

JUnit best practices: put Cactus tests in their own directories

It's a good strategy to put the Cactus tests in their own directory structure, such as in `src/test-cactus`. The main reason is that integration unit tests and pure JUnit tests have different running life cycles: Pure JUnit tests are usually run more often than Cactus tests, because they execute faster. Separate directories allow the build system to perform different build actions on either type of test. Most Cactus test runners expect the Cactus tests to be in a specific directory, because they need a different setup than pure JUnit tests (such as packaging and deploying them in a container, and so forth).

class, which lets you run `setUp` and `tearDown` code before any test is executed and at the end of the test suite execution. `JettyTestSetup` uses that to start Jetty before the suite starts, and it stops Jetty when the suite terminates. Listing 8.4 demonstrates how to create a test suite to run Cactus tests inside the Jetty container.

Listing 8.4 Test suite to run Cactus test inside the Jetty container

```
package junitbook.container;

import org.apache.cactus.extension.jetty.JettyTestSetup;

import junit.framework.Test;
import junit.framework.TestSuite;

public class TestAllWithJetty
{
    public static Test suite()
    {
        System.setProperty("cactus.contextURL",
                           "http://localhost:8080/test");

        TestSuite suite = new TestSuite("All tests with Jetty");
        suite.addTestSuite(TestSampleServletIntegration.class);
        return new JettyTestSetup(suite);
    }
}
```

- 1 Define the context under which the web application containing the Cactus tests will run. The `JettyTestSetup` class uses this information to set up a listener on the port defined and to create a context. Because `cactus.contextURL` is a `System` property, it's also possible to pass it by using a `-Dcactus.contextURL=...` flag when you start the JVM used for the tests.

- ② Create a JUnit test suite and add all the tests found in the `TestSampleServletIntegration` class.
- ③ Wrap the JUnit test suite with the Cactus `JettyTestSetup` so that Jetty will be started and stopped during the execution of the tests.

Now that you have all the required source files, let's see what you need in order to execute the tests with Cactus/Jetty. Cactus requires some jars to be present on your classpath: the Cactus jar, the Commons Logging jar, the Commons HttpClient jar, and the AspectJ runtime jar. You'll be running the Jetty servlet engine, so you also need to have the Jetty jar on your classpath. If you're following the directory structure and the Eclipse projects as defined in appendix A, you should find the jars in the `junitbook/repository` directory, as shown in figure 8.5. Notice that we created an Eclipse project (named `junitbook-repository`) for the `junitbook/repository` directory to make it easier to include these jars on the `junitbook-container` project classpath (but doing so is not mandatory).

Let's now add the required jars to `junitbook-container` project, as shown in figure 8.6. In order to add these jars, right-click your project and select Properties. In the dialog that appears (on the Libraries tab), select Add JARs; then, in the next dialog that appears, open the `junitbook-repository` project and select all the jars you need.

You can now enjoy running the Cactus tests by clicking the Run button in the toolbar. The result is shown in figure 8.7. The console shows that Cactus started Jetty automatically before running the tests. Don't worry if you don't understand how Cactus works at this point. We'll come back to the Cactus mechanism later.

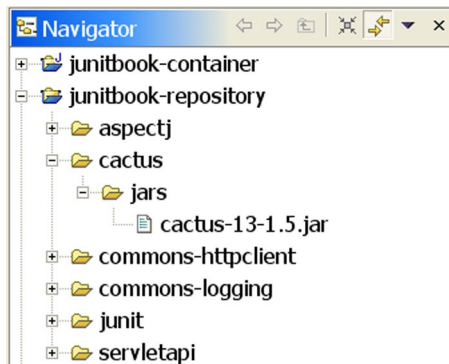


Figure 8.5
The Eclipse project named `junitbook-repository`. It's a placeholder for the jars used in all the other projects.



Figure 8.6 Jars required for the `junitbook-container` project in Eclipse

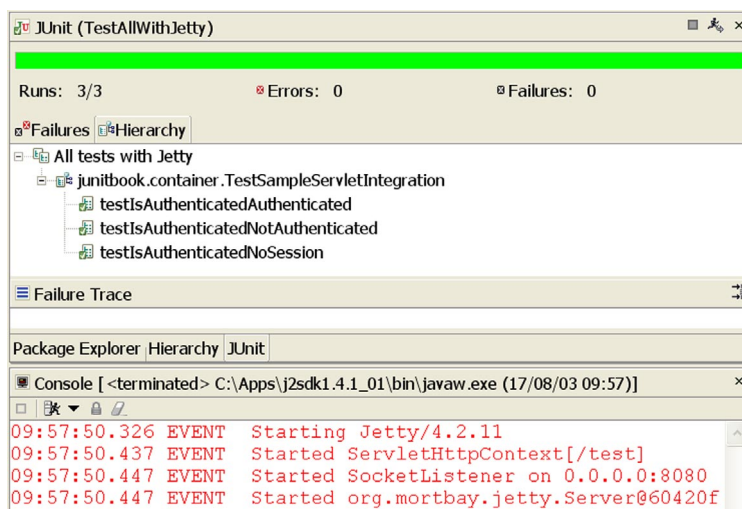


Figure 8.7 Cactus tests running in the Jetty container from inside Eclipse

8.5.3 Drawbacks of in-container testing

So far, we've shown you the advantages of in-container unit-testing. However, there are also a few disadvantages.

Specific tools required

A major drawback is that although the concept is generic, the tools that implement in-container unit-testing are very specific to the underlying API being tested. In the J2EE world, the de facto standard is Jakarta Cactus (introduced in the previous section). However, if you wish to write integration unit tests for another

component model, chances are that no such framework will exist, and you may have to build one yourself. On the other hand, the mock-objects approach is completely generic and will work for almost any API.

Longer execution time

Another disadvantage is speed of execution. For a test to run in the container, you need to start the container beforehand, and that can take some time. How much time depends on the container: Jetty starts in less than 1 second, Tomcat starts in about 5 seconds, and WebLogic starts in about 30 seconds. The startup lag doesn't end with the container. If your unit tests hit a database or other remote resource, the database must be in a valid state before the test (see chapter 11, which covers database application testing). In terms of execution time, integration unit tests cost more than a mock-objects approach. Consequently, you may not run them as often as logic unit tests.

Complex configuration

The biggest drawback of in-container testing may be the configuration complexity. The tests run inside the container, so you need to package your application (usually as a war or an ear) before you can run the tests, deploy the tests to the server, start the server, and then start the tests.

However, there are some positive counter-arguments. Our favorite is that for production, you need to deploy your application. To do so you must package it, deploy it, and start the server—the same steps required for running a Cactus test! Our belief is that this exercise should be practiced from day one of the project, because it's one of the most complex tasks of a J2EE project. It needs to be done as often as possible and automated as much as possible in order to be able to perform deployments easily. Cactus acts as a triggering element in favor of setting up this process at the beginning of the project; doing so is good and is completely in line with the spirit of *continuous integration*.

The second counter-argument is that the Cactus development team has recognized that it can be a daunting task to set up everything before you can run your first Cactus test. Cactus provides several front ends that hide much (if not all) of the complexity and will run the test for you at the click of a button.

8.6 How Cactus works

The following chapters show how to use Cactus to unit-test servlets, filters, JSPs, database code, and EJBs. However, before we rush into the details, you need to understand a bit more about how Cactus works.

The life cycle of a Cactus test is shown in figure 8.8.

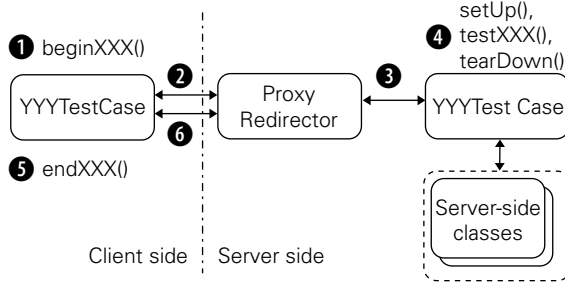


Figure 8.8
Life cycle of a Cactus test

We'll describe the different steps using the `TestSampleServletIntegration` Cactus test from listing 8.3. Say you have already deployed the application in the container, and the container is started. You can submit the Cactus test to a JUnit test runner, and the runner starts the tests.

8.6.1 Executing client-side and server-side steps

The life cycle is divided into steps that are executed on the client side and others that are executed on the server side (inside the container JVM). *Client side* refers to the JVM in which you have started the JUnit test runner.

On the client side, the Cactus logic is implemented in the `YYYTestCase` classes that your tests extend (where `YYY` can be `Servlet`, `Jsp`, or `Filter`). More specifically, `YYYTestCase` overrides `JUnit TestCase.runBare`, which is the method called by the JUnit test runner to execute one test. By overriding `runBare`, Cactus can implement its own test logic, as described later.

On the server side, the Cactus logic is implemented in a *proxy redirector* (or *redirector* for short).

8.6.2 Stepping through a test

For each test (`testXXX` methods in the `YYYTestCase` classes), the six steps shown in figure 8.8 take place. Let's step through all six.

Step 1: execute `beginXXX`

If there is a `beginXXX` method, Cactus executes it. The `beginXXX` method lets you pass information to the redirector. The `TestSampleServletIntegration` example extends `ServletTestCase` and connects to the Cactus servlet redirector. The servlet redirector is implemented as a servlet; this is the entry point in the container. The Cactus client side calls the servlet redirector by opening an HTTP connection to it. The `beginXXX` method sets up HTTP-related parameters that are set in the

HTTP request received by the servlet redirector. This method can be used to define HTTP POST/GET parameters, HTTP cookies, HTTP headers, and so forth. For example:

```
public void beginXXX(WebRequest request)
{
    request.addParameter("param1", "value1");
    request.addCookie("cookie1", "value1");
    [...]
}
```

In the `TestSampleServletIntegration` class, we have used the `beginXXX` method to tell the redirector not to create an HTTP session (the default behavior creates one):

```
public void beginIsAuthenticatedNoSession(WebRequest request)
{
    request.setAutomaticSession(false);
}
```

Step 2: open the redirector connection

The `YYYTestCase` opens a connection to its redirector. In this case, the `ServletTestCase` code opens an HTTP connection to the servlet redirector (which is a servlet).

Step 3: create the server-side `TestCase` instance

The redirector creates an instance of the `YYYTestCase` class. Note that this is the second instance created by Cactus; a first one has been created on the client side (by the `JUnit TestRunner`). Then, the redirector retrieves container objects and assigns them in the `YYYTestCase` instance by setting class variables.

In the servlet example, the servlet redirector creates an instance of `TestSampleServletIntegration` and sets the following objects as class variables in it: `HttpServletRequest`, `HttpServletResponse`, `HttpSession`, and so forth. The servlet redirector is able to do this because it is a servlet. When it's called by the Cactus client side, it has received a valid `HttpServletRequest`, `HttpServletResponse`, `HttpSession`, and other objects from the container and is passing them to the `YYYTestCase` instance. It acts as a proxy/redirector (hence its name).

The redirector then starts the test (see step 4). Upon returning from the test, it stores the test result in the `ServletConfig` servlet object along with any exception that might have been raised during the test, so the test result can later be retrieved. The redirector needs a place to temporarily store the test result because the full Cactus test is complete only when the `endXXX` method has finished executing (see step 5).

Step 4: call setUp, testXXX, and tearDown on the server side

The redirector calls the JUnit `setUp` method of `YYYTestCase`, if there is one. Then it calls the `testXXX` method. The `testXXX` method calls the class/methods under test, and finally the redirector calls the JUnit `tearDown` method of the `TestCase`, if there is one.

Step 5: execute endXXX

Once the client side has received the response from its connection to the redirector, it calls an `endXXX` method (if it exists). This method is used so that your tests can assert additional results from the code under test. For example, if you're using a `ServletTestCase`, `FilterTestCase`, or `JspTestCase` class, you can assert HTTP cookies, HTTP headers, or the content of the HTTP response:

```
public void endXXX(WebResponse response)
{
    assertEquals("value",
        response.getCookie("cookieName").getValue());
    assertEquals("...", response.getText());
    [...]
}
```

Step 6: Gathering the test result

In step 3, the redirector saves the test result in a variable stored with the `Servlet-Config` object. The Cactus client side now needs to retrieve the test result and tell the JUnit test runner whether the test was successful, so the result can be displayed in the test runner GUI or console. To do this, the `YYYTestCase` opens a second connection to the redirector and asks it for the test result.

This process may look complex at first glance, but this is what it takes to be able to get inside the container and execute the test from there. Fortunately, as users, we are shielded from this complexity by the Cactus framework. You can simply use the provided Cactus front ends to start and set up the tests.

8.7 Summary

When it comes to unit-testing container applications, pure JUnit unit tests come up short. A mock-objects approach (see chapter 7) works fine and should be used. However, it misses a certain number of tests—specifically integration tests, which verify that components can talk to each other, that the components work when run inside the container, and that the components interact properly with the container. In order to perform these tests, an in-container testing strategy is required.

In the realm of J2EE components, the de facto standard framework for in-container unit-testing is Jakarta Cactus.

In this chapter, we ran through some simple tests using both a mock-objects approach and Cactus, in order to get a flavor for how it's done. We also discussed how Cactus works. We're now ready to move to the following chapters, which will let you practice unit-testing J2EE components like web applications and EJBs using both mock objects and Cactus.

Part 3

Testing components

Part 3 lets you practice the the testing knowledge acquired in parts 1 and 2 on J2EE components. You'll see not only how to write unit tests for the whole gamut of J2EE components but also how to set up your projects and how to run and automate the tests with Ant, Maven, and Eclipse.

Chapter 9 focuses on servlets and filters. Chapter 10 will teach you how to test JSPs and taglibs. In chapter 11, you'll learn about an aspect common to almost all applications: unit-testing code that calls the database. Chapter 12 takes you through the journey of EJB unit-testing.

After reading part 3, you'll know how to completely unit-test full J2EE applications. You'll also be familiar with the tradeoffs that exist and when to use one testing strategy over another.

Unit-testing servlets and filters

This chapter covers

- Demonstrating the Test-Driven Development (TDD) approach
- Writing servlet and filter unit tests with Cactus and mock objects
- Running Cactus tests with Maven
- Choosing when to use Cactus and when to use mock objects

The only time you don't fail is the last time you try anything—and it works.

—William Strong

When you unit-test servlet and filter code, you must test not only these objects, but also any Java class calling the Servlet/Filter API, the JNDI API, or any back-end services. Starting in this chapter, you'll build a real-life sample application that will help demonstrate how to unit-test each of the different kinds of components that make up a full-blown web application. This chapter focuses on unit-testing the servlet and filter parts of that application. Later chapters test the other common components (JSPs, taglibs, and database access).

In this chapter, you'll learn how to unit-test servlets and filters using both the Cactus in-container testing approach and the mock-objects approach with the DynaMock framework from <http://www.mockobjects.com/>. You'll also learn the pros and cons of each approach and when to use them.

9.1 Presenting the Administration application

The goal of this sample Administration application is to let administrators perform database queries on a relational database. Suppose that the application it administers already exists. Administrators can perform queries such as listing all the transactions that took place during a given time interval, listing the transactions that were out of Service Level Agreement (SLA), and so forth. We set up a typical web application architecture (see figure 9.1) to demonstrate how to unit-test each type of component (filter, servlet, JSP, taglib, and database access).

The application first receives from the user an HTTP request containing the SQL query to execute. The request is caught by a security filter that checks

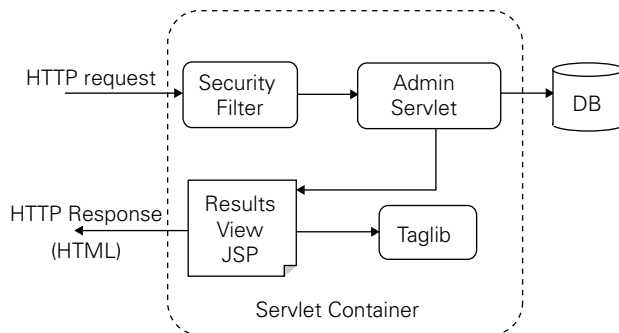


Figure 9.1 The sample Administration application. You'll use it as a base sample in this chapter and following chapters to see how to unit-test servlets, filters, JSPs, taglibs, and database applications.

whether the SQL query is a `SELECT` query (to prevent modifying the database). If not, the user is redirected to an error page. If the query is a `SELECT`, the `AdminServlet` servlet is called. The servlet performs the requested database query and forwards the result to a JSP page, which displays the results. The page uses JSP tags to iterate over the returned results and to display them in HTML tables. JSP tags are used for all the presentation logic code. The JSPs contain only layout/style tags (no Java code in scriptlets).

You'll start by unit-testing the `AdminServlet` servlet. Then, in section 9.4, you'll unit-test your security filter. You'll test the other components of the Administration application in following chapters.

9.2 Writing servlet tests with Cactus

In this section, we'll focus on using Cactus to unit-test the `AdminServlet` servlet (shaded in figure 9.2) from the Administration application.

Let's test `AdminServlet` by writing the tests before you write the servlet code. This strategy is called Test-Driven Development (TDD) or Test-First, and it's very efficient for designing extensible and flexible code and making sure the unit test suite is as complete as possible. (See chapter 4 for an introduction to TDD.)

Before you begin coding the test, let's review the requirement for `AdminServlet`. The servlet should extract the needed parameter containing the command to execute from the HTTP request (in this case, the SQL command to run). Then it should fetch the data using the extracted command. Finally, it should pass the control to the JSP page for display, passing the fetched data. Let's call the methods corresponding to these actions `getCommand`, `executeCommand`, and `callView`, respectively.

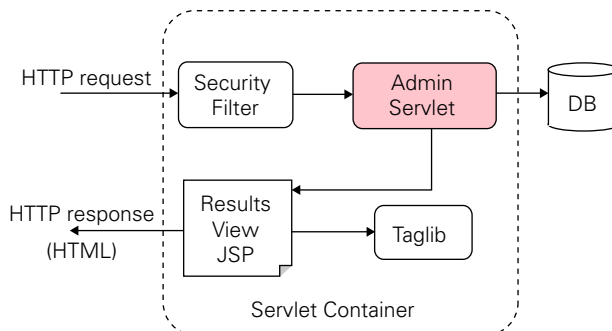


Figure 9.2
Unit-testing `AdminServlet` of
the Administration application

9.2.1 Designing the first test

Listing 9.1 shows the unit tests for the `getCommand` method. Remember that you have not yet written the code under test. The `AdminServlet` class doesn't exist, and your code doesn't compile (yet).

Listing 9.1 Designing and testing the `getCommand` method

```
package junitbook.servlets;

import javax.servlet.ServletException;
import org.apache.cactus.ServletTestCase;
import org.apache.cactus.WebRequest;

public class TestAdminServlet extends ServletTestCase
{
    public void beginGetCommandOk(WebRequest request)
    {
        request.addParameter("command", "SELECT...");
    }

    public void testGetCommandOk() throws Exception
    {
        AdminServlet servlet = new AdminServlet();
        String command = servlet.getCommand(request);

        assertEquals("SELECT...", command);
    }

    public void testGetCommandNotDefined
    {
        AdminServlet servlet = new AdminServlet();

        try
        {
            servlet.getCommand(request);
            fail("Command should not have existed");
        }
        catch (ServletException expected)
        {
            assertTrue(true);
        }
    }
}
```

Test valid case: command defined as HTTP parameter

Test invalid case: no command parameter defined

If you've typed this code in Eclipse, you'll notice that Eclipse supports what it calls *Quick Fixes*. Quick Fixes are corrections that the Java editor offers to problems found while you're typing and after compiling. The Quick Fix is visible as a lightbulb in



Figure 9.3 Working by intention with Eclipse's Quick Fixes

the left gutter. In figure 9.3, a lightbulb appears on the lines referring to the `AdminServlet` class (which does not yet exist). Clicking the lightbulb shows the list of Quick Fixes offered by Eclipse. Here, Eclipse proposes to automatically create the class for you. The same operation can then be repeated for the `getCommand` method. This is very efficient when you're using the TDD approach. (Other IDEs, like IntelliJ IDEA, also support this feature.)

Listing 9.2 shows the code Eclipse generates for you, to which you have made some modifications:

- Added the `throws ServletException` clause. You need it because the `testGetCommandNotDefined` test clearly shows that if the command parameter is not found, the `getCommand` method should return a `ServletException` exception.
- The request object in `TestAdminServlet` comes from the `ServletTestCase` class and is of type `HttpServletRequestWrapper`. This Cactus class transparently wraps an `HttpServletRequest` object and provides additional features that are useful for unit testing. Eclipse thus generated a signature of `getCommand(HttpServletRequestWrapper request)`, but what you really want is `getCommand(HttpServletRequest request)`.

Listing 9.2 Minimum code to make the `TestAdminServlet` compile

```

package junitbook.servlets;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;

public class AdminServlet extends HttpServlet
{

```



```
    public String getCommand(HttpServletRequest request)
        throws ServletException
    {
        return null;
    }
}
```

This is the minimum code that allows the `TestAdminServlet` to compile successfully.

Before you continue with the other test cases and implement the minimal application that satisfies your tests, try to run the Cactus test. It should fail, but at least you'll know it does, and that your test correctly reports a failure. Then, when you implement the code under test, the tests should succeed, and you'll know you've accomplished something. It's a good practice to ensure that the tests fail when the code fails.

JUnit best practice: always verify that the test fails when it should fail

It's a good practice to always verify that the tests you're writing work. Be sure a test fails when you expect it to fail. If you're using the Test-Driven Development (TDD) methodology, this failure happens as a matter of course. After you write the test, write a skeleton for the class under test (a class with methods that return null or throw runtime exceptions). If you try to run your test against a skeleton class, it should fail. If it doesn't, fix the test (ironically enough) so that it does fail! Even after the case is fleshed out, you can vet a test by changing an assertion to look for an invalid value that should cause it to fail.

9.2.2 Using Maven to run Cactus tests

In chapter 8, you used the Cactus/Jetty integration to run Cactus tests from an IDE. This time, you'll try to run the tests with Tomcat using the Maven Cactus plugin (<http://maven.apache.org/reference/plugins/cactus/>). Tomcat is a well-known servlet/JSP engine (it's also the reference implementation for the Servlet/JSP specifications) that can be downloaded at <http://jakarta.apache.org/tomcat/>. (For a quick reference to Maven, see chapter 5.)

The Maven Cactus plugin is one of the easiest ways to run Cactus tests. Everything is automatic and transparent for the user: *cactification* of your application war file, starting your container, deploying the cactified war, executing the Cactus tests, and stopping your container. (*Cactification* is the automatic addition of the Cactus jars and the addition of Cactus-required entries in your `web.xml` file.)

Figure 9.4 shows the directory structure. It follows the directory structure conventions introduced in chapters 3 and 8.

By default, the Maven Cactus plugin looks for Cactus tests under the `src/test-cactus` directory, which is where we have put the `TestAdminServlet` Cactus Servlet-`TestCase` class. You place under `src/webapp` all the metadata and resource files needed for your web app. For example, `src/webapp/WEB-INF/web.xml` is the application's `web.xml`. Note that the Maven Cactus plugin automatically adds Cactus-related definitions to `web.xml` during the test, which is why you must provide a `web.xml` file in your directory structure, even if it's empty. The `web.xml` content for the Administration application is as follows:

```
<?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>

    <servlet>
        <servlet-name>AdminServlet</servlet-name>
        <servlet-class>junitbook.servlets.AdminServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>AdminServlet</servlet-name>
        <url-pattern>/AdminServlet</url-pattern>
    </servlet-mapping>

</web-app>
```

You'll be running the Cactus tests with Maven, so you need to provide a valid `project.xml` file, as shown in listing 9.3 (see chapter 5 for details on Maven's `project.xml`).

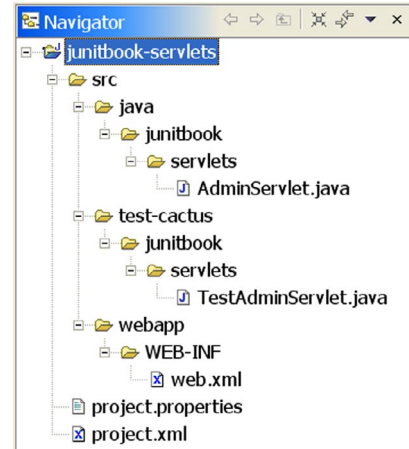


Figure 9.4 Maven directory structure for running Cactus tests

Listing 9.3 project.xml for running Maven on the junitbook-servlets project

```

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

<project>
  <pomVersion>3</pomVersion>
  <id>junitbook-servlets</id>
  <name>JUnit in Action - Unit Testing Servlets and Filters</name>
  <currentVersion>1.0</currentVersion>
  <organization>
    <name>Manning Publications Co.</name>
    <url>http://www.manning.com/</url>
    <logo>http://www.manning.com/front/dance.gif</logo>
  </organization>
  <inceptionYear>2002</inceptionYear>
  <package>junitbook.servlets</package>
  <logo>/images/jia.jpg</logo>

  <description>[...]</description>
  <shortDescription>[...]</shortDescription>
  <url>http://sourceforge.net/projects/junitbook/servlets</url>
  <developers/>

  <dependencies>

    <dependency>
      <groupId>commons-beanutils</groupId>
      <artifactId>commons-beanutils</artifactId>
      <version>1.6.1</version>
      <properties>
        <war.bundle>true</war.bundle>
      </properties>
    </dependency>

    <dependency>
      <groupId>servletapi</groupId>
      <artifactId>servletapi</artifactId>
      <version>2.3</version>
    </dependency>

    <dependency>
      <groupId>easymock</groupId>
      <artifactId>easymock</artifactId>
      <version>1.0</version>
    </dependency>

    <dependency>
      <groupId>mockobjects</groupId>
      <artifactId>mockobjects-core</artifactId>
      <version>0.09</version>
    </dependency>

  </dependencies>

```

① Jars needed in execution classpath

② Jar to be included in the war

```

<build>
  <sourceDirectory>src/java</sourceDirectory>
  <unitTestSourceDirectory>src/test</unitTestSourceDirectory>
  <unitTest>
    <includes>
      <include>**/Test*.java</include>
    </includes>
    <excludes>
      <exclude>**/Test*All.java</exclude>
    </excludes>
  </unitTest>
</build>

<reports>
  <report>maven-cactus-plugin</report>
</reports>

</project>

```

③ JUnit tests to include/exclude

④ Tell Maven to generate Cactus report

- ① Define the jars you need when compiling and running the Cactus tests. You don't need to include the Cactus-related jars (the Cactus jars, the Commons HttpClient jar, the Commons Logging jar, and so on), because they are automatically included by the Maven Cactus plugin.
- ② The Maven Cactus plugin uses the war plugin. The `<war.bundle>` element tells Maven to include the Commons BeanUtils jar in the generated production war, which is cactified by the Maven Cactus plugin. Note that you include a dependency on Commons BeanUtils, because you'll use it later in your servlet code.
- ③ Include/exclude the pure JUnit tests to match the test you wish to run. These settings only impact the tests found in `src/test` (defined by the `unitTestSourceDirectory` XML element). Thus these include/excludes have no effect on the selection of the Cactus tests (found in `src/test-cactus`). If you wish to precisely define what Cactus tests to include/exclude, you need to define the `cactus.src.includes` and `cactus.src.excludes` properties. The default values for these properties are as follows:

```

# Default Cactus test files to include in the test
cactus.src.includes = **/*Test*.java

# Default Cactus test files to exclude from the test
cactus.src.excludes = **/AllTests.java,**/Test*All.java

```

- ④ List the reports to generate. If you don't explicitly define a `reports` element in your `project.xml`, Maven will generate default reports. However, these reports

don't include Cactus tests. To generate a Cactus report as part of the web site generation (`maven site`), you must explicitly define it.

Before running the Cactus plugin, you need to tell it where Tomcat is installed on your machine, so that it can run the Cactus tests from within that container. Maven reads a `project.properties` file you put at the same level as your `project.xml` file. The Cactus plugin needs the following line added to your `project.properties`:

```
cactus.home.tomcat4x = C:/Apps/jakarta-tomcat-4.1.24
```

`C:/Apps/jakarta-tomcat-4.1.24` is the actual path where you have installed Tomcat (you can use any version of Tomcat—we're using 4.1.24 as a sample). If you don't already have it on your system, it's time to download and install it. The installation is as simple as unzipping the file anywhere on your disk.

To start the Cactus tests in Maven, type `maven cactus:test` in `project.xml`'s directory. The result of the run is shown in figure 9.5. As expected, the tests fail, because you have not yet written the code that is tested.

Maven can also generate an HTML report of the Cactus tests (see figure 9.6). By default, the Cactus plugin stops on test failures. To generate the test report, you need to add `cactus.halt.on.failure = false` to your `project.properties` (or `build.properties`) file so the build doesn't stop on test failures. Then, generate the site by typing `maven site`, which generates the web site in the `servlets/target/docs` directory.

You have executed the Cactus tests using the Tomcat container. However, the Maven Cactus plugin supports lots of other containers you can use to run your Cactus tests. Check the plugin documentation for more details (<http://maven.apache.org/reference/plugins/cactus/>).

```
cactus:test:
[cactus] -----
[cactus] Running tests against Tomcat 4.1.24
[cactus] -----
[cactus] Testsuite: junitbook.servlets.TestAdminServlet
[cactus] Tests run: 2, Failures: 2, Errors: 0, Time elapsed: 1.272 sec
[cactus]
[cactus] Testcase: testGetCommandOk(junitbook.servlets.TestAdminServlet): FAILED
[cactus] expected:<SELECT...> but was:<null>
[cactus] junit.framework.ComparisonFailure: expected:<SELECT...> but was:<null>
[cactus]   at junitbook.servlets.TestAdminServlet.testGetCommandOk(TestAdminServlet.java:20)
[cactus]   at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
[cactus]   at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
```

Figure 9.5 Executing the failing Cactus tests using Maven

Summary				
[summary] [package list] [test cases]				
Tests	Errors	Failures	Success rate	Time(s)
2	0	2	0.00%	1.50
Note: <i>failures</i> are anticipated and checked for with assertions while <i>errors</i> are unanticipated.				
Package List				
[summary] [package list] [test cases]				
junitbook.servlets	2	2	0	1.50
Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.				
junitbook.servlets				
TestAdminServlet	2	0	2	1.502
Test Cases				
[summary] [package list] [test cases]				
TestAdminServlet				
testGetCommandOk		Failure		0.89
expected:<SELECT...> but was:<null>;				
testGetCommandNotDefined		Failure		0.09
Command should not have existed				

Figure 9.6 Cactus HTML report showing the test results

Let's return to the test. Listing 9.4 shows the code for `getCommand`. It's a minimal implementation that passes the tests.

JUnit best practice: use TDD to implement The Simplest Thing That Could Possibly Work

The Simplest Thing That Could Possibly Work is an Extreme Programming (XP) principle that says over-design should be avoided, because you never know what will be used effectively. XP recommends designing and implementing the minimal working solution and then refactoring mercilessly. This is in contrast to the *monumental methodologies*,¹ which advocated fully designing the solution before starting development.

When you're developing using the TDD approach, the tests are written first—you only have to implement the bare minimum to make the test pass, in order to achieve a fully functional piece of code. The requirements have been fully expressed as test cases, and thus you can let yourself be led by the tests when you're writing the functional code.

¹ For more on agile methodologies versus monumental methodologies, read <http://www.martinfowler.com/articles/newMethodology.html>.

Listing 9.4 Implementation of getCommand that makes the tests pass

```
package junitbook.servlets;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;

public class AdminServlet extends HttpServlet
{
    public static final String COMMAND_PARAM = "command";

    public String getCommand(HttpServletRequest request)
        throws ServletException
    {
        String command = request.getParameter(COMMAND_PARAM);
        if (command == null)
        {
            throw new ServletException("Missing parameter ["
                + COMMAND_PARAM + "]");
        }
        return command;
    }
}
```

Running the tests again by typing `maven cactus:test` yields the result shown in figure 9.7.

```
cactus:test:
[cactus] -----
[cactus] Running tests against Tomcat 4.1.24
[cactus] -----
[cactus] Testsuite: junitbook.servlets.TestAdminServlet
[cactus] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 1.262 sec
[cactus]
[cactus] Testcase: testGetCommandOk took 0.671 sec
[cactus] Testcase: testGetCommandNotDefined took 0.091 sec
BUILD SUCCESSFUL
Total time: 20 seconds
```

Figure 9.7 Execution of successful Cactus tests using Maven

9.2.3 Finishing the Cactus servlet tests

At the beginning of section 9.2, we mentioned that you need to write three methods: `getCommand`, `executeCommand`, and `callView`. You implemented `getCommand` in listing 9.4. The `executeCommand` method is responsible for obtaining data from the database. We'll defer this implementation until chapter 11, "Unit-testing database applications."

That leaves the `callView` method, along with the servlet `doGet` method, which ties everything together by calling your different methods. One way of designing the application is to store the result of the `executeCommand` method in the HTTP servlet request. The request is passed to the JSP by the `callView` method (via `request.forward()`). The JSP can then access the data to display by getting it from the request (possibly using a `useBean` tag). This is a typical MVC Model 2 pattern used by many applications and frameworks.

Design patterns in action: MVC Model 2 pattern

MVC stands for Model View Controller. This pattern is used to separate the core business logic layer (the Model), the presentation layer (the View), and the presentation logic (the Controller), usually in web applications. In a typical MVC Model 2 design, the Controller is implemented as a servlet and handles all incoming HTTP requests. It's in charge of calling the core business logic services and dynamically choosing the right view (often implemented as a JSP). The Jakarta Struts framework (<http://jakarta.apache.org/struts/>) is a popular implementation of this pattern.

You still need to define what objects `executeCommand` will return. The `BeanUtils` package in the Jakarta Commons (<http://jakarta.apache.org/commons/beanutils/>) includes a `DynaBean` class that can expose public properties, like a regular `JavaBean`, but you don't need to hard-code getters and setters. In a Java class, you access one of the dyna-properties using a map-like accessor:

```
DynaBean employee = ...
String firstName = (String) employee.get("firstName");
employee.set("firstName", "vincent");
```

The `BeanUtils` framework is nice for the current use case because you'll retrieve arbitrary data from the database. You can construct dynamic `JavaBeans` (or `dyna beans`) that you'll use to hold database data. The actual mapping of a database to `dyna beans` is covered in chapter 11.

Testing the `callView` method

There's enough in place now that you can write the tests for `callView`, as shown in listing 9.5.

Listing 9.5 Unit tests for callView

```

package junitbook.servlets;
[...]
```

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

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

```

public class TestAdminServlet extends ServletTestCase
{
[...]
```

private Collection createCommandResult() **throws** Exception

{

 List results = **new** ArrayList();

 DynaProperty[] props = **new** DynaProperty[] {

new DynaProperty("id", String.class),

new DynaProperty("responsetime", Long.class)

 };

 BasicDynaClass dynaClass = **new** BasicDynaClass("requesttime",

null, props);

 DynaBean request1 = dynaClass.newInstance();

 request1.set("id", "12345");

 request1.set("responsetime", **new** Long(500));

 results.add(request1);

 DynaBean request2 = dynaClass.newInstance();

 request1.set("id", "56789");

 request1.set("responsetime", **new** Long(430));

 results.add(request2);

return results;

}

```

public void testCallView() throws Exception
{
    AdminServlet servlet = new AdminServlet();

    // Set the result of the execution of the command in the
    // HTTP request so that the JSP page can get the data to
    // display
    request.setAttribute("result", createCommandResult());
    servlet.callView(request);
}
}

```

Create objects to be returned by execute-Command

2 Set execution results in HTTP requests

To make the test easier to read, you create a `createCommandResult` private method. This utility method creates arbitrary DynaBean objects, like those that will be returned by `executeCommand`. In `testCallView`, you place the dyna beans in the HTTP request where the JSP can find them.

There is nothing you can verify in `testCallView`, so you don't perform any asserts there. The call to `callView` forwards to a JSP. However, Cactus supports asserting the result of the execution of a JSP page. So, you can use Cactus to verify that the JSP will be able to display the data that you created in `createCommandResult`. Because this would be JSP testing, we'll show how it works in chapter 10 ("Unit-testing JSPs and taglibs").

Listing 9.6 shows the simplest code that makes the `testCallView` test pass.

Listing 9.6 Implementation of `callView` that makes the tests pass

```
package junitbook.servlets;
[...]
```

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

```
    public void callView(HttpServletRequest request)
    {
    }
}
```

You don't have a test yet for the returned result, so not returning anything is enough. That will change once you test the JSP.

Testing the `doGet` method

Let's design the unit test for the `AdminServlet` `doGet` method. To begin, you need to verify that the test results are put in the servlet request as an attribute. Here's how you can do that:

```
Collection results = (Collection) request.getAttribute("result");
assertNotNull("Failed to get execution results from the request",
    results);
assertEquals(2, results.size());
```

This code leads to storing the command execution result in `doGet`. But where do you get the result? Ultimately, from the execution of `executeCommand`—but it isn't implemented yet. The typical solution to this kind of deadlock is to have an `executeCommand` that does nothing in `AdminServlet`. Then, in your test, you can implement `executeCommand` to return whatever you want:

```
AdminServlet servlet = new AdminServlet()
{
```

```

        public Collection executeCommand(String command)
            throws Exception
        {
            return createCommandResult();
        }
    };

```

You can now store the result of the test execution in `doGet`:

```

public void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException
{
    try
    {
        Collection results = executeCommand(getCommand(request));
        request.setAttribute("result", results);
    }
    catch (Exception e)
    {
        throw new ServletException(
            "Failed to execute command", e);
    }
}

```

Notice that you need the catch block because the Servlet specification says `doGet` must throw a `ServletException`. Because `executeCommand` can throw an exception, you need to wrap it into a `ServletException`.

If you run this code, you'll find that you have forgotten to set the command to execute in the HTTP request as a parameter. You need a `beginDoGet` method, such as this:

```

public void beginDoGet(WebRequest request)
{
    request.addParameter("command", "SELECT...");
}

```

The completed unit test is shown in listing 9.7.

Listing 9.7 Unit test for `doGet`

```

package junitbook.servlets;
[...]
public class TestAdminServlet extends ServletTestCase
{
    [...]
    public void beginDoGet(WebRequest request)
    {
        request.addParameter("command", "SELECT...");
    }
}

```

```

public void testDoGet() throws Exception
{
    AdminServlet servlet = new AdminServlet()
    {
        public Collection executeCommand(String command)
            throws Exception
        {
            return createCommandResult();
        }
    };

    servlet.doGet(request, response);

    // Verify that the result of executing the command has been
    // stored in the HTTP request as an attribute that will be
    // passed to the JSP page.
    Collection results =
        (Collection) request.getAttribute("result");
    assertNotNull("Failed to get execution results from the "
        + "request", results);
    assertEquals(2, results.size());
}
}

```

The doGet code is shown in listing 9.8.

Listing 9.8 Implementation of doGet that makes the tests pass

```

package junitbook.servlets;
[...]
```

public class AdminServlet extends HttpServlet

```

{
[...]
```

public Collection executeCommand(String command)

```

    throws Exception
    {
        throw new RuntimeException("not implemented");
    }
}

```

public void doGet(HttpServletRequest request,

```

    HttpServletResponse response) throws ServletException
    {
        try
        {
            Collection results =
                executeCommand(getCommand(request));
            request.setAttribute("result", results);
        }
        catch (Exception e)
        {

```

Throws exception if called; not implemented yet

```
        throw new ServletException(  
            "Failed to execute command", e);  
    }  
}
```

There are two points of note. First, the call to `callView` is not present in `doGet`; the tests don't yet mandate it. (They will, but not until you write the unit tests for your JSP.) Second, you throw a `RuntimeException` object if `executeCommand` is called. You could return `null`, but throwing an exception is a better practice. An exception clearly states that you have not implemented the method. If the method is called by mistake, there won't be any surprises.

JUnit best practice: throw an exception for methods that aren't implemented

When you're writing code, there are often times when you want to execute the code without having finished implementing all methods. For example, if you're writing a mock object for an interface and the code you're testing uses only one method, you don't need to mock all methods. A very good practice is to throw an exception instead of returning `null` values (or not returning anything for methods with no return value). There are two good reasons: Doing this states clearly to anyone reading the code that the method is not implemented *and* ensures that if the method is called, it will behave in such a way that you cannot mistake skeletal behavior for real behavior.

9.3 Testing servlets with mock objects

You have seen how to write servlet unit tests using Cactus. Let's try to do the same exercise using only a mock-objects approach. We'll then define some rules for deciding when to use the Cactus approach and when to use mock objects.

In chapter 8, you used EasyMock to write mock objects. This time you'll use the DynaMock API, which is part of the MockObjects.com framework (<http://www.mockobjects.com/>). They both use Dynamic Proxies to generate mock objects at runtime. However, the DynaMock framework has several advantages over EasyMock: Its API is more comprehensive (notably in the definition of the expectations), and it results in more concise code. The downside is that it's slightly more complex to use (at least initially), and it's a less mature framework. (However, we haven't resisted the temptation to show you how to use it, because we think it has a great future.)

EasyMock vs. DynaMock

- DynaMock provides more concise code (about half as much code as EasyMock).
- EasyMock provides strong typing, which is useful for auto-completion and when interfaces change.
- DynaMock has a more comprehensive API (especially for expectations).
- EasyMock is more mature, because it has been around for several years. DynaMock is very new, and its API is not completely stabilized (as of this writing).

9.3.1 Writing a test using DynaMocks and DynaBeans

Listing 9.9 shows the re-implementation of `testGetCommandOk` and `testGetCommandNotDefined` from listing 9.1.

Listing 9.9 Tests for `AdminServlet.getCommand` using the DynaMock API

```
package junitbook.servlets;
[...]
```

```
import com.mockobjects.dynamic.C;
import com.mockobjects.dynamic.Mock;

public class TestAdminServletMO extends TestCase
{
    private Mock mockRequest;
    private HttpServletRequest request;
    private AdminServlet servlet;

    protected void setUp()
    {
        servlet = new AdminServlet();

        mockRequest = new Mock(HttpServletRequest.class);
        request = (HttpServletRequest) mockRequest.proxy();
    }

    protected void tearDown()
    {
        mockRequest.verify();
    }

    public void testGetCommandOk() throws Exception
    {
        mockRequest.expectAndReturn("getParameter", "command",
                                   "SELECT...");
    }
}
```

1

2

3

```

        String command = servlet.getCommand(request);
        assertEquals("SELECT...", command);
    }

    public void testGetCommandNotDefined()
    {
        mockRequest.expectAndReturn("getParameter",
            C.isA(String.class), null);

        try
        {
            servlet.getCommand(request);
            fail("Command should not have existed");
        }
        catch (ServletException expected)
        {
            assertTrue(true);
        }
    }
}

```

4

- ❶ You're using an `HttpServletRequest` object in the code to test; so, because you aren't running inside a container, you need to create a mock for it. Here you tell the DynaMock API to generate an `HttpServletRequest` mock for you.
- ❷ Ask your mock to verify the expectations you have set on it and to verify that the methods for which you have defined behaviors have been called.
- ❸ Tell the mock to return "SELECT..." when the `getParameter` method is called with the "command" string as parameter.
- ❹ Tell the mock request to return null when `getParameter` is called with a string parameter passed to it.

9.3.2 Finishing the DynaMock tests

Let's finish transforming the other tests from listing 9.1 into DynaMock tests. Listing 9.10 shows the results.

Listing 9.10 Tests for `callView` and `doGet` with dynamic mocks

```

package junitbook.servlets;
[...]
```

public class TestAdminServletMO **extends** TestCase

```

{
[...]
```

private Mock mockResponse;

private HttpServletResponse response;

1

```

protected void setUp()
{
    servlet = new AdminServlet()
    {
        public Collection executeCommand(String command)
            throws Exception
        {
            return createCommandResult();
        }
    };
[...]
```

mockResponse = new Mock(HttpServletResponse.class);
response = (HttpServletResponse) mockResponse.proxy(); ❶

```

[...]
```

private Collection createCommandResult() throws Exception
{
 // Same as in listing 9.5
}

public void testCallView() throws Exception
{
 servlet.callView(request);
}

public void testDoGet() throws Exception
{
 mockRequest.expectAndReturn("getParameter", "command",
 "SELECT..."); ❶
 // Verify that the result of executing the command has been
 // stored in the HTTP request as an attribute that will be
 // passed to the JSP page.
 mockRequest.expect("setAttribute", C.args(C.eq("result"),
 C.isA(Collection.class))); ❶
 servlet.doGet(request, response);
}

```

}
```

- ❶ You need a new mock for the HttpServletResponse class (used in doGet).
- ❷ Set the behaviors of the mock HttpServletRequest object. You also tell DynaMock to verify that the methods are called and that the parameters they are passed match what is expected. For example, you verify that the setAttribute method call is passed a first parameter matching the "result" string and that the second parameter is a Collection object.

You now have a fully working test suite using mock objects that exercises your servlet code.

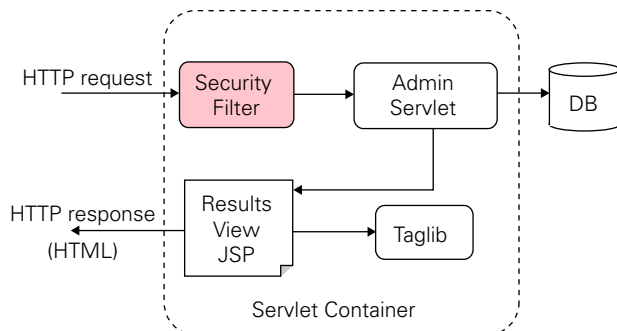


Figure 9.8
Unit-testing the
SecurityFilter of the
Administration application

9.4 Writing filter tests with Cactus

Now that you know how to unit-test servlets, let's change the focus to filters—in particular, the SecurityFilter specified by figure 9.8.

The requirement for the SecurityFilter is to intercept all HTTP requests and verify that the incoming SQL statement doesn't contain any harmful commands. For now, you'll only check whether the SQL query contains a SELECT statement; if it doesn't, you'll forward to an error page (see listing 9.11).

Listing 9.11 SecurityFilter.java

```

package junitbook.servlets;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class SecurityFilter implements Filter
{
    private String securityErrorPage;

    public void init(FilterConfig theConfig) throws ServletException
    {
        this.securityErrorPage =
            theConfig.getInitParameter("securityErrorPage");
    }

    public void doFilter(ServletRequest theRequest,
        ServletResponse theResponse, FilterChain theChain)
        throws IOException, ServletException
  
```

Get name
of error
page from
web.xml

```

    {
        String sqlCommand =
            theRequest.getParameter(AdminServlet.COMMAND_PARAM);

        if (!sqlCommand.startsWith("SELECT"))
        {
            // Forward to an error page
            RequestDispatcher dispatcher =
                theRequest.getRequestDispatcher(
                    this.securityErrorPage);
            dispatcher.forward(theRequest, theResponse);
        }
        else
        {
            theChain.doFilter(theRequest, theResponse);
        }
    }

    public void destroy()
    {
    }
}

```

Redirect to error page

Testing this filter with Cactus is very similar to the tests you have already performed on the AdminServlet. The main difference is that the `TestCase` extends `FilterTestCase` instead of `ServletTestCase`. This change allows the test to get access to the Filter API objects (`FilterConfig`, `Request`, `Response`, and `FilterChain`).

9.4.1 Testing the filter with a **SELECT** query

Listing 9.12 tests the `doFilter` method when the SQL query that is passed is a `SELECT` query.

Listing 9.12 TestSecurityFilter.java (testDoFilterAllowedSQL)

```

package junitbook.servlets;

import java.io.IOException;

import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

import org.apache.cactus.FilterTestCase;
import org.apache.cactus.WebRequest;

public class TestSecurityFilter extends FilterTestCase
{
    public void beginDoFilterAllowedSQL(WebRequest request)
    {
    }
}

```

```
{
    request.addParameter("command", "SELECT [...]"); ❶
}

public void testDoFilterAllowedSQL() throws Exception
{
    SecurityFilter filter = new SecurityFilter();
    FilterChain mockFilterChain = new FilterChain()
    {
        public void doFilter(ServletRequest theRequest,
            ServletResponse theResponse) throws IOException,
            ServletException
        {
        }

        public void init(FilterConfig theConfig)
        {
        }

        public void destroy()
        {
        }
    };

    filter.doFilter(request, response, mockFilterChain);
}
}
```

- ❶ Use the Cactus `beginXXX` method to add the SQL command to the HTTP request that is processed by your filter. (Note that your SQL query is a `SELECT` query.)
- ❷ For this test, you don't want your filter to call the next filter in the chain (or the target JSP/servlet). Thus you create an empty implementation of a `FilterChain`. You could also let the filter call the next element in the chain. However, a filter is completely independent from other filters or any JSP/servlet that might be called after it in the processing chain. Thus, it makes more sense to test this filter in isolation, especially given that the filter doesn't modify the returned HTTP response.

9.4.2 Testing the filter for other query types

So far, you have tested only one scenario from your filter. You also need to verify that the behavior is correct when the SQL command that is passed is not a `SELECT` query (see listing 9.13). In that case, the filter behavior should be to redirect the user to an error page. For example, here is the code for `securityError.jsp`, the JSP error page you're forwarding to in the `testDoFilterForbiddenSQL` test in listing 9.13:

```

<html>
  <head>
    <title>Security Error Page</title>
  </head>
  <body>
    <p>
      Only SELECT SQL queries are allowed!
    </p>
  </body>
</html>

```

Listing 9.13 TestSecurityFilter.java (testDoFilterForbiddenSQL)

```

package junitbook.servlets;
[...]
public class TestSecurityFilter extends FilterTestCase
{
  [...]
  public void beginDoFilterForbiddenSQL(WebRequest request)
  {
    request.addParameter("command", "UPDATE [...]"); ❶
  }

  public void testDoFilterForbiddenSQL() throws Exception
  {
    config.setInitParameter("securityErrorPage",
      "/securityError.jsp"); ❷
    SecurityFilter filter = new SecurityFilter();
    filter.init(config); ❸
    filter.doFilter(request, response, filterChain);
  }

  public void endDoFilterForbiddenSQL(WebResponse response)
  {
    assertTrue("Bad response page",
      response.getText().indexOf(
        "<title>Security Error Page</title>" > 0); ❹
  }
}

```

- ❶ Pass a SQL query that is not a SELECT.
- ❷ Use a Cactus-specific API (`config.setInitParameter`) to simulate an init parameter that represents the name of the security error page. Note that this is the equivalent of defining the init parameter in your web application's `web.xml` file, like this:

```

<filter>
  <filter-name>FilterRedirector</filter-name>
  <filter-class>

```

```

        org.apache.cactus.server.FilterTestRedirector
    </filter-class>
    <init-param>
        <param-name>securityErrorPage</param-name>
        <param-value>/securityError.jsp</param-value>
    </init-param>
</filter>

```

Notice that you add the `init` parameter to the Cactus `FilterRedirector` definition—not to the `SecurityFilter` definition. This is because in the test, you instantiate your `SecurityFilter` class as a plain old Java object (POJO), not as a filter. Cactus, under the hood, calls a Cactus filter redirector that it uses to provide valid filter objects (`Request`, `Response`, `FilterConfig`, `FilterChain`) to your `testXXX` method. However, it's simpler to use the `setInitParameter` method as shown here.

- 3 You have instantiated your filter as a POJO, so you need to call its `init(FilterConfig)` method to correctly initialize the filter. (This is what the container would have done internally.)
- 4 To verify that the `SecurityFilter` has correctly forwarded you to the error page, check that the returned HTTP response body contains elements that you expect to be present in the error page.

9.4.3 Running the Cactus filter tests with Maven

Running your filter tests with Maven is easy. The only prerequisite is to put your sources in the directory structure expected by Maven. Figure 9.9 demonstrates this structure.

The filter code under test is located under `src/java`, the Cactus tests are in `src/test-cactus`, and the web-app resources (`web.xml` and JSPs, for example) are located in `src/webapp`. These are the default locations where Maven expects to find the different sources.

Running the Cactus tests is simply a matter of opening a shell in the `junitbook-servlets` project directory and entering **maven cactus:test**. The Maven `cactus:test` goal automatically calls the Maven `war` goal, which packages your application in a war file. Then the Cactus plugin repackages this war (by

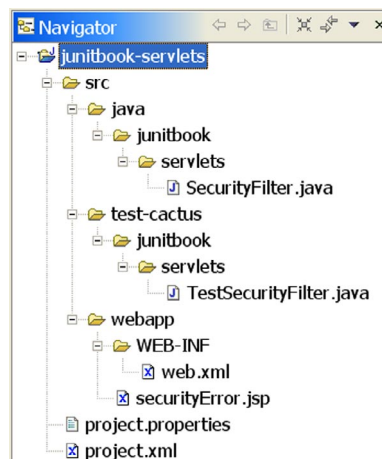


Figure 9.9 Directory structure for the Cactus filter tests

```
cactus:test:
[cactus] -----
[cactus] Running tests against Tomcat 4.1.24
[cactus] -----
[cactus] Testsuite: junitbook.servlets.TestSecurityFilter
[cactus] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 19.218 sec
[cactus]
[cactus] Testcase: testDoFilterAllowedSQL took 0.641 sec
[cactus] Testcase: testDoFilterForbiddenSQL took 18.086 sec
BUILD SUCCESSFUL
Total time: 38 seconds
```

Figure 9.10 Run the Cactus tests using the Maven Cactus plugin.

adding the Cactus jars along with definitions for the Cactus redirectors in the `web.xml` file), deploys it to your target container, starts the container, runs the tests, and stops the container. Figure 9.10 shows the result.

9.5 When to use Cactus, and when to use mock objects

At this point, you must be wondering whether to use Cactus or mock objects to test your servlets and filter. Both approaches have advantages and disadvantages:

- The main difference is that Cactus performs not only unit tests but also integration tests and, to some extent, functional tests. The added benefits come at the cost of added complexity.
- Mock-object tests are usually harder to write, because you need to define the behavior of all calls made to the mocks. For example, if your method under test makes 10 calls to mocks, then you need to define the behavior for these 10 calls as part of the test setup.
- Cactus provides real objects for which you only need to set some initial conditions.
- If the application to unit-test is already written, it usually has to be refactored to support mock-object testing. Extra refactoring is generally not needed with Cactus.

A good strategy is to separate the business-logic code from the integration code (code that interacts with the container), and then:

- Use mock objects to test the business logic.
- Use Cactus to test the integration code.

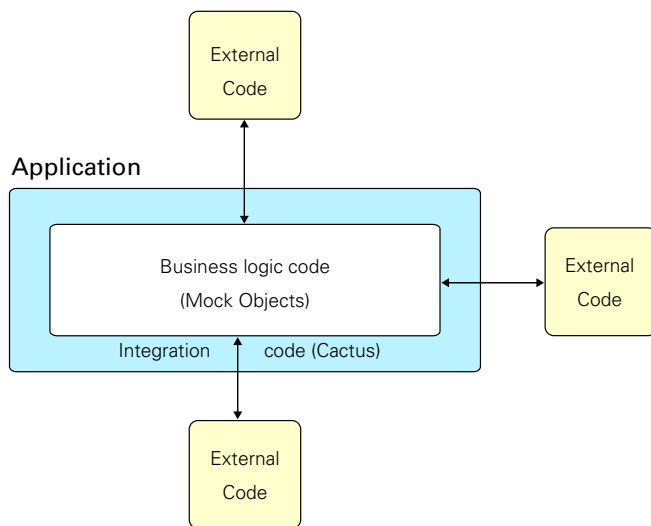


Figure 9.11
Where to use Cactus, and
where to use mock objects

Figure 9.11 illustrates this concept. In the example, the business-logic code is the `executeCommand` method; the rest is integration code.

The question is not so much *whether* you should use Cactus *or* mock objects, but rather *where* to use Cactus *and* mock objects. The approaches are not exclusive but complementary. Both can be used to serve the prime objective: increasing overall application quality by discovering as many bugs as early as possible.

9.6 Summary

In this chapter, we have demonstrated how to unit-test servlets and filters and, more generally, any code that uses the Servlet/Filters API. You can create and run these kinds of tests using mock objects or Jakarta Cactus. Although the mock-objects approach can unit-test servlet and filter code, the tests cannot achieve the wide range that's possible with Cactus-based tests. In practice, the approaches are complementary. Use mock objects to unit-test business-logic code at a very fine-grained level, and use Cactus to unit-test integration code (code that interacts with the container).

In the next chapter, we'll continue unit-testing the Administration application by moving the focus to unit-testing the JavaServer Pages and Taglib APIs.

10

Unit-testing JSPs and taglibs

This chapter covers

- Unit-testing a JSP in isolation with Cactus and mock objects
- Running Cactus JSP tests with Maven
- Unit-testing taglibs with Cactus
- Unit-testing taglibs with mock objects and MockMaker

A test that can't be repeated is worthless.

—Brian Marick

In this chapter, we'll continue with the Administration application we introduced in chapter 9. In chapter 9, we focused on unit-testing the servlet component of the application. In this chapter, we concentrate on the view components—namely the JavaServer Pages (JSPs) and custom tag libraries (taglibs).

We'll cover unit-testing JSPs and taglibs with both Cactus and mock objects. The two techniques are complementary. Mock objects excel at writing focused, fine-grained unit tests against the business logic. Meanwhile, Cactus can perform integration unit tests against the target environment. The integration unit tests are essential in order to ensure that all components work properly when run in their target containers.

10.1 Revisiting the Administration application

We'll base our examples on the Administration application (introduced in chapter 9). Its architecture is shown in figure 10.1, which also highlights the parts for which you'll write unit tests (shaded boxes).

You use the application by sending an HTTP request (from your browser) to the AdminServlet. You pass an SQL query to run as an HTTP parameter, which is retrieved by the AdminServlet. The security filter intercepts the HTTP request and verifies that the SQL query is harmless (that is, it's a SELECT query). Then, the servlet executes the query on the database, stores the resulting objects in the HTTP Request object, and calls the Results View page. The JSP takes the results from the Request and displays them, nicely formatted, using custom JSP tags from your tag library.

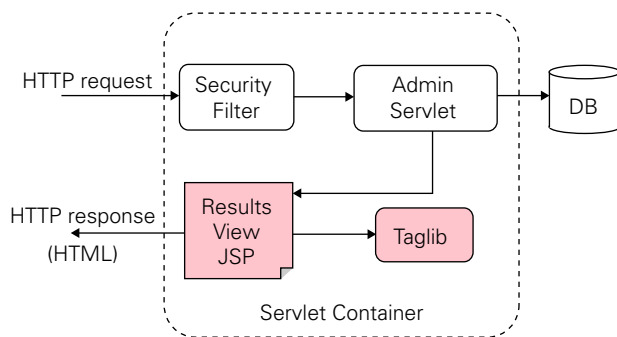


Figure 10.1
Unit-testing the Results View JSP
from the Administration application

10.2 What is JSP unit testing?

First, let's remove any doubt: What we call *unit-testing a JSP* is not about unit-testing the servlet that is generated by the compilation of the JSP. We also assume that the JSP is well designed, which means there is no Java code in it. If the page must handle any presentation logic, the logic is encapsulated in a JavaBean or in a taglib. You can perform two kinds of tests to unit-test a JSP: test the JSP page itself in isolation and/or test the JSP's taglibs.

You can isolate the JSP from the back end by simulating the JavaBeans it uses and then verifying that the returned page contains the expected data. We'll use Cactus (see chapter 8) to demonstrate this type of test. Because mock objects (see chapter 7) operate only on Java code, you can't use a pure mock-objects solution to unit-test your JSP in isolation.

You could also write functional tests for the JSP using a framework such as HttpUnit. However, doing so means going all the way to the back end of the application, possibly to the database. With a combination of Cactus and mock objects, you can prevent calling the back end and keep your focus on unit-testing the JSPs themselves.

You can also unit-test the custom tags used in the JSP. You'll do this with both Cactus and mock objects. Both have pros and cons, and they can be used together effectively.

10.3 Unit-testing a JSP in isolation with Cactus

The strategy for unit-testing JSPs in isolation with Cactus is defined in figure 10.2.

Here is what happens. The Cactus test case class must extend `ServletTestCase` (or `JspTestCase`):

- ❶ In the `testXXX` method (called by Cactus from inside the container), you create the mock objects that will be used by the JSP. The JSP gets its dynamic information either from one container-implicit object (`HttpServletRequest`, `HttpServletResponse`, or `ServletConfig`) or from a taglib. (We handle the taglib case in section 10.4.)
- ❷ Still in `testXXX`, you perform a forward to call the JSP under test. The JSP then executes, getting the mock data set up in ❶.

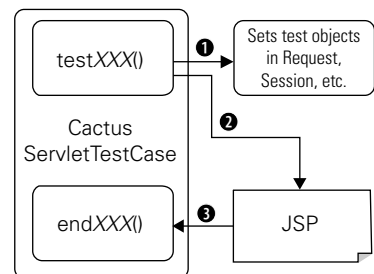
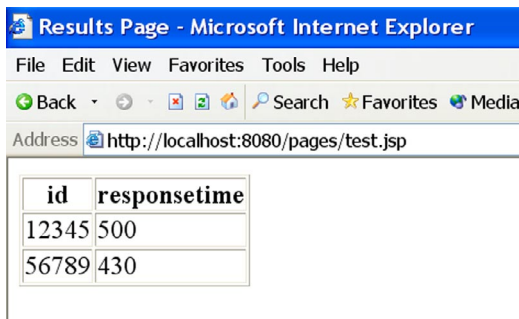


Figure 10.2 Strategy to unit-test JSPs with Cactus

- 3 Cactus calls `endXXX`, passing to it the output from the JSP. This allows you to assert the content of the output and verify that the data you set up found its way to the JSP output, in the correct location on the page.

10.3.1 Executing a JSP with SQL results data

Let's see some action on the Administration application. In chapter 9 ("Unit-testing servlets and filters"), you defined that the results of executing the SQL query would be passed to the JSP by storing them as a collection of DynaBean objects in the `HttpServletRequest` object. Thanks to the dynamic nature of dyna beans, you can easily write a generic JSP that will display any data contained in the dyna beans. Dyna beans provide metadata about the data they contain. You can create a generic table with columns corresponding to the fields of the dyna beans, as shown in listing 10.1. The result of executing this JSP (using arbitrary SQL results data) is shown in figure 10.3.



id	responsetime
12345	500
56789	430

Figure 10.3
Result of executing `results.jsp`
with arbitrary data that comes from
the execution of a SQL query

Listing 10.1 Results View JSP (`results.jsp`)

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib prefix="c"
    uri="http://jakarta.apache.org/taglibs/core" %>
<%@ taglib prefix="d" uri="/dynabeans" %>

<html>
  <head>
    <title>Results Page</title>
  </head>
  <body bgcolor="white">
    <table border="1">

      <d:properties var="properties"
        item="${requestScope.results[0]}" />

      <tr>
        <c:forEach var="property" items="${properties}">
```

1

```

        <th><c:out value="\${property.name}"/></th>
      </c:forEach>
    </tr>

    <c:forEach var="result" items="\${requestScope.results}">
      <tr>
        <c:forEach var="property" items="\${properties}">
          <td><d:getProperty name="\${property.name}"
            item="\${result}"/></td>
        </c:forEach>
      </tr>
    </c:forEach>
  </table>
</body>
</html>

```

You use both JSTL tags and custom taglibs to write the JSP: The JSTL tag library is a standard set of useful and generic tags. It's divided into several categories (core, XML, formatting, and SQL). The category used here is the core, which provides output, management of variables, conditional logic, loops, text imports, and URL manipulation. The JSTL implementation used is the Jakarta Standard 1.0 implementation (<http://jakarta.apache.org/taglibs/>) of the JSTL specifications (<http://java.sun.com/products/jsp/jstl/>).

You also write two custom tags (<d:properties> and <d:getProperty>), which are used to extract information from the dyna beans. <d:properties> (❶) extracts the name of all properties of a dyna bean, and <d:getProperty> (❷) extracts the value of a given dyna bean property.

There are two reasons for writing these custom tags. The primary reason is that it isn't possible to extract dyna bean information without (*ouch!*) embedding Java code in the JSP (at least, not with the current implementation of the JSTL tags and the DynaBean package). The second reason is that it gives you a chance to write and unit-test custom taglibs of your own. (Of course, the Struts 1.1 tags are dyna-bean-aware, and you could use those, but we decided not to overload this chapter with yet another framework.)

10.3.2 Writing the Cactus test

Now let's write a Cactus ServletTestCase for the JSP. In chapter 9, you defined a method named `callView` from the `AdminServlet` class. The `callView` method forwards control to the Results View JSP, as shown in listing 10.2.

Listing 10.2 AdminServlet.callView implementation

```

package junitbook.pages;
[...]
import java.io.IOException;

public class AdminServlet extends HttpServlet
{
    [...]

    public void callView(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException
    {
        request.getRequestDispatcher("/results.jsp")
            .forward(request, response);
    }
}

```

Listing 10.3 shows a unit test for `callView` that sets up the DynaBean objects in the Request, calls `callView`, and then verifies that the JSP output is what you expect.

Listing 10.3 TestAdminServlet.java: unit tests for results.jsp

```

package junitbook.pages;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

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

public class TestAdminServlet extends ServletTestCase
{
    private Collection createCommandResult() throws Exception
    {
        List results = new ArrayList();

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

        DynaBean request1 = dynaClass.newInstance();
        request1.set("id", "12345");
        request1.set("responsetime", new Long(500));
    }
}

```

Create test input
data for JSP



```

        results.add(request1);

        DynaBean request2 = dynaClass.newInstance();
        request2.set("id", "56789");
        request2.set("responsetime", new Long(430));
        results.add(request2);

        return results;
    }

    public void testCallView() throws Exception
    {
        AdminServlet servlet = new AdminServlet();
        request.setAttribute("results", createCommandResult());
        servlet.callView(request, response);
    }

    public void endCallView(
        com.meterware.httpunit.WebResponse response)
        throws Exception
    {
        assertTrue(response.isHTML());

        assertEquals("tables", 1, response.getTables().length);
        assertEquals("columns", 2,
            response.getTables()[0].getColumnCount());
        assertEquals("rows", 3,
            response.getTables()[0].getRowCount());

        assertEquals("id",
            response.getTables()[0].getCellAsText(0, 0));
        assertEquals("responsetime",
            response.getTables()[0].getCellAsText(0, 1));

        assertEquals("12345",
            response.getTables()[0].getCellAsText(1, 0));
        assertEquals("500",
            response.getTables()[0].getCellAsText(1, 1));
        assertEquals("56789",
            response.getTables()[0].getCellAsText(2, 0));
        assertEquals("430",
            response.getTables()[0].getCellAsText(2, 1));
    }
}

```

Create test input data for JSP

Use HttpUnit integration for asserting HTTP response

You use the Cactus HttpUnit integration in the `endCallView` method to assert the returned HTML page. When Cactus needs to execute the `endXXX` method, first it looks for an `endXXX(org.apache.cactus.WebResponse)` signature. If this signature is found, Cactus calls it; if it isn't, Cactus looks for an `endXXX(com.meterware.httpunit.WebResponse)` signature and, if it's available, calls it. Using the

`org.apache.cactus.WebResponse` object, you can perform asserts on the content of the HTTP response, such as verifying the returned cookies, the returned HTTP headers, or the content. The Cactus `org.apache.cactus.WebResponse` object sports a simple API. The HttpUnit web response API (`com.meterware.httpunit.WebResponse`) is much more comprehensive. With HttpUnit, you can view the returned XML or HTML pages as DOM objects. In listing 10.3, you use the provided HTML DOM to verify that the returned web page contains the expected HTML table.

10.3.3 Executing Cactus JSP tests with Maven

Let's run the Cactus tests with the Maven plugin for Cactus (introduced in chapter 9). The Maven directory structure for this chapter is shown in figure 10.4. The figure lists not only the `AdminServlet` and `TestAdminServlet` classes but also tag library classes that you'll develop in section 10.4.

As usual, you put the Java source files in `src/java` (as required by Maven) and the Cactus tests in `src/test-cactus` (as required by the Maven Cactus plugin). Internally, the Maven Cactus plugin calls the Maven war plugin, which requires the web application resource and configuration files to be put in `src/webapp`. The Maven project configuration files (`project.xml` and `project.properties`) are put in the root directory. Table 10.1 describes the different project files.

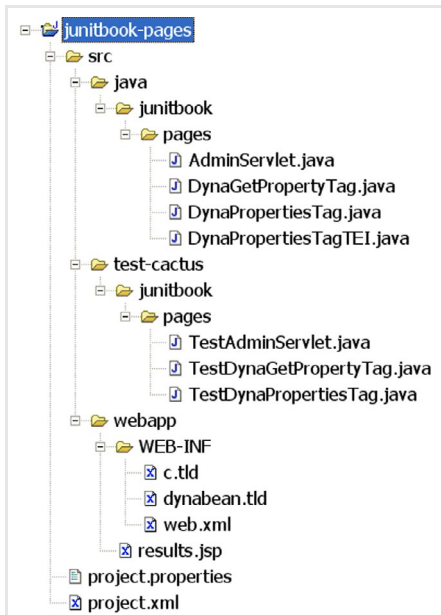


Figure 10.4
Directory structure for the JSP tests
showing how to set up a web app for the
Maven Cactus plugin

Table 10.1 Source files and directories for the JSP sample project

File and directory locations	Description
<code>src/java/junitbook/pages/</code>	Main runtime Java sources
<code>AdminServlet.java</code>	Administration servlet that forwards to the <code>results.jsp</code> JSP
<code>Dyna*.java</code>	Custom taglib implementations described in detail in section 10.4
<code>src/test-cactus/junitbook/pages/</code>	Cactus unit tests
<code>TestAdminServlet.java</code>	Unit test class for unit-testing the <code>callView</code> method of the administration servlet
<code>TestDyna*.java</code>	Unit tests for the custom taglibs
<code>src/webapp/results.jsp</code>	Results View JSP that you want to unit test
<code>src/webapp/WEB-INF/</code>	Web app configuration files
<code>c.tld</code>	Configuration file for the JSTL Core taglib
<code>dynabean.tld</code>	Configuration file for the custom taglib
<code>web.xml</code>	Main web app configuration file containing the taglib mapping between the URIs used in the JSP and the taglib configuration files (<code>.tld</code> files)
<code>project.properties</code>	Maven configuration file
<code>project.xml</code>	Maven project descriptor

NOTE For conciseness, the `.tld`, `web.xml`, and `project.xml` file contents are not shown here. However, they can be downloaded from the book's web site (see appendix A for details).

Before you execute your tests, you need to tell the Maven Cactus plugin what servlet container to use to execute the Cactus tests. To do so, add a property in your `project.properties` or `build.properties` file. This property defines where the container is installed on the local hard disk. For example, if you want to run the tests in Tomcat 4.1.24, you need to add the following property (assuming you've installed Tomcat in `c:/Apps/jakarta-tomcat-4.1.24`):

```
cactus.home.tomcat4x = C:/Apps/jakarta-tomcat-4.1.24
```

Starting the Maven Cactus plugin is as simple as opening a shell in the `junitbook/pages/` directory and typing **maven cactus:test**. Figure 10.5 shows the result.


```

cactus:test:
[cactus] -----
[cactus] Running tests against Tomcat 4.1.24
[cactus] -----
[cactus] Testsuite: junitbook.pages.TestAdminServlet
[cactus] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 5.468 sec
[cactus]
[cactus] Testcase: testCallUView took 4.997 sec
[cactus] Testsuite: junitbook.pages.TestDynaGetPropertyTag
[cactus] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 2.454 sec
[cactus]
[cactus] Testcase: testDoStartTag took 1.963 sec
[cactus] Testsuite: junitbook.pages.TestDynaPropertiesTag
[cactus] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 1.051 sec
[cactus]
[cactus] Testcase: testDoStartTag took 0.571 sec
BUILD SUCCESSFUL
Total time: 29 seconds

```

Figure 10.5 Cactus test results for the JSP (`results.jsp`) using the Maven Cactus plugin

10.4 Unit-testing taglibs with Cactus

Figure 10.6 depicts how you unit-test a tag from a taglib with Cactus.

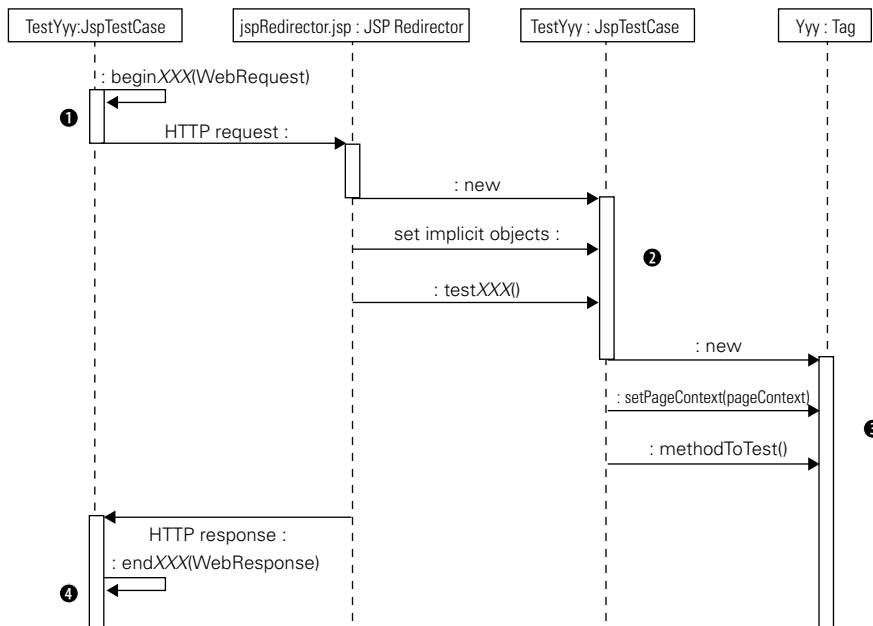


Figure 10.6 Sequence diagram of taglib testing with Cactus

- ❶ Cactus instantiates the test class, which must extend `JspTestCase`. You must configure any HTTP parameter needed by the tag you are testing in a `beginXXX` method. For example, if the tag extracts information from an HTTP parameter, you need to define this parameter in `beginXXX`.
- ❷ Under the hood, the Cactus `JspTestCase` class calls the Cactus JSP Redirector (which is a JSP). The JSP Redirector is in charge of instantiating the `JspTestCase` class on the server side, passing to it the JSP implicit objects (mainly the `PageContext` object). Then, it calls the `testXXX` test method.
- ❸ In the `testXXX` method, you write code to unit-test the JSP tag. The typical steps for testing a tag are as follows: instantiate the tag by calling `new`, set the `PageContext` by calling `setPageContext`, call the method to test, and perform server-side assertions. For example, if the tag sets some objects in the HTTP session, you can assert that the object is there.
- ❹ The Cactus JSP Redirector returns the output of the tag to the client side in an HTTP response. You can then assert the tag output by writing an `endXXX` method in the `JspTestCase` class. Cactus provides a tight integration with `HttpUnit`, which allows very fine-grained assertions on the returned content of the tag.

10.4.1 Defining a custom tag

The Administration application displays the query results on a page called the Results View JSP (`results.jsp`). In the Results View JSP, the first tag class you use is `DynaPropertiesTag`. This tag extracts all the properties of a `DynaBean` object into an array. The properties, which are `DynaProperty` objects, are stored in the `PageContext` under a name passed to the tag. Here's how the tag is used:

```
<d:properties var="properties" item="${dynaBean}"/>
```

where `properties` is the variable name to use for the array of `DynaProperty` objects and `dynaBean` is the `DynaBean` instance from which to extract your properties.

The `DynaPropertiesTag` code is shown in listing 10.4.

Listing 10.4 `DynaPropertiesTag.java`

```
package junitbook.pages;

import org.apache.commons.beanutils.DynaBean;
import org.apache.taglibs.standard.lang.support.
    → ExpressionEvaluatorManager;

import javax.servlet.jsp.tagext.TagSupport;
import javax.servlet.jsp.JspException;

public class DynaPropertiesTag extends TagSupport
{
```