTestNG {
	@Test
	public void test1() {
		Assert.assertTrue(true);
	}

	@Test(dependsOnMethods = { "test1" })
	public void test2() {
		System.out.println("Test2 pass");
	}

	@Test(dependsOnMethods = { "test1", "test2" })
	public void test3() {
		System.out.println("Test3 pass");
	}

	@Test
	public void test4() {
		Assert.assertTrue(false);
	}

	@Test(dependsOnMethods = { "test4" })
	public void test5() {
		System.out.println("Test5 fail");
	}

}
```

Output:

```
Test2 pass
Test3 pass
PASSED: test1
PASSED: test2
PASSED: test3
FAILED: test4
java.lang.AssertionError: expected [true] but found [false]
	at org.testng.Assert.fail(Assert.java:96)
	at org.testng.Assert.failNotEquals(Assert.java:776)
	at org.testng.Assert.assertTrue(Assert.java:44)
	at org.testng.Assert.assertTrue(Assert.java:54)
	at
com.testNG.DependentsOnMethodsTestNG.test4(DependentsOnMethodsTestNG.java:25)
	at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
	at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
```

```
	at org.testng.remote.AbstractRemoteTestNG.run(AbstractRemoteTestNG.java:132)
	at org.testng.remote.RemoteTestNG.initAndRun(RemoteTestNG.java:230)
	at org.testng.remote.RemoteTestNG.main(RemoteTestNG.java:76)

SKIPPED: test5
java.lang.Throwable: Method DependentsOnMethodsTestNG.test5()[pri:0, 
instance:com.testNG.DependentsOnMethodsTestNG@548b7f67] depends on not 
successfully finished methods
	at org.testng.internal.Invoker.invokeTestMethods(Invoker.java:878)
	at 


===============================================
    Default test
    Tests run: 5, Failures: 1, Skips: 1
===============================================


===============================================
Default suite
Total tests run: 5, Failures: 1, Skips: 1
===============================================
```

Data Provider in TestNG

Marks a method as supplying data for a test method. The annotated method must return an Object[][] where each Object[] can be assigned the parameter list of the test method. The @Test method that wants to receive data from this DataProvider needs to use a dataProvider name equals to the name of this annotation.

```
package com.testNG;

import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class DataProviderInTestNG {

	@DataProvider
	public Object[][] testData(){
		Object[][] data = new Object[2][2];
		
		data[0][0] = "Test1";
		data[0][1] = "Test2";
```

```
                data[1][0] = "Test3";
                data[1][1] = "Test4";

                return data;
        }

        @DataProvider
        public Object[][] testData1(){
                Object[][] data = new Object[3][3];

                data[0][0] = 1;
                data[0][1] = 2;
                data[0][2] = 3;

                data[1][0] = 4;
                data[1][1] = 5;
                data[1][2] = 6;

                data[2][0] = 7;
                data[2][1] = 8;
                data[2][2] = 9;

                return data;
        }

        @Test(dataProvider="testData")
        public void test(String s1, String s2){
                System.out.println("s1"+" "+s1+" s2"+" "+s2);
        }

        @Test(dataProvider="testData1")
        public void test1(int s1, int s2, int s3){
                System.out.println("s1"+" "+s1+" s2"+" "+s2+" s3"+" "+s3);
        }

}
```

Output:

```
s1 Test1 s2 Test2
s1 Test3 s2 Test4
s1 1 s2 2 s3 3
s1 4 s2 5 s3 6
s1 7 s2 8 s3 9
PASSED: test("Test1", "Test2")
PASSED: test("Test3", "Test4")
```

```
PASSED: test1(1, 2, 3)
PASSED: test1(4, 5, 6)
PASSED: test1(7, 8, 9)

===============================================
    Default test
    Tests run: 5, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 5, Failures: 0, Skips: 0
===============================================
```

Design TestNG xml to Run Testcases

```java
package com.testNG;

import org.testng.annotations.Test;

public class RunTestThroughTestNgXML {

	@Test
	public void test1() {

	}

	@Test
	public void test2() {

	}

}
```

RunTestThroughTestNgXML.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="TestNgTest">
	<test name="MyFirstTest">
		<classes>
			<class name="com.testNG.RunTestThroughTestNgXML"></class>
		</classes>
	</test>
```

```
</suite>
```

Grouping In TestNG

TestNG allows you to perform sophisticated groupings of test methods. Not only can you declare that methods belong to groups, but you can also specify groups that contain other groups. Then TestNG can be invoked and asked to include a certain set of groups (or regular expressions) while excluding another set. This gives you maximum flexibility in how you partition your tests and doesn't require you to recompile anything if you want to run two different sets of tests back to back.

```java
package com.testNG;

import org.testng.annotations.Test;

public class GroupingInTestNG {

	@Test(groups={"sanity"})
	public void test1() {

	}
	@Test(groups={"sanity","regression"})
	public void test2() {

	}

	@Test(groups={"regression"})
	public void test3() {

	}

}
```

groupinginTestNG.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="Groupting in TestNG">
	<groups>
		<run>
			<include name="regression"></include>
```

```
				<include name="sanity"></include>
			</run>
		</groups>
		<test name="Groupting in TestNG">
			<classes>
				<class name="com.testNG.GroupingInTestNG"></class>
			</classes>
		</test>

</suite>
```

Priority In TestNG

This will help us to decide which test case to be executed first and so on.

```
package com.testNG;

import org.testng.annotations.Test;

public class PriorityInTestNg {

	@Test(priority = 2)
	public void test1() {

	}

	@Test(priority = 1)
	public void test2() {

	}

	@Test(priority = 0)
	public void test3() {

	}
}
```

Parameterization in TestNG

@Parameters will help us to supply parameter to the test from testing.xml file. Like username, Password, DB connection url

```
package com.testNG;

import org.testng.annotations.Optional;
import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

public class ParametersInTestNG {


            @Test
            @Parameters({"data","data1"})
            public void test1(@Optional("default") String data, @Optional("default1") String
data1) {
        System.out.println(data);
        System.out.println(data1);
            }
}
```

parametersinTestNg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="Suite">
   <parameter name="data" value="TestData"></parameter>
   <parameter name="data1" value="TestData1"></parameter>
  <test name="Test">
      <classes>
         <class name="com.testNG.ParametersInTestNG"></class>
      </classes>
  </test> <!-- Test -->
</suite> <!-- Suite -->
```

Output:

```
TestData
TestData1

===============================================
Suite
Total tests run: 1, Failures: 0, Skips: 0
===============================================
```

Listener In TestNG

Listener is defined as interface that modifies the default TestNG's behaviour. 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.

Types of Listeners in TestNG

There are many types of listeners which allows you to change the TestNG's behavior.

Below are the few TestNG listeners:

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.

```
package com.testNG;

import org.testng.ITestContext;
import org.testng.ITestListener;
import org.testng.ITestResult;
import org.testng.Reporter;

public class ListnerInTestNG implements ITestListener{

	public void onFinish(ITestContext arg0) {
		System.out.println("All test finished");
		Reporter.log("All test finished");

	}

	public void onStart(ITestContext arg0) {
		System.out.println("All Test Execution started");
		Reporter.log("All Test Execution started");
	}

	public void onTestFailedButWithinSuccessPercentage(ITestResult arg0) {
		System.out.println("Test is pass absed on percentage
"+getMethodName(arg0));
		Reporter.log("Test is pass absed on percentage "+getMethodName(arg0));
	}

	public void onTestFailure(ITestResult arg0) {
		System.out.println("Test failed "+getMethodName(arg0));
		Reporter.log("Test failed "+getMethodName(arg0));
	}

	public void onTestSkipped(ITestResult arg0) {
		System.out.println("Test Skipped "+getMethodName(arg0));
		Reporter.log("Test Skipped "+getMethodName(arg0));
	}

	public void onTestStart(ITestResult arg0) {
		System.out.println("Starting test is "+getMethodName(arg0));
		Reporter.log("Starting test is "+getMethodName(arg0));
	}

	public void onTestSuccess(ITestResult arg0) {
		System.out.println("Test Passed "+getMethodName(arg0));
		Reporter.log("Test Passed "+getMethodName(arg0));
	}

	private static String getMethodName(ITestResult arg0){
		return arg0.getMethod().getConstructorOrMethod().getName();
	}


}

package com.testNG;

import org.testng.Assert;
import org.testng.SkipException;
import org.testng.annotations.Test;

public class ListenerTestClass {
	int i = 0;
	@Test
	public void test1() {

	}

	@Test
	public void test2() {
		throw new SkipException("I am skipping the test");
	}

	@Test(successPercentage = 60, invocationCount = 5)
	public void test3() {
		i++;
		System.out.println("test3 test method, invocation count: " + i);
		if (i == 1 || i == 2) {
			//System.out.println("test3 failed!");
			Assert.assertEquals(i, 8);
		}
	}

}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="Suite">
    <listeners>
        <listener class-name="com.testNG.ListnerInTestNG"></listener>
    </listeners>
  <test name="Test">
      <classes>
          <class name="com.testNG.ListenerTestClass"></class>
      </classes>
  </test> <!-- Test -->
</suite> <!-- Suite -->
```

Retry In TestNG

Retry is used to retry failed test cases multiple time based on configuration.

```
package com.testNG;
import org.testng.IRetryAnalyzer;
import org.testng.ITestResult;

public class RetryAnalyzer implements IRetryAnalyzer {
	int counter = 0;
	int retryLimit = 3;
	/*
	 * This method decides how many times a test needs to be rerun.
	 * TestNg will call this method every time a test fails. So we
	 * can put some code in here to decide when to rerun the test.
	 *
	 * Note: This method will return true if a tests needs to be retried
	 * and false it not.
	 *
	 */

	public boolean retry(ITestResult result) {

		if(counter < retryLimit)
		{
			counter++;
			return true;
		}
		return false;
	}
}

package com.testNG;

import java.lang.reflect.Constructor;
import java.lang.reflect.Method;

import org.testng.IAnnotationTransformer;
import org.testng.annotations.ITestAnnotation;

public class Retry implements IAnnotationTransformer {


	public void transform(ITestAnnotation annotation, Class testClass, Constructor
testConstructor, Method testMethod) {
		annotation.setRetryAnalyzer(RetryAnalyzer.class);

	}
}

package com.testNG;

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

```java
public class RetryTest {
	int i = 0;

	@Test
	public void test1() {
		System.out.println("retrying");
		i++;
		System.out.println("counter i is: "+i);
		if (i < 3) {
			System.out.println("Failure happened");
			Assert.assertTrue(false);
		}
		else{
			System.out.println("passed");
			Assert.assertTrue(true);
		}
	}

}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">


<suite name="Retry Suite">
  <listeners>
    <listener class-name="com.testNG.Retry"></listener>
  </listeners>
  <test name="Retry Test">
    <classes>
      <class name="com.testNG.RetryTest"></class>
    </classes>
  </test>
</suite>
```