WSRPtk

Table of contents

1 About....2
1.1 Maintaining this Website....2
1.2 Project Roles....2
2 WSRP V1 Conformance....3
2.1 WSRP V1 Conformance Test Kit....3
2.2 Getting Started....4
2.3 Executing the Test Kit....5
2.4 Project Directory Structure....7
2.5 Architecture....7
2.6 Development....8
2.7 WSRP Conformance Test Kit Tutorial....19
3 All....21

1. About

1.1. Maintaining this Website

1.1.1. Under Construction

This site is under construction. It is generated from source files using Apache Forrest (version 0.7). The source files for this website are stored in the WSRPtk project's CVS repository. The generated site is uploaded to the Source Forge website using SFTP.

1.2. Project Roles

1.2.1. Source Forge User Types

Unauthenticated User

A member of the general public who downloads and uses the test kit software.

Authenticated User

A person who has established a Source Forge user id and password. In order to obtain a user id, the person must sign Terms and Conditions, indicating that they promise to abide by the license terms of any project to which them make a contribution.

Developer

An Authenticated User who has obtained commit priveleges on a Source Forge Project.

Release Technicial

A Developer who also has the privelege to use the File Release System to create and post releases.

Project Administrators

The Authenticated User that created the project in the first place. The Project Administrator has all priveleges and can grant Developers Project Administrator priveleges. The Project Administrator also grants commit priveleges to Authenticated Users to enable them to be Developers.

1.2.2. Becoming a Developer

The steps to becoming a developer on the WSRPtk Project are as follows:

1. Obtain a SourceForge account to become an Authenticated User and sign the Terms and Conditions
2. Request Developer priveleges by sending an email to the Project Administrator
   - The OASIS Technical Committee Conformance Sub Committee must vote to grant developer priveleges to the requester.
   - The requester must be an employee of an OASIS TC member company.
3. The Project Administrator will add priveleges to the requester's account.

Note:

Some Developers will also be granted Release Technician priveleges, according to the wishes of the Conformance Sub Committee.

1.2.3. Conformance Sub Committee

The OASIS Technical Committee Conformance Sub Committee is involved in the management of the WSRPtk project in the following ways:

1. Chair of the Conformance Sub Committee will be granted Project Administrator priveleges.
2. The Conformance Sub Committee votes to grant Project Administrator priveleges to one or more Developers.
3. The Conformance Sub Committee votes to release each version of the Test Kit
4. The Conformance Sub Committee designates a Developer (i.e. Release Technician) to publish the release to the File Release System
5. The Conformance Sub Committee reviews and votes to accept 3rd Party Contributions

1.2.4. Handling 3rd Party Contributions

If a person who is not a Developer wishes to contribute code to the project:

1. The Conformance Sub Committee shall review the code contribution to verify the origin of the code and that it can be released under the Common Public License.
2. The Conformance Sub Committee votes to accept it.
3. A Developer then commits the code to the project.

2. WSRP V1 Conformance

2.1. WSRP V1 Conformance Test Kit

2.1.1. Introduction

The V1 Conformance Test Kit is designed to verify conformance of running WSRP

Producers and Consumers to the V1 WSRP Specification. The Test Kit does not provide 100% coverage of the V1 conformance statements. Refer to the WSRPtk project website under Feature Requests for a list of assertions that are outstanding.

This website provides documentation for the V1 Test Kit, but is also a work in progress. Information on how this website is produced can be found here.

Refer to links on left hand side of this page to find information on how to download the Test Kit and its prerequisites and how to configure and execute the Test Kit.

A rudimentary export of the Power Point tutorial is also provided. Most of this site's initial content has come from this tutorial. The documentation of how to write test assertions has not been incorporated into these web pages yet, so refer to the tutorial for help with development.

A brief overview of the architecture is available.

The development content is a work in progress and has not yet been extracted from the tutorial.

2.2. Getting Started

2.2.1. Obtaining the Release

Download the release from the website. Unzip the file in any directory.

2.2.2. Obtaining and Installing Prerequisites

2.2.2.1. Eclipse Web Tools

1. Download Web Tools 1.0.1 from here.
2. Unpack Zip File
3. Copy from:
   `eclipse/plugins/org.eclipse.wst.wsi_1.0.1.v200602030355.jar`
   to: `wsrptk/lib`

Note:

Note: the name of this jar file has changed over time. You may have wsi-core.jar and wsi-validate.jar already on your system if you have previously installed Web Tools. These jars *should* work also.

2.2.2.2. WS-I Testing Tools

1. Download the WS-I Testing Tools 1.1 from here.

2. Unpack Zip File
3. Copy from: `wsi-test-tools/common/xsl/*.*` to: `wsrptk/xsl`
4. Copy from: `wsi-test-tools/common/schemas/*.*` to: `wsrptk/schemas`

2.2.2.3. XSL Stylesheet Issues

- The namespace definitions in the XSL stylesheets obtained from WS-I are out of date with respect to the jar files from Eclipse.
- The embedded style directives use ALL CAPS for HTML tag names, and they have to be lower case to get proper formatting.
- The WSRPTK project has patch files that can be applied to make these corrections.

2.3. Executing the Test Kit

2.3.1. Summary of Steps

Executing the Test Kit happens in 2 phases. The first phase is the Monitoring phase, during which a message log is created from a running WSRP service. The second phase is the Analysis phase, during which the log file is analyzed for conformance.

Here is an architectural diagram of the components:

Diagram of Test Kit Architecture

In general, these are the steps for running the test kit:

1. Configure the Producer to listen on a particular port (e.g. 8080).
2. Configure the Consumer to write to a particular port (e.g. 8081).
3. Configure the Monitor to listen to the Consumer (e.g. on 8081) and send to the Producer (e.g. on 8080) by setting the port numbers in the config file.
4. Start your Producer (e.g. WSRP4J on Tomcat)
5. Start the Monitor (bin/Monitor.bat)
6. Start your Consumer (e.g. WSRP4J Swing Consumer)
7. Exercise the User Interface
8. Exit the Consumer
9. Stop the Monitor (type "exit")
10. Run the Analyzer (bin/Analyzer.bat)
11. View the Report (logs/AnalyzerLog.xml)

2.3.2. The Monitor Component

The Monitor is a WS-I Testing Tools component that intercepts web service traffic between parties and writes the messages to a log file.

The Functional Specification for the Monitor component can be found here.

The Monitor gets parameters from a configuration file, the path to which is passed in at runtime. The config file is found in `wsrptk/config`. Here is a sample of configuration file:

```
. . .
<wsi-monConfig:logFile replace="true" location="../logs/MonitorLog.xml">
<wsi-common:addStyleSheet href="../xsl/log.xsl" type="text/xsl"/>
</wsi-monConfig:logFile>
. . .
<wsi-monConfig:manInTheMiddle>
<wsi-monConfig:redirect>
<wsi-monConfig:listenPort>8081</wsi-monConfig:listenPort>
<wsi-monConfig:schemeAndHostPort>http://localhost:8080</wsi-monConfig:schemeAndHostPor
<wsi-monConfig:maxConnections>1000</wsi-monConfig:maxConnections>
<wsi-monConfig:readTimeoutSeconds>15</wsi-monConfig:readTimeoutSeconds>
</wsi-monConfig:redirect>
</wsi-monConfig:manInTheMiddle>
. . .
```

2.3.3. The Analyzer Component

The Analyzer is a WS-I Testing Tools component that intercepts web service traffic between parties and writes the messages to a log file.

The Functional Specification for the Analyzer component can be found here.

The Analyzer gets parameters from a configuration file, the path to which is passed in at runtime. The config file is found in `wsrptk/config`. Here is a sample of configuration file:

```
<wsi-analyzerConfig:configuration name="Sample WSRP Profile Analyzer
Configuration"
xmlns:wsi-analyzerConfig="http://www.ws-i.org/testing/2003/03/analyzerConfig/">
. . . .
<wsi-analyzerConfig:reportFile replace="true"
location="../logs/AnalyzerLog.xml">
<wsi-analyzerConfig:addStyleSheet href="../xsl/report.xsl"
type="text/xsl"/>
</wsi-analyzerConfig:reportFile>
<wsi-analyzerConfig:testAssertionsFile>
../profiles/WSRPProfileTestAssertions.xml
</wsi-analyzerConfig:testAssertionsFile>
<wsi-analyzerConfig:logFile correlationType="endpoint">
../logs/MonitorLog.xml
</wsi-analyzerConfig:logFile>
```

```
. . . .
</wsi-analyzerConfig:configuration>
```

2.4. Project Directory Structure

The WSRPTK project has the following directories:

Directory	Contents
bin	Batch files for running command line programs
config	Configuration files for the Monitor and Analyzer
lib	Jar files
logs	Generated message logs and reports
profiles	Profile assertions document
schemas	Schema files for WS-I tools as well as WSRP
wsdl	WSDL files for WSRP
xsl	Stylesheets for viewing logs and reports in HTML
src	Java source for WSRPTK
site	Source and intermediate build results for the WSRPTK web site
classes	Intermediate build results for WSRPTK
doc	This tutorial
javadoc	Generated javadoc for WSRPTK

2.5. Architecture

2.5.1. Introduction

The WSRPtk makes use of the Web Services Interoperability (WS-I) Testing Tools, by using a man-in-the-middle architecture to capture the web service message flow between Producer and Consumer. The Monitor tool is used to log messages between Producer and Consumer. It passes the messages unchanged between them. The messages are saved in the Message Log.

The Message Log file is then analyzed offline by the Analyzer tool to see if the messages

meet the conformance requirements. The WSRPtk extends the WS-I Analyzer by implementing test assertions that evaluate the messages with respect to the WSRP V1 Specification. The Profile is an XML file that contains a description of each test assertion, and is used to tell the Analyzer which test assertions to execute. The logic for each test assertion is written in a Java class which the Analyzer loads at runtime.

The Analyzer creates an XML file containing the Report which summarizes the results of the test assertions.

Diagram of Test Kit Architecture

2.6. Development

2.6.1. Development

2.6.1.1. Introduction

This section covers information needed by developers to add new test assertions to the V1 Conformance Test Kit. It is assumed that the reader is familiar with the test kit architecture and WSRP.

Creating a new Test Assertion requires 2 basic steps:

1. Enable the Test Assertion in the Profile, or add it to the Profile if it doesn't already exist.
2. Write the Java code to implement the Test Assertion logic.

Additional steps are required if the Test Assertion requires the use of the Specialized Portlet:

1. Add the JSP file to the webapp.
2. Add the JSP to the flow_script_master.xml and flow_script.xml files.

2.6.2. WSRP Profile

2.6.2.1. Introduction

The Profile is an XML document that contains a description of each Test Assertion in the test kit. The Profile is found in the `profile` directory and is called `WSRPProfileTestAssertions.xml`. If you open the file in a browser window, an XSL stylesheet is loaded which converts the XML to HTML for more convenient viewing.

The Profile was created initially from the conformance statements found in the WSRP Specification. The conformance statements are documented in a spreadsheet on the OASIS WSRP site. The conformance statements were boiled down into Test Assertions and documented in the Profile.

The Profile is read at runtime by the Analyzer to determine which Test Assertions should be executed. Test Assertions which are "enabled" are executed, the rest are skipped.

Types of Test Assertions

There are 3 types of Test Assertions. The primary focus the V1 Conformance Test Kit is to verify the message content of a running WSRP service. As a result, the vast majority of Test Assertions are of the Message type.

- Discovery -- Verifies UDDI entries
  - WSRP Profile has no UDDI assertions
- Description -- Verifies WSDL entries
  - WSRP Profile has some WSDL assertions
- Message -- Verifies message content in the log file
  - WSRP Profile has many message assertions
  - Primary focus of test kit.

If the configuration options for the Analyzer requires WSDL processing, then the Analyzer reads the WSDL and execute the Test Assertions from the Profile that are in the description artifact.

After the WSDL is evaluated, the Analyzer processes the message log file one message at a time. In general, for each message in the log file, the Analyzer executes each enabled Test Assertion in the Profile, although the entryType attribute results in some high level filtering. Test Assertions which have an entryType of requestMessage are executed only for requests, those having an entryType of responseMessage are executed only for responses, and those with an entryType of anyMessage, are executed for all messages.

A Test Asssertion evaluates each artifact passed to it, and generates a result. The result can be Pass, Fail, or Not Applicable.

The Analyzer finds the proper Java class to execute for a Test Assertion based on a naming convention where the Java class has the same name as the assertion id in the Profile.

Sample Test Assertion

```
<testAssertion id="RP0070" entryType="responseMessage" type="required"
enabled="false">
  <context>getServiceDescriptionResponse</context>
  <assertionDescription>
    If the Consumer uses a Producer who has set requiresInitCookie
    to a value other than "none", it shall:
    1. Invoke initCookie for each portlet from such a
```

```
    Producer for each new end-user;
    2. Return the set cookie(s) only for this end-user
    (see AS006, AS023 and AS024).
  </assertionDescription>
  <failureMessage>
    When a Producer sets ServiceDescription.requiresInitCookie
    to a value other than "none", the Consumer
    has to invoke initCookie()
  </failureMessage>
  <failureDetailDescription>{SOAP message}{any XML parser error
messages}</failureDetailDescription>
  <additionalEntryTypeList>
    <messageInput>none</messageInput>
    <wsdlInput>none</wsdlInput>
    <uddiInput>none</uddiInput>
  </additionalEntryTypeList>
  <prereqList/>
  <referenceList>
    <reference profileID="WSRP1"/>
  </referenceList>
  <comments/>
</testAssertion>
```

Elements and Attributes of a Test Assertion

Many of the elements and attributes in the Test Assertion are not used, but the ones that are important are:

Name	Description
id attribute	This is the unique name of the Test Case. Naming conventions are discussed in more detail below.
Type attribute	This is determines to which type of message log td the Test Case applies. For example: responseMessage, requestMessage, binding.
type attribute	"required" or "recommended". Used to distinguish MUST from SHOULD in Conformance Statements.
enabled attribute	"true" or "false". If "true", execute the test case on the data. If "false", just skip it. This is used when the Test Case is added to the Profile but the code hasn't been written yet. The Test Case will show up on the report as not enabled.

assertionDescription element	This is the same as the Conformance Statement text.
failureMessage element	This text is output in the report when a Test Case fails.
prereqList element	This element is used to list other Test Cases (by id) that must be run before this one. If a prerequisite Test Case fails, then so does this one.

2.6.3. Programming New Test Assertions

2.6.3.1. Introduction

The following sections provide an overview of the organization of the Java classes which makeup the test kit. The test kit extends the WS-I implementation and relies on the behaviour of existing base classes wherever possible.

2.6.3.2. Factory

There is an extension point in the WS-I framework which allows us to insert our own factory implementation during initialization. We use these factories to cause our specialized classes to be constructed. The `wsi.properties` file specifies our factory:

```
# ----------------------------------------------------------------------
# Profile validation factory.
# ----------------------------------------------------------------------
wsi.profile.validator.factory=org.oasis.wsrp.test.impl.WSRPValidatorFactoryImpl
```

2.6.3.3. WS-I Base Validator

Refer to the diagram below. The classes in yellow are the WS-I base classes and the classes in white are the WSRP extensions. The `WSRPValidatorFactoryImpl` creates the WSRP specific extensions when the Analyzer is started. The `WSRPMessageValidatorImpl` class plays an important role in the execution of Test Assertions because there is only one instance of this class for all message Test Assertions. The `WSRPMessageValidatorImpl` instance is used to share data between Test Assertions or between messages. This is discussed in more detail in a subsequent section.

Relationships among base validateor implementations and extensions

In the WSRPTK, there are very few Test Assertions which deal with the WSDL. The vast majority of Test Assertions are focused on examining the messages in the log file. In general, there is a symmetrical treatment of message artifacts and description artifacts. For the next

few sections, only the structure and relationships among the classes involved in the message processing side of the system are discussed. Classes which handle description artifacts may be included on the diagrams, but are not distinguished further.

2.6.3.4. Extending AssertionProcess

Refer to the diagram below. Again, the yellow classes are WS-I base classes, and the white classes are WSRP specific. The small class, `RP0090`, at the bottom of the diagram is an example of a Test Assertion.

Test Assertions ultimately extend AssertionProcess and get a reference to WSRPMessageValidatorImpl

Ultimately, all Test Assertions are derived from `AssertionProcess`. This class declares the `validate()` method that all Test Assertions must implement. The `AssertionProcess` class also declares a member variable to hold an instance of `BaseValidatorImpl`.

WSRPTK introduces 2 intermediate classes between Test Assertions and `AssertionProcess`. `WSRPBaseAssertionProcess` is used to implement convenience methods such as `pass()`, `fail()`, etc, for setting the result of a Test Assertion, regardless of whether the Test Assertion is on the message artifact side or the description artifact side.

2.6.3.5. Constructor Hack

`WSRPMessageAssertionProcess` extends the `WSRPBaseAssertionProcess` in order to declare the instance of the specialized `WSRPMessageValidatorImpl` to which all the Test Assertions need access for data sharing. The `BaseValidatorImpl` is responsible for parsing and loading the message log file and then creating and invoking the Test Assertions in the Profile. When a Test Assertion Java class, e.g. `RP0090`, gets initialized, the `BaseValidatorImpl` passes an instance of `BaseMessageValidatorImpl` into the constructor. The Test Assertion calls the constructor on it's superclass, `WSRPMessageAssertionProcess`, passing the `BaseMessageValidatorImpl`. The superclass forces a cast of the `BaseMessageValidatorImpl` to `WSRPMessageValidatorImpl` and saves it in a protected member variable called `validator`.

This hack is necessary for historical reasons. The original WS-I framework was organized so that all the Test Assertion classes were inner classes of the `WSRPMessageValidatorImpl` class, and so they had direct access to its methods and state. At some point, the architecture was changed so that the Test Assertions were separated into individual files and this hack became necessary to preserve access to the specialized

methods and state of the `WSRPMessageValidatorImpl`.

2.6.3.6. WSRPMessageValidatorImpl

As explained above, there is a single instance of this class which is available to all Test Assertions through the use of the protected variable `validator`. This class offers many convenience methods as well as saving state between messages. Let's look at the following Test Assertion:

RP0510	The value supplied in MarkupParams.windowState shall be either "wsrp:view" or one of the values from the PortletDescription.markupTypes[].windowStates for a matching mime type, which has to have a value

This Test Assertion looks at a `getMarkupRequest` and has to check the value of window state against a set of valid window states which were supplied in the `getServiceDescriptionResponse`. In the following code fragment, `RP0510` uses convenience methods from the `WSRPMessageValidatorImpl` to locate the `offeredPortlet` from the saved `getServiceDescriptionResponse`.

```
//-----------------------------------
// we got a portlet handle out of the request, so
// lookup the offeredPortlet in the getServiceDescriptionResponse which
was
// previously saved
regHandle = validator.findRegistrationHandle((Node)
doc.getDocumentElement());
portletHandle = validator.checkForClonedPortlet(portletHandle,
regHandle);
offeredPortlet = validator.findOfferedPortlet(portletHandle);
if (offeredPortlet == null) {
// this means the portlet was not found in the serviceDescriptionResponse
fail("Cannot find portlet with handle: " +
portletHandle +
" in serviceDescriptionResponse");
done = true;
}
```

This Test Assertion relies on the fact that another Test Assertion, `RP0320` saves the `serviceDescriptionResponse`. Note that if a log file is analyzed that does not contain the `serviceDescriptionResponse` message, then any Test Assertion that relies on it will fail.

The `WSRPMessageValidatorImpl` has many methods which are beyond the scope of this tutorial. The reader is encouraged to examine the source code. The following diagram

shows the `WSRPMessageValidatorImpl` and all the classes it references. Not all the methods are visible.

WSRPMessageValidatorImpl

2.6.3.7. Choosing a Base Class to Extend

There are several choices as to which class to extend to create a new Test Assertion. The `WSRPBaseAssertionProcess` class, the `BaseRP` class, `CustomItemChecker`, or an existing Test Assertion. The diagram below shows several Test Assertions and the classes they extend. `CustomItemChecker` is specialized and is used to parse the markup have a listener called when certain things are found. The `BaseRP` class checks the message type and then delegates handling of the message to the Test Assertion subclass. This design pattern was introduced after many Test Assertions were already implemented, so you may discover Test Assertions that would have benefited from this approach.

WSRPMessageAssertionProcess

2.6.3.8. Worked Example

Let's look at a simple Test Assertion.

RP0090	Every time an extension element appears in a WSRP message, it's child element shall use the xsi:type attribute to declare its type.

Here is the source code:

```
/*
* Copyright (c) 2002-2005 IBM Corporation. All rights reserved.
*
* ==========================================
*
* @author Julie MacNaught jmacna@us.ibm.com
* @author Martin Fanta Martin.Fanta@cz.ibm.com
*
*/
package org.oasis.wsrp.test.impl.message;
import javax.xml.transform.TransformerException;
import org.eclipse.wst.wsi.internal.core.WSIException;
import org.eclipse.wst.wsi.internal.core.profile.TestAssertion;
import org.eclipse.wst.wsi.internal.core.profile.validator.EntryContext;
import
org.eclipse.wst.wsi.internal.core.profile.validator.impl.BaseMessageValidator;
import org.eclipse.wst.wsi.internal.core.report.AssertionResult;
import org.w3c.dom.Document;
import org.w3c.dom.NamedNodeMap;
```

```
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
/**
* Assertion: Every time an extension element appears in a WSRP message,
it's child
* element shall use the xsi:type attribute to declare its type.
*
* @author Julie MacNaught (jmacna@us.ibm.com)
* @author Martin Fanta (Martin.Fanta@cz.ibm.com)
*
*/
public class RP0090 extends WSRPMessageAssertionProcess {
public RP0090(BaseMessageValidator impl) {
  super(impl);
}
public AssertionResult validate(TestAssertion testAssertion, EntryContext
entryContext) throws WSIException {
  if (validator.isOneWayResponse(entryContext)) {
    na();
  } else {
    try {
      // parse the request message
      Document reqDoc = entryContext.getRequestDocument();
      // get all the extension elements included in the request
      NodeList extNodes = NodeUtils.getNodes(reqDoc, "extension");
      int extNodeCount = extNodes.getLength();
      if (extNodeCount < 1) {
        na();
      } else {
        Node extNode, childNode;
        NamedNodeMap attribs;
        for (int ii = 0; ii < extNodeCount; ii++) {
          extNode = extNodes.item(ii);
          childNode = extNode.getFirstChild();
          if (childNode != null) {
            attribs = childNode.getAttributes();
            if (attribs.getNamedItem("xsi:type") != null) {
              // it is probably sufficient to assume that if the
attribute is present,
              // the condition of this test case is fulfilled
              pass();
            } else {
              fail("If an extension element is present, its child has to
have the 'xsi:type' attribute defined");
            }
          } else {
            warn("Extension element has no children");
```

```
          }
        }
      }
    } catch (TransformerException te) {
      te.printStackTrace();
      fail(te.getMessage());
    }
  }
  return createAssertionResult(testAssertion, result,
failureDetailMessage);
}
}
```

This Test Assertion extends `WSRPBaseAssertionProcess` and implements the `validate()` method. It also implements a constructor so it can establish a handle to the `WSRPMessageValidatorImpl` class. It uses a class called `NodeUtils` which implements various shortcuts to XPATH for finding nodes in messages.

2.6.4. Specialized Portlet

2.6.4.1. Introduction

The purpose of the Specialized Portlet is to cause certain conditions to occur so that the Analyzer can look a the messages and detemine if the Consumer handled the situation according to the spec. The Specialized Portlet generates markup with particular values that the Analyzer can look for in the message log.

The Specialized Portlet works by reading a file called `flow_script.xml` which tells it which JSP file to load. The JSP file actually contains the markup that the Specialized Portlet supplies on a getMarkupRequest. The `flow_script.xml` file allows test cases to be repeated a particular number of times.

Here is a sample of the `flow_script.xml` file:

2.6.4.2. flowscript.xml

The SpecializedPortlet reads an XML file, called `flow_script.xml`, which tells the portlet which tests to execute. The test cases are actually represented by JSP files. Here is a sample of the `flow_script.xml` file:

```
<flow>
  <case id="134">
    <pass repeatCount="2">
      <jspInclude name="test_case_1340_1.jsp" />
    </pass>
```

```
  </case>
</flow>
```

A Specialized Portlet JSP typically contains a form and has javascript logic that causes the form to be automatically submitted. This allows the Specialized Portlet to execute all the tests without further human intervention.

2.6.4.3. Specialized Portlet Example

Let's look at an example:

RP1340	When an activated portlet URL has specified the wsrp-navigationalState portlet URL parameter, the Consumer shall supply its value unchanged in the MarkupParams.navigationalState field.

Here is the JSP file for this test (line feeds have been added to improve readability):

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri='/WEB-INF/tld/portlet.tld' prefix='portlet'%>
<portlet:defineObjects/>
<script>
setTimeout('wsrp_rewrite_reload()',5000);
function wsrp_rewrite_reload()
{
  // only the first instance of the portlet shall submit the form
  var form = document.forms.wsrp_test_submit_form;
  if((form.wsrp_test_counter.value == "0") &&
(form.wsrp_test_autoSubmit.value == "true"))
  {
    form.wsrp_test_autoSubmit.value = "false";
    form.wsrp_test_counter.value = "1";
    // callback function for JSP's which are including this javascript
    wsrp_test_do(form);
    form.submit();
  }
}
function wsrp_test_do(form)
{
}
</script>
<form name="wsrp_test_submit_form" method="post"
  action="wsrp_rewrite?
    wsrp-urlType=blockingAction&
    wsrp-mode=wsrp:view&
    wsrp-windowState=wsrp:normal&
    wsrp-secureURL=false&
```

```
    wsrpnavigationalState=wsrp_test_navigationalState
  /wsrp_rewrite
    &wsrp_test_expectedNavigationalState=wsrp_test_navigationalState">
  Test case #134.<br>
  <input type="hidden" name="wsrp_test_autoSubmit" value="true"></input>
  <input type="hidden" name="wsrp_test_currentCase"
    value="<%=(String)renderRequest.getAttribute("wsrp_test_currentCase")%>"></input>
  <input type="hidden" name="wsrp_test_passCounter"
    value="<%=(String)renderRequest.getAttribute("wsrp_test_passCounter")%>"></input>
  <input type="hidden" name="wsrp_test_counter" value="0"></input>
  <input type="submit" value="Submit"></input>
</form>
```

Notice that the rewrite URL contains navigationalState. Here is the Test Assertion code that examines the log:

```
private static final String CODEM_NAME_EXPECTED_NAVIGATIONAL_STATE =
"wsrp_test_expectedNavigationalState";
protected void processPerformBlockingInteractionRequest(Document reqDoc)
  throws WSIException, TransformerException {
  // find the interactionParams node
  Node iaParamsNode = NodeUtils.getNode(reqDoc, "interactionParams");
  if (iaParamsNode != null) {
    // find all form codemeters
    NodeList formParamNodes = NodeUtils.getNodes(iaParamsNode,
"formParameters");
    int formParamCount = formParamNodes.getLength();
    Node formParamNode;
    String formParamName, expectedNavStateValue = null;
    // go through the form codemeters until we find the one that
interests us (if any)
    for (int ii = 0; ii < formParamCount; ii++) {
      formParamNode = formParamNodes.item(ii);
      formParamName = NodeUtils.getAttributeValue(formParamNode, "name");
      if (CODEM_NAME_EXPECTED_NAVIGATIONAL_STATE.equals(formParamName)) {
        // find the value subnode
        Node valueNode = NodeUtils.getTextNode(formParamNode, "value");
        expectedNavStateValue =
NodeUtils.getSafeTextNodeValue(valueNode);
        break;
      }
    } // for ii
    if (expectedNavStateValue != null) {
      // compare the expected navigational state value to the one
actaully supplied
      // find the markupParams node
      Node markupParamsNode = NodeUtils.getNode(reqDoc, "markupParams");
      if (markupParamsNode != null) {
```

```
        // find the navigational state subnode
        Node navStateNode = NodeUtils.getTextNode(markupParamsNode,
"navigationalState");
        String navState = NodeUtils.getSafeTextNodeValue(navStateNode);
        if (expectedNavStateValue.equals(navState)) {
          pass();
        } else {
          fail(
            "Expected navigational state: "
              + expectedNavStateValue
              + "; supplied navigational state: "
              + navState);
        }
      } else {
        fail("Missing markupParams node");
      }
    } // expected navigational state value not null
  } // interaction codems not null
} // processPerformBlockingInteractionRequest()
```

The Test Assertion looks at the value of an interaction parameter for the value that the navigationalState should contain.

2.7. WSRP Conformance Test Kit Tutorial

2.7.1.

Click here to start

Table of contents

WSRP Conformance Test Kit Tutorial

Topics

Test Kit Requirements

WS-I Test Tools

WSRP Conformance Test Kit Architecture

WSRP Profile

Running Everything End to End

WS-I Components

Monitor

Analyzer

Author: Julie MacNaught

E-mail: jmacna@us.ibm.com

Homepage: http://wsrptk.sourceforge.net/

Download presentation

Analyzer Configuration File Profile Details Sample Test Assertion Test Assertion Implementation WSRPValidatorFactoryImpl WS-I Base Classes Assertion Constructor Magic Using WSRPMessageValidatorImpl for State Extending Base Classes Test Assertion Example Slide 21 NodeUtils Specialized Portlet XML Script File Specialized Portlet Example Slide 26 Slide 27 Running the Specialized Portlet WSRPTK Source Forge Project WSRPTK Directory Organization Prerequisites Prerequisites, contâ€™d Slide 33 Contributing to the WSRPTK Project Source Forge User Types Becoming a Developer Project Administration Release Process	

Handling 3rd Party Contributions Project TODOs	

3. All