TagUnit

User Guide

September 2003
http://www.tagunit.org

Contents

1 Introduction

1.1 Overview

In the same way that JUnit allows you to write unit tests for Java classes, TagUnit allows you to unit test JSP custom tags, inside the container. In essence, TagUnit is a tag library for testing custom tags within JSP pages.

Even with tools like Cactus, JUnitEE and HttpUnit, testing Java Servlets and JSP pages is hard, particularly if they contain specific business or presentation logic that needs to be tested. Best practices around J2EE development suggest that logic should be encapsulated in JavaBeans or JSP custom tags for better separation of concerns, maintainability, reusability and to facilitate easier testing. Although JUnit can be used to test JavaBeans, testing custom tags by simply invoking their methods doesn't make sense. Custom tags are components and therefore need to be tested at that level, in the way that they would normally be used from within a JSP page.

TagUnit provides a way to perform assertions on the content that custom tags generate and the side-effects that they have on the environment such as the introduction of scoped (request/page/session/application) attributes, cookies and so on. In addition to this, assertions can be made on the constraints specified within the tag library descriptor file that give us a way to verify the contract that a tag provides. In just a four line JSP page, TagUnit can automatically perform tests such as asserting whether the tag handler class is loadable and that it has setter methods for all declared attributes. To supplement this, user defined tests provide a way to perform assertions on the description of a tag, such as its body content and the details of any attributes.

1.2 Purpose

This document presents a user guide to TagUnit, including details on how to install and use the framework to test your own custom tags.

1.3 References and Further Information

Further information about the TagUnit framework can be found at the project website, http://www.tagunit.org.

2 TagUnit Distributions

This section takes a look at how to download and install the TagUnit framework and examples onto your own development environment.

2.1 Binary Distribution

A binary version of TagUnit is available from the Sourceforge file release system via http://www.tagunit.org – the TagUnit project website. The TagUnit binary distribution is shipped as a ZIP file containing the following items:

- General information (readme.txt)
- Change log (changes.txt)
- User Guide (doc/tagunit-userguide.pdf, this document)
- Getting Started Guide (doc/getting-started.pdf)
- TagUnit with Ant Guide (doc/tagunit-with-ant.pdf)
- TagUnit classes (lib/tagunit.jar)
- TagUnit examples web application (tagunit-examples directory)
- TagUnit blank web application (tagunit-blank directory)
- Licensing details (license.txt)
- Sample Ant build script for running TagUnit (test.xml)

2.2 Source Code Distribution

A source code only version is also available from the website, and additionally through direct anonymous `pserver` access to the CVS repository. This contains everything that you need to build TagUnit yourself, including Java sources, TLD files and Ant build scripts. The source distribution can be downloaded from the TagUnit website, while CVS access to the latest code can be obtained with the following CVS root and module name.

CVS root	`:pserver:anonymous@cvs.sourceforge.net:/cvsroot/tagunit`
Module name	`tagunit`

3 Installing TagUnit

3.1 Prerequisites

Installing TagUnit is a straightforward process, requiring the following software:

- Java™ 2 Standard Edition, SDK 1.3 or above[1]
- JavaServer Pages (JSP) 1.2 compatible web/application server such as Jakarta Tomcat 4.0

For the automated running of TagUnit tests, you will additionally require:

- Apache Ant (tested against version 1.5.3)

3.2 Installing the TagUnit Examples

The TagUnit distribution includes a set of pre-built tests, wrapped up and ready to deploy as a web application. Running these examples is a good way to become familiar with TagUnit and its capabilities.

To install the examples, simply take the `tagunit-examples.war` file from the binary distribution and deploy it into your chosen web/application server.

> For Tomcat, this simply involves copying the `tagunit-examples.war` file into the `TOMCAT_HOME/webapps` directory and possibly restarting the server.

To run the examples, simply point your browser to the deployed web application.

> With a default Tomcat installation, the URL for this would be `http://localhost:8080/tagunit-examples/`.

[1] J2SE 1.4 is required to use the regular expression facilities offered by the <`tagunit:assertMatch`> tag.

4 Creating a TagUnit Web Application

This section explains how to set up a TagUnit web application, ready for testing.

4.1 Creating a New Web Application

Creating a separate web application in which to run your tests, although slightly more work is the preferred approach since it keeps the tests away from the web application containing your application specific code. After all, you probably won't the tests to be deployed onto the production environment by accident.

Setting up a web application specifically for testing your tags is as simple as setting up any other web application, although for convenience, the TagUnit distribution contains a web application called `tagunit-blank` that can be used as a starting point. Unextract the `tagunit-blank.war` file and copy the tags that you wish to test into the new web application. Depending on your development environment/process, this will probably involve one of the following:

- Copying a pre-packaged JAR file (containing the tag handler classes and TLDs) into the `WEB-INF/lib` directory of the web application
- Copying a JAR file containing the tag handler classes directly into the `WEB-INF/lib` directory and the relevant TLD files into the `WEB-INF` directory
- Copying the tag handler classes directly into the `WEB-INF/classes` directory and the relevant TLD files into the `WEB-INF` directory

The important point here is that the tag libraries are being deployed into the new web application and any method can be used here.

4.2 Using an Existing Web Application

It is also possible to add TagUnit to an existing web application, such as one that already contains the tags that you would like to test.

Here, you will need to take `tagunit.jar` from the `WEB-INF/lib` directory of the `tagunit-blank` web application distribution and copy it into the `WEB-INF/lib` directory of your web application. Next, copy the contents of the `test` sub-directory (again from `tagunit-blank`) and copy it into the root of your existing web application.

Finally, you will need to copy the following elements into the `web.xml` file of your web application. These elements just set up the TagUnit controller servlet through which all tests are executed.

```
<servlet>
  <servlet-name>TagUnitTestController</servlet-name>
  <servlet-class>org.tagunit.controller.FrontController</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>TagUnitTestController</servlet-name>
  <url-pattern>/test/servlet/*</url-pattern>
</servlet-mapping>
```

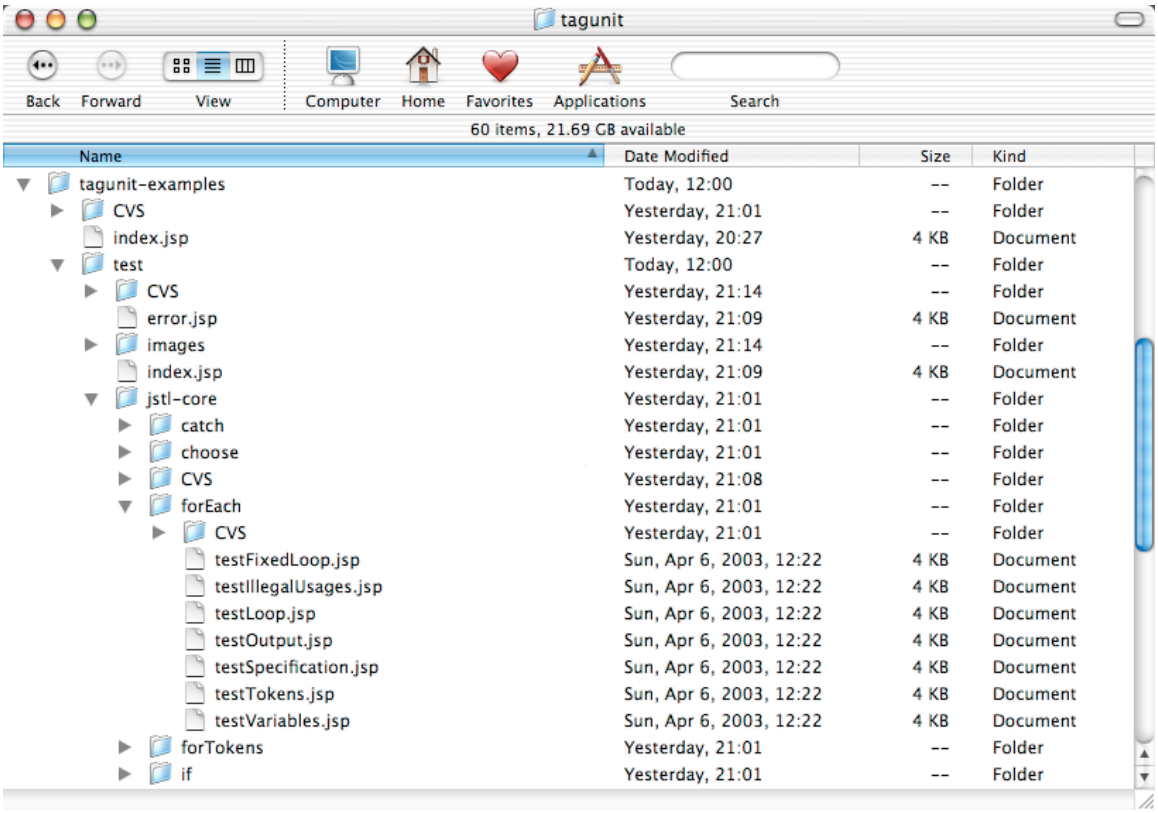
You are now ready to start testing your tags.

5 TagUnit Web Application Structure

Like JUnit, for TagUnit to be able to pick up your tests, there is a defined structure that must be followed. Let's look at this in the context of the TagUnit examples web application.

5.1 Overall Structure

The first step in setting up a web application is to define the overall structure. Using the `tagunit-examples` web application as an example, the screenshot below shows the structure that has been adopted.



With the `tagunit-examples` directory being the root of the web application, all of the tests for the JSTL core taglib are located in a directory called `jstl-core`, underneath a directory called `test`. This defines the top level directory that will contain all of the tests for the JSTL core taglib, and underneath this are further sub-directories, one for each tag that is to be tested. It's here that the actual test pages are placed.

5.2 Deploying the Tag Library to be Tested

To test a tag library, you must remember to deploy that tag library into the web application. This can be achieved in several ways as follows.

- Copying a pre-packaged JAR file (containing the tag handler classes and TLDs) into the `WEB-INF/lib` directory of the application

- Copying a JAR file containing the tag handler classes directly into the `WEB-INF/lib` directory and the relevant TLD files underneath the `WEB-INF` directory
- Copying the tag handler classes directly into the `WEB-INF/classes` directory and the relevant TLD files underneath the `WEB-INF` directory

For the purposes of the `tagunit-examples` web application, the JAR files containing the JSTL (`jstl.jar` and `standard.jar`) have been copied into the `WEB-INF/lib` directory.

5.3 Specifying the Tag Library to be Tested

With the structure of the web application defined, the next step is to tell TagUnit which tag library should be tested. In order to do this, create a JSP file (e.g. `index.jsp`) and place it underneath the top level directory that represents the tag library to be tested. For example, to test the JSTL core taglib, an `index.jsp` file has been created underneath the `jstl-core` directory.

The contents of this JSP page tell TagUnit which taglib is to be tested, and this is achieved using the following syntax.

```
<%@ taglib uri="http://www.tagunit.org/tagunit/core" prefix="tagunit" %>

<tagunit:testTagLibrary uri="/test/jstl-core">
  <tagunit:tagLibraryDescriptor jar="standard.jar" name="c.tld"/>
</tagunit:testTagLibrary>
```

Here, we're using some of the tags from the TagUnit tag library to specify the location of the taglib to be tested. In this example, we're pointing TagUnit to the taglib defined in the `c.tld` file inside `standard.jar`, which itself can be found in the `WEB-INF/lib` directory. See section 6.1 for further details on the various ways in which tag libraries can be specified using the `testTagLibrary` tag.

5.4 Running the Tests

With the tag library specified, it is now possible to run the tests as they stand. Although no tests have actually been written yet, TagUnit is able to perform some basic, automatic, tests of the tag library to check that the TLD file is correctly specified, that tag handler classes are available, setter methods have been specified for tag attributes and so on. To run the tests, start up your JSP container and point your browser to the following address. Note that this may be slightly different depending on your environment.

```
http://localhost:8080/tagunit-examples/test/servlet/RunTests?uri=/test/jstl-core/index.jsp
```

All that this URL does is pass the URI of the JSP page specifying the tag library to be tested to the controller responsible for actually initiating the tests. After a short delay, the web browser should display a page containing the results of running the automatic tests, and details of exactly what was tested.

5.5 Writing Tests

Now that the web application is set up, the final step is to write some tests. Like JUnit, TagUnit provides a way of breaking up the tests into separate units so that tests can be logically and physically arranged according to their intent.

All TagUnit tests are written as JSP files, and like JUnit test methods, these JSP files need to adhere to a specific naming convention. As mentioned earlier, the tests for a specific tag are placed underneath the directory representing that tag. For example, the tests for the JSTL core `forEach` tag are located within a sub-directory of `jstl-core` called `forEach`. Each of these directories may contain zero, one or more test JSP files, each of which must be prefixed with `test` and have a `.jsp` extension. Each test JSP may contain one or more assertions, in a similar way to the way that JUnit test methods may contain one or more assertions.

As far as naming conventions go, it is best to try and choose a name that states something about the intent of the tests contained with the JSP page. For example, you may want to have a JSP that performs assertions on the specification of a tag, perhaps checking its attributes, body content and so on. Such a JSP might be called `testSpecification.jsp`, highlighting that fact that it tests the specification. Sticking with the `forEach` tag, you might want a JSP to test that tokenized strings are correctly iterating over. This page might be called `testTokens.jsp`, for example.

The following example shows a snippet from the `testTokens.jsp` page that is used to test the `forEach` tag.

```
<%@ taglib uri="http://www.tagunit.org/tagunit/core" prefix="tagunit" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>

<tagunit:assertEquals name="Loop over 3 tokens">
  <tagunit:actualResult><c:forEach var="myVar" items="1,2,3"><c:out
value="${myVar}"/></c:forEach></tagunit:actualResult>
  <tagunit:expectedResult>123</tagunit:expectedResult>
</tagunit:assertEquals>

<tagunit:assertEquals name="Loop over zero tokens">
  <tagunit:actualResult><c:forEach var="myVar" items=""><c:out
value="${myVar}"/></c:forEach></tagunit:actualResult>
  <tagunit:expectedResult/>
</tagunit:assertEquals>
```

As this illustrates, TagUnit tests are typically nothing more than simple usages of the tag being tested, wrapped up inside some of the various assertion tags that are provided by the TagUnit framework. For a full listing of the assertions available, please see the next sections.

5.6 Setting up and tearing down

The final point to mention is that, like JUnit, there are often times when you have common set up and tear down logic within your tests. Where JUnit allows `setUp` and `tearDown` methods to be written, TagUnit provides the ability to place similar logic in JSP pages called `setUp.jsp` and `tearDown.jsp` respectively. These should be placed in the directory representing the appropriate tag, next to the `testXXX.jsp` pages containing the tests for that tag. As with JUnit, these pages get called before every

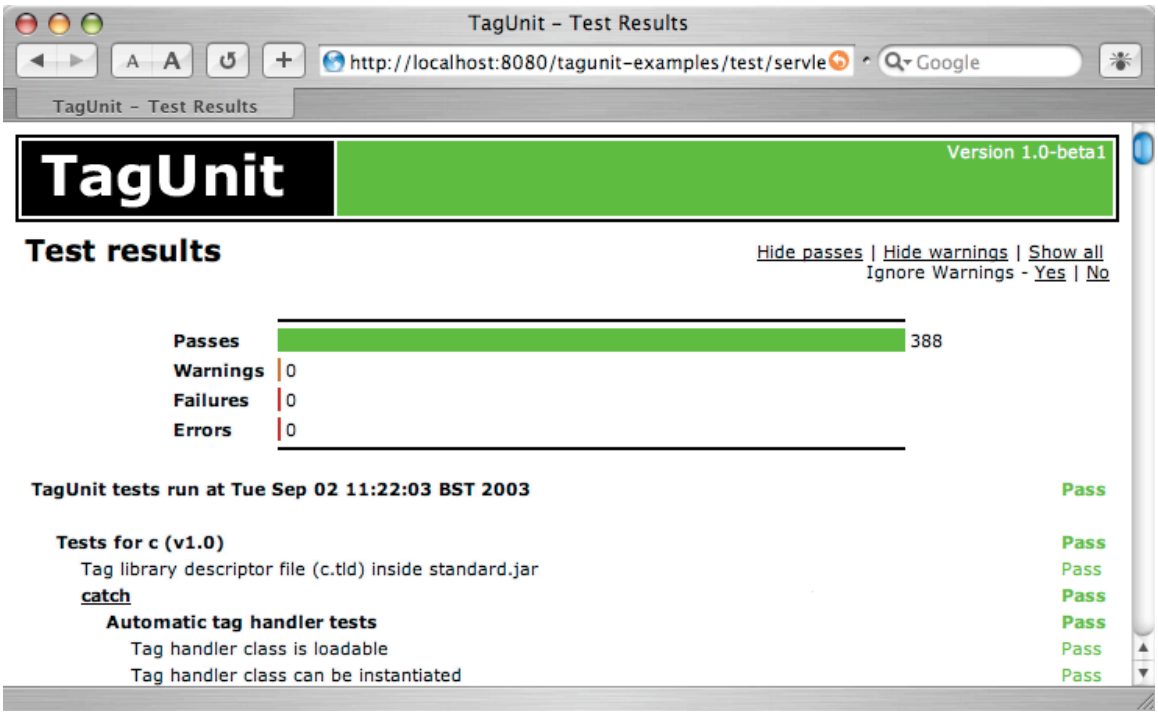
`testXXX.jsp` page. For example, you may want to initialize a request scoped object in the set up stage, and destroy this during tear down, rather than doing this inside each and every `testXXX.jsp` page.

5.7 Test Results

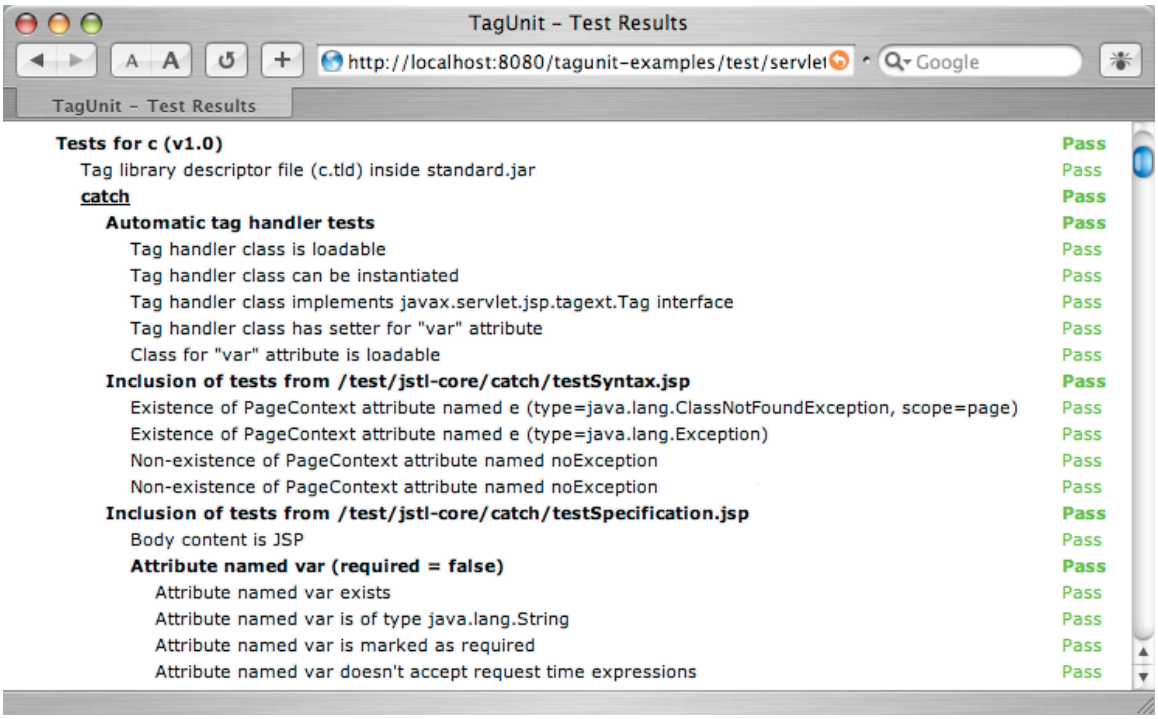
After the tests have successfully been executed, a web page will be displayed in your browser detailing the results of those tests. The top part of the page is colour-coded to represent the overall test results. Pass, warning or fail are coded as green, yellow or red respectively. Underneath this is a small bar chart summarizing the number of tests for a particular status as follows.

Test status	Description
Pass	The tests passed.
Warning	The tests passed although a warning was raised. The most usual situation for this to occur is when only the automatic tests have been performed on a tag. In other words, you haven't yet written any assertions to test a particular tag.
Failure	The test or assertion failed.
Error	The test or assertion could not be completed because of an error. A error message or stack trace will be given further down the page.

The following screenshot shows an example of the web page displayed after running some tests.



Drilling down further into the information provided by the test results, the web page contains a section for each tag in the tag library being tested, regardless of whether you have written any tests for them yet. The following screenshot shows the results for the JSTL `catch` tag.



First are the results from the automatic TagUnit tests. In this example, TagUnit has tested that the tag handler class is loadable, can be instantiated and implements the `Tag` interface. In addition to this, TagUnit also runs some tests on the attributes of any given tag, in this case checking that the appropriate setter method is available on the tag handler class for the `var` attribute. As mentioned before, all of these tests are performed automatically and transparently – no additional work is necessary.

Following this are the results from the tests that we have written ourselves. In this example, tests have been written in two JSP files, one called `testSyntax.jsp` and one called `testSpecification.jsp`, both of which contain various assertions using the assertion tags detailed in section 7.

6 Tags for Defining TagUnit Tests

There are several types of tags within TagUnit – those that are used to define tests and those that perform assertions. This section covers those tags used to setup and define the tests that are to be executed, all of which are a part of the core tag library, imported with the following taglib directive:

```
<%@ taglib uri="http://www.tagunit.org/tagunit/core" prefix="tagunit" %>
```

6.1 testTagLibrary

The `<tagunit:testTagLibrary>` tag is the outermost container for defining the suite of tests that are to be performed on the tags within a tag library. It defines the tag library that is to be tested, and specifies the top-level location from which the tests for the tags in the tag library will be found.

The processing of this tag includes looking for the tags that are defined in the corresponding TLD file (see <tagunit:tagLibraryDescriptor> tag below) and executing the tests that are defined for them. In addition to this, some automatic tests are performed on each tag behind the scenes. These include the following:

- Checks that the tag handler corresponding to the tag exists and is available on the classpath for the web application.
- Checks that the tag handler corresponding to the tag implements the javax.servlet.jsp.tagext.Tag interface.
- Checks that the tag extra info class associated with the tag (if applicable) exists and is available on the classpath for the web application.
- Checks that the tag extra info class associated with the tag (if applicable) extends the javax.servlet.jsp.tagext.TagExtraInfo class.
- For each attribute of the tag, checks that a corresponding public setter method is available on the tag handler class.

6.1.1 Attributes

Name	Required	Request-time Expression	Description
uri	Yes	No	This is the uri pointing to the directory in which the tests for the tags in the tag library are to be found.

6.1.2 Usage Notes

There are no special usage notes for this tag.

6.1.3 Example

The following JSP snippet is an example of how the `<tagunit:testTagLibrary>` tag is used.

```
<tagunit:testTagLibrary uri=”/test/jstl-core”>
  ...
</tagunit:testTagLibrary>
```

Here, the `uri` attribute points to the directory that contains the actual tests for the tag library, which in this case is `/test/jstl-core`. Underneath this directory are directories containing the tests for each tag, and they must be named in accordance with the tag that they are testing. In this example, the JSP pages containing the tests for a tag called `forEach` would need to be located in the `/test/jstl-core/forEach` directory, as described in section *5.1, Overall Structure*.

6.2 tagLibraryDescriptor

The `<tagunit:tagLibraryDescriptor>` tag is nested within the `<tagunit:testTagLibrary>` tag and tells the TagUnit framework where to find the tag library descriptor for the tag library that is being tested. Without this, none of the automatic TagUnit assertions can be performed and TagUnit won't know which tag library is being tested.

6.2.1 Attributes

Name	Required	Request-time Expression	Description
uri	No	No	The uri that points to the location of the TLD file.
jar	No	No	The name of the JAR file (residing in `WEB-INF/lib`) that contains the TLD file.
name	No	No	The name of the TLD file.

6.2.2 Usage Notes

The only valid combinations of attributes permitted for usages of this tag are as follows:

- uri

OR

- jar and name

In other words, you either specify a URI to the TLD file, or the name of a JAR file and the name of the TLD file that resides inside that JAR file. Any other combination will raise an exception.

6.2.3 Example

The following JSP snippet is an example of how the `<tagunit:tagLibraryDescriptor>` tag would be used. Note that it must be nested within a `<tagunit:testTagLibrary>` tag.

```
<tagunit:testTagLibrary uri=”/test/jstl-core”>
  <tagunit:tagLibraryDescriptor jar="standard.jar" name="c.tld"/>
  ...
</tagunit:testTagLibrary>
```

In this example, the framework will look for the `WEB-INF/lib/standard.jar` file, and then look for the tag library descriptor (`c.tld`) in the `META-INF` directory of the JAR file. This usage is useful if you are using a prepackaged tag library, where a complete, ready-to-deploy JAR file is used.

If, however, the TLD file is simply located somewhere underneath the `WEB-INF` directory, the following usage should be adopted.

```
<tagunit:testTagLibrary uri=”/test/jstl-core”>
  <tagunit:tagLibraryDescriptor uri="/WEB-INF/c.tld"/>
  ...
</tagunit:testTagLibrary>
```

In this example, the framework will look for the tag library descriptor file (`c.tld`) within the `WEB-INF` directory of your web application.

7 Tags for Testing Custom Tag Characteristics

Although most of the time it will be the functionality of a custom tag that is being tested, it can also be useful to perform assertions based upon the specification and characteristics of a particular tag. This section covers the tags that make this possible, all of which are again a part of the core tag library, imported with the following taglib directive:

```
<%@ taglib uri="http://www.tagunit.org/tagunit/core" prefix="tagunit" %>
```

All of the following tags appear within the testXXX.jsp pages that implement the tests for a specific tag.

7.1 assertNoAttributes

The `<tagunit:assertNoAttributes/>` asserts that the tag that is currently being tested doesn't take any attributes. In other words, it tests that the description of that tag in the TLD file doesn't declare any attributes.

7.1.1 Attributes

There are no attributes for this tag.

7.1.2 Usage Notes

This tag has a body content of `empty`, and therefore must be used on the page without content between its opening and closing tag.

7.1.3 Examples

This tag is used standalone, and without body content as follows.

```
<tagunit:assertNoAttributes/>
```

Given the context of the current tag being tested, this assertion evaluates to true if the tag has no attributes, and false otherwise.

7.2 assertAttribute

This tag represents the opposite of the previous tag, and asserts that a named attribute exists on the tag currently being tested.

7.2.1 Attributes

Name	Required	Request-time Expression	Description
name	Yes	No	The name of the attribute that is being asserted.
type	No	No	The fully qualified name of the class (type) of the attribute.
required	Yes	No	Whether the attribute is marked as required.
rtexprvalue	Yes	No	Whether the attribute is marked as

			accepting a request-time value.

7.2.2 Usage Notes

There are no special usage notes for this tag.

7.2.3 Examples

This tag is used standalone, and without body content as follows.

```
<tagunit:assertAttribute name="scope" required="false"
rtexprvalue="false"/>
```

Given the context of the current tag being tested, this assertion evaluates to true if the named attribute exists, and is defined as specified.

7.3 assertBodyContent

This tag performs an assertion on the type of body content that the tag currently being tested can accept. In other words, it checks the body content declared for the tag in the tag library descriptor file.

7.3.1 Attributes

Name	Required	Request-time Expression	Description
name	Yes	No	The name of the body content – "empty", "JSP", "tagdependent".

7.3.2 Usage Notes

There are no special usage notes for this tag.

7.3.3 Examples

The following example asserts that the tag currently being tested has a body content of `empty`.

```
<tagunit:assertBodyContent name="empty"/>
```

7.4 assertInterface

This tag asserts that the tag handler of the tag currently being tested implements the named interface (or class). This is particularly useful for those tags that are to be extended, or for those tags that cooperate with one another through a predefined interface.

7.4.1 Attributes

Name	Required	Request-time Expression	Description
name	Yes	No	The fully qualified name of the class/interface.

7.4.2 Usage Notes

There are no special usage notes for this tag.

7.4.3 Examples

The following example asserts that the tag handler for the current tag implements the `javax.servlet.jsp.jstl.core.LoopTagSupport` class.

```
<tagunit:assertInterface
  name="javax.servlet.jsp.jstl.core.LoopTagSupport"/>
```

8 Tags for Testing Custom Tag Behaviour

The final set of tags that make up the TagUnit core tag library are those that perform the actual assertions. For example, these tags allow conditions to be checked, and actual/expected content to be compared. This section covers the tags that make this possible, all of which are again a part of the core tag library, imported with the following taglib directive:

```
<%@ taglib uri="http://www.tagunit.org/tagunit/core" prefix="tagunit" %>
```

Again, all of the following tags appear within the testXXX.jsp pages that implement the tests for a specific tag.

8.1 assertEquals (including actualResult and expectedResult)

One of the simplest types of assertion that can be performed with a testing framework is to compare an actual result to an expected result. With TagUnit, this is performed using the `assertEquals` tag, in conjunction with two other tags - `actualResult` and `expectedResult`.

8.1.1 Attributes

Name	Required	Request-time Expression	Description
name	Yes	No	The name of the test.
ignoreWhitespace	No	No	True if whitespace should be ignored during comparisons, false otherwise.

8.1.2 Usage Notes

The body content of the `actualResult` and `expectedResult` tags is `JSP`, allowing other custom tags and JSP elements to be used when assembling content for comparison.

In additon, `expectedResult` can be used with an attribute called uri that points to a text file containing the expected result. This is useful when the expected result is very large and you would like to keep it away from the JSP page containing the assertions.

8.1.3 Examples

The following snippet is an example of how the `<tagunit:assertEquals>` tag might be used.

```
<tagunit:assertEquals name="Simple test of generated content">
  <tagunit:expectedResult>...</tagunit:expectedResult>
  <tagunit:actualResult>...</tagunit:actualResult>
</tagunit:assertEquals>
```

The body content of this tag is simply made up of two other tags that explicitly demarcate what the actual and expected results will be. At runtime, the body content

of the <tagunit:actualResult> and <tagunit:expectedResult> tags are evaluated and a comparison is made. If the evaluated body content is equal, the assertion is true, otherwise it evaluates to false.

Here is an example of how the JSTL <c:if> tag could be tested.

```
<tagunit:assertEquals name="Simple condition (always true)">
  <tagunit:expectedResult>
    aaa
  </tagunit:expectedResult>
  <tagunit:actualResult>
    <c:if test="${1 < 2}">aaa</c:if>
  </tagunit:actualResult>
</tagunit:assertEquals>
```

Here, the condition used within the <c:if> tag always evaluates to true and therefore the body content of aaa is included. This is then compared to the expected result of aaa and the assertion subsequently evaluates to true.

The final example shown here demonstrates how to specify an external file that contains the expected results.

```
<tagunit:assertEquals name="A comparison of lots of content">
  <tagunit:expectedResult uri="/test/someFileContainingLotsOfContent"/>
  <tagunit:actualResult>
    <someTagThatGeneratesLotsOfContent/>
  </tagunit:actualResult>
</tagunit:assertEquals>
```

8.2 assertNotEquals

This performs the opposite of the assertEquals tag – it checks that actual content doesn't equal the expected content. Like the assertEquals tag, this is performed in conjunction with two other tags - actualResult and expectedResult.

8.2.1 Attributes

Name	Required	Request-time Expression	Description
name	Yes	No	The name of the test.
ignoreWhitespace	No	No	True if whitespace should be ignored during comparisons, false otherwise.

8.2.2 Usage Notes

The body content of the actualResult and expectedResult is JSP, allowing other custom tags and JSP elements to be used when assembling content for comparison.

In additon, expectedResult can be used with an attribute called uri that points to a text file containing the expected result. This is useful when the expected result is very large and you would like to keep it away from the JSP page containing the assertions.

8.2.3 Examples

The following snippet is an example of how the `<tagunit:assertNotEquals>` tag might be used.

```
<tagunit:assertNotEquals name="Simple test of generated content">
  <tagunit:actualResult>...</tagunit:actualResult>
  <tagunit:expectedResult>...</tagunit:expectedResult>
</tagunit:assertNotEquals>
```

The body content of this tag is simply made up of two other tags that explicitly indicate what the actual and expected results will be. At runtime, the body content of the `<tagunit:actualResult>` and `<tagunit:expectedResult>` tags are evaluated and a comparison is made. If the evaluated body content is not equal, the assertion is true, otherwise it evaluates to false.

8.3 assertContains

Where the `assertEquals` tag tests for equality between an actual and expected result, the `assertContains` tag tests that the actual result contains the expected result. As with the `assertEquals` tag, `assertContains` is used in conjunction with the `actualResult` and `expectedResult` tags.

8.3.1 Attributes

Name	Required	Request-time Expression	Description
name	Yes	No	The name of the test.
ignoreWhitespace	No	No	True if whitespace should be ignored during comparisons, false otherwise.

8.3.2 Usage Notes

The body content of the `actualResult` and `expectedResult` is `JSP`, allowing other custom tags and JSP elements to be used when assembling content for comparison.

8.3.3 Examples

The following snippet is an example of how the `<tagunit:assertContains>` tag might be used.

```
<tagunit:assertContains name="Simple test of generated content">
  <tagunit:expectedResult>
   lots of content
  </tagunit:expectedResult>
  <tagunit:actualResult>
    Here is lots of content that is automatically generated
  </tagunit:actualResult>
</tagunit:assertContain>
```

The body content of this tag is simply made up of two other tags that explicitly indicate what the actual and expected results will be. At runtime, the body content of the `<tagunit:actualResult>` and `<tagunit:expectedResult>` tags are evaluated

and a comparison is made. If the actual result contains the expected result, the assertion is true, otherwise it evaluates to false.

8.4 assertMatches

Where the `assertEquals` tag tests for equality between an actual and expected result, the `assertMatches` tag tests that the actual result matches the expected result (a regular expression). As with the `assertEquals` tag, `assertMatches` is used in conjunction with the `actualResult` and `expectedResult` tags.

8.4.1 Attributes

Name	Required	Request-time Expression	Description
name	Yes	No	The name of the test.
ignoreWhitespace	No	No	True if whitespace should be ignored during comparisons, false otherwise.

8.4.2 Usage Notes

The body content of the `actualResult` and `expectedResult` is `JSP`, allowing other custom tags and JSP elements to be used when assembling content for comparison.

8.4.3 Examples

The following snippet is an example of how the `<tagunit:assertMatches>` tag might be used.

```
<tagunit:assertMatches name="Simple test of generated content">
  <tagunit:expectedResult>^Some .*</tagunit:expectedResult>
  <tagunit:actualResult>Some content</tagunit:actualResult>
</tagunit:assertMatches>
```

The body content of this tag is simply made up of two other tags that explicitly indicate what the actual and expected results will be. At runtime, the body content of the `<tagunit:actualResult>` and `<tagunit:expectedResult>` tags are evaluated and a comparison is made. If the actual result matches the expected result (a regular expression), the assertion is true, otherwise it evaluates to false.

8.5 assertPageContextAttribute

This tag asserts that a named attribute is available in a specific scope (page, request, session or application) and can be used when tags use the page context as a location for sharing information, confirming that the correct objects are placed in the appropriate scope. In addition, this tag can also be used to confirm that specific properties (of a JavaBean) or elements (of a java.util.Map) are set correctly.

8.5.1 Attributes

Name	Required	Request-time Expression	Description
name	Yes	No	The name of the attribute to be tested for.
type	Yes	No	The type of the named object.
scope	No	No	The scope of the named object (“page”,

			"request", "session" or "application"). If omitted, the default is "page".
value	No	Yes	The value to be compared against. This can be a simple string or an object passed by a request-time expression.
property	No	No	If the named object is a JavaBean, this represents the name of the property on that bean to test. If the named object is a java.util.Map, this represents the key of the value to test.

8.5.2 Usage Notes

There are no special usage notes for this tag.

8.5.3 Examples

The following example asserts that a variable called "myString" of type `java.lang.String` exists at page scope.

```
<tagunit:assertPageContextAttribute name="myString"
type="java.lang.String"/>
```

The next example asserts the same, except that it tests for the object in session scope.

```
<tagunit:assertPageContextAttribute name="myString"
type="java.lang.String" scope="session"/>
```

Building upon this, the following example illustrates how the value of an object can be asserted too.

```
<tagunit:assertPageContextAttribute name="myString"
type="java.lang.String" value="Hello" scope="session"/>
```

The next two examples show how a JavaBean property, and an entry in a `java.util.Map` can be tested.

```
<tagunit:assertPageContextAttribute name="bean" property="name"
  type="org.mycompany.MyJavaBean" value="simon"/>

<tagunit:assertPageContextAttribute name="map" property="name"
  type="java.util.HashMap" value="simon"/>
```

8.6 assertNoPageContextAttribute

This tag performs the opposite assertion of the `<tagunit:assertPageContextAttribute/>` tag in that it checks that an object with the specified name does not exist in the specified scope.

8.6.1.1 Attributes

Name	Required	Request-time Expression	Description

name	Yes	No	The name of the object to test for.
scope	No	No	The scope of the named object (“page”, “request”, “session” or “application”). If omitted, the default is "page".

8.6.1.2 Usage Notes

There are no special usage notes for this tag.

8.6.1.3 Examples

This example asserts that an object called “myString” does not exist in session scope.

```
<tagunit:assertNoPageContextAttribute name="myString" scope=”session”/>
```

8.7 assertException

This tag provides a way to ensure that the correct exceptions are thrown from a custom tag, given the appropriate error condition in incorrect usage. At runtime, this tag captures any exceptions that are thrown during processing so that assertions can be performed on the `java.lang.Throwable` instance.

8.7.1.1 Attributes

Name	Required	Request-time Expression	Description
name	Yes	No	The name of the test.
exception	No	No	The fully qualified name of the exception that is being tested for.
message	No	No	The text representing the message provided by the exception. If specified, this is compared to the string returned from the `getMessage()` method of the exception that was thrown.

8.7.1.2 Usage Notes

The `exception` and `message` attributes of this tag are optional and, if specified, provide the tag with more information with which to perform assertions. In its most basic form (by just specifying the `name` attribute), this tag simply checks that an exception was thrown. Specifying the `exception` and/or `message` attributes allow the tag to perform assertions on the type (class) and message of the exception instance respectively.

8.7.1.3 Examples

The following example demonstrates how the `assertException` tag might be used to test that an exception is thrown when a custom tag is used incorrectly. Here, the TagUnit `expectedResult` tag is used outside of an `assertEquals`/`assertNotEquals`/etc tag.

```
<tagunit:assertException name="Incorrect nesting"
exception="javax.servlet.jsp.JspTagException" message="expectedResult
tag must be nested.">
  <tagunit:expectedResult/>
```

```
</tagunit:assertException>
```

If the named exception is thrown inside the body content, the assertion evaluates to true, otherwise it evaluates to false to indicate that an exception wasn't thrown.

8.8 fail

This tag provides a way for an assertion to be instantly failed, which can be useful to indicate that a tag hasn't worked as expected. For example, it could be used to test a custom tag that provides conditional behaviour by placing the fail tag in the branch that should not be evaluated to true.

8.8.1 Attributes

Name	Required	Request-time Expression	Description
name	Yes	No	The name of the test.
message	Yes	No	A message to help determine what went wrong.

8.8.2 Usage Notes

There are no special usage notes for this tag.

8.8.3 Examples

This example tests that a conditional tag chooses the correct branch, demonstrating how the fail tag can be used to catch failures. This example is based upon the JSTL `<c:choose>` tag.

```
<c:set var="i" value="1"/>
<c:choose>
    <c:when test="${i < 5}">
      1st block chosen
    </c:when>
    <c:otherwise>
      <tagunit:fail name="Matching when block exists" message="The
otherwise block was chosen"/>
    </c:otherwise>
</c:choose>
```

If the test works as expected, the `when` block is chosen and the test passes. On the other hand, if there is a problem with the tag and the `otherwise` block is executed, the `fail` tag tells TagUnit that the test has failed.

8.9 assertCustom (including param)

The assertion tags provided by TagUnit allow a wide range of simple tests to be performed. However, there may be times when some additional functionality is needed and there are two options available. The first is to download the source code for TagUnit and extend the existing tags. The other option is to use the `assertCustom` tag in conjunction with your own class that encapsulates the functionality you need.

In order to do this, you will need to write a Java class that extends the `org.tagunit.test.CustomTestPackage` class, providing an implementation of the following method that realizes the functionality of your test(s).

```
public void executeTest(TestContextContainer testContext);
```

The parameter to this method (of type `TestContextContainer`) is a container for storing the results from tests that you might perform. In using this class, you can set various properties of the test context (e.g. name, pass/fail/warning/etc) as well as build up a hierarchy of nested, sub-contexts so that a group of assertions can be organized appropriately. In addition, various methods on the superclass (`org.tagunit.test.CustomTestPackage`) provide a way to get access to the current HTTP request, the page context and so on. For more information about functionality provided by this and related classes, please see the source code and/or javadocs.

8.9.1.1 Attributes

Name	Required	Request-time Expression	Description
name	Yes	No	The name of the test.
type	Yes	No	The fully qualified name of the class implementing the custom tests and assertions. This class must extend the `org.tagunit.test.CustomTestPackage` class.

8.9.1.2 Usage Notes

The class representing the custom assertion must be available on the classpath of the web application. This means that the class must either reside inside a JAR in the `WEB-INF/lib` directory, or in the appropriate directory hierarchy underneath the `WEB-INF/classes` directory.

8.9.1.3 Examples

Here is an example of how to build a simple custom test class.

```
package com.yourcompany.tagunit;

import javax.servlet.jsp.PageContext;

import org.tagunit.TestContextContainer;
import org.tagunit.TestContext;

/**
 * A simple custom test package example.
 *
 * @author    Simon Brown
 */
public class CustomTestPackageExample extends CustomTestPackage {

  /**
   * Creates a new test package.
   *
   * @param name    the name of the test package
   */
```

```
  public CustomTestPackageExample(String name) {
    super(name);
  }

  /**
   * Runs the tests that are part of this test package.
   *
   * @param testContext   the context in which these tests are to be run
   */
  public void executeTest(TestContextContainer testContext) {
    // get access to the current JSP page context
    PageContext pageContext = getPageContext();

    // get the name of the object we need to look for
    String objectName = (String)getParameter("name");

    // use the page context in some way
    Object someObject = pageContext.findAttribute(objectName);

    // and finally perform some assertions on it,
    // setting the result as appropriate
    testContext.setStatus(TestContext.PASS);
  }

}
```

All that this test class does is look up an object from the page context, the name of which is specified as a parameter when the tag is used (see below). Some assertions would then be performed (e.g. testing the object isn't null, etc) and the status of the `TestContextContainer` is updated accordingly. With the class written, it needs to be compiled and placed into the classpath of the web application, as described in the special usage notes above. To use a custom test class, the `assertCustom` tag can be used as follows.

```
<%@ taglib uri="http://www.tagunit.org/tagunit/core" prefix="tagunit" %>

<tagunit:assertCustom name="An example custom test" type="
com.yourcompany.tagunit.CustomTestPackageExample">
  <tagunit:param name="name" value="someObject" />
</tagunit:assertCustom>
```

The `assertCustom` tag allows you to specify the fully qualified name of the class that contains your testing logic, while the nested `param` tag allows you to specify parameters at runtime. In this example, a single parameter called `name` is supplied, with a value of `someObject`. To specify more parameters, just nest further `param` tags.

9 License Details

TagUnit is distributed under a BSD style license, meaning that it can be used, modified and distributed freely provided that the original license details and copyright notice (as follows) are included:

For further information about this, and other open source issues, please take a look at `http://www.opensource.org`.