Unit Testing J2EE platform components with JUnit and JUnitEE frameworks in IBM Rational Application Developer Version 6.0.2: Part 1. Unit testing Java and EJB applications

Abraham WoldeMichael
John Diamond

January 23, 2007

This is Part 1 of a two-part series of articles. It demonstrates how you can use an open source test framework, such as JUnit and JUnitEE, to unit test Java and EJB applications by using IBM Rational Application Developer Version 6 in an IBM WebSphere Application Server Version 6 environment.

View more content in this series

Reasons for unit testing J2EE components

As we move to a distributed, multi-tiered and heterogeneous computing, the Java™ 2 platform, Enterprise Edition (J2EE™) technology, has become the most popular for developing component-based multi-tiered, distributed enterprise application. The J2EE technology has integrated application clients and applets, Web components (JSP and servlets), and Enterprise JavaBeans™ (EJB™) components. Web and EJB components run on an application server, such as the IBM® WebSphere® Application Server software. As people use J2EE technology to develop large, complex enterprise applications, it is inevitable that different components are assembled cohesively to produce one integrated application.

Before this assembly takes place, it is necessary and critical to properly unit test each component independently. Unit testing each component in isolation significantly reduces bugs and helps to ensure high-quality software. Some developers or testers might argue that unit testing J2EE components is too time-consuming, labor intensive, and error prone. Although it is beyond the scope of this article to delve into the details of unit testing various J2EE components, it will show you how easy and effective it is to unit test the EJB components with the JUnit and JUnitEE test frameworks.

Unit testing is a piece of code written by a developer that exercises a very small, specific area of functionality of the code being tested. Usually, a unit test exercises some particular method in a particular context, thus it falls in the broader category of white box testing. Since the need to unit test J2EE components has become apparent, developers have been exploiting the benefits of the
JUnit test framework to perform these tests. The JUnit test framework, originally written by Enrich Gamma and Kent Beck, is a framework for unit testing a client-side Java application. It offers several benefits:

- Simple framework for writing automated, self-verifying tests in Java
- Support for test assertions
- Test suite development
- Immediate test reporting

JUnit provides a text-based command line, as well as AWT-based and Swing-based graphical test reporting mechanisms. The IBM Rational Application Developer integrated development platform (IDE) includes JUnit.

However, for developers who want to unit test within the application server's containers and display results in HTML or XML, the JUnit framework is limited and ineffective. That is why JUnitEE, an open source development community, has extended the JUnit capabilities by implementing the JUnitEE test framework that enables you to run JUnit unit tests within the application server's containers.

About this series of articles

This series of two articles explains how you can use IBM tools and technologies to unit test Java applications, EJBs and Web services by using JUnit and JUnitEE test frameworks.

- **Part 1** (this article) starts with an overview of JUnit and JUnitEE, and then demonstrates the use of JUnit and JUnitEE within Rational Application Developer IDE to unit test a simple Java application and a stateless session bean. First, we present a brief overview of the structure of JUnit and JUnitEE followed by how to develop, configure and setup unit tests in JUnit and JUnitEE test framework. Finally, we demonstrate how to deploy and execute out unit tests in application server environment.

- **Part 2** discusses unit testing Web services by using the JUnit and JUnitEE test frameworks within Rational Application Developer platform, as well as how to deploy and execute unit tests in the IBM WebSphere Application Server Version 6 environment.

Prerequisites

To benefit from this article, you need a basic understanding of Web services and of Rational Application Developer. The IBM® developersWorks® Web site and Related topics at the end of this article provide introductory materials.

This tutorial was developed and tested using using the final release of JUnitEE Version 1.10 and Rational Application Developer Version V 6.0.2. You can download the trial version of Rational Application Developer and the binary files for this example (see Related topics).

JUnit test framework

JUnit is now the unofficial standard for unit testing Java-based applications. Although the Junit.org Web site provides more compressive information and tutorials (see Related topics), this section gives you an overview of the JUnit test framework. The principal purpose of the set up APIs that
comprise JUnit is to make writing Java unit test cases fast and easy. At minimum, a JUnit test case has the common structure shown in code Listing 1.

**Listing 1. JUnit common structure**

1. Import junit.framework.TestCase; 2. 3. Public class AddJavaTest extends TestCase { 4. 5. protected void setUp() throws Exception 6. { 7. // create some object 8. } 9. protected void tearDown() throws Exception 10. { 11. // release any recourse that was created in 12. setup() 13. } 14. public AddJavaTest (String name){ 15. super (name); 16. 17. public void testSimpleAddition (){ 18. assertTrue (expect == actual); 19. } }

As Listing 1 shows in line 3, all Java™ cases must extend the junit.framework.TestCase which is the core class for JUnit. In line 5, TestCase.setUp() is overridden to initialize or instantiate the object under test. Conversely, in line 9, TestCase.tearDown() is overridden to release any allocated resource. In line 14, a test case must have a single string parameter constructor that passes the argument to its parent class (TestCase) for the purpose of displaying the test case name in a log.

A test case method must be declared public void with no formal parameters. In addition, it is desirable to prefix the test method name with “test” so that the test runner will execute all methods automatically. Lastly, in line 18 an assertion statement is issued to determine the success or failure of a test case. The method assert compares an expected value to an actual value for the particular test scenario. You can use the fail() method to force a test case to fail, for example if you want to force a timeout of an operation. JUnit provides an additional mechanism to determine the success or failure of a test case. Table 1 shows a sampling of different assert and fail method signatures.

**Table 1. Assert methods**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>static void</td>
<td>assertEquals (boolean expected, boolean actual)</td>
</tr>
<tr>
<td></td>
<td>Asserts that two booleans are equal.</td>
</tr>
<tr>
<td>static void</td>
<td>assertFalse (boolean condition)</td>
</tr>
<tr>
<td></td>
<td>Asserts that a condition is false.</td>
</tr>
<tr>
<td>static void</td>
<td>assertNotNull (java.lang.Object object)</td>
</tr>
<tr>
<td></td>
<td>Asserts that an object isn't null.</td>
</tr>
<tr>
<td>static void</td>
<td>notSame (java.lang.Object expected, java.lang.Object actual)</td>
</tr>
<tr>
<td></td>
<td>Asserts that two objects refer to the same object.</td>
</tr>
<tr>
<td>static void</td>
<td>assertNotNull (java.lang.Object object)</td>
</tr>
<tr>
<td></td>
<td>Asserts that an object is null.</td>
</tr>
<tr>
<td>static void</td>
<td>assertSame (java.lang.Object expected, java.lang.Object actual)</td>
</tr>
<tr>
<td></td>
<td>Asserts that two objects refer to the same object.</td>
</tr>
<tr>
<td>static void</td>
<td>assertTrue (boolean condition)</td>
</tr>
<tr>
<td></td>
<td>Asserts that a condition is true.</td>
</tr>
<tr>
<td>static void</td>
<td>fail (java.lang.String message)</td>
</tr>
<tr>
<td></td>
<td>Fails a test with the given message.</td>
</tr>
</tbody>
</table>

Unit Testing J2EE platform components with JUnit and JUnitEE frameworks in IBM Rational Application Developer Version 6.0.2: Part 1. Unit testing Java and EJB applications
JUnit provides a `TestRunner` class to execute test cases. There are various ways to execute tests. Test reports are displayed using graphics and text. To get the most popular graphical results, use `junit.swingui.TestRunner` and `junit.awtgui.TestRunner`. For the less-popular text-based results, use `junit.textui.TestRunner`, instead.

Additionally, JUnit Test cases may be run from Rational Application Developer's IDE or automated in an Ant build script.

JUnit also provides a way to group tests into a suite by using the `junit.framework.TestSuite` class. A test suite is a composite of related tests that you can run during the same session. There are two convenient ways that JUnit runs test cases:

- With the first way, you pass the test class to the TestSuite constructor by using this command:
  ```java
  TestSuite suite = new TestSuite(testAddJava.class)
  ```
  In this case, the TestRunner extracts all test methods that have a test prefix, and then executes each test case automatically.

- The alternative way is add each test case by using the TestSuite.addTest method:
  ```java
  TestSuite suite = new TestSuite(); suite.addTest(new AddJavaTest("testSimpleAddition"));
  ```

### JUnitEE test framework

JUnit provides an effective and easy way to unit test client-side Java applications, but it has some limitations; therefore, testing in each application server container becomes a tedious process. The IBM Rational Application Developer platform features a Web-based Universal Test Client (UTC) that provides a seamless and integrated mechanism for unit testing Enterprise JavaBeans (EJBs). However, the Rational Application Developer UTC is an interactive unit-testing mechanism, thus it falls short when it comes to automating unit tests.

The JUnitEE test framework addresses these limitations, as well as the tedious process. This framework extends the standard JUnit so that it can execute unit tests in an application server container. It is configured in the J2EE Web module of a unit test application, and it uses a TestRunner to output HTML or XML test results. It also includes a TestServlet for an entry point to JUnit test cases. As result, according to JUnitEE.org, building your test harness as a standard J2EE Web offers several benefits:

- Tests are packaged in a J2EE Web module (in a WAR file), which is easy to deploy and execute.
• Test cases look just like production code, and they can use the same Java beans that you use as a facade for your EJBs.
• Tests can be automated by using an Ant script.

**Example of using JUnit and JUnitEE for unit testing**

This section presents an example of how you can take advantage of the JUnit and JUnitEE test frameworks to unit test a simple stateless session EJB that was developed in the Rational Application Developer IDE. The EJB simply adds two numbers and returns the sum of those numbers. Although this is a trivial function, it includes everything necessary to illustrate this example.

For simplicity, this example is a unit test of a stateless session bean. However, you can unit test other types of EJBs using JUnit and JUnitEE test framework, too. The simplified approach here includes these steps:

1. Create a simple Java bean calculator.
2. Create an EJB for a basic calculator.
3. Generate an EJB client.
4. Develop JUnit test cases to test both the Java application and the calculator EJB.
5. Create and configure a JUnitEE test module.
6. Deploy and execute test cases within the WebSphere Application Server environment.

You can download the source code for all examples in this article (see [Related topics](#)).

**Step 1. Create a simple Java bean calculator**

The first example is a simple Java application that adds two numbers and returns their sum. Code Listing 2 shows the class implementation.

**Listing 2. Implementation of the Basic Calculator Java Bean**

```java
package calc; public class BasicCalculator { public double addTwoNumbers(double first, double second) { return first + second; } }
```

**Step 2. Create an EJB for a basic calculator**

Use Rational Application developer to generate a stateless session bean called BasicCalculator.

Code Listing 3 contains a sample of what the EJB should look like.

The `BasicCalculatorBean` EJB class contains one business method, `addTwoNumbers`, for calculator function. This business method is added to a remote interface to give clients access to this method. EJBs can be remote, distributed objects. The Rational Application Developer IDE generates client artifacts to access remote objects easily. The test application may be in a separate Enterprise Archive (EAR) so we use remote rather than local interfaces to the EJB.
Listing 3. Implementation of the Basic Calculator EJB

```java
package ejbs; /** * Bean implementation class for Enterprise Bean: BasicCalculator */ public class BasicCalculatorBean implements javax.ejb.SessionBean { private javax.ejb.SessionContext mySessionCtx; public javax.ejb.SessionContext getSessionContext() { return mySessionCtx; } public void setSessionContext(javax.ejb.SessionContext ctx) { mySessionCtx = ctx; } public void ejbCreate() throws javax.ejb.CreateException { } public void ejbActivate() { } public void ejbPassivate() { } public void ejbRemove() { } public double addTwoNumbers(double first, double second) { return first + second; }
}
```

Step 3. Generate an EJB client

In this step, follow the Rational Application Developer’s smart guides to generate the client artifacts.

Step 4. Develop JUnit test cases to test the Java and calculator EJB

After you have implemented the Java Bean Calculator, Basic Calculator EJB and client EJB, you're ready to write JUnit test cases.

Unit testing a Java application with JUnit

Here, you take advantage of the JUnit test structure outlined in Listing 1 to write three test methods to unit test the BasicCalculator Java Bean and EJB.

Listing 4 unit tests illustrate various assert methods to perform positive and negative unit testing. It has the following test methods:

- The `testSimpleAddition()` method contains two assertions. The first one asserts that the calculator instance object is not null, i.e. it exists. The second one verifies that the calculator is correctly adding 2 plus 2 by comparing the result to the expected result, 4.
- The `testSimpleAdditionNotSame()` method demonstrates negative testing to ensure that two distinct values are not the same. That is, the sum of two numbers (2+2) does not equal 5.
- The `testDesignedToFail()` method demonstrates what a failed test case looks like in the JUnitEE framework. In this test case, two arguments, 1 and 1 are summed by the BasicCalculator, and then the result is compared to 3. This test fails. The output shows you exactly where and why it failed.

Listing 4. JUnit Test for Calc Java Bean

```java
package calc.test; import calc.BasicCalculator; import junit.framework.TestCase; public class BasicCalculatorTest extends TestCase { BasicCalculator aBasicCalculator = null; /* * Setup before each test case */ protected void setUp() throws Exception { super.setUp(); aBasicCalculator = new BasicCalculator(); } public void testSimpleAdditionNotSame() throws Exception { double result = aBasicCalculator.addTwoNumbers(2, 2); assertNotSame("2+2 does not = 5", new Double(result), new Double(5)); } public void testDesignedToFail() throws Exception { double result = aBasicCalculator.addTwoNumbers(1, 1); assertTrue("1 + 1 = 3", result == 3); } public void testSimpleAddition() throws Exception { assertEquals("2+2=4", aBasicCalculator.addTwoNumbers(2, 2), 4); }
```
Listing 5. JUnit source code to test Basic Calculator EJB

```java
package calculator.test; import java.rmi.RemoteException; import com.ibm.etools.service.locator.ServiceLocatorManager; import ejbs.BasicCalculator; import ejbs.BasicCalculatorHome; import junit.framework.TestCase; public class BasicCalculatorTest extends TestCase { BasicCalculator aBasicCalculator = null; /* * Setup before each test case */ protected void setUp() throws Exception { super.setUp(); aBasicCalculator = createBasicCalculator(); } protected void tearDown() throws Exception { aBasicCalculator = null; } public void testSimpleAddition() throws Exception { assertTrue("Calculator instance is not null", aBasicCalculator != null); double result = aBasicCalculator.addTwoNumbers(2, 2); assertTrue("2+2=4", result == 4); } public void testSimpleAdditionNotSame() throws Exception { double result = aBasicCalculator.addTwoNumbers(2, 2); assertNotNull("2+2 does not = 5", new Double(result), new Double(5)); } public void testDesignedToFail() throws Exception { double result = aBasicCalculator.addTwoNumbers(1, 1); assertTrue("1 + 1 = 3", result == 3); } /* * Rational generated code snippet to access EJB */ private BasicCalculator createBasicCalculator() { BasicCalculatorHome aBasicCalculatorHome = (BasicCalculatorHome) ServiceLocatorManager .getRemoteHome(STATIC_BasicCalculatorHome_REF_NAME, STATIC_BasicCalculatorHome_CLASS); try { if (aBasicCalculatorHome != null) { return aBasicCalculatorHome.create(); } } catch (javax.ejb.CreateException ce) { ce.printStackTrace(); } catch (RemoteException re) { re.printStackTrace(); } return null; } private final static String STATIC_BasicCalculatorHome_REF_NAME = "ejb/BasicCalculator"; private final static Class STATIC_BasicCalculatorHome_CLASS = BasicCalculatorHome.class; }
```

Unit testing EJBs with JUnit

Because EJBs are managed by their containers they must be looked up in a JNDI directory. Unit testing can be a tedious and difficult process. Fortunately, Rational Application Developer provides code snippets that perform the task of creating an EJB instance for the developer.

The calculator methods of the stateless session EJB are the same as the Java bean implementation. Therefore, you can reuse the underlying JUnit test cases for your EJB unit tests. But you must make a few changes in the `setUp()` method to locate and instantiate the EJB object. Do you recall that the purpose of a `setUp()` method is to initialize the object under test. In this method, a call to `createBasicCalculator()` is made, which returns a EJB instance of the calculator. The code in this method was generated by Rational Application Developer. The method is shown below in Listing 6.

Listing 6. Locating and accessing the EJB home

```java
private BasicCalculator createBasicCalculator() { BasicCalculatorHome aBasicCalculatorHome = (BasicCalculatorHome) ServiceLocatorManager .getRemoteHome(STATIC_BasicCalculatorHome_REF_NAME, STATIC_BasicCalculatorHome_CLASS); try { if (aBasicCalculatorHome != null) { return aBasicCalculatorHome.create(); } } catch (javax.ejb.CreateException ce) { ce.printStackTrace(); } catch (RemoteException re) { re.printStackTrace(); } return null; } private final static String STATIC_BasicCalculatorHome_REF_NAME = "ejb/BasicCalculator"; private final static Class STATIC_BasicCalculatorHome_CLASS = BasicCalculatorHome.class; }
```

Step 5. Create and configure the JUnitEE test module

To execute your JUnit test cases within the WebSphere Application Server software, you can take advantage of the JUnitEE framework. As mentioned previously, JUnitEE is an extension of the JUnit framework that executes tests in an application server and uses a TestRunner interface to display test results in either HTML or XML format. This section shows how to use both test frameworks together to create a single deployable Web archive (WAR) file that contains your tests and a JUnitEE test servlet to run your Java and EJB unit test cases.

Creating and configuring a JUnitEE test module in Rational Application Developer software is an intuitive process.
1. First, create a dynamic Web project in Rational Application Developer, named BasicAddJUnitEEWeb.
2. Place junit.jar and junitee.jar files in the project classpath by placing them in the WEB-INF/lib directory of the Web project.
3. Create a jar file, named BasicAddUnitTest.jar, containing the two Test case classes. Copy the jar file, BasicAddUnitTest.jar into the WEB-INF/lib directory.
4. Include the XML code shown below in Listing 7 in the web.xml deployment descriptor.

Listing 7. JUnitEE test servlet mapping in the web.xml Web deployment descriptor

```
1. &lt;servlet&gt; 2. &lt;servlet-name&gt;JUnitEEServlet&amp;lt;/servlet-name&amp;gt; 3. &lt;display-
name&gt;JUnitEEServlet&amp;lt;/display-name&amp;gt; 4. &lt;servlet-class&gt;org.junitee.servlet.JUnitEEServlet 5.
 &lt;/servlet-class&gt; 6. &lt;init-param&gt; 7. &lt;param-name&gt;searchResources&amp;lt;/param-name&amp;gt; 8.
 &lt;param-value&gt;BasicAddUnitTest.jar&amp;lt;/param-value&gt; &lt;/init-param&gt; 9. &lt;/servlet&gt; 10.
 &lt;servlet-mapping&gt; 11. &lt;servlet-name&gt;JUnitEEServlet&amp;lt;/servlet-name&amp;gt; 12. &lt;url-pattern&gt;/
 TestServlet/*&lt;/url-pattern&gt; &lt;/servlet-mapping&gt;
```

JUnitEE test cases can be executed using either smart or basic mode within the application server environment. Using the smart mode, the JUnitEE TestRunner automatically locates all test classes that end with "Test" or "Tests" and executes them. The smart mode is configured by specifying the name of the JAR file in the deployment descriptor as outlined in Listing 7, lines 6-8. If there are more than one JAR file, delimit them with a comma. During the servlet initialization, the container activates the servlet by calling the init method once. The init method initializes a set of parameters for the servlet instance. The parameter name searchResources is declared in line 7, and in line 8, the value BasicAddUnitTest.jar is assigned to this parameter. As a result, whenever the container activates the JunitEEServlet, JUnitEE looks for all test classes in the BasicAddUnitTest.jar file.

An alternative to the smart test mode is basic test mode. In the basic test mode, the TestRunner executes only test classes that are specified in WEB-INF/testCase.txt file, which is a simple file that contains one JUnit test class name per line, as Listing 8 shows.

Listing 8. Text file with JUnit test package and class names

```
```

In the final step, before deploying, make sure that the build path and all JAR dependencies are in place. In addition, make sure that the EAR file contains all of the necessary utility JAR files, as well as the JUnitEE Web module, packaged in a WAR file. When this is complete, proceed to the next section to deploy your tests in the WebSphere Application Server environment.

Step 6. Deploy and execute test cases in WebSphere Application Server

Thus far, you have completed all of the groundwork, and you have an EAR file that contains your configured and packaged JUnitEE test module (as a WAR file).

After you have deployed the EAR file on the application server, launch your browser and point to the JUnitEE servlet URL. The URL format is http://&amp;lt;hostname&amp;gt;:&amp;lt;portnumber&amp;gt;/
BasicAddJUnitEEWeb/TestServlet, where BasicAddJUnitEEWeb is the name of the Web context,
the name of the dynamic Web project and TestServlet is the JUnitEE TestRunner servlet. This example uses http://localhost:9080/BasicAddJUnitEEWeb/TestServlet. Figure 1 shows a conceptual view of how information is exchanged between J2EE components.

**Figure 1. Component interaction**

![Component interaction diagram]

A successful run displays all of the tests in JUnitEE TestRunner view, as depicted by Figure 2. The screen shows different ways to execute your unit test. Select the test option that you prefer or enter the name of your test suite in the field provided to start running test cases.

**Figure 2. JUnitEE TestRunner screen**

![JUnitEE TestRunner screenshot]

If there are no errors with configurations or JNDI name lookup, the test runner executes selected tests and displays their result in the JUnit Test Results window. This view provides comprehensive information on executed tests. If a test passes, it is denoted with a green check mark. If it fails, it is denoted with a red X. Clicking the X provides stack trace information for debugging. We deliberately caused one of the tests to fail to show how JUnit displays a failed test, as illustrated by Figure 3.
Summary

Examples in this article demonstrated how you can quickly and easily develop, configure, and execute unit tests of Java and EJB applications by using IBM technologies and open source JUnit and JUnitEE test frameworks to deliver accurate, higher-quality J2EE applications.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample code</td>
<td>sourceCode.zip</td>
<td>256 KB</td>
</tr>
</tbody>
</table>
Related topics

- JUnit framework
- JUnitEE expands the capabilities of the JUnit framework
- Free trial version Rational Application Developer

© Copyright IBM Corporation 2007

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)