TestNG – Overview of major features

TestNG is a testing framework inspired from JUnit and NUnit but introducing some new functionalities;

* Annotations.
* Run your tests in arbitrarily big thread pools with various policies available (all methods in their own thread, one thread per test class, etc...).
* Flexible test configuration.
* Support for data-driven testing (with @DataProvider).
* Support for parameters.
* Powerful execution model (no more TestSuite).
* Supported by a variety of tools and plug-ins (Eclipse, IDEA, Maven, etc...).

**Lifecycle 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.

public class CalculatorTest {

private Calculator calculator;

public CalculatorTest() {

System.*out*.println("CONSTRUCTOR");

}

@BeforeTest

public void beforeTest() {

calculator = new Calculator();

System.*out*.println("BEFORE\_TEST");

}

@AfterTest

public void afterTest() {

System.*out*.println("AFTER\_TEST");

}

@BeforeClass

public void beforeClass() {

System.*out*.println("BEFORE\_CLASS");

}

@AfterClass

public void afterClass() {

System.*out*.println("AFTER\_CLASS");

}

@BeforeMethod

public void beforeMethod() {

System.*out*.println("BEFORE\_METHOD");

}

@AfterMethod

public void afterMethod() {

System.*out*.println("AFTER\_METHOD");

}

@Test

public void testmultiply() {

System.*out*.println("TEST");

Assert.*assertEquals*(3.96, calculator.multiply(3.3, 1.2), 0.01);

}

@Test

public void testdivide() {

System.*out*.println("TEST");

Assert.*assertEquals*(2.75, calculator.divide(3.3, 1.2));

}

}

Suite xml configuration;

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

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

<test name=*"Test"*>

<classes>

<class name=*"com.ntier.util.CalculatorTest"*/>

</classes>

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

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

Attributes;

alwaysRun: For before methods (beforeSuite, beforeTest, beforeTestClass and beforeTestMethod, but not beforeGroups): If set to true, this configuration method will be run regardless of what groups it belongs to.   
For after methods (afterSuite, afterClass, ...): If set to true, this configuration method will be run even if one or more methods invoked previously failed or was skipped.

dependsOnGroups: The list of groups this method depends on

dependsOnGroups:The list of groups this method depends on

dependsOnMethods: The list of methods this method depends on

Enabled: Whether methods on this class/method are enabled

Groups: The list of groups this class/method belongs to

TestNG allows you to perform sophisticated groupings of test methods. Groups are specified in your testng.xml file and can be found either under the <test> or <suite> tag. Groups specified in the <suite> tag apply to all the <test> tags underneath.

Xml suite config, include groups

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

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

<test name=*"Test"*>

<groups>

<run>

<include name=*"firstSet"* />

<include name=*"anotherSet"* />

</run>

</groups>

<classes>

<class name=*"com.ntier.util.CalculatorTest"*/>

</classes>

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

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

**Include or Exclude Methods**

You can also exclude or include individual methods:

<test name="Test1">

<classes>

<class name="example1.Test1">

<methods>

<include name=".\*enabledTestMethod.\*"/>

<exclude name=".\*brokenTestMethod.\*"/>

</methods>

</class>

</classes>

</test>

JUnit

TestNG can run JUnit 3 and JUnit 4 tests.  All you need to do is put the JUnit jar file on the classpath, specify your JUnit test classes in the testng.classNames property and set the testng.junit property to true:

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

<suite name="Suite" parallel="false">

<test name="Test" junit="true">

<classes>

<class name="com.ntier.util.CalculatorTest"/>

</classes>

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

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

You run the xml file;

The behavior of TestNG in this case is similar to JUnit depending on the JUnit version found on the class path:

JUnit 3:

* All methods starting with test\* in your classes will be run
* If there is a method setUp() on your test class, it will be invoked before every test method
* If there is a method tearDown() on your test class, it will be invoked before after every test method
* If your test class contains a method suite(), all the tests returned by this method will be invoked

JUnit 4:

* TestNG will use the org.junit.runner.JUnitCore runner to run your tests

**Dependencies**

Hard dependencies. All the methods you depend on must have run and succeeded for you to run. If at least one failure occurred in your dependencies, you will not be invoked and marked as a SKIP in the report.

Soft dependencies. You will always be run after the methods you depend on, even if some of them have failed. This is useful when you just want to make sure that your test methods are run in a certain order but their success doesn't really depend on the success of others. A soft dependency is obtained by adding "alwaysRun=true" in your @Test annotation.

@Test(dependsOnMethods={"testmultiply"}, dependsOnGroups={"anotherGrp"})

public void testAdd() {

Assert.assertEquals(4.0, calculator1.add(2,2), 0.01);

}

@Test

public void testmultiply() {

Assert.assertEquals(4.0, calculator1.multiply(2,2), 0.01);

}

@Test(dependsOnMethods={"testmultiply", "testAdd"}, alwaysRun=true)

public void testNothing() {

System.out.println("Nothing here");

}

@Test(groups="anotherGrp")

public void testAnother() {

Assert.assertEquals(4.0, calculator1.multiply(2,2), 0.01);

}

Alternatively, you can specify your group dependencies in the testng.xml file. You use the <dependencies> tag to achieve this:

|  |  |
| --- | --- |
| <test name="My suite"> | |
| <groups> |

|  |
| --- |
| <dependencies> |
| <group name="c" depends-on="a  b" /> | |

|  |  |
| --- | --- |
| <group name="z" depends-on="c" /> | |
| </dependencies> |

|  |
| --- |
| </groups> |

</test>

|  |
| --- |
| Parameters  Test methods don't have to be parameterless. You can use an arbitrary number of parameters on each of your test method, and you instruct TestNG to pass you the correct parameters with the @Parameters annotation. |

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

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

<test name=*"Test"*>

<parameter name=*"arg1"* value=*"2.2"*/>

<parameter name=*"arg2"* value=*"2.0"*/>

<classes>

<class name=*"com.ntier.util.CalculatorTest"*/>

</classes>

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

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

@Test

@Parameters({"arg1", "arg2"})

public void testmultiply(double arg1, double arg2) {

System.out.println("TEST");

Assert.assertEquals(4.4, calculator.multiply(arg1, arg2), 0.01);

}

**DataProvider**

The Data Provider method can return one of the following two types:

* An array of array of objects (Object[][]) where the first dimension's size is the number of times the test method will be invoked and the second dimension size contains an array of objects that must be compatible with the parameter types of the test method. This is the cast illustrated by the example above.
* An Iterator<Object[]>. The only difference with Object[][] is that an Iterator lets you create your test data lazily. TestNG will invoke the iterator and then the test method with the parameters returned by this iterator one by one. This is particularly useful if you have a lot of parameter sets to pass to the method and you don't want to create all of them upfront.

private Calculator calculator;

@BeforeTest

public void beforeTest() {

calculator = new Calculator();

}

@Test(dataProvider="provideNumbers")

public void testmultiply(double arg1, double arg2, double expected) {

System.out.println("TEST");

Assert.assertEquals(expected, calculator.multiply(arg1, arg2), 0.01);

}

@DataProvider(name = "provideNumbers")

public Object[][] provideData() {

return new Object[][] {

{ 2, 2, 4.0 },

{ 3, 3, 9.0 },

{ 4, 5, 20.0 }

};

}

**Run tests in Parallel**

Data providers can run in parallel with the attribute parallel:

@DataProvider(parallel = true)

You can instruct TestNG to run your tests in separate threads in various ways.

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

<suite name="Suite" parallel="methods" thread-count="5">

<test name="Test">

<groups>

<run>

<include name="anotherSet" />

</run>

</groups>

<classes>

<class name="com.ntier.util.CalculatorTest"/>

<class name="com.ntier.util.AnotherTest"/>

</classes>

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

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

The parallel attribute on the <suite> tag can take one of following values:

<suite name="My suite" parallel="methods" thread-count="5">

<suite name="My suite" parallel="tests" thread-count="5">

<suite name="My suite" parallel="classes" thread-count="5">

<suite name="My suite" parallel="instances" thread-count="5">

parallel="methods": TestNG will run all your test methods in separate threads. Dependent methods will also run in separate threads but they will respect the order that you specified.

parallel="tests": TestNG will run all the methods in the same <test> tag in the same thread, but each <test> tag will be in a separate thread. This allows you to group all your classes that are not thread safe in the same <test> and guarantee they will all run in the same thread while taking advantage of TestNG using as many threads as possible to run your tests.

parallel="classes": TestNG will run all the methods in the same class in the same thread, but each class will be run in a separate thread.

parallel="instances": TestNG will run all the methods in the same instance in the same thread, but two methods on two different instances will be running in different threads.

**Listeners**

There are several interfaces that allow you to modify TestNG's behavior. These interfaces are broadly called "TestNG Listeners".  Here are a few listeners:

* IAnnotationTransformer
* IAnnotationTransformer2
* IHookable
* IInvokedMethodListener
* IMethodInterceptor
* IReporter
* ISuiteListener
* ITestListener

Here is how you can define listeners in your testng.xml file:

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

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

<listeners>

<listener class-name=*"com.ntier.util.MyInterceptor"* />

</listeners>

<test name=*"Test"*>

<classes>

<class name=*"com.ntier.util.CalculatorTest"*/>

</classes>

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

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

public class MyInterceptor implements IMethodInterceptor{

@Override

public List<IMethodInstance> intercept(List<IMethodInstance> methods,

ITestContext context) {

System.*out*.println("Intercepted");

for (IMethodInstance iMethodInstance : methods) {

System.*out*.println(iMethodInstance.getMethod().getMethodName());

}

return methods;

}

}

@Listeners({MyListener.class})

public class CalculatorTest {

public class MyListener implements ITestListener{

@Override

public void onTestStart(ITestResult result) {

System.*out*.println(result.getStartMillis());

}

@Override

public void onTestSuccess(ITestResult result) {

System.*out*.println(result.getEndMillis());

}

@Override

public void onTestFailure(ITestResult result) {

System.*out*.println("FAIL");

}

@Override

public void onTestSkipped(ITestResult result) {

System.*out*.println("SKIP");

}

@Override

public void onTestFailedButWithinSuccessPercentage(ITestResult result) {

System.*out*.println("FAILED");

}

@Override

public void onStart(ITestContext context) {

System.*out*.println("START");

}

@Override

public void onFinish(ITestContext context) {

System.*out*.println("FINISH");

}

}