

```

    private String varName;
    private String item;

    public void setVar(String varName)
    {
        this.varName = varName;
    }

    public String getVar()
    {
        return this.varName;
    }

    public void setItem(String item)
    {
        this.item = item;
    }

    public String getItem()
    {
        return this.item;
    }

    public int doStartTag() throws JspException
    {
        // Evaluate the item attribute (an EL expression) which
        // must result in a DynaBean object.
        DynaBean bean =
            (DynaBean) ExpressionEvaluatorManager.evaluate(
                "item", getItem(), DynaBean.class, this,
                this.pageContext);

        // Get the DynaBean meta-properties and store them in the
        // variable pointed to by the "var" attribute.
        this.pageContext.setAttribute(getVar(),
            bean.getDynaClass().getDynaProperties());
        return SKIP_BODY;
    }

    public int doEndTag() throws JspException
    {
        return EVAL_PAGE;
    }
}

```

To be consistent with the JSTL library (<http://java.sun.com/products/jsp/jstl/>), you implement the content of the `item` attribute as an Expression Language (EL) expression. EL is the expression language used in the JSTL tags. It's very convenient for passing variables from one JSP tag to another. For example, in the tag shown before listing 10.4, you pass the `dynaBean` variable (`${dynaBean}`) to the

<d:properties> tag. This is a very simple usage of EL, but you can use any valid EL expression in this item attribute. The beauty of it is that you need only one line of code to implement this functionality (as shown in the `doStartTag` method).

### 10.4.2 Testing the custom tag

Your next challenge is to unit-test this custom tag. What do you need to test? Well, you need to verify whether this tag correctly stores the properties of the DynaBean object passed in `PageContext` scope. See the `testDoStartTag` method in listing 10.5 for this example.

**Listing 10.5** Unit tests for `DynaPropertiesTag`

```
package junitbook.pages;

import org.apache.cactus.JspTestCase;
import org.apache.commons.beanutils.DynaProperty;
import org.apache.commons.beanutils.BasicDynaClass;
import org.apache.commons.beanutils.DynaBean;

import javax.servlet.jsp.tagext.Tag;

public class TestDynaPropertiesTag extends JspTestCase
{
    private DynaBean createDynaBean() throws Exception
    {
        DynaProperty[] props = new DynaProperty[] {
            new DynaProperty("id", String.class),
            new DynaProperty("responsetime", Long.class)
        };
        BasicDynaClass dynaClass = new BasicDynaClass("requesttime",
            null, props);

        DynaBean bean = dynaClass.newInstance();
        bean.set("id", "12345");
        bean.set("responsetime", new Long(500));

        return bean;
    }

    public void testDoStartTag() throws Exception
    {
        DynaPropertiesTag tag = new DynaPropertiesTag();
        tag.setPageContext(pageContext);
        pageContext.setAttribute("item", createDynaBean());
        tag.setItem("${item}");
        tag.setVar("var");

        int result = tag.doStartTag();
    }
}
```

Create instance  
of tag to test

Set PageContext to initialize tag

Set  
environmental  
parameters for tag

```

        assertEquals(Tag.SKIP_BODY, result);
        assertTrue(pageContext.getAttribute("var")
            instanceof DynaProperty[]);

        DynaProperty[] props = (DynaProperty[])
            pageContext.getAttribute("var");
        assertEquals(props.length, 2);
    }
}

```

**Assert server-side  
environment after  
execution**

The tag is simple and implements a single life cycle `doStartTag` method. The full life cycle of a simple tag (a tag that does not need to manipulate its body) is as follows:

```

ATag t = new ATag();
t.setPageContext(...);
t.setParent(...);
t.setProperty1(value1);
[...]
t.setPropertyN(valueN);
t.doStartTag();
t.doEndTag();
t.release();

```

In the example (listing 10.5), you instantiate the tag, call `setPageContent`, set the needed properties, and call `doStartTag`. Because the tag isn't a nested tag, you don't call `setParent`, because you don't need to fetch anything from a superclass. (For more about collaborating tags, see section 10.4.4.) You also don't implement `release`. If you did, you'd write tests for it the same way you wrote a unit test for the `doStartTag` method. The `doEndTag` method implementation is "too simple to break" (see listing 10.4), so you don't even need to test it!

### 10.4.3 Unit-testing tags with a body

So far, we've demonstrated how to unit-test simple tags (tags without a body). Let's see now how you can unit-test a tag with a body. A *body tag* is a tag that encloses content, which can be text or other tags. Let's take the example of a `<sortHtmlTable>` tag, which sorts column elements in an HTML table:

```

<table>

  <d:sortHtmlTable order="ascending" column="1">
    <c:forEach items="{customers}" var="customer">
      <tr>
        <td><c:out value="{customer.getLastName()}" /></td>
        <td><c:out value="{customer.getFirstName()}" /></td>
      </tr>
    </c:forEach>
  </d:sortHtmlTable>

</table>

```

A quick implementation that skips the details of the sorting algorithm is shown in listing 10.6.

**Listing 10.6** Skeleton for SortHtmlTableTag (leaves out the sort algorithm)

```
package junitbook.pages;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.BodyContent;
import javax.servlet.jsp.tagext.BodyTagSupport;

public class SortHtmlTableTag extends BodyTagSupport
{
    private String sortOrder = "ascending";
    private int sortColumn = 1;

    public void setOrder(String sortOrder)
    {
        this.sortOrder = sortOrder;
    }

    public void setColumn(int sortColumn)
    {
        this.sortColumn = sortColumn;
    }

    public int doAfterBody() throws JspException
    {
        // The body content has been evaluated, now we need to
        // parse it and sort the table lines.
        BodyContent body = getBodyContent();
        String content = body.getString();
        body.clearBody();
        try
        {
            getPreviousOut().print(sortHtmlTable(content));
        }
        catch (Exception e)
        {
            throw new JspException("Failed to sort body content ["
                + content + "]", e);
        }

        return SKIP_BODY;
    }

    private String sortHtmlTable(String content)
    {
        // Algorithm skipped :-)
        return content;
    }
}
```

**Retrieve body content as a String**

**Clear body content to replace it with transformed content**

**Output new modified body content**

You have two options to test this: reproduce some parts of the container life cycle and write a focused unit test that exercises the `doAfterBody` method, or use a Cactus helper class called `JspTagLifecycle` that reproduces the full container life cycle for you. The latter solution provides more coarse-grained tests. All the tag life cycle methods (`doStartTag`, `doInitBody`, and so forth) are called in sequence. You can perform assertions only once all the methods have been called.

### **Testing a tag life cycle method**

In order to write focused tests for a given method of the tag life cycle, you have to understand what you need to set up prior to calling that method. Thus, you need to understand the *body tag container* life cycle (the order in which the container calls the different tag methods):

```
ATag t = new ATag();
t.doStartTag();
out = pageContext.pushBody();
t.setBodyContent(out);
// perform any initialization needed after body content is set
t.doInitBody();
t.doAfterBody();
// while doAfterBody returns EVAL_BODY_BUFFERED we
// iterate body evaluation
[...]
t.doAfterBody();
t.doEndTag();
t.pageContext.popBody();
t.release();
```

In the `SortHtmlTableTag` example, you only needed a `doAfterBody` method. Reading the previous life cycle, you need to instantiate the tag, call `pageContext.pushBody` to create a `BodyContent` object, assign the body content to the tag object, put some content in the `BodyContent` object by calling one of its print methods, and call the `doAfterBody` method. To send back the generated output in the HTTP response, you must also be sure to call `pageContext.popBody` when you're finished. Listing 10.7 integrates all these steps into a test case.

#### **Listing 10.7 Cactus tests for `SortHtmlTableTag` using a fine-grained approach**

```
package junitbook.pages;

import javax.servlet.jsp.tagext.BodyContent;
import javax.servlet.jsp.tagext.Tag;

import org.apache.cactus.JspTestCase;
import org.apache.cactus.WebResponse;
```

```

public class TestSortHtmlTableTag extends JspTestCase
{
    private SortHtmlTableTag sortTag;
    private BodyContent bodyContent;

    protected void setUp()
    {
        sortTag = new SortHtmlTableTag();
        sortTag.setPageContext(pageContext);
        bodyContent = pageContext.pushBody();
        sortTag.setBodyContent(bodyContent);
    }

    protected void tearDown()
    {
        pageContext.popBody();
    }

    public void testDoAfterBody() throws Exception
    {
        bodyContent.print("<tr><td>Vincent</td></tr>"
            + "<tr><td>Ted</td></tr>");

        int result = sortTag.doAfterBody();

        assertEquals(Tag.SKIP_BODY, result);
    }

    public void endDoAfterBody(WebResponse response)
    {
        String expectedString = "<tr><td>Ted</td></tr>"
            + "<tr><td>Vincent</td></tr>";

        assertEquals(expectedString, response.getText());
    }

    // Other tests to write to be complete: empty body content,
    // already ordered list, only one line in table to order,
    // bad content (i.e. not a table).
}

```

- ❶ Factorize the tag life cycle methods common to all tests in `setUp` and `tearDown`.
- ❷ Define the input data you will feed to the tag.
- ❸ Verify that the result is sorted. Of course, because you have not yet implemented the sorting algorithm (the implementation in listing 10.6 simply returns the content, untouched), the test will fail. To make it work, you need to either implement the sorting algorithm or modify the `expectedString` to put *Vincent* before *Ted*.

**Testing all the life cycle methods at once**

Cactus provides a helper class called `JspTagLifecycle` that automatically performs all the initialization steps—like setting the page context and creating body content—and calls the different life cycle methods in the right order. It also provides some expectation methods to verify whether the tag body was evaluated, and so forth.

Rewriting the previous example (`TestSortHtmlTableTag`) leads to listing 10.8 (changes from listing 10.7 are shown in bold).

**Listing 10.8** Cactus tests for `SortHtmlTableTag` using the `JspTagLifecycle` approach

```
package junitbook.pages;

import org.apache.cactus.JspTestCase;
import org.apache.cactus.WebResponse;
import org.apache.cactus.extension.jsp.JspTagLifecycle;

public class TestSortHtmlTableTag2 extends JspTestCase
{
    private SortHtmlTableTag sortTag;
    private JspTagLifecycle lifecycle;

    protected void setUp()
    {
        sortTag = new SortHtmlTableTag();
        lifecycle = new JspTagLifecycle(pageContext, sortTag); ❶
    }

    public void testDoAfterBody() throws Exception
    {
        lifecycle.addNestedText("<tr><td>Vincent</td></tr>"
            + "<tr><td>Ted</td></tr>"); ❷
        lifecycle.expectBodyEvaluated(); ❸
        lifecycle.invoke(); ❹
    }

    public void endDoAfterBody(WebResponse response)
    {
        String expectedString = "<tr><td>Ted</td></tr>"
            + "<tr><td>Vincent</td></tr>";

        assertEquals(expectedString, response.getText());
    }

    // Other tests to write to be complete: empty body content,
    // already ordered list, only one line in table to order,
    // bad content (i.e. not a table).
}
```

- ❶ Create the `JspTagLifecycle` helper, passing to it the `PageContext` object and the tag instance you're testing.
- ❷ ❸ Configure the `JspTagLifecycle` object and tell it what to expect. In ❷, you tell it that the tag contains some nested text; in ❸, you tell it that you expect the tag body to be evaluated once (the `doAfterBody` tag method is called once and only once).
- ❹ When you call `JspTagLifecycle.invoke`, the `JspTagLifecycle` object executes the standard tag life cycle, calling the life cycle methods one after another. It also verifies that the expectations are met.

The advantage of this approach is that there is minimal setup in the test and the full tag life cycle is executed. However, you still benefit from the unit-test approach because you can perform server-side assertions after you have called `lifecycle.invoke`, such as verifying that an attribute the tag is supposed to set is effectively set, and so forth.

#### 10.4.4 Unit-testing collaboration tags

A *collaboration tag* is nested within another tag and needs to communicate with the parent tag in order to retrieve some value. Unit-testing a collaboration tag requires that you call `setParent` on the tag in your test case. Once this is done, the tag will correctly work when it invokes `findAncestorWithClass` or `getParent` to find the parent tag.

For example, you could write the following in the `JspTestCase` class:

```
MyParentTag parentTag = new MyParentTag();
parentTag.setXXX(value);
MyChildTag childTag = new MyChildTag();
childTag.setParent(parentTag);
[...]
```

### 10.5 Unit-testing taglibs with mock objects

---

You have seen that it isn't possible to unit-test a JSP purely with the mock-objects approach because a JSP isn't Java code. On the other hand, tag libraries (taglibs) are pure Java code, which should make them easy to test with mock objects. However, it isn't that easy.... You need a mock object for `PageContext`, which is an *abstract* Java class.

In chapters 8 and 9, you learned that it's easy to use mock objects when they are generated on the fly by frameworks such as EasyMock and DynaMock. However, these frameworks use the JDK 1.3+ Dynamic Proxy feature, which can only generate proxies for *interfaces*. It doesn't work for classes.



You'll hit the same limitation if you're stuck with a JDK older than version 1.3. You can always write mock objects by hand (tedious, but not so bad). However, there is a better solution: Use a mock-objects generation framework, like MockMaker, that generates mocks from classes.

### 10.5.1 Introducing MockMaker and installing its Eclipse plugin

MockMaker (<http://mockmaker.org/>) is a *build-time* (as opposed to runtime) mock-object generation tool. It generates source files that need to be added to your project's test source files before compilation. Under the hood, MockMaker uses the MockObjects.com framework, so the mocks it generates use that syntax and those conventions.

You can run MockMaker three ways: by running it as a Java application on the command line, by using the provided Ant task, or by using its Eclipse plugin. This section demonstrates how to run it as an Eclipse plugin, because that's probably the easiest way to use MockMaker (at least, if you're already using Eclipse!).

Installing the Eclipse plugin is easy: Get it from <http://mockmaker.org/> and unzip the plugin in your `ECLIPSE_HOME/plugins` directory. (`ECLIPSE_HOME` is the directory where you have installed Eclipse.)

### 10.5.2 Using MockMaker to generate mocks from classes

In this section, you'll use MockMaker to generate a mock implementation of `PageContext`. In order to generate the `PageContext` mock using the Eclipse plugin (see figure 10.7), you need to select the class to mock first (`PageContext`). Then, right-click and choose `MockMaker→Select Package` to identify the output directory where MockMaker will generate the mock. Select the `junitbook-pages/src/test/junitbook/pages` output directory. MockMaker generates a mock class named `PageContext` in that directory.

**NOTE** The current version of MockMaker (v1.12) doesn't generate the Java imports needed by the generated mocks, so you'll have to perform this step manually.

Let's now write a mock-objects test for the `DynaPropertiesTag` class (see listing 10.9).

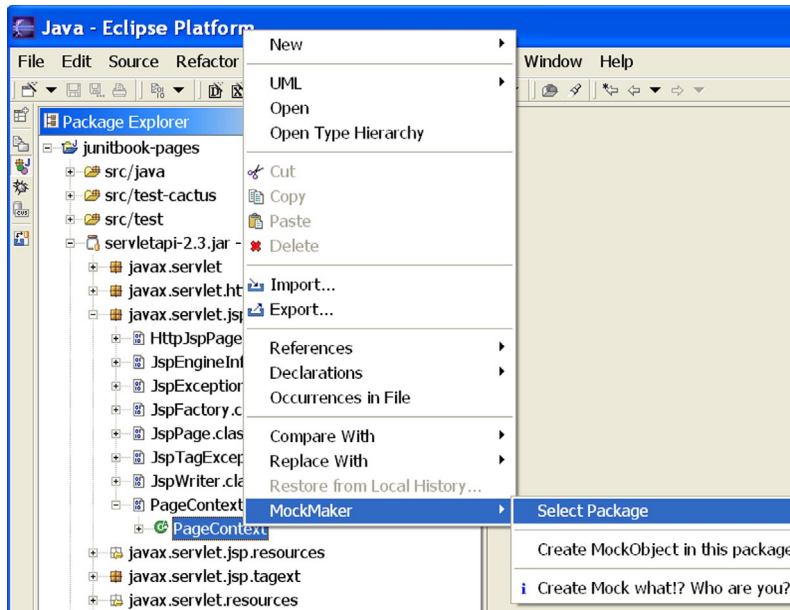


Figure 10.7 Generating the PageContext mock using the MockMaker Eclipse plugin

#### Listing 10.9 Mock-objects test for DynaPropertiesTag

```
package junitbook.pages;

import junit.framework.TestCase;
import org.apache.commons.beanutils.DynaBean;
import org.apache.commons.beanutils.DynaProperty;
import org.apache.commons.beanutils.BasicDynaClass;

import javax.servlet.jsp.tagext.Tag;

public class TestDynaPropertiesMO extends TestCase
{
    private DynaPropertiesTag tag;
    private MockPageContext mockPageContext;

    private DynaBean createDynaBean() throws Exception
    {
        DynaProperty[] props = new DynaProperty[] {
            new DynaProperty("id", String.class),
            new DynaProperty("responsetime", Long.class)
        };
        BasicDynaClass dynaClass = new BasicDynaClass("requesttime",
            null, props);

        DynaBean bean = dynaClass.newInstance();
        bean.set("id", "12345");
    }
}
```

```

        bean.set("responsetime", new Long(500));

        return bean;
    }

    protected void setUp()
    {
        tag = new DynaPropertiesTag();
        mockPageContext = new MockPageContext();
        tag.setPageContext(mockPageContext);
    }

    public void testDoStartTag() throws Exception
    {
        DynaBean bean = createDynaBean();

        mockPageContext.setupFindAttribute(bean); ❶
        mockPageContext.addExpectedFindAttributeValues("item"); ❷
        mockPageContext.addExpectedSetAttributeStringObjectValues( ❸
            "var", bean.getDynaClass().getDynaProperties());

        tag.setItem("${item}");
        tag.setVar("var");

        int result = tag.doStartTag();

        assertEquals(Tag.SKIP_BODY, result);
    }

    protected void tearDown()
    {
        mockPageContext.verify(); ❹
    }
}

```

As with all mock-objects tests, you need to go over all the mock-object methods that will be called and tell the mocks how to behave. In this simple case, only one mock method is called: `PageContext.findAttribute` (❶). The second typical step with mock objects is to tell the mocks what values they should expect (❷ and ❸), so you can verify that the mocks methods were actually called with the expected values (❹).

If you're observant, you may have noticed a slight difference between the Cactus test implementation and the mock-objects implementation. Mocks usually need a deeper knowledge of the implementation than the Cactus tests. In ❶, you have to tell the mock that `PageContext.findAttribute` will be called and that it should return `item`. However, you aren't calling `findAttribute` anywhere in the implementation of `DynaPropertiesTag`! That's because you're calling `ExpressionEvaluatorManager.evaluate("item", ..., pageContext, ...)`. `ExpressionEvaluatorManager` is a utility class from the JSTL library. It uses the passed `PageContext`

to search for an `item` attribute in the page, request, session (if valid), or application scope(s).

## 10.6 When to use mock objects and when to use Cactus

---

One good rule is to always separate, as much as possible, integration code from business logic code. If you're coding a tag that retrieves a list of users from a database, you should implement the process with two Java classes. One class can handle the business logic and avoid dependencies on the Taglib API. A second class can implement the actual tag.

The *separation of concerns* strategy permits reuse of classes in more than one context *and* simplifies testing. You can test the business logic class with JUnit and mock objects, in the usual way. The integration code method that implements the tag can be handled separately, using Cactus.

Cactus requires more setup than mock objects but is well worth the effort. You may not run the Cactus tests as often, but they can confirm that your tags will work in the target environment the way you expect them to.

Sometimes, you may be tempted to skip testing the taglib components. "After all," someone might say, "they will eventually be tested as a side effect of the application's general functional tests, won't they?" We recommend that you fight this temptation. Taglib components deserve the benefits of unit testing as much as any other components. These benefits include:

- Fine-grained tests that can be run over and over and that tell you whether something has broken and exactly where it broke. Integration code is at least as complex as business logic code, and it should also be exercised with unit tests. Cactus provides an easy way of doing so.
- Ability to fully test your taglibs, not only for the successful cases but for failure cases as well. Given the example of a tag accessing a database, you should confirm that the tag behaves well when the connection with the database is broken (for example). Something like this is hard to test in automated functional tests, but easy to test when you combine Cactus and mock objects.

## 10.7 Summary

---

Cactus provides a unique ability to unit-test JSPs by allowing the interception of the JSP calls on the server side, thus providing a hook to set up objects in the HTTP request, HTTP session, or the JSP page context. Cactus enables unit-testing

of JSPs in isolation. Cactus provides a `JspTestCase` class—an extension to the `JUnit TestCase` class—that allows unit-testing of taglibs.

In this chapter, we also demonstrated how to automate unit-testing of JSPs and taglibs using Maven and how to unit-test taglibs using mock objects. You used `MockMaker` to generate a `PageContext` mock, even though `PageContext` is a class and not an interface. (The JDK 1.3 Dynamic Proxy feature only works with interfaces.) Taglibs need more than a Java class to run. Taglibs also require deployment descriptors (`web.xml` and `.tld` files). This point is where Cactus shows all its strength. Cactus can quickly build automated JSP and taglibs test suites that not only verify the code at the unit level but also verify that the deployment descriptors are correct, and that the code runs correctly in the target container.

# 11

## *Unit-testing database applications*

---

### ***This chapter covers***

- Unit-testing in isolation from the database with mock objects
- Performing integration tests with Cactus and DbUnit
- Improving integration tests' performances

*Dependency is the key problem in software development at all scales.... [E]liminating duplication in programs eliminates dependency.*

—Kent Beck, *Test-Driven Development: By Example*

For the combined 22 years that we have been writing business applications, we can't recall a single project that did not use a database of some sort! How do people unit-test code that calls a database? Most of them don't. Many developers deem the database problem to be too complex and rely solely on functional tests.

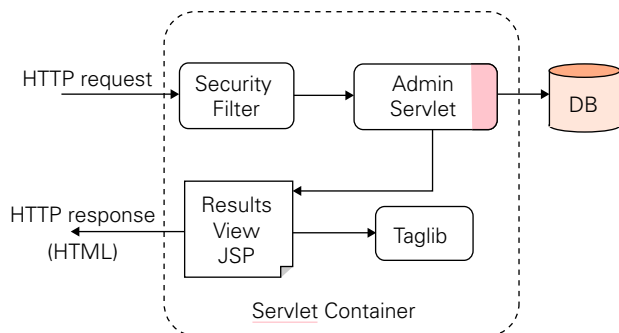
Our goal in this chapter is to show you not only that unit-testing database access code is possible, but also that you can use several different solutions. After we explore the approaches for unit-testing databases, we'll provide some guidelines for deciding which one to use for your particular application.

## 11.1 Introduction to unit-testing databases

We'll use the simple Administration application that we started in chapter 9 (see figure 11.1) to demonstrate how to unit-test database applications. By the end of this chapter, we will have completely covered the unit-testing of the Administration application.

The Administration application is a typical web app. A SQL query is contained in the HTTP request. A security filter verifies that the query found in the request is safe to execute. The processing logic is in the AdminServlet. The servlet receives the request (if it passes the filter), extracts the SQL query, calls the database using JDBC, and forwards to a JSP to display the result. The JSP uses tags to help render the dynamic data as HTML.

In chapters 9 and 10, we covered how to unit-test the servlet, filter, and JSP components of this application. Here, we'll focus on showing you how to unit-test the JDBC component (the shaded part in figure 11.1).

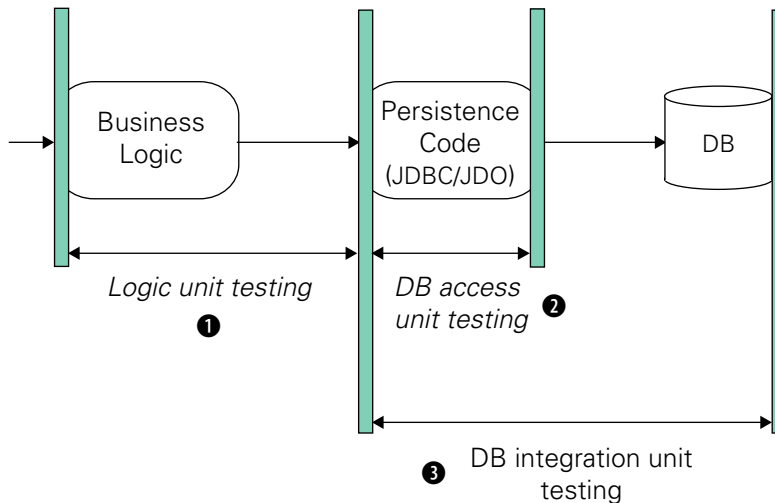


**Figure 11.1**  
Unit-testing the database  
access part of the  
Administration application

Suppose your database access code has been cleanly separated from your business logic code. Separation of concerns is a good practice here, because it lets you change the persistence strategy without changing the rest of the application, and it simplifies unit testing.

You can write different types of tests involving database access (see figure 11.2):

- *Logic unit tests*—The goal of these tests is to unit-test business logic code in isolation of database access code (called *persistence code* in figure 11.2). The strategy is to mock database access code using a mock-objects approach.
- *Database access unit tests*—The database access code uses a persistence API (the JDBC API in this example) to access your database. The goal of this type of test is to validate that you're correctly using the persistence API. Using a mock-objects strategy, you can mock the persistence API, allowing you to run these tests without being connected to a database and without running inside a container.
- *Database integration unit tests*—These types of tests check the database functionality: connectivity, queries, stored procedures, triggers, constraints, and referential integrity. These tests must be run from within the container in order to test the code in the same context from which it will be run. This approach allows you to use the same mechanism to get a `DataSource` (connection pooling) and test transactions. Cactus provides a solution for



**Figure 11.2** Different types of unit tests involving database access: logic unit tests, database access unit tests, and database integration unit tests



in-container testing, and DbUnit (as you'll see) offers a way to preload the database with test data.

In this chapter, we'll demonstrate how to perform each type of database unit test. We'll begin with testing the application's business logic.

## 11.2 Testing business logic in isolation from the database

---

The goal here is to unit-test the business logic code without involving the database access code. Although this type of test isn't a database test per se, it's a good strategy to test the business logic in isolation of harder-to-test database code. Fortunately, this task is easy if you separate the database access layer from the business logic layer. Let's see what this means on the Administration application. The `AdminServlet` class has the following signatures:

```
public class AdminServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException
    {
    }

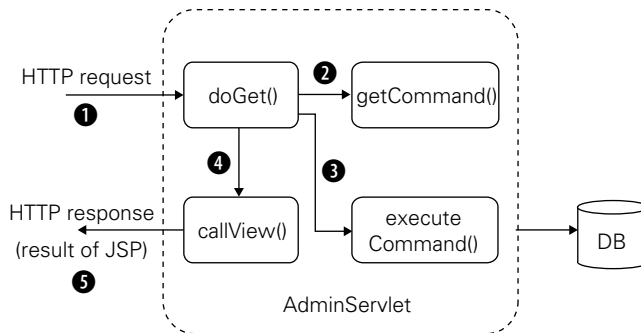
    public String getCommand(HttpServletRequest request)
        throws ServletException
    {
    }

    public void callView(HttpServletRequest request)
    {
    }

    public Collection executeCommand(String command)
        throws Exception
    {
    }
}
```

The execution flow of the different methods is described in figure 11.3. The `doGet` method is the main entry point. It receives the HTTP requests (❶) and calls the `getCommand` method to extract the SQL query from it (❷). It then calls `executeCommand` (❸) to execute the database call (using the extracted SQL query) and return the results as a `Collection`. The results are then put in the HTTP request (as a request attribute) and, at last, `doGet` calls `callView` (❹) to invoke the JSP page that presents the results to the user (❺).

The challenge is to be able to unit-test `doGet`, `getCommand`, and `callView` without executing the database access code that is run by `executeCommand`. The easiest



**Figure 11.3**  
Execution flow for the  
AdminServlet class, showing  
the order of the method calls

and most generic solution is to create a database access layer that handles all database access. (In this case, it's a single class.) The trick is then to create an interface for the database access layer. Given an interface, you can unit-test database access using the mock-objects strategy (see chapter 7).

### 11.2.1 Implementing a database access layer interface

Let's call this interface `DataManager` (listing 11.1) and the implementation `JdbcDataManager`.

#### Listing 11.1 Isolating the database access layer: `DataManager.java`

```

package junitbook.database;

import java.util.Collection;

public interface DataManager
{
    Collection execute(String sql) throws Exception;
}

```

Now that you have a data access interface, you need to refactor the `AdminServlet` class to use that interface and to instantiate the `JdbcDataManager` implementation of `DataManager`. Listing 11.2 demonstrates this refactoring.

#### Listing 11.2 Isolating the database access layer: `AdminServlet.java`

```

package junitbook.database;

import java.util.Collection;

import javax.naming.NamingException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;

```

```
public class AdminServlet extends HttpServlet
{
    // [...]

    private DataAccessManager dataManager;

    public void init() throws ServletException
    {
        super.init();

        try
        {
            this.dataManager = new JdbcDataAccessManager();
        }
        catch (NamingException e)
        {
            throw new ServletException(e);
        }
    }

    public Collection executeCommand(String command)
        throws Exception
    {
        return this.dataManager.execute(command);
    }
}
```

Writing a unit test for a method of the `AdminServlet` class is now easy. All you need to do is create a mock-object implementation of `DataAccessManager`. The only tricky part is deciding how to pass the mock instance to the `AdminServlet` class so that it uses the mock instead of the real `JdbcDataAccessManager` implementation.

### 11.2.2 Setting up a mock database interface layer

There are several strategies you could use to pass a `DataAccessManager` mock object to `AdminServlet`:

- Create a constructor that accepts the `DataAccessManager` interface as a parameter.
- Create a setter method (`setDataAccessManager(DataAccessManager manager)`).
- Extend `AdminServlet` to override the `executeCommand` method and return the result of calling the mock's `executeCommand` method.
- Make the data access manager implementation a parameter of your application by defining the class name in the `web.xml` as an `AdminServlet` initialization parameter.

The challenge is to use the most natural solution—either the one that makes the class more flexible and extensible *or* the one that requires the fewest changes. The constructor solution is really not natural in this case, because the class is a servlet and a servlet must only have a default constructor. Extending `AdminServlet` sounds nice, because it doesn't involve modifying the `AdminServlet` class; but this approach has the drawback that the test won't exercise `AdminServlet` but rather the class that extends it. The `web.xml` solution sounds even nicer; but then you have to make the `DataAccessManager` implementation class an application parameter, which may not be your intent. (In this case, it isn't.)

In this case, the setter solution sounds most natural. Listing 11.3 shows the implementation. The refactoring performed is shown in bold; all you do is add a `setDataAccessManager` method.

**Listing 11.3 Setter approach to introduce `DataAccessManager` mock**

```
package junitbook.database;

import java.util.Collection;

import javax.naming.NamingException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;

public class AdminServlet extends HttpServlet
{
    // [...]

    private DataAccessManager dataManager;

    public void setDataAccessManager(DataAccessManager manager)
    {
        this.dataManager = manager;
    }

    public void init() throws ServletException
    {
        super.init();

        try
        {
            setDataAccessManager(new JdbcDataAccessManager());
        }
        catch (NamingException e)
        {
            throw new ServletException(e);
        }
    }

    public Collection executeCommand(String command)
        throws Exception
```

```
    {  
        return this.dataManager.execute(command);  
    }  
}
```

---

### 11.2.3 Mocking the database interface layer

Listing 11.4 is a test case template that demonstrates how to create and use a mock `DataAccessManager` class for unit-testing the `AdminServlet` class. You can now write any mock object unit test using this canvas. The listing demonstrates writing mocks using the DynaMock API (see chapter 9). However, the strategy will also work with any other mock-object framework (EasyMock, for example).

#### Listing 11.4 Canvas for mocking the database layer using the DynaMock API

```
package junitbook.database;  
  
import java.util.ArrayList;  
  
import com.mockobjects.dynamic.Mock;  
import com.mockobjects.dynamic.C;  
  
import junit.framework.TestCase;  
  
public class TestAdminServletDynaMock extends TestCase  
{  
    public void testSomething() throws Exception  
    {  
        Mock mockManager = new Mock(DataAccessManager.class);  
        DataAccessManager manager =  
            (DataAccessManager) mockManager.proxy();  
  
        mockManager.expectAndReturn("execute", C.ANY_ARGS,  
            new ArrayList());  
  
        AdminServlet servlet = new AdminServlet();  
        servlet.setDataAccessManager(manager);  
  
        // Call the method to test here. For example:  
        // manager.doGet(request, response)  
  
        // [...]  
    }  
}
```

---

You start by creating a `DataAccessManager` mock using the DynaMock API. Next, you tell the mock object to return an empty `ArrayList` when the `execute` method

is called. Then, you set up the mock manager using the `setDataAccessManager` method introduced in listing 11.3.

### 11.3 Testing persistence code in isolation from the database

---

You saw in the previous section that separating the database access layer from the business layer is a good practice. This strategy allows you to use mock objects to easily test the business logic in isolation from the database access code. However, that still leaves you with some untested code: the database access logic code itself.

How do you test database access logic code? Is it desirable to test it at all? The “how” is relatively easy, because the JDBC API is well designed and uses Java interfaces. JDBC lends itself very well to the mock-objects strategy. We will answer the “why” question later, in section 11.7, “Overall database unit-testing strategy.”

Let’s implement database access unit tests using the `MockObjects.com` JDBC API. (See the `com.mockobjects.sql` package.) You could use the `DynaMock` API instead; it yields test code of about the same complexity as the conventional `MockObjects.com` API, but in a few more lines of code. The advantage of the conventional SQL mock objects from `MockObjects.com` is that they don’t need to be created dynamically (because they already exist), and they are tuned for unit-testing JDBC code.

Let’s see what the code to test looks like. Listing 11.5 shows the first implementation of `JdbcDataAccessManager.java`.

#### Listing 11.5 `JdbcDataAccessManager.java`

```
package junitbook.database;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Collection;

import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;

import org.apache.commons.beanutils.RowSetDynaClass;

public class JdbcDataAccessManager implements DataAccessManager
{
    private DataSource dataSource;

    public JdbcDataAccessManager() throws NamingException
    {
        this.dataSource = getDataSource();
    }
}
```

```

    }

    protected DataSource getDataSource() throws NamingException
    {
        InitialContext context = new InitialContext();
        DataSource dataSource =
            (DataSource) context.lookup("java:/DefaultDS");
        return dataSource;
    }

    protected Connection getConnection() throws SQLException
    {
        return this.dataSource.getConnection();
    }

    public Collection execute(String sql) throws Exception
    {
        Connection connection = getConnection();

        // For simplicity, we'll assume the SQL is a SELECT query
        ResultSet resultSet =
            connection.createStatement().executeQuery(sql);

        RowSetDynaClass rsdc = new RowSetDynaClass(resultSet);
        resultSet.close();
        connection.close();

        return rsdc.getRows();
    }
}

```

**Get Data-Source from JNDI**

**Execute SQL query**

**Wrap ResultSet in RowSetDynaClass dyna bean for easy retrieval of data**

**Return a collection of dyna beans, one for each result set row**

As you can see, the execute method is simple and generic. The simplicity stems from use of the BeanUtils package. BeanUtils provides a RowSetDynaClass that wraps a ResultSet and maps database columns to bean properties. You can then use the DynaBean API to access the columns as properties.

The RowSetDynaClass class automatically copies the ResultSet columns to dyna bean properties, allowing you to close the database connection as soon as you have instantiated a RowSetDynaClass object.

### 11.3.1 Testing the execute method

Let's write unit tests for the execute method. The idea is to provide mocks for all calls to the JDBC API. Listing 11.5 shows that you need to mock the Connection object. The mocked Connection can then return other mocks (mock ResultSet, mock Statement, and so forth). Thus, the first question you need to ask is how to pass a mock Connection object to the JdbcDataAccessManager class.

### Passing the mock Connection object

You could pass the mock `Connection` object as a parameter to the `execute` method. However, in this example, you want to keep the creation of the `Connection` object contained within the `JdbcDataAccessManager` class. There are two other valid solutions:

- *Create a new component class*—You can create a new `DataSourceComponent` class with a `getConnection` method and then add a `JdbcDataAccessManager(DataSourceComponent)` constructor.
- *Create a wrapper class*—You can mark `getConnection` as `protected` (instead of `private`) and then create a `TestableJdbcDataAccessManager` class that extends `JdbcDataAccessManager` and adds a `setConnection(Connection)` method. You can set the mock `Connection` by bypassing the use of the `DataSource` to get the connection.

The first solution sounds too complex for the simple needs of this example. Listing 11.6 demonstrates the wrapper-class solution.

#### Listing 11.6 Wrapper to make `JdbcDataAccessManager` testable

```
package junitbook.database;

import java.sql.Connection;
import java.sql.SQLException;

import javax.naming.NamingException;
import javax.sql.DataSource;

public class TestableJdbcDataAccessManager
    extends JdbcDataAccessManager
{
    private Connection connection;

    public TestableJdbcDataAccessManager() throws NamingException
    {
        super();
    }

    public void setConnection(Connection connection)
    {
        this.connection = connection;
    }

    protected Connection getConnection() throws SQLException
    {
        return this.connection;
    }

    protected DataSource getDataSource() throws NamingException
    {

```



```

        return null;
    }
}

```

You now have your way in the `execute` method, so you can begin writing the first unit test for it. As with any test using mock objects, the hard part is finding what methods you need to mock. In other words, you need to understand exactly what methods of your mocked API are being called, in order to provide mocked responses. Often, trial and error is enough to get you started. Begin with the minimum mocks, run the tests (which usually fail), and then refactor, one step at a time.

### **Creating a first test**

To demonstrate, listing 11.7 shows the first test implementation for the `execute` method.

**Listing 11.7** First try at unit-testing `JdbcDataAccessManager.execute`

```

package junitbook.database;

import java.util.Collection;
import java.util.Iterator;

import org.apache.commons.beanutils.DynaBean;

import com.mockobjects.sql.MockConnection2;
import com.mockobjects.sql.MockStatement;

import junit.framework.TestCase;

public class TestJdbcDataAccessManagerMO1 extends TestCase
{
    private MockStatement statement;
    private MockConnection2 connection;
    private TestableJdbcDataAccessManager manager;

    protected void setUp() throws Exception
    {
        statement = new MockStatement();
        connection = new MockConnection2();
        connection.setupStatement(statement);

        manager = new TestableJdbcDataAccessManager();
        manager.setConnection(connection);

    }

    public void testExecuteOk() throws Exception
    {
        String sql = "SELECT * FROM CUSTOMER";

        Collection result = manager.execute(sql);
    }
}

```

```

        Iterator beans = result.iterator();
        assertTrue(beans.hasNext());
        DynaBean bean1 = (DynaBean) beans.next();
        assertEquals("John", bean1.get("firstname"));
        assertEquals("Doe", bean1.get("lastname"));
        assertTrue(!beans.hasNext());
    }
}

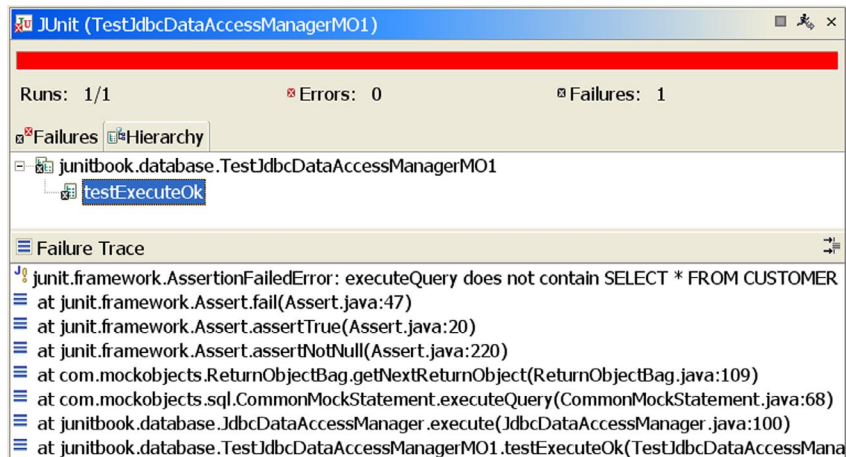
```

5

- ❶ Create the Statement and Connection mocks.
- ❷ Have the Connection mock return the mock Statement object.
- ❸ Instantiate the wrapper class (wrapping the class to test) and call the `setConnection` method (added in listing 11.6) to pass the mock Connection object.
- ❹ Call the method to unit-test.
- ❺ Assert the results by browsing the returned Collection.

Obviously, this test isn't finished. For example, you haven't told the mock Statement what to return when the `executeQuery` method is called (see listing 11.5). Let's run the test and see what happens (figure 11.4).

As expected, you get an error, because you haven't told the mock Statement what to return when `executeQuery` is called. (The stack trace indicates that line 68 of `CommonMockStatement` is trying to return a `ResultSet`, but it hasn't been told what to return.)



**Figure 11.4** Result of running an incomplete mock-object test

### Refining the test

Let's now add the mock `ResultSet` (see listing 11.8; additions are in bold).

**Listing 11.8** Second try at unit-testing `JdbcDataAccessManager.execute`

```
package junitbook.database;
[...]
```

```
import com.mockobjects.sql.MockSingleRowResultSet;
[...]
```

```
public class TestJdbcDataAccessManagerMO2 extends TestCase
{
    private MockSingleRowResultSet resultSet;
    [...]

    protected void setUp() throws Exception
    {
        resultSet = new MockSingleRowResultSet();

        statement = new MockStatement();
    }
    [...]

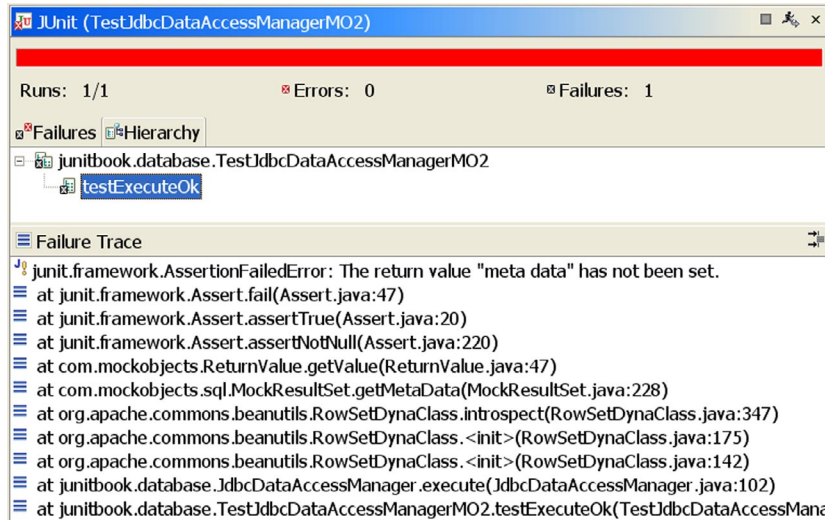
    public void testExecuteOk() throws Exception
    {
        String sql = "SELECT * FROM CUSTOMER";
        statement.addExpectedExecuteQuery(sql, resultSet);

        String[] columnsLowercase =
            new String[] {"firstname", "lastname"};
        resultSet.addExpectedNamedValues(columnsLowercase,
            new Object[] {"John", "Doe"});

        Collection result = manager.execute(sql);
    }
    [...]
}
```

Note that you use the `MockSingleRowResultSet` implementation. Mock-Objects.com provides two implementations: `MockSingleRowResultSet` and `MockMultiRowResultSet`. As their names indicate, the first implementation is used to simulate a result set with one row, and the second implementation is used to simulate a result set with several rows.

Running the test still fails (see figure 11.5), but you have made some progress—even if it isn't yet apparent! The error means that `ResultSet.getMetaData` is called somewhere in the flow of the test. But, you don't call `getMetaData` anywhere in the test class or in the class under test. Thus, it must be called by another package you're using. As the stack trace shows, the `org.apache.commons.beanutils.RowSetDynaClass.introspect` method calls `MockResultSet.getMetaData`. Further



**Figure 11.5** The test still fails after you add a mock `ResultSet`.

investigation shows that `introspect` is called when the `RowSetDynaClass` class is instantiated.

Tracing this error illustrates one of the potential issues with using mock objects: You often need intimate knowledge of the implementation of classes calling your mocks. You can discover indirect calls to your mocks through trial and error, as we just demonstrated. There are two other solutions: get access to the source code, or mock at a different level.

### Discovering indirect calls in source code

Getting the source code isn't always possible and may be time consuming.<sup>1</sup> In this specific case, the Commons BeanUtils is an open source project. The relevant portion of the code is shown in listing 11.9.

#### Listing 11.9 Code from `RowSetDynaClass` highlighting the call to `getMetaData`

```

343 protected void introspect(ResultSet resultSet)
    throws SQLException {
344
345     // Accumulate an ordered list of DynaProperties
346     ArrayList list = new ArrayList();
347     ResultSetMetaData metadata = resultSet.getMetaData();
348     int n = metadata.getColumnCount();

```

<sup>1</sup> A nice decompiler plugin for Eclipse called Jadclipse allows you to see code for which you don't have the source. Be aware that doing this may be illegal in some cases.

```

349         for (int i = 1; i <= n; i++) { // JDBC is one-relative!
350             DynaProperty dynaProperty =
                    createDynaProperty(metadata, i);
351             if (dynaProperty != null) {
352                 list.add(dynaProperty);
353             }
354         }

```

The culprit for the error is at line 347. Looking at the code, note that the next issue will be the call to `getColumnCount` (line 348). You need to set that up in the test case, too.

### ***Mocking at a different level***

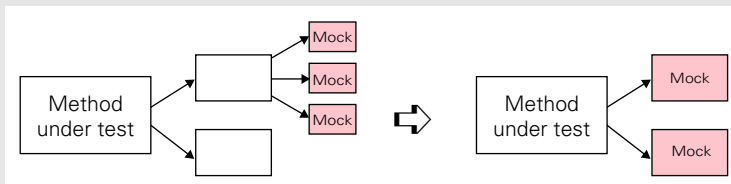
The other solution, mocking at a different level, can be very useful. Your goal here isn't to test the `RowSetDynaClass` class but the `execute` method. One solution is to create a mock `RowSetDynaClass` and pass it (somehow) to the `execute` method.

In the case at hand, it seems easier to set up two additional methods (`getMetaData` and `getColumnCount`). However, mocking at a different level is often the solution to follow when the fixture for a given test becomes long and complex.

### ***JUnit best practices: refactor long setups when using mock objects***

When you're using mock objects, if the amount of setup you need to perform before being able to call the method under test is becoming excessive, you should consider refactoring. A long setup usually means one of two things: either the method under test does too much, so it's difficult to set up its environment, or the mocks you're using are not the correct ones. This happens if you're mocking too deep. In that case, you should introduce new mocks that are directly used by the method under test.

The following illustration demonstrates that mocking at the wrong level forces you to create more mocks:



The more mocks you have, the more setup you need.

**Fixing the test**

Let's modify the test case to support the calls to `getMetaData` and `getColumnCount`. Listing 11.10 show the additions in bold.

**Listing 11.10 Adding a `ResultSetMetaData` mock**

```
package junitbook.database;
[...]
```

```
import com.mockobjects.sql.MockResultSetMetaData;
[...]
```

```
public class TestJdbcDataAccessManagerMO3 extends TestCase
{
[...]
```

```
    private MockResultSetMetaData resultSetMetaData;

    protected void setUp() throws Exception
    {
        resultSetMetaData = new MockResultSetMetaData();
        resultSet = new MockSingleRowResultSet();
        resultSet.setupMetaData(resultSetMetaData);
[...]
```

```
    }

    public void testExecuteOk() throws Exception
    {
        String sql = "SELECT * FROM CUSTOMER";
        statement.addExpectedExecuteQuery(sql, resultSet);

        String[] columnsLowercase =
            new String[] {"firstname", "lastname"};
        String[] columnsUppercase = new String[] {"FIRSTNAME",
            "LASTNAME"};
        String[] columnClassNames = new String[] {
            String.class.getName(), String.class.getName()};

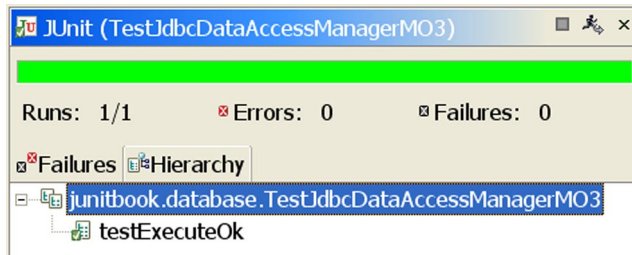
        resultSetMetaData.setupAddColumnNames(columnsUppercase);
        resultSetMetaData.setupAddColumnClassNames(
            columnClassNames);
        resultSetMetaData.setupGetColumnCount(2);

        resultSet.addExpectedNamedValues(columnsLowercase,
            new Object[] {"John", "Doe"});

        Collection result = manager.execute(sql);
[...]
```

```
    }
}
```

The test now succeeds (see figure 11.6).



**Figure 11.6** Successful test for unit-testing the `JdbcDataManager`'s `execute` method in isolation from the database, by using `MockObjects.com` JDBC mocks

### 11.3.2 Using expectations to verify state

Although the test was successful, there are still assertions that you may want to verify as part of the test. For example, you may want to verify that the database connection was closed correctly (and only once), that the query string executed was the one you passed during the test, that a `PreparedStatement` was created only once, and so forth. These kinds of verifications are called *expectations* in `MockObjects.com` terminology. (See section 7.5 in chapter 7 for more about expectations.) Almost all mock objects from `MockObjects.com` expose an Expectation API in the form of `addExpectedXXX` or `setExpectedXXX` methods. The expectations are confirmed by calling the `verify` method on the respective mocks. Some of the `MockObjects.com` mocks perform default verifications. For example, when you write `statement.addExpectedExecuteQuery(sql, resultSet)`, the `MockObjects` code verifies at the end of the test that the `ResultSet` was accessed and the SQL executed was the query passed as parameter. Otherwise, the mock raises an `AssertionFailedError`.

#### Adding expectations

Let's modify listing 11.10 to add some expectations. The result is shown in listing 11.11 (changes are in bold).

#### Listing 11.11 Adding expectations to the `testExecuteOk` method

```
public void testExecuteOk() throws Exception
{
    String sql = "SELECT * FROM CUSTOMER";
    statement.addExpectedExecuteQuery(sql, resultSet); ①

    String[] columnsUppercase = new String[] {"FIRSTNAME",
        "LASTNAME"};
    String[] columnsLowercase = new String[] {"firstname",
        "lastname"};
```

```

String[] columnClassNames = new String[] {
    String.class.getName(), String.class.getName()};

resultSetMetaData.setupAddColumnNames(columnsUppercase);
resultSetMetaData.setupAddColumnClassNames(
    columnClassNames);
resultSetMetaData.setupGetColumnCount(2);

resultSet.addExpectedNamedValues(columnsLowercase,
    new Object[] {"John", "Doe"});

connection.setExpectedCreateStatementCalls(1); ❷
connection.setExpectedCloseCalls(1); ❸

Collection result = manager.execute(sql);

[...]
}

protected void tearDown()
{
    connection.verify();
    statement.verify();
    resultSet.verify();
}

```

- ❶ Add an expectation to verify that the SQL string executed is the one passed by the test, unmodified. This expectation isn't new; you've used it since listing 11.8. It was needed earlier because the call to `addExpectedExecuteQuery` performs two actions: It sets what SQL query the mock `Statement` should simulate and verifies that the SQL query that is executed is the one expected. However, because you were not calling the `verify` method on your `Statement` mock, the expectation wasn't verified.
- ❷ Verify that only one `Statement` is created.
- ❸ Verify that the `close` method is called and that it's called only once.
- ❹ Verify the expectations set on mocks in ❶, ❷, and ❸.

So far, you have only tested a valid execution of the `execute` method. Obviously this isn't enough for a real-world test class. Alternate execution paths are a leading source of bugs in software. It's extremely important to unit-test exception paths. Put yourself in Murphy's shoes: Ask yourself what could possibly go wrong, and then write a test for that possibility.

### Testing for errors

Here is a non-exhaustive list of things that can go wrong in your code and that therefore deserve a test case:



- The `getConnection` may fail with a `SQLException`. (There may be no connections left in the pool, the database may be offline, and so forth.)
- The creation of the `Statement` may fail.
- The execution of the query may fail.

These types of errors are easy to discover. However, there are always other errors that aren't so obvious. The subtle, unexpected errors can only be exposed through experience (and bug reports).

In the database application realm, there is a well-known error that can happen: The database connection may not be closed when an exception is raised. This is a common error, so let's write a test for it (see listing 11.12).

**Listing 11.12** Verifying that the connection is closed even when an exception is raised

```
public void testExecuteCloseConnectionOnException()
    throws Exception
{
    String sql = "SELECT * FROM CUSTOMER";

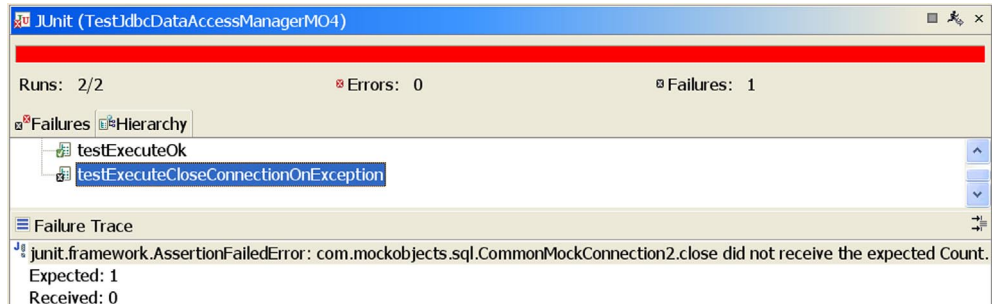
    statement.setupThrowExceptionOnExecute(
        new SQLException("sql error"));

    connection.setExpectedCloseCalls(1);

    try
    {
        manager.execute(sql);
        fail("Should have thrown a SQLException");
    }
    catch (SQLException expected)
    {
        assertEquals("sql error", expected.getMessage());
    }
}
```

- ❶ To verify that the database connection is correctly closed when a database exception is raised, tell the `MockStatement` to throw a `SQLException` when it's executed.
- ❷ Add an expectation to verify that a close call on the mock `Connection` happens once and only once.

If you try to run the test now, it fails as shown in figure 11.7. (Flip back to listing 11.5, where you implemented the `execute` method, and you may see why.) You can easily fix the code by wrapping it in a `try/finally` block:



**Figure 11.7** Failure to close the connection when a database exception is raised

```
public Collection execute(String sql) throws Exception
{
    ResultSet resultSet = null;
    Connection connection = null;
    Collection result = null;

    try
    {
        connection = getConnection();

        resultSet =
            connection.createStatement().executeQuery(sql);

        RowSetDynaClass rsdc = new RowSetDynaClass(resultSet);

        result = rsdc.getRows();
    }
    finally
    {
        if (resultSet != null)
        {
            resultSet.close();
        }
        if (connection != null)
        {
            connection.close();
        }
    }

    return result;
}
```

That's much better. You can now be confident that the database access code performs as expected. But there is more to database access than SQL code. For example, are you sure the connection pool is set up correctly? Do you have to wait for functional tests to discover that something else doesn't work? The focus of the next section is the third type of test: database integration unit testing.

## **11.4 Writing database integration unit tests**

---

Executing database integration unit tests means executing unit tests with a live database. The previous section demonstrated how to write unit tests for database access code without the need for a live database. Running unit tests on a live database allows you to check the following:

- Integration issues, such as verifying that the connection with the database is working
- Business logic located in the database as stored procedures is working properly
- Database triggers are set up correctly
- Database constraints work as intended
- Referential integrity is properly set up in the database

You may find that writing both unit tests in isolation from the database and database integration unit tests is redundant and time-consuming—and you would be right. It's important to define an overall unit-testing strategy that prevents you from having to write too many tests, thus making the process fastidious. In the next section, we look at choosing an appropriate strategy.

### **11.4.1 Filling the requirements for database integration tests**

Say you need to write integration unit tests. In order to run integration unit tests, you require two features from the testing framework:

- The ability to preset the database with test data.
- The ability to start the test from within the running container. This step verifies that the component can talk with the database.

You can use two frameworks for this task: Cactus and DbUnit (<http://www.dbunit.org/>). Cactus allows you to start the test inside the container. (See chapter 8.) DbUnit is a database unit-testing framework that provides two main features: the ability to easily preload the database with test data, and the ability to compare the content of the database after the test with reference data.

Because you want to run the tests in a container, let's pick a J2EE container and a database for the Administration application example. Two free open source tools that are very easy to use are JBoss and Hypersonic SQL. Better yet, JBoss comes preconfigured with Hypersonic SQL. This pair is a particularly good choice because this chapter isn't about how to set up a database—it's about testing a database. Be assured that everything demonstrated here should work with any J2EE container and with any database.

As for the build tool, you'll run the full scenario using the well-known Ant (see chapter 5). You'll start Cactus from Ant and use the Cactus/Ant integration module.

Moreover, in this section you'll continue to use the Administration application. You'll also write integration unit tests for the `JdbcDataAccessManager` `execute` method that you have already unit-tested using mock objects (see section 11.3), but this time you'll use Cactus.

### 11.4.2 Presetting database data

You could preset the database data using JDBC Java code, as part of the test code—in a `setUp` method, for example. This way, you would keep the test code and data in a single place (the Java class). However, there are some drawbacks:

- The database data cannot easily be shared among several test cases. It requires the creation of helper classes.
- Java code isn't the best place to write database queries. You would need to find a framework that lets you easily send SQL code. DbUnit doesn't support creating the dataset from Java. (Its preferred strategy is to use an external file containing the data.)
- The database data cannot be shared between different types of tests: integration unit tests, functional tests, and stress tests. (More information is provided in section 11.7, "Overall database unit-testing strategy.")

In the rest of this section, we'll demonstrate how to preset the database with data defined in an XML file that you load using DbUnit. Listing 11.13 shows a Cactus test that exercises the `JdbcDataAccessManager`'s `execute` method.

**Listing 11.13** Cactus test case for testing the `execute` method

```
package junitbook.database;

import java.util.Collection;
import java.util.Iterator;

import org.apache.cactus.ServletTestCase;
import org.apache.commons.beanutils.DynaBean;

public class TestJdbcDataAccessManagerIC extends ServletTestCase ❶
{
    public void testExecuteOk() throws Exception
    {
        JdbcDataAccessManager manager =
            new JdbcDataAccessManager();
        Collection result =
            manager.execute("SELECT * FROM CUSTOMER");
    }
}
```

```

        Iterator beans = result.iterator();

        assertTrue(beans.hasNext());
        DynaBean bean1 = (DynaBean) beans.next();
        assertEquals("John", bean1.get("firstname"));
        assertEquals("Doe", bean1.get("lastname"));

        assertTrue(!beans.hasNext());
    }
}

```

- ❶ Run the test from the context of a servlet by extending the Cactus ServletTestCase class.
- ❷ The main difference from the previous mock tests is that you're not mocking the DataAccessManager interface anymore; you're using the real implementation that goes to the database.

### **Connecting to the database**

The JdbcDataAccessManager implementation uses a DataSource to connect to the database, as shown in listing 11.14 (new code in bold).

#### **Listing 11.14 DataSource implementation in JdbcDataAccessManager**

```

package junitbook.database;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Collection;

import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;

import org.apache.commons.beanutils.RowSetDynaClass;

public class JdbcDataAccessManager implements DataAccessManager
{
    private DataSource dataSource;

    public JdbcDataAccessManager() throws NamingException
    {
        this.dataSource = getDataSource();
    }

    protected DataSource getDataSource() throws NamingException
    {
        InitialContext context = new InitialContext();
        DataSource dataSource =
            (DataSource) context.lookup("java:/DefaultDS");
    }
}

```

```

        return dataSource;
    }

    protected Connection getConnection() throws SQLException
    {
        return this.dataSource.getConnection();
    }

    public Collection execute(String sql) throws Exception
    {
        [...]
    }
}

```

When you execute `new JdbcDataAccessManager()` in the `TestJdbcDataAccessManagerIC` class (listing 11.13), the `DataSource` is looked up with JNDI. Note that you're looking up the `DataSource` using the `java:/DefaultDS` JNDI key. By default, JBoss defines a `Hypersonic DataSource` at this location.

If you run this test as is, it will fail because you haven't preset the database with the correct data. Let's do this next.

### Setting up the database with data

Listing 11.15 shows how to use `DbUnit` to preset the database with the correct data. You use `DbUnit` in your `TestCase`'s `setUp` method so that the data will be pre-set before each test.

**Listing 11.15** Presetting database data in the `setUp` method

```

package junitbook.database;
[...]
import org.dbunit.database.DatabaseDataSourceConnection;
import org.dbunit.database.IDatabaseConnection;
import org.dbunit.dataset.IDataset;
import org.dbunit.dataset.xml.FlatXmlDataSet;
import org.dbunit.operation.DatabaseOperation;

public class TestJdbcDataAccessManagerIC extends ServletTestCase
{
    protected void setUp() throws Exception
    {
        IDatabaseConnection connection =
            new DatabaseDataSourceConnection(new InitialContext(),
            "java:/DefaultDS");

        IDataset dataSet = new FlatXmlDataSet(
            this.getClass().getResource(
                "/junitbook/database/data.xml"));
    }
}

```

1

2

```
        try
        {
            DatabaseOperation.CLEAN_INSERT.execute(connection,
                                                    dataSet);
        }
        finally
        {
            connection.close();
        }
    }

    public void testExecuteOk() throws Exception
    {
        [...]
    }
}
```

- ❶ Create a database connection using the `DataSource` bound with the `java:/DefaultDS` JNDI name.
- ❷ Load the XML data file.
- ❸ Apply its content to the database by using the `CLEAN_INSERT` strategy. This strategy ensures that the content of the database is exactly the same as what you have in the `data.xml` file.
- ❹ Remember to close the connection!

You're putting the XML data file next to the `TestCase` in the directory structure. (See section 11.5.1 on the project directory structure for more.) This is why you load it as a Java resource. Its content is very simple for this test, because you need only to fill the `CUSTOMER` table with one record for John Doe to make the test pass (listing 11.16).

**Listing 11.16** `Data.xml` containing a single record

```
<dataset>
  <CUSTOMER FIRSTNAME="John" LASTNAME="Doe"/>
</dataset>
```

DbUnit supports several XML file formats, but the simplest is the Flat XML format used in listing 11.16. Each XML element represents a record. The element name is the name of the table where this record applies, the attributes are column names, and the attribute values are the record values.

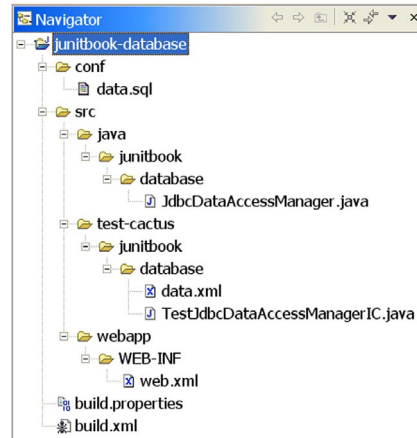
## 11.5 Running the Cactus test using Ant

Let's run the Cactus test using Ant. But first, let's review the project directory structure.

### 11.5.1 Reviewing the project structure

Figure 11.8 shows the project directory structure for the Cactus testing. You cleanly separate the main classes (in `src/java`) from the Cactus test classes (in `src/test-cactus`). Plain JUnit test classes would go in `src/test` to separate them from the other type of classes. Of course, you're free to use whatever directory structure you like. The one we show here is a commonly used structure that we consider a best practice.

Table 11.1 describes the files for the project. The last files in table 11.1 are new to the project, so let's discuss what they do.



**Figure 11.8** Project directory structure for the database integration unit-testing project

**Table 11.1** The project directory for the database unit tests using Cactus and Ant contains seven files.

File	Description
<code>JdbcDataAccessManager.java</code>	The class you're testing
<code>TestJdbcDataAccessManagerIC.java</code>	The Cactus test case class
<code>data.xml</code>	The data to preset in the database, read by the <code>TestDataAccessManagerIC</code> class
<code>data.sql</code>	Used to create the database schema; contains the commands to create the <code>CUSTOMER</code> table
<code>web.xml</code>	The web application descriptor
<code>build.xml</code>	The Ant build file
<code>build.properties</code>	The properties file for the build, which contains Ant properties that are environment dependent



The `conf/data.sql` file creates the database schema. Although DbUnit sets up the data in the database tables, it doesn't create the schema. Happily, you can get Ant to create the schema using the Ant `sql` task. To get you started, `data.sql` needs only a single SQL command:

```
CREATE TABLE CUSTOMER (lastname varchar primary key,
                        firstname varchar);
```

If you execute the `data.sql` file a second time, you need to remove the table before creating it again, or you'll get an error. To remove the table first, you can insert another SQL command:

```
DROP TABLE CUSTOMER;
CREATE TABLE CUSTOMER (lastname varchar primary key,
                        firstname varchar);
```

You'll use the Ant `war` task to create the web application. Unsurprisingly, the Ant `war` task requires a `web.xml` file. For this example, you can leave the file empty:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
</web-app>
```

In production, the `web.xml` file would register the servlets (like the `AdminServlet`) and taglibs, among other things.

### 11.5.2 Introducing the Cactus/Ant integration module

The Cactus/Ant integration module is a jar containing a set of Ant tasks that you can use in your own Ant buildfiles to run Cactus tests. It provides the following tasks:

- `cactifywar`—Transforms an ordinary war file into a Cactus-ready war. This means adding all Cactus-required jars to the original war, adding the Cactus tests, and modifying the `web.xml` descriptor to add the definition for the Cactus redirectors and other miscellaneous entries. This process is called *cactification*.
- `cactus`—Executes Cactus tests. The `cactus` task performs several actions: It configures the container in which running you're the Cactus tests, deploys a *cactified* war into the container, starts the container, runs the tests, and stops

the running container. The cactus task is an extension of the Ant junit task used to automatically run Cactus tests.

You can obtain the Cactus/Ant integration jar by downloading the jakarta-cactus-13-<version>.zip file from the Cactus web site (<http://jakarta.apache.org/cactus/>). The 13 stands for the J2EE API 1.3, which you are using here. The zip contains a cactus-ant-<version>.jar file with the Ant tasks we discussed. You can use these tasks directly in your Ant buildfile via the Ant taskdef syntax. For more about the Cactus/Ant integration module, see <http://jakarta.apache.org/cactus/integration/ant/index.html>.

### 11.5.3 Creating the Ant build file step by step

Let's create the Ant buildfile step by step. Because the Cactus/Ant integration module does 99% of the work for you, the steps are neither long nor complex:

- 1 Create the database schema.
- 2 Create the web application war.
- 3 Compile the Cactus tests.
- 4 Run the Cactus tests.

#### Creating the database schema

Ant's sql task makes this step easy, because it can execute arbitrary SQL commands. The Ant target shown in listing 11.17 reads the SQL from the data.sql file and executes it against the Hypersonic SQL database defined by the \${database} property.

**Listing 11.17** Ant target to create the database schema

```
<?xml version="1.0"?>

<project name="Database" default="createdb" basedir=".">

  <property file="build.properties"/>
  <property name="conf.dir" location="conf"/>

  <target name="createdb">
    <sql driver="org.hsqldb.jdbcDriver"
        url="jdbc:hsqldb:${database}"
        userid="sa"
        password="">
      <fileset dir="${conf.dir}">
        <include name="data.sql" />
      </fileset>
    </sql>
    <classpath>
      <pathelement location="${hsqldb.jar}" />
    </classpath>
  </target>
</project>
```

```
</classpath>
</sql>
</target>

</project>
```

To set this up, you use the `build.properties` file to define two Ant properties: `${database}` (location of the Hypersonic database) and `${hsqldb.jar}` (location of the Hypersonic jar):

```
cactus.home.jboss3x = C:/Apps/jboss-3.2.1
hsqldb.jar = ${cactus.home.jboss3x}/server/default/lib/hsqldb.jar
database =
→ ${cactus.home.jboss3x}/server/default/data/hypersonic/default
```

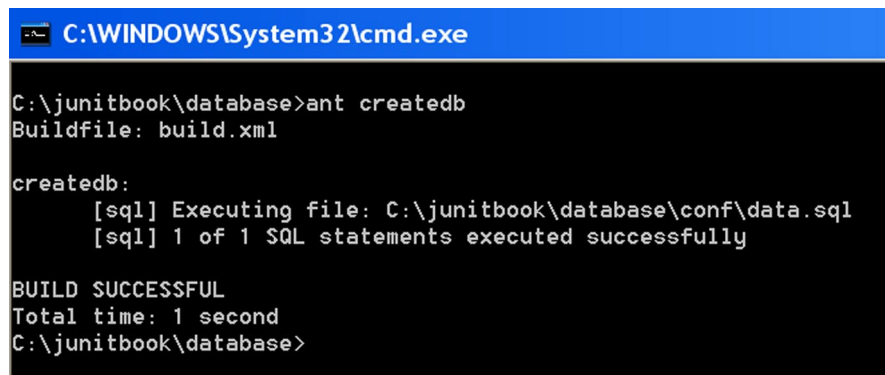
You use the Hypersonic default database, located in the JBoss directory hierarchy. You also use the Hypersonic jar (`hsqldb.jar`) located in the JBoss directory hierarchy.

If you open a command shell and run the `createdb` target by entering **ant createdb**, you get the result shown in figure 11.9.

### ***Creating the web application war***

This is a web application. So, you need a target to create a war to deploy the application. You don't need to include anything related to Cactus in the war, because you'll use the Cactus Ant integration `cactifywar` task to Cactus-enable the war.

To create the war, you need to compile the sources first. Thus you create two targets: `compile` and `war`, shown in listing 11.18.



```
C:\WINDOWS\System32\cmd.exe

C:\junitbook\database>ant createdb
Buildfile: build.xml

createdb:
  [sql] Executing file: C:\junitbook\database\conf\data.sql
  [sql] 1 of 1 SQL statements executed successfully

BUILD SUCCESSFUL
Total time: 1 second
C:\junitbook\database>
```

**Figure 11.9** Result of executing the `createdb` Ant target

**Listing 11.18 Ant targets to create the application war**

```

<?xml version="1.0"?>

<project name="Database" default="war" basedir=".">

[...]

  <target name="compile">
    <mkdir dir="target/classes"/>
    <javac destdir="target/classes" srcdir="src/java">
      <classpath>
        <pathelement location="${beanutils.jar}"/>
        <pathelement location="${servlet.jar}"/>
      </classpath>
    </javac>
  </target>

  <target name="war" depends="compile">
    <war destfile="target/database.war"
        webxml="src/webapp/WEB-INF/web.xml">
      <classes dir="target/classes"/>
      <lib file="${beanutils.jar}"/>
      <lib file="${collections.jar}"/>
    </war>
  </target>

</project>

```

To set this up, you define three Ant properties in the build.properties file—`${beanutils.jar}`, `${servlet.jar}`, and `${lib.dir}` (the location of the jars):

```

lib.dir = ../repository

beanutils.jar =
→   ${lib.dir}/commons-beanutils/jars/commons-beanutils-1.6.1.jar
collections.jar =
→   ${lib.dir}/commons-collections/jars/commons-collections-2.1.jar
servlet.jar = ${lib.dir}/servletapi/jars/servletapi-2.3.jar

```

**Compiling the Cactus tests**

Before you can run the Cactus tests, you need to add a target to compile them, like the one shown here:

```

<target name="compile.cactustest">
  <mkdir dir="target/cactus-test-classes"/>
  <javac destdir="target/cactus-test-classes"
        srcdir="src/test-cactus">
    <classpath>
      <pathelement location="target/classes"/>
      <pathelement location="${beanutils.jar}"/>
    </classpath>
  </javac>
</target>

```

```

        <pathelement location="${dbunit.jar}"/>
        <pathelement location="${cactus.jar}"/>
    </classpath>
</javac>
<copy todir="target/cactus-test-classes">
    <fileset dir="src/test-cactus">
        <include name="**/*.xml"/>
    </fileset>
</copy>
</target>

```

- ❶ You need to add the jars for DbUnit and Cactus to the compilation classpath, because classes from these jars are used in the test cases. So, you define two more properties in the build.properties file, `${dbunit.jar}` and `${cactus.jar}`:

```

cactus.jar = ${lib.dir}/cactus/jars/cactus-13-1.5.jar
dbunit.jar = ${lib.dir}/dbunit/jars/dbunit-1.5.5.jar

```

- ❷ Back in listing 11.15, you loaded the `data.xml` file as a Java resource. This means it has to be in the test classpath when you execute the tests. So, here you copy any XML file in the test source directory structure (`src/test-cactus`) to the target compilation directory (where the compiled test classes are put).

### Running the Cactus tests

You're almost ready to execute the tests using the `cactus` Ant task from the Cactus/Ant integration module. However, you need to pass a cactified war to the `cactus` task. The `cactifywar` Ant task can handle this for you. Before you can use the Cactus/Ant integration tasks, you need to define them in your buildfile as follows:

```

<target name="test" depends="war, compile.cactustest">
    <taskdef resource="cactus.tasks">
        <classpath>
            <pathelement location="${cactus.ant.jar}"/>
            <pathelement location="${cactus.jar}"/>
            <pathelement location="${logging.jar}"/>
            <pathelement location="${aspectjrt.jar}"/>
            <pathelement location="${httpclient.jar}"/>
        </classpath>
    </taskdef>

```

You have added all the Cactus jars to the classpath of the `taskdef`. This classpath will be used by both `cactifywar` and `cactus`. The `cactifywar` task puts the Cactus jars in the cactified war, and the `cactus` task puts the Cactus jars in the test execution classpath.

As usual, you create Ant properties to hold the path to your jars in the `build.properties` file:

```

cactus.ant.jar = ${lib.dir}/cactus/jars/cactus-ant-13-1.5.jar
aspectjrt.jar = ${lib.dir}/aspectj/jars/aspectjrt-1.0.6.jar
logging.jar =
→   ${lib.dir}/commons-logging/jars/commons-logging-1.0.3.jar
httpclient.jar =
→   ${lib.dir}/commons-httpclient/jars/commons-httpclient-2.0.jar

```

Writing the cactification task is easy:

```

<target name="test" depends="war, compile.cactustest">

  <taskdef resource="cactus.tasks">
[...]
```

```

  </taskdef>

  <cactifywar srcfile="target/database.war"
    destfile="target/test.war">
    <classes dir="target/cactus-test-classes"/>
    <lib file="${dbunit.jar}"/>
    <lib file="${exml.jar}"/>
  </cactifywar>

```

The `cactifywar` task transforms the `target/database.war` war into a Cactus-enabled `target/test.war` war. This task adds all the jars you have defined in the `taskdef` to the war. Here, you use the nested `lib` element to add the `${dbunit.jar}` and `${exml.jar}` jars.

DbUnit needs the Electric XML jar (`exml.jar`) at runtime. So, you add it to the `build.properties` file:

```
exml.jar = ${lib.dir}/dbunit/jars/exml-dbunit-1.5.5.jar
```

The content of the resulting `test.war` is shown in figure 11.10.

The last step is to execute the Cactus tests by using the `cactus` task:

```

<target name="test" depends="war, compile.cactustest">

  <taskdef resource="cactus.tasks">
[...]
```

```

  </taskdef>

  <cactifywar srcfile="target/database.war"
[...]
```

```

  </cactifywar>

  <cactus warfile="target/test.war" fork="yes" printsummary="yes"
    haltonerror="true" haltonfailure="true">
    <containerset>
      <jboss3x dir="${cactus.home.jboss3x}"
        output="target/jbossresult.txt"/>
    </containerset>
    <formatter type="brief" usefile="false"/>
    <batchtest>

```

```


















        <fileset dir="src/test-cactus">
            <include name="**/TestJdbcDataAccessManagerIC.java"/>
        </fileset>
    </batchtest>
    <classpath>
        <pathelement location="target/classes"/>
        <pathelement location="target/cactus-test-classes"/>
        <pathelement location="${dbunit.jar}"/>
    </classpath>
</cactus>

</target>

```

The cactus task extends the junit Ant task. Thus, all attributes available to the junit task are also directly available to the cactus task. For example, you use the `fork`, `printsummary`, `haltonerror`, and `haltonfailure` attributes from the junit task. You also use the `formatter`, `batchtest`, and `classpath` elements from the junit task.

You need to tell the cactus task what container you wish to run the tests with. In this case, you're using JBoss 3.2.1; hence the `jboss3x` element. The cactus task supports several other containers, such as Tomcat (3.x, 4.x, 5.x),

Name	Path
 jspRedirector.jsp	
 MANIFEST.MF	META-INF\
 web.xml	WEB-INF\
 AdminServlet.class	WEB-INF\classes\junitbook\database\
 data.xml	WEB-INF\classes\junitbook\database\
 DataAccessManager.class	WEB-INF\classes\junitbook\database\
 JdbcDataAccessManager.class	WEB-INF\classes\junitbook\database\
 TestJdbcDataAccessManagerIC.class	WEB-INF\classes\junitbook\database\
 aspectjrt-1.0.6.jar	WEB-INF\lib\
 cactus-13-1.5.jar	WEB-INF\lib\
 commons-beanutils-1.6.1.jar	WEB-INF\lib\
 commons-collections-2.1.jar	WEB-INF\lib\
 commons-httpclient-2.0.jar	WEB-INF\lib\
 commons-logging-1.0.3.jar	WEB-INF\lib\
 dbunit-1.5.5.jar	WEB-INF\lib\
 exml-dbunit-1.5.5.jar	WEB-INF\lib\
 junit.jar	WEB-INF\lib\

**Figure 11.10** Content of the cactified war. The `jspRedirector.jsp` file, the test classes, and the Cactus-related jars have been added by the cactification. In addition, the `web.xml` file has been modified to include the Cactus redirector definitions and mappings.

Orion (1.x, 2.x), Resin (2.x), and WebLogic (7.x), to name a few. (See the Cactus web site for full information.)

The output attribute of the `jboss3x` element redirects all container output to the `target/jbossresult.txt` file. This keeps `stdout` from becoming cluttered.

The cactus task automatically adds the jars you have defined in the `taskdef` to the execution classpath (the classpath of the JVM where the JUnit test runner executes). You need to explicitly add the DbUnit jar because it's referenced from your Cactus `TestJdbcDataAccessManagerIC` test case class.

The full listing of the test target is shown in listing 11.19.

**Listing 11.19** Ant target that calls the Cactus/Ant integration module to run the tests

```
<?xml version="1.0"?>
<project name="Database" default="test" basedir=".">
[... ]

<target name="test" depends="war,compile.cactustest">

  <taskdef resource="cactus.tasks">
    <classpath>
      <pathelement location="${cactus.ant.jar}"/>
      <pathelement location="${cactus.jar}"/>
      <pathelement location="${logging.jar}"/>
      <pathelement location="${aspectjrt.jar}"/>
      <pathelement location="${httpClient.jar}"/>
    </classpath>
  </taskdef>

  <cactifywar srcfile="target/database.war"
    destfile="target/test.war">
    <classes dir="target/cactus-test-classes"/>
    <lib file="${dbunit.jar}"/>
    <lib file="${exml.jar}"/>
  </cactifywar>

  <cactus warfile="target/test.war" fork="yes" printsummary="yes"
    haltonerror="true" haltonfailure="true">
    <containerset>
      <jboss3x dir="${cactus.home.jboss3x}"
        output="target/jbossresult.txt"/>
    </containerset>
    <formatter type="brief" usefile="false"/>
    <batchtest>
      <fileset dir="src/test-cactus">
        <include name="**/TestJdbcDataAccessManagerIC.java"/>
      </fileset>
    </batchtest>
  </cactus>
</target>
</project>
```



```

C:\WINDOWS\System32\cmd.exe

C:\junitbook\database>ant test
Buildfile: build.xml

compile:
  [mkdir] Created dir: C:\junitbook\database\target\classes
  [javac] Compiling 4 source files to C:\junitbook\database\target\classes

war:
  [war] Building war: C:\junitbook\database\target\database.war

compile.cactustest:
  [mkdir] Created dir: C:\junitbook\database\target\cactus-test-classes
  [javac] Compiling 4 source files to C:\junitbook\database\target\cactus-test-classes
  [copy] Copying 1 file to C:\junitbook\database\target\cactus-test-classes

test:
[cactifywar] Analyzing war: C:\junitbook\database\target\database.war
[cactifywar] Building war: C:\junitbook\database\target\test.war
[cactus] -----
[cactus] Running tests against JBoss 3.2
[cactus] -----
[cactus] Running junitbook.database.TestJdbcDataAccessManagerIC
[cactus] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 3.816 sec
[cactus] Testsuite: junitbook.database.TestJdbcDataAccessManagerIC
[cactus] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 3.816 sec

[cactus] Shutdown complete

BUILD SUCCESSFUL
Total time: 49 seconds
C:\junitbook\database>

```

**Figure 11.11** Result of executing Cactus tests automatically using Ant against JBoss 3.2.1

```

    <pathelement location="target/classes"/>
    <pathelement location="target/cactus-test-classes"/>
    <pathelement location="${dbunit.jar}"/>
  </classpath>
</cactus>

</target>

</project>

```

This example needs only a few of the many features available through the Cactus/Ant integration module. To learn more, visit the Cactus web site (<http://jakarta.apache.org/cactus/>).

### 11.5.4 Executing the Cactus tests

You now have a full-fledged build system that automatically executes the Cactus tests. Let's run it by entering **ant test** from a command shell. The result is shown in figure 11.11. You can now run the database integration unit tests automatically from Ant whenever you like.

## 11.6 Tuning for build performance

This example includes only a single integration test. The test takes 49 seconds to run, so you'll probably run it as often as needed. But in a production application with hundreds of test cases, the total time of execution becomes significant. If the tests take too long to run, you may not run them as often as you should.

Several strategies can help tune test execution performance:

- Factor out read-only data.
- Group tests in functional test suites.
- Use an in-memory database.

### 11.6.1 Factoring out read-only data

Some of the data in a database is only read and never modified by the application. Instead of initializing this data in the `setUp` method (which is called once for every test), a better solution is to preset it in a `TestSuite setUp` method (which is executed once per test suite run).

For example, the `execute` method in the `JdbcDataAccessManager` class only reads data and doesn't perform any deletes or updates. You can refactor `TestJdbcDataAccessManagerIC` (from listing 11.15) and move the database initialization into a `JUnit TestSetup` class, as shown by listing 11.20.

**Listing 11.20** Factorizing read-only database data in a JUnit TestSetup class

```
package junitbook.database;

import javax.naming.InitialContext;

import junit.extensions.TestSetup;
import junit.framework.Test;

import org.dbunit.database.DatabaseDataSourceConnection;
import org.dbunit.database.IDatabaseConnection;
import org.dbunit.dataset.IDataset;
import org.dbunit.dataset.xml.FlatXmlDataSet;
import org.dbunit.operation.DatabaseOperation;

public class DatabaseTestSetup extends TestSetup
{
    public DatabaseTestSetup(Test suite)
    {
        super(suite);
    }

    protected void setUp() throws Exception
    {

```