



J A D E™

Web Services Tips and Techniques

Jade Software Corporation Limited cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages, or loss of profits. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Jade Software Corporation Limited.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Copyright © 2010 Jade Software Corporation Limited.

All rights reserved.

JADE is a trademark of Jade Software Corporation Limited. All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

Contents

Web Services Tips and Techniques	3
Overview	3
Session Management.....	3
Session Definition	3
WSDL Generation.....	4
Runtime Processing	5
Using the Session Object.....	6
Timing Out Sessions.....	6
Removing Sessions	6
Performance Testing.....	7
soapUI	7
JADE Monitor.....	14
Debugging	15
JADE Debugger.....	15
Logging	15
IIS.....	15
Jadehttp	18
Web Application	18
Method Re-implementation.....	19
Fiddler.....	19
Message Sizes	21
Web Server Setup.....	22
Create an Application Pool.....	23
Create a Virtual Directory.....	24
Set Up a Handler Mapping.....	25
Set Up ISAPI Restrictions	27
Usage Statistics	28
Web Service Client	28
Web Service Application	31
Consumer Asynchronous Calls	32
WSDL Import	33
Setting up a Worker Application	34
Coding Example.....	35
Using SOAP Headers.....	38
The SOAP Header Element	38
Information to Put in the Header	39
Defining SOAP Headers	40
Inserting Authentication Headers.....	44
Denial of Service.....	45
Badly-Formed Messages.....	45
Handling Badly-Formed Messages.....	47
Exception Handling.....	48

Web Services Tips and Techniques

Overview

This white paper complements the white paper “Web Services in JADE” and is intended to provide useful information to assist a JADE Web services developer. These tips and techniques are based on user feedback about the areas for which people have had the most questions. The areas that are covered in this paper are:

- Session management
- Performance measurement
- Debugging
- Setting up a Web server
- Usage statistics
- Calling a Web service asynchronously
- Using SOAP headers
- Message sizes
- Non-malicious denial of service
- Exception handling

Please note that all references to the JADE platform in the following sections are relative to JADE 6.3.

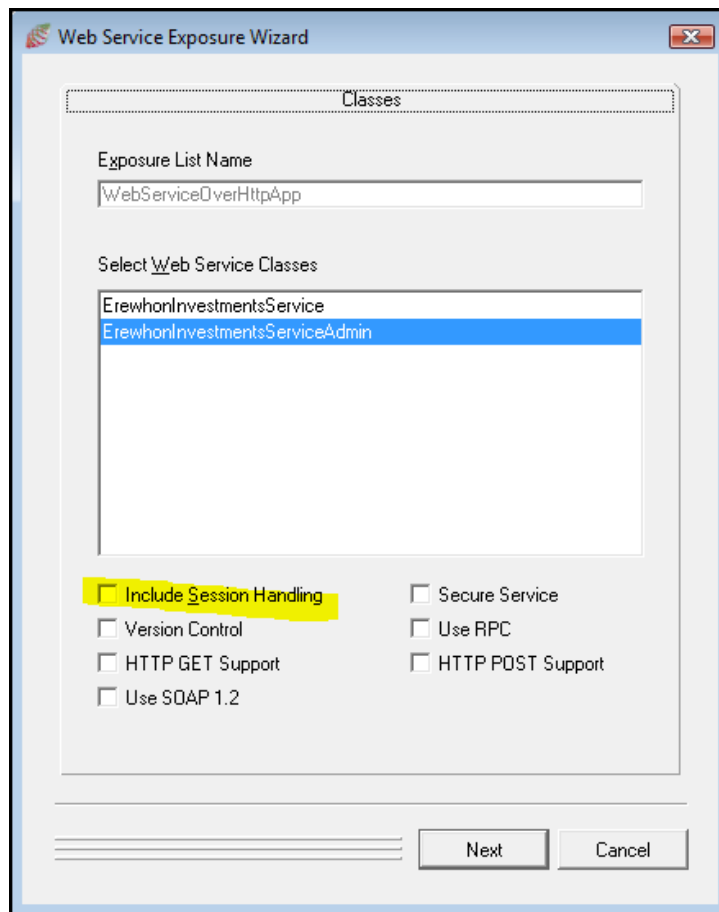
Session Management

Session management in Web services is optional and if required, it must be specified at design time; that is, this is not a deployment option. The reason is because the generated WSDL has information relating to sessions.

Session Definition

The inclusion of session management in a Web service is defined as part of the exposure list definition.

The following dialog illustrates this.



Check the **Include Session Handling** check box, to include session handling.

WSDL Generation

The WSDL that is generated will now include session handling information against every method.

Below is a snippet of a WSDL containing session-related information.

In the <types> definition section:

```
</xsd:complexType>
  <xsd:element name="JadeSessionHeader" type="tns:JadeSessionHeader"/>
  <xsd:complexType name="JadeSessionHeader">
    <xsd:sequence>
      <xsd:element name="sessionId" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
```

In the message section:

```
<message name="getClientJadeSessionHeader">
  <part name="sessionId" element="tns:JadeSessionHeader"/>
</message>
```

In the <binding> section:

```
<operation name="getClient">
  <soap:operation
    soapAction="urn:JadeWebServices/WebServiceOverHttpApp/getClient"
    style="document"/>
    <input>
      <soap:body use="literal" />
      <soap:header use="literal"
        message="tns:getClientJadeSessionHeader" part="sessionId"/>
    </input>
    <output>
      <soap:body use="literal" />
      <soap:header use="literal"
        message="tns:getClientJadeSessionHeader" part="sessionId"/>
    </output>
  </operation>
```

This information tells the target system to create a class called **JadeSessionHeader** with a single string property on import and to generate a SOAP header containing the session id at execution time.

Note As the <header> tag is defined in the <input> and <output> sections, this SOAP header is Input-Output.

Runtime Processing

At run time, the following processing occurs.

- The Web service client sends a request.
- The Web service receives the request and if there is no SOAP header or the SOAP header does not have a session id, the Web service generates a header with a new session id in the response. If there is session id information, this is used to obtain the session object.
- The Web service client must send the generated session id with every request or a new session will be created each time. Generally, your code does not have to do anything – the framework will handle the passing of this information.

An example of a SOAP message with session information is shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <JadeSessionHeader
      xmlns="urn:JadeWebServices/WebServiceOverHttpApp/">
      <sessionId>0fd1c8984243d2f8</sessionId>
    </JadeSessionHeader>
  </soap:Header>
  <soap:Body>
    .....
  </soap:Body>
</soap:Envelope>
```

Using the Session Object

When session management is used, a persistent instance of the subclass of **WebSession** in the current schema is created. This instance can be referred to in code using the system variable **currentSession**. Additional properties and methods can be added to this class for storing state information. As this is a persistent instance, any updating of these properties must be done in transaction state.

Note The **currentSession** object is valid only for the duration of the Web service call. Any attempt to reference its properties or methods outside of this call (for example, in notification code) will result in a null object reference error.

Timing Out Sessions

Sessions can be set to automatically time out based on a configuration setting. This can be set in the Define Application dialog or dynamically configured at run time. When the timeout is set and there is no activity for the session within that time, the session is deleted. If this session is subsequently referenced, a SOAP fault (error code 11007) will be returned to the Web service client.

It is recommended that session timeouts be used to clean up unused sessions. The default message for this can be changed by re-implementing the

Application::timedOutSessionMessage method. Note, however, that the string returned by this method must be a SOAP message. The response returned by this message is what will be returned to the Web service client if a subsequent request uses the session id of the timed out session. See the tip in the next section, "Removing Sessions".

Note When all Web service applications are shut down or the database node is shut down, all Web sessions are deleted.

Removing Sessions

There may be situations where a session needs to be removed immediately; for example, when a user logs off. To do this, you can call the **WebSession::removeSession** method.

If you are removing the current session and want to return a specific message, you can call the **WebSession::removeSessionWithMessage** method, passing the message you want to return.

In all other cases, if you want to send a message that differs from the default, you can reimplement the **Application::removeSessionMessage** method. Note, however that the string returned by this method must be a SOAP message. The response returned by this message is what will be returned to the Web service client if a subsequent request uses the session id of the removed session.

Tip One way of creating a SOAP message in this case would be to create a **JadeSOAPException** instance, populate the **errorCode**, **errorItem**, and **extendedErrorText** properties, and then call the **JadeSOAPException** class **createSOAPMessage** method to generate the SOAP formatted message.

Performance Testing

soapUI

Performance and scalability of JADE Web services can be tested by using the freely available soapUI tool. The use of this tool will be briefly demonstrated in this section.

The version of soapUI used in this example is 3.0.

Before we start, we should do the following.

1. Disable logging at the Web server.

You do this in IIS by using the MMC snap-in. See the "Debugging" section, later in this document, for details.

2. Minimize logging by **jadehttp**.

If you turned tracing on in **jadehttp**, turn this off. See the "Debugging" section, later in this document, for the configuration options.

3. Turn off logging by the Web service application. See the "Debugging" section, later in this document, for configuration options.

4. Change your Web service provider applications to be Web-enabled non-GUI applications.

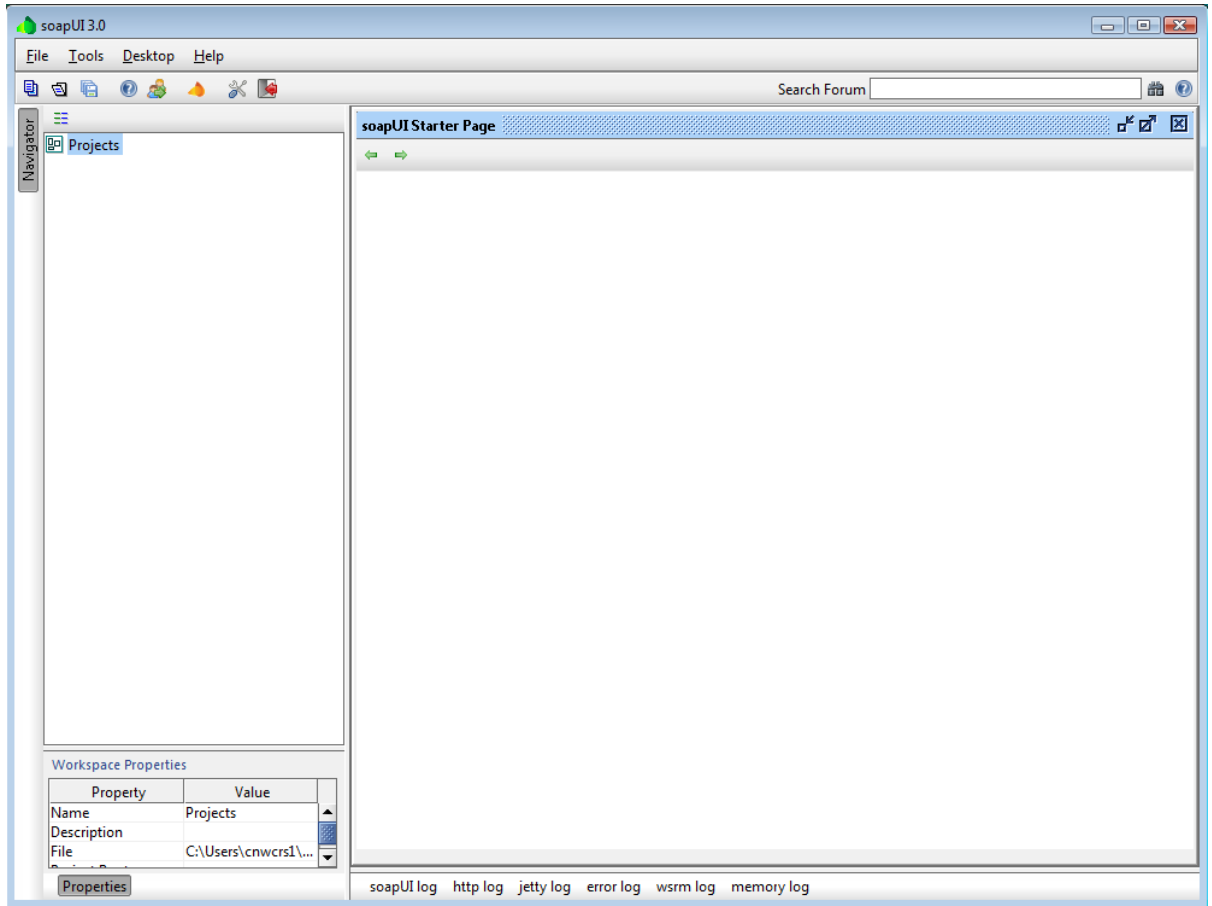
If you use the GUI version, your performance will be degraded, as the Web monitor status window is continually updated.

We will use **ErewhonInvestmentsViewSchema** in the Erewhon sample system to demonstrate the use of soapUI.

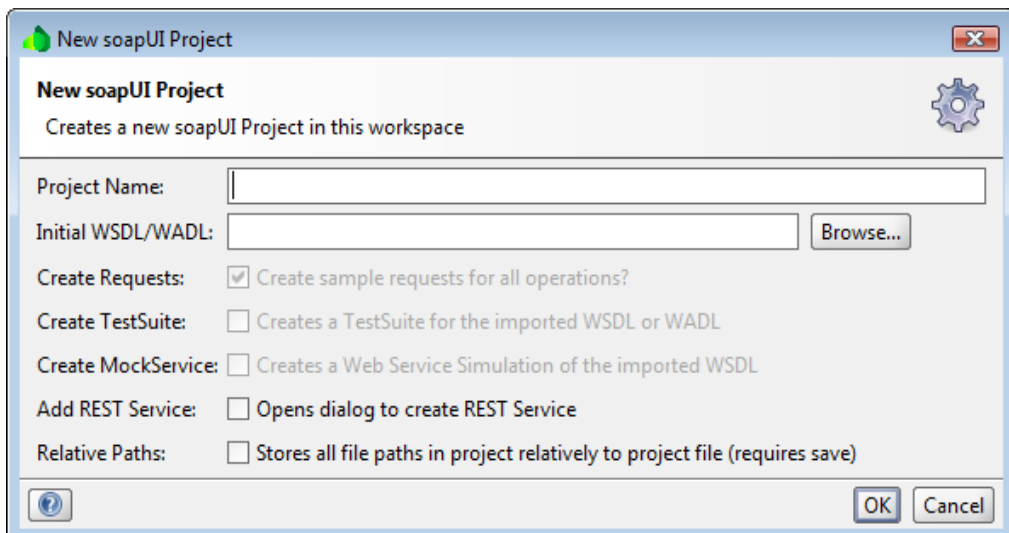
1. From the Application Browser, select the **WebServiceOverHttpApp** application.
2. On the **Web Options** sheet of the Define Application dialog, change the number of application copies to **5**.
3. Generate the WSDL.

Now run the application. You should have five copies of the application running.

Now bring up soapUI, which will display the following window.



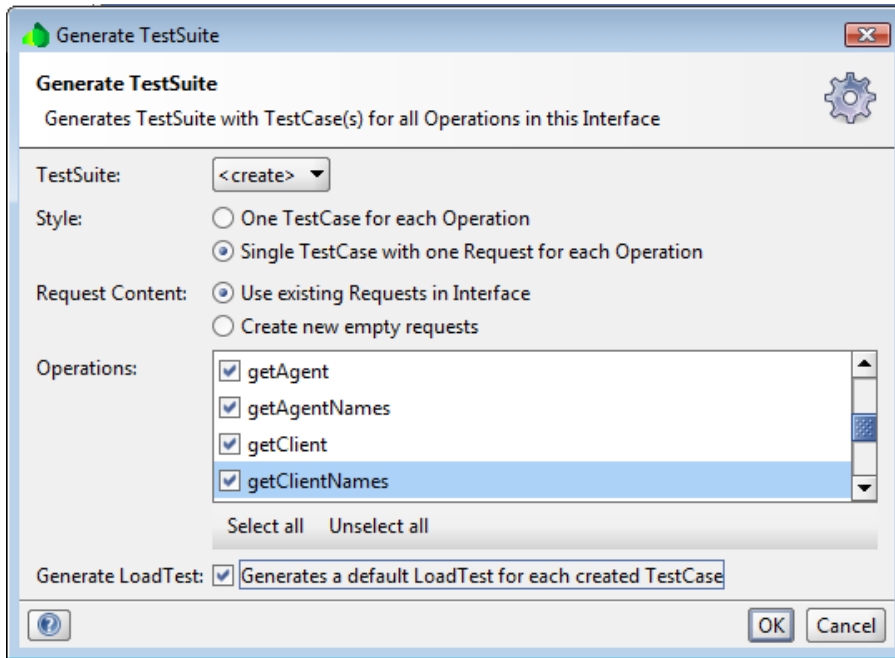
1. Select **New soapUI project** from the File menu, which will bring up the following dialog.



2. Enter the project name of **ErewhonWebServiceTest**.
3. Enter the WSDL path of the WSDL that was extracted previously.

4. Check the **Create TestSuite** check box.
5. Click **OK**.

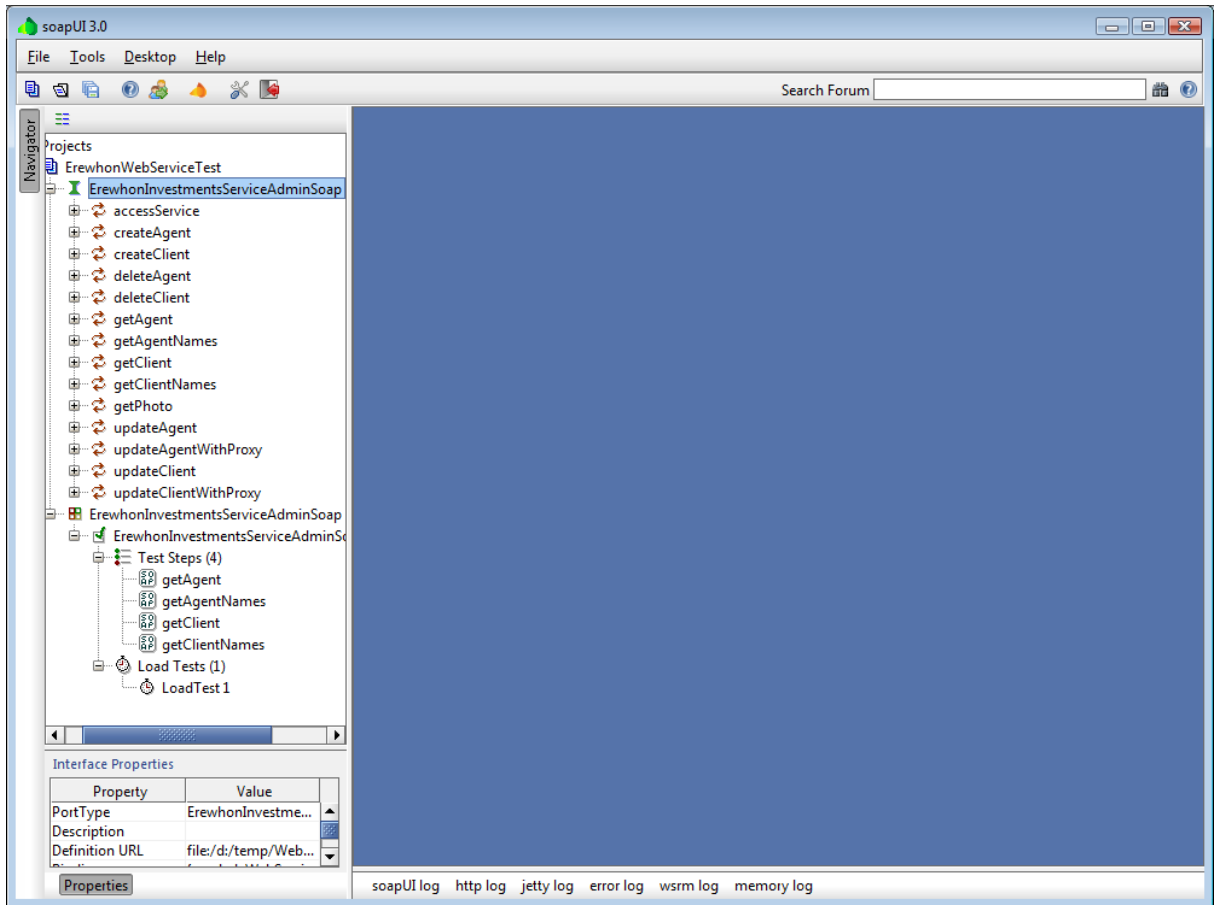
This will bring up the following dialog.



To keep this example simple, we create a test only for the four operations selected above. Unselect all other operations and then click **OK**.

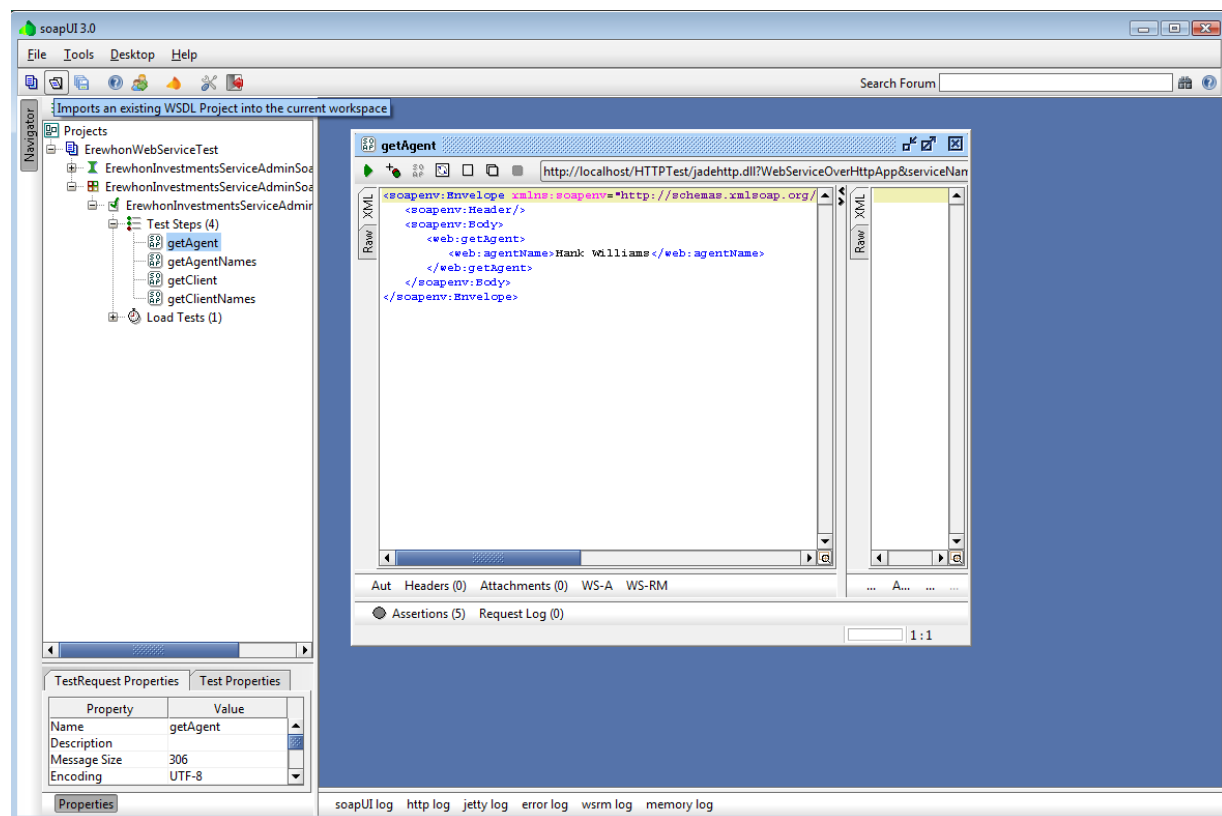
When the request for a name for the test suite comes up, just click **OK** on the dialog.

You should now have a window that looks like the following (you may need to expand your project to see this).



We will need to enter some data for the **getAgent** and **getClient** methods.


The following diagram shows an example of the **getAgent** operation.

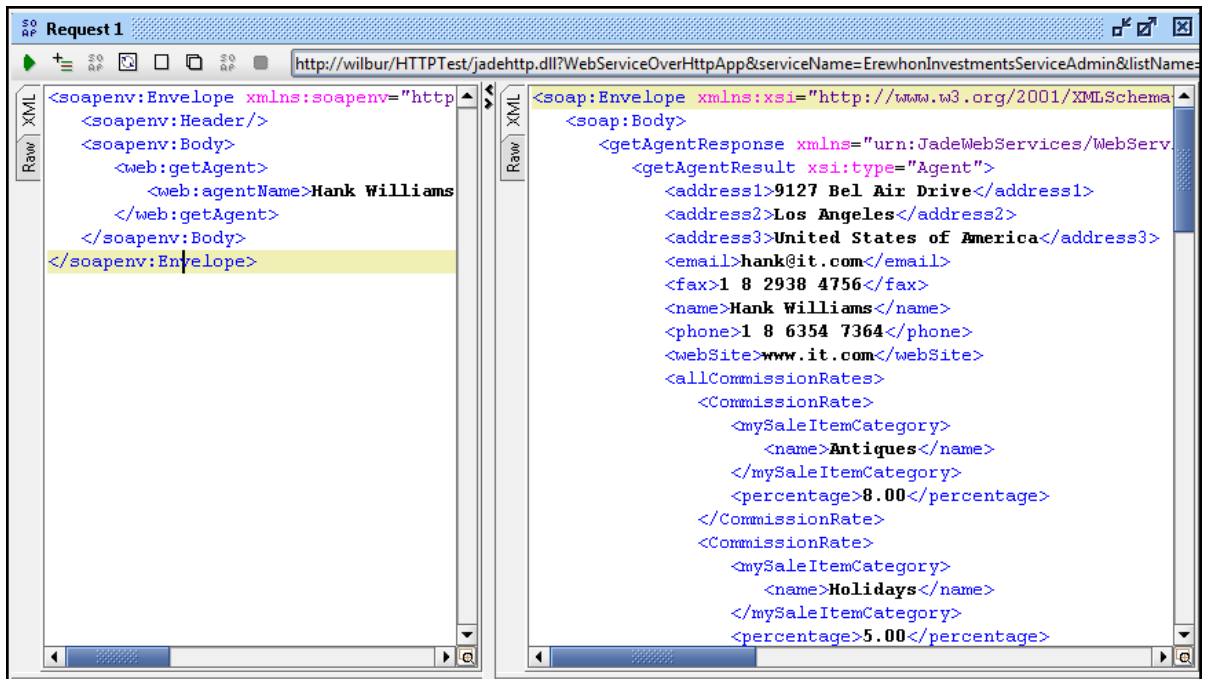


To do this, double-click on the **getAgent** entry under **Test Steps** and then enter **Hank Williams** between the `<web:agentName>` and the `</web:agentName>` tags.

Similarly, enter **Brian Olsen** for the **getClient** request.

We are now ready to run the test.

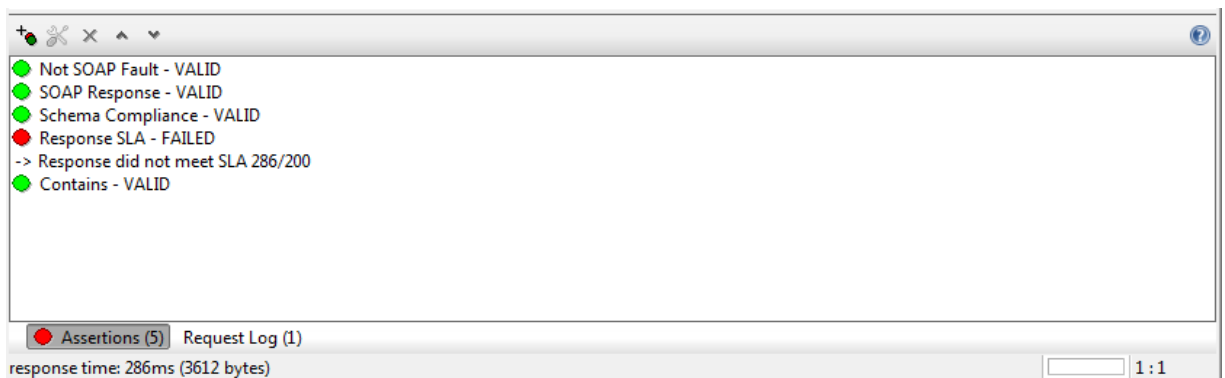
Click the  icon on the getAgent window. This will call the Web service and return a SOAP message on the right pane, as shown in the following diagram.



You can also set up assertions for the call. In this example, I set up five assertions as follows,

- Not SOAP fault
- SOAP Response
- Schema Compliance (WSDL compliance)
- Response SLA – set to 200 ms
- Response contains **Hank Williams**

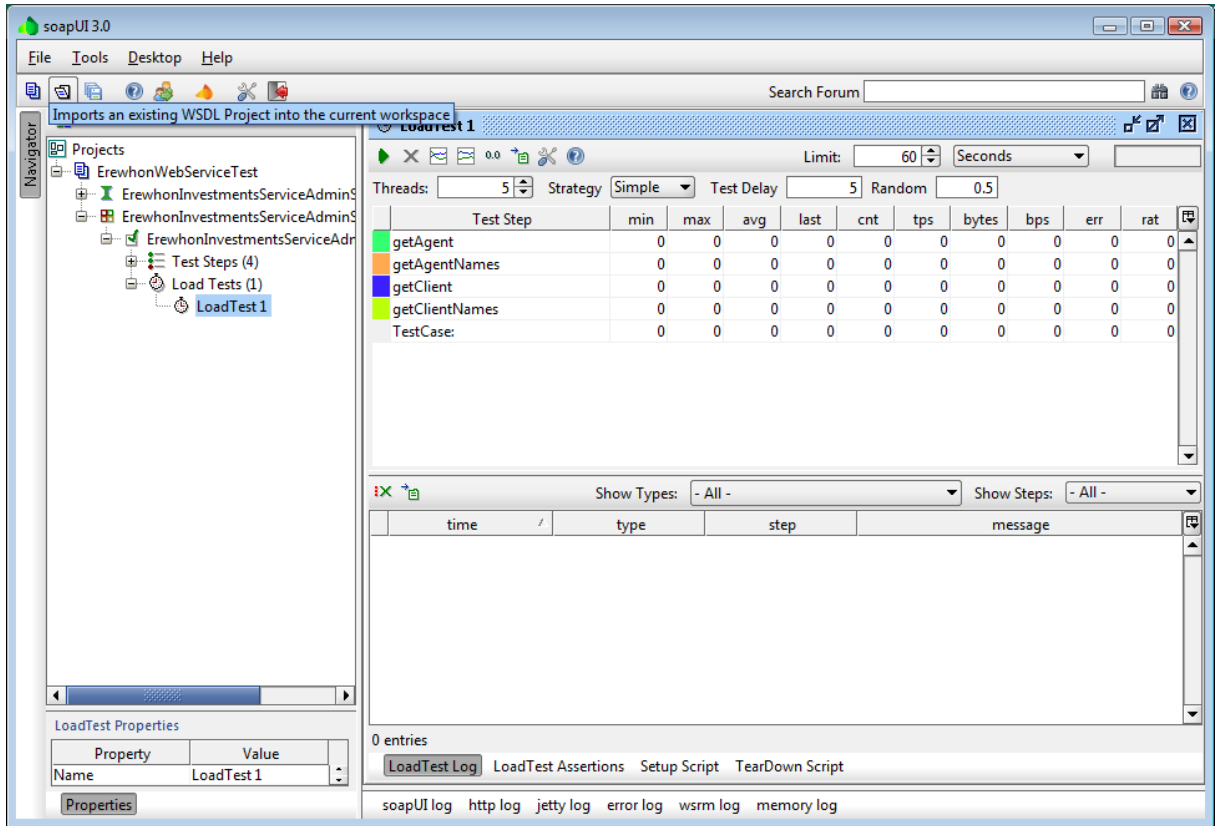
The resulting assertions from this call are shown in the following diagram.



The response time was greater than 200 ms (286ms) and hence the Response SLA assertion failed.

See the <http://www.soapui.org/userguide/functional/response-assertions.html> soapUI online reference for documentation about assertions.

You can also run the tests multiple times.



Double-clicking on the **LoadTest1** item in the left pane will bring up the load test window on the right-hand window.

All of the items that we selected earlier are available for load testing.

There are several options to the test; for example, number of threads, limit, strategy, and so on.

We are going to run the simple tests with a test delay of five ms, five threads, and running for one minute. Running this produces the following result.

The screenshot displays the soapUI 3.0 interface. The main window shows the configuration for 'LoadTest 1'. The test is set to run for 60 seconds with 5 threads using a 'Simple' strategy and a 5ms test delay. The results table below the configuration shows the following data:

Test Step	min	max	avg	last	cnt	tps	bytes	bps	err	rat
getAgent	5	1721	10.31	12	8565	142.75	30936...	515630	3	0
getAgentNames	2	225	6.71	6	8562	142.7	6318756	105316	0	0
getClient	2	41	6.09	9	8562	142.7	4640604	77345	0	0
getClientNames	3	220	6.9	6	8562	142.7	10068...	167820	0	0
TestCase:	12	2207	30.02	33	8565	142.75	51965...	866113	3	0

The bottom pane shows a log of test events. The log entries are as follows:

time	type	step	message
2009-09-09 14:29:52:086	Message		LoadTest started at Wed Sep 09 14:29:52 NZST ...
2009-09-09 14:30:28:754	Step Status	getAgent	TestStep [getAgent] result status is FAILED; [[Re...
2009-09-09 14:30:28:760	Step Status	getAgent	TestStep [getAgent] result status is FAILED; [[Re...
2009-09-09 14:30:28:760	Step Status	getAgent	TestStep [getAgent] result status is FAILED; [[Re...
2009-09-09 14:30:52:089	Message		LoadTest ended at Wed Sep 09 14:30:52 NZST 2...

The results show that we are getting transactions per second (a tps) of around 140. Note that in the **getAgent** case, there are three errors recorded and the bottom pane shows the reason. In this instance, all three errors relate to one of the assertions we added to the **getAgent** operation exceeding the required response time of 200ms. Putting in a longer delay will eliminate these errors but the tps will then be fewer.

There are scripts that can be written to perform any initialization before the test is run (Setup Script) and to perform any finalization when the test is complete (TearDown Script). These tests can also be run without the UI. Refer to the soapUI documentation for further information.

The above discussion relates to measuring performance when providing a Web service. If you are now consuming an external Web service and want to measure performance as a JADE consumer, currently you will have to write your own test framework. If, however, you want to test just the performance of the Web service, you can obviously use the soapUI tool by the process mentioned above.

JADE Monitor

The JADE Monitor is an extremely useful tool when it comes to measuring performance. It can provide you with statistical information, method analysis, and file analysis in order to help you determine the bottlenecks in your system. Refer to the *JADE Monitor User's Guide (JadeMonitor.pdf)* for more details.

Debugging

There are several debugging aids available to help debug your Web service applications.

- JADE Debugger
- IIS logs, jadehttp logs, Web application logs
- Method reimplementations to capture additional information
- Fiddler tool

JADE Debugger

The JADE Debugger can be used to step through JADE code in a Web service application or a Web service consumer application. The usage of the debugger should be familiar to JADE users, and will not be described here. See Chapter 7 of the *JADE Development Environment User's Guide (UserGuide.pdf)* for more details.

Logging

IIS

The discussion below refers only to the use of IIS 7. For other releases, consult the appropriate documentation. Logging can be enabled using the IIS Manager from the MMC console. From here, we can get to the logging options window by selecting **Default Web Site** and then selecting **Logging**, which brings up the following window.

The screenshot shows the 'Logging' configuration window in IIS Manager. The window title is 'Logging' and it includes a globe icon. Below the title, there is a description: 'Use this feature to configure how IIS logs requests on the Web server.' The 'One log file per:' dropdown is set to 'Site'. The 'Log File' section contains a 'Format:' dropdown set to 'IIS' with a 'Select Fields' button, a 'Directory:' text box containing '%SystemDrive%\inetpub\logs\LogFiles' and a 'Browse...' button, and an 'Encoding:' dropdown set to 'UTF-8'. The 'Log File Rollover' section has a radio button selected for 'Schedule:' with a 'Daily' dropdown, and two other radio buttons for 'Maximum file size (in bytes):' and 'Do not create new log files'. There is also a checkbox for 'Use local time for file naming and rollover'. On the right side, there is an 'Actions' pane with buttons for 'Apply', 'Cancel', 'Disable', 'View Log Files...', 'Help', and 'Online Help'. At the bottom left, there are tabs for 'Features View' and 'Content View'.

Set up options, as required. IIS format for the log file gives you the most information. Make sure that logging is enabled (the Actions window will show **Disable**, if it is).

Making requests to your Web service will now log information similar to the following.

```

File Edit Format View Help
127.0.0.1, -, 9/10/2009, 10:36:56, W3SVC1, WILBUR, 127.0.0.1, 1897, 659, 3779, 200, 0, POST, /HTTPTest/jadehttp.dll,
WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp,
127.0.0.1, -, 9/10/2009, 10:36:56, W3SVC1, WILBUR, 127.0.0.1, 1898, 659, 3779, 200, 0, POST, /HTTPTest/jadehttp.dll,
WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp,
127.0.0.1, -, 9/10/2009, 10:36:56, W3SVC1, WILBUR, 127.0.0.1, 1898, 659, 3779, 200, 0, POST, /HTTPTest/jadehttp.dll,
WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp,
127.0.0.1, -, 9/10/2009, 10:36:56, W3SVC1, WILBUR, 127.0.0.1, 1897, 659, 3779, 200, 0, POST, /HTTPTest/jadehttp.dll,
WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp,
127.0.0.1, -, 9/10/2009, 10:36:56, W3SVC1, WILBUR, 127.0.0.1, 1905, 659, 3779, 200, 0, POST, /HTTPTest/jadehttp.dll,
WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp,
127.0.0.1, -, 9/10/2009, 10:36:57, W3SVC1, WILBUR, 127.0.0.1, 3, 659, 3779, 200, 0, POST, /HTTPTest/jadehttp.dll,
WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp,
127.0.0.1, -, 9/10/2009, 10:36:57, W3SVC1, WILBUR, 127.0.0.1, 43, 652, 745, 500, 0, POST, /HTTPTest/jadehttp.dll,
WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp,
127.0.0.1, -, 9/10/2009, 10:36:57, W3SVC1, WILBUR, 127.0.0.1, 6, 596, 1343, 200, 0, POST, /HTTPTest/jadehttp.dll,
127.0.0.1, -, 9/10/2009, 10:36:57, W3SVC1, WILBUR, 127.0.0.1, 3, 659, 3779, 200, 0, POST, /HTTPTest/jadehttp.dll,
WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp,
127.0.0.1, -, 9/10/2009, 10:36:57, W3SVC1, WILBUR, 127.0.0.1, 2, 594, 904, 200, 0, POST, /HTTPTest/jadehttp.dll,
WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp,
127.0.0.1, -, 9/10/2009, 10:36:57, W3SVC1, WILBUR, 127.0.0.1, 4, 652, 745, 500, 0, POST, /HTTPTest/jadehttp.dll,
WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp,
127.0.0.1, -, 9/10/2009, 10:36:57, W3SVC1, WILBUR, 127.0.0.1, 3, 659, 3779, 200, 0, POST, /HTTPTest/jadehttp.dll,
WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp,
127.0.0.1, -, 9/10/2009, 10:36:57, W3SVC1, WILBUR, 127.0.0.1, 2, 596, 1343, 200, 0, POST, /HTTPTest/jadehttp.dll,
WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp,
127.0.0.1, -, 9/10/2009, 10:36:57, W3SVC1, WILBUR, 127.0.0.1, 2, 594, 904, 200, 0, POST, /HTTPTest/jadehttp.dll,
WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp,

```

The above example is in IIS format. The IIS log file format is a fixed ASCII text-based format, so you cannot customize it.

The IIS log file format records the following data:

- Client IP address
- User name
- Date
- Time
- Service and instance
- Server name
- Server IP address
- Time taken
- Client bytes sent
- Server bytes sent
- Service status code (a value of 200 indicates that the request was fulfilled successfully)
- Windows status code (a value of 0 indicates that the request was fulfilled successfully)
- Request type
- Target of operation
- Parameters (the parameters that are passed to a script)

Not all fields will contain information. For fields for which there is no information, a hyphen (-) is displayed as a placeholder. If a field contains a non-printable character, it is replaced with a plus sign (+) to preserve the log file format. This typically occurs with virus attacks, when, for example, a malicious user sends carriage returns and line feeds that, if not replaced with the plus sign (+), would break the log file format.

Taking a single entry from the above output:

```
127.0.0.1, -, 9/10/2009, 10:36:56, W3SVC1, WILBUR, 127.0.0.1, 1897, 659,
3779, 200, 0, POST, /HTTPTest/jadehttp.dll,
WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName
=WebServiceOverHttpApp,
```

Field	Appears As	Description
Client IP address	127.0.0.1	The IP address of the client.
User name	-	The user is anonymous.
Date	9/10/2009	This log file entry was made on September 10, 2009.
Time	10:36:56	This log file entry was recorded at 10:36 A.M.
Service and instance	W3SVC1	This is a Web site, and the site instance is 1.
Server name	WILBUR	The name of the server.
Server IP	127.0.0.1	The IP address of the server.
Time taken	1897	This action took 1,897 milliseconds.
Client bytes sent	659	The number of bytes sent from the client to the server.
Server bytes sent	3779	The number of bytes sent from the server to the client.
Service status code	200	The request was fulfilled successfully.
Windows status code	0	The request was fulfilled successfully.
Request type	POST	The user issued a POST command.
Target of operation	/HTTPTest/jadehttp.dll	The user wants to connect to jadehttp.
Parameters	WebServiceOver	Parameters passed.

A free tool provided by Microsoft, Log Parser, can be used to analyze the IIS logs. Log Parser is a powerful, versatile tool that provides universal query access to text-based data.

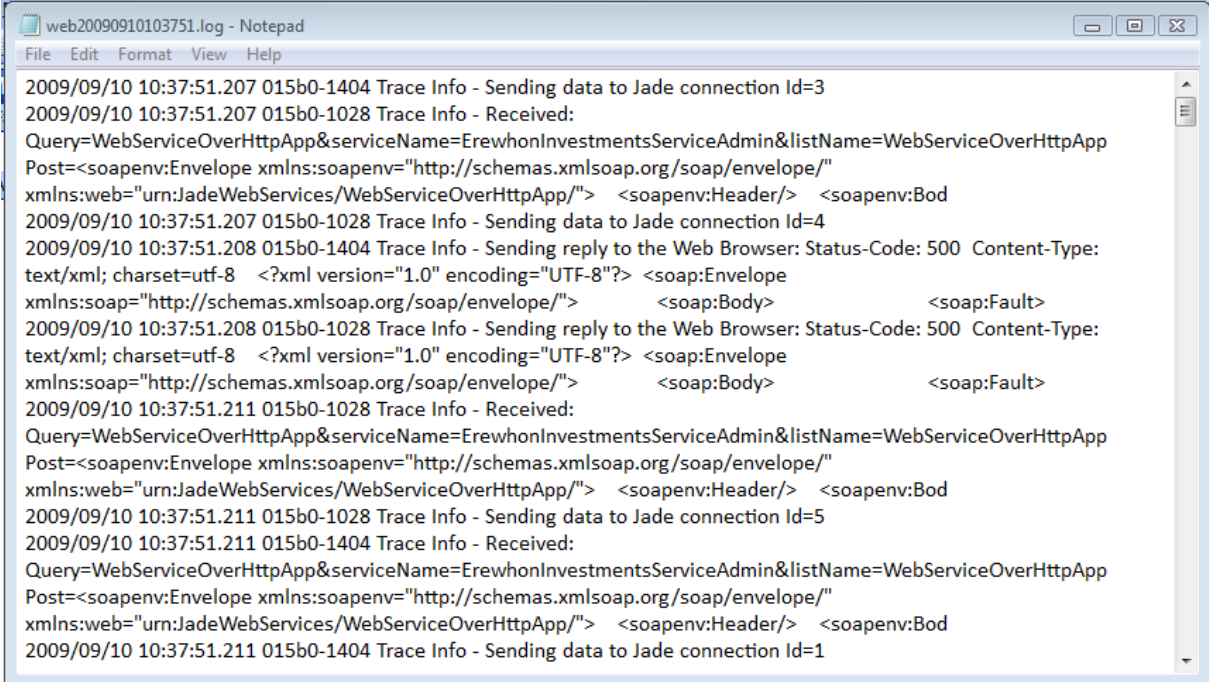
Jadehttp

Tracing can be turned on in **jadehttp**, by a setting in the **jadehttp.ini** file, as follows.

```
[Jadehttp Logging]
trace=true
traceFile=d:\temp\web.log
traceFileSize=1000000
```

These settings specify that tracing is to be turned on, the trace file to use is in **d:\temp\web.log**, and the maximum file size before switching logs is 1,000,000 bytes.

Using these settings, we will get a file with contents similar to the following.



```
web20090910103751.log - Notepad
File Edit Format View Help
2009/09/10 10:37:51.207 015b0-1404 Trace Info - Sending data to Jade connection Id=3
2009/09/10 10:37:51.207 015b0-1028 Trace Info - Received:
Query=WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp
Post=<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:web="urn:JadeWebServices/WebServiceOverHttpApp/"> <soapenv:Header/> <soapenv:Body
2009/09/10 10:37:51.207 015b0-1028 Trace Info - Sending data to Jade connection Id=4
2009/09/10 10:37:51.208 015b0-1404 Trace Info - Sending reply to the Web Browser: Status-Code: 500 Content-Type:
text/xml; charset=utf-8 <?xml version="1.0" encoding="UTF-8"?> <soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"> <soap:Body> <soap:Fault>
2009/09/10 10:37:51.208 015b0-1028 Trace Info - Sending reply to the Web Browser: Status-Code: 500 Content-Type:
text/xml; charset=utf-8 <?xml version="1.0" encoding="UTF-8"?> <soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"> <soap:Body> <soap:Fault>
2009/09/10 10:37:51.211 015b0-1028 Trace Info - Received:
Query=WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp
Post=<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:web="urn:JadeWebServices/WebServiceOverHttpApp/"> <soapenv:Header/> <soapenv:Body
2009/09/10 10:37:51.211 015b0-1028 Trace Info - Sending data to Jade connection Id=5
2009/09/10 10:37:51.211 015b0-1404 Trace Info - Received:
Query=WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp
Post=<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:web="urn:JadeWebServices/WebServiceOverHttpApp/"> <soapenv:Header/> <soapenv:Body
2009/09/10 10:37:51.211 015b0-1404 Trace Info - Sending data to Jade connection Id=1
```

The log will mainly consist of the following three types of messages.

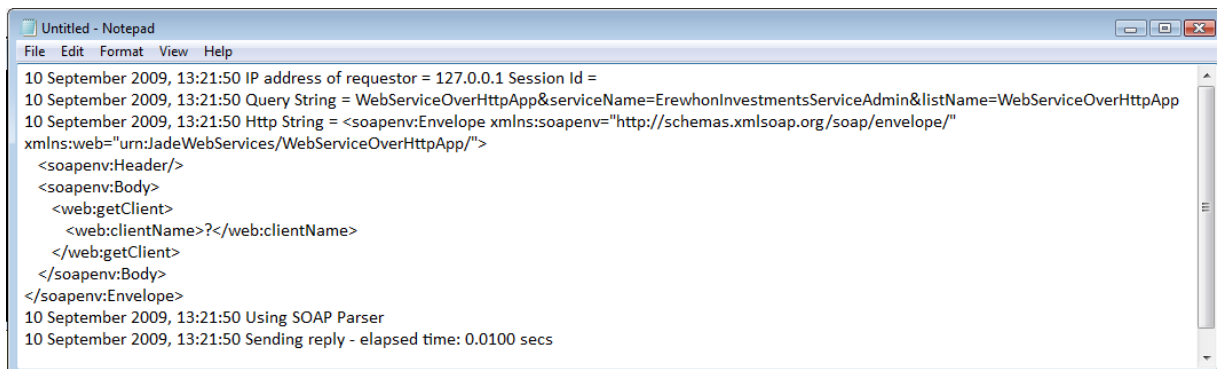
- Sending data to Jade connection Id=
- Received:
- Sending reply to the Web Browser:

Up to 200 characters of the data received and sent will be displayed. The log parser tool can also be used to analyze this data. Note that sensitive data such as passwords can also be logged if they appear in the first 200 characters of the message.

Web Application

The information that is shown in the Web application monitor log when running the application as a GUI application can be captured to a file, either in GUI or non-GUI mode. A configuration file setting, **log_file_name**, allows this output to be captured. Refer to Chapter 3 of the *JADE Web Application Guide (WebApps.pdf)* for details.

A sample output would look like the following.

A screenshot of a Notepad window titled "Untitled - Notepad". The window contains several lines of text representing SOAP message logs and XML content. The logs include timestamps, IP addresses, session IDs, query strings, and HTTP strings. The XML content is a SOAP envelope with a header and a body containing a `getClient` request.

```
10 September 2009, 13:21:50 IP address of requestor = 127.0.0.1 Session Id =
10 September 2009, 13:21:50 Query String = WebServiceOverHttpApp&serviceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp
10 September 2009, 13:21:50 Http String = <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:web="urn:JadeWebServices/WebServiceOverHttpApp/">
  <soapenv:Header/>
  <soapenv:Body>
    <web:getClient>
      <web:clientName?</web:clientName>
    </web:getClient>
  </soapenv:Body>
</soapenv:Envelope>
10 September 2009, 13:21:50 Using SOAP Parser
10 September 2009, 13:21:50 Sending reply - elapsed time: 0.0100 secs
```

The messages captured by the application are as follows.

- IP address of the client.
- Query String – equivalent to the URL on a browser window.
- Http String – body of the POST message. This will have the SOAP request.
- Elapsed time from receiving the message to sending the response.
- Queue Depth – number of requests waiting to be processed.

This information can be analyzed to determine elapsed times, queue depth over time, and so on.

Method Re-implementation

The **JadeWebServiceProvider::processRequest** method and the **JadeWebServiceProvider::reply** method can be re-implemented and the incoming and outgoing messages can be logged, along with other relevant information. In the **processRequest** method, log the **incomingMessage** property as the incoming Web service message and then call **inheritMethod**. In the **reply** method, call **inheritMethod** and log the returned string as the outgoing Web service message.

Similarly, on the consumer side, the **JadeWebServiceConsumer::invoke** method and the **JadeWebServiceConsumer::processReply** method can also be re-implemented to log information. In the **invoke** method, log the **inputMessage** parameter as the outgoing Web service message and then call **inheritMethod**. In the **processReply** method, log the **soapResponse** as the incoming Web service message and then call **inheritMethod**.

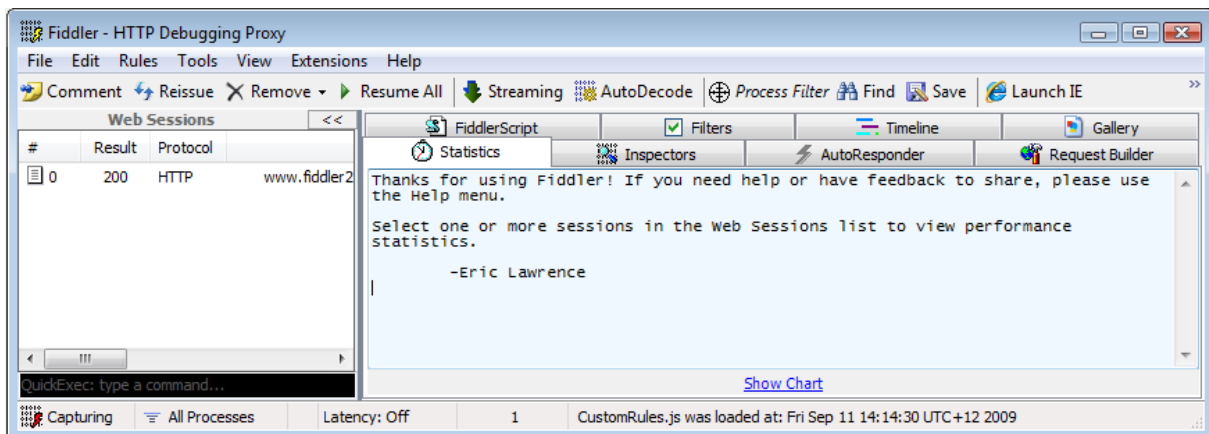
Fiddler

Fiddler is a Web Debugging Proxy that logs all HTTP or HTTPS traffic between your computer and the Internet.

Fiddler allows you to inspect all HTTP or HTTPS traffic, set breakpoints, and "fiddle" with incoming or outgoing data.

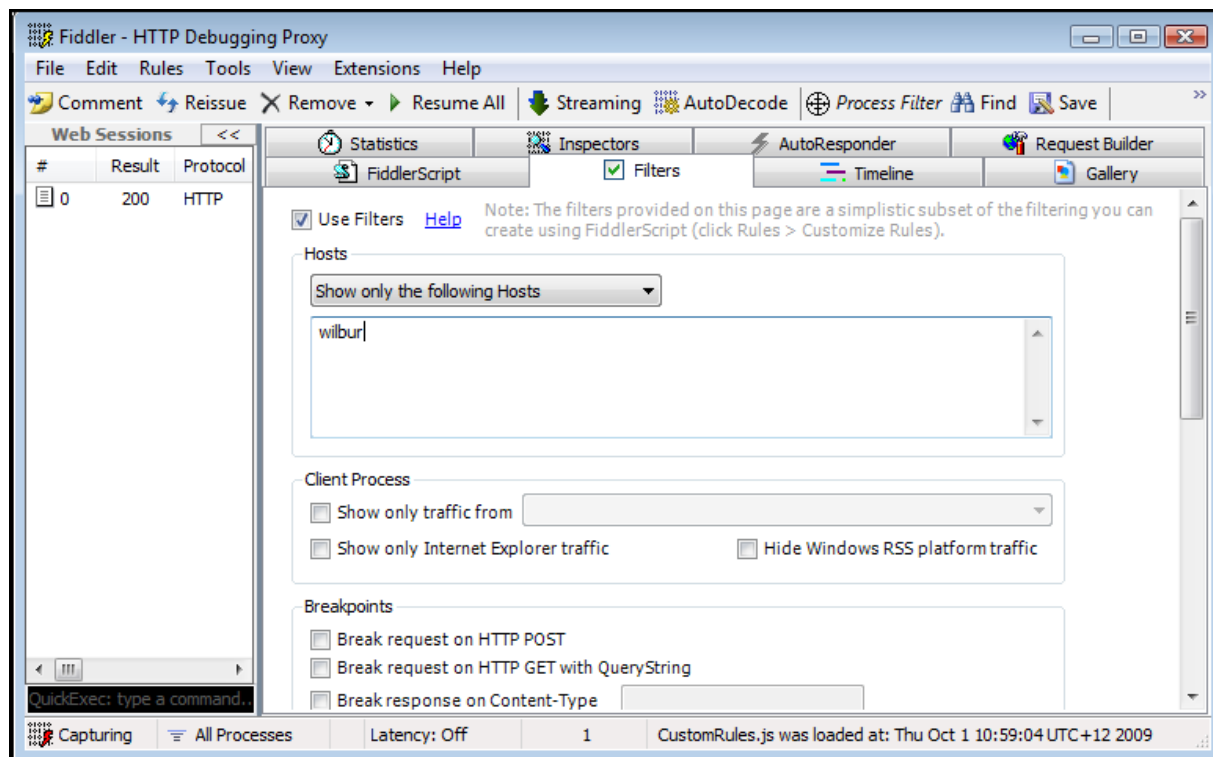
Fiddler includes a powerful event-based scripting subsystem. Fiddler is freeware that can be downloaded from <http://www.fiddler2.com/fiddler2/version.asp>.

The start-up screen for Fiddler looks like the following.



By default, Fiddler will capture all traffic.

To restrict the traffic to that in which you are interested in, you can set up filters. In this example, we are going to restrict traffic to **wilbur**.



We will now select the **Inspectors** tab, to inspect the traffic. In this example, we have set up the Erewhon Web service on host wilbur.

Now when a Web service client calls the Web service for the **getClient** operation on host wilbur, the following information can be observed in Fiddler.

The screenshot shows the Fiddler interface with a SOAP request and response visible. The request is a POST to `http://www.fiddler2.com/crwc/ServiceOverHttpApp/ServiceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp` with a SOAP body containing a `getClient` operation. The response is an HTTP 200 OK with a SOAP body containing a `getClientResponse` with a `getClientResult` of type `Client`. The response body includes details such as address, email, phone, website, and retail sales information.

```

POST /jade/jadehttp.dll?WebServiceOverHttpApp&ServiceName=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp HTTP/1.1
Accept: text/plain, text/html, text/xml
Content-Type: text/xml; charset=utf-8
SOAPAction: "urn:JadeWebServices/WebServiceOverHttpApp/getClient"
User-Agent: Jade/6.3.04
Host: crwcrcs1a
Content-Length: 452
Connection: Keep-Alive
Pragma: no-cache

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:tns="urn:JadeWebServices/WebServiceOverHttpApp/" xmlns:s1="urn:JadeWebServices/WebServiceOverHttpApp/">
  <soap:Body>
    <s1:getClient>
      <s1:clientName>Brian Olsen</s1:clientName>
    </s1:getClient>
  </soap:Body>
</soap:Envelope>

HTTP/1.1 200 OK
Content-Length: 1781
Content-Type: text/xml; charset=utf-8
Server: Microsoft-IIS/7.0
X-Powered-By: ASP.NET
Date: Fri, 11 Sep 2009 02:20:50 GMT

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:tns="urn:JadeWebServices/WebServiceOverHttpApp/">
  <soap:Body>
    <getClientResponse xmlns="urn:JadeWebServices/WebServiceOverHttpApp/">
      <getClientResult xsi:type="Client">
        <address1>2834 The Palace</address1>
        <address2>San Diego</address2>
        <address3>United States of America</address3>
        <email>bo@w01.com</email>
        <fax>64 2 2930 9393</fax>
        <name>Brian Olsen</name>
        <phone>1 2 3848 8384</phone>
        <website>www.w01olsen.com</website>
        <allRetailSales>
          <mySaleItem xsi:type="RetailSaleItem">
            <shortDescription>Jungle Hideaway</shortDescription>
          </mySaleItem>
        </allRetailSales>
      </getClientResult>
    </getClientResponse>
  </soap:Body>
</soap:Envelope>

```

We can see that another entry has been added to the left pane. Selecting this gives us the details of this call. Note that the **Raw** tab is selected for both the request and response in the above example. This will give us the HTTP headers as well as the body of the messages.

This tool has many other features, but this brief example shows how you can view messages from and to a Web service client and it will prove to be an extremely valuable tool for debugging Web services.

Message Sizes

The maximum message size that can be handled by a JADE Web service is about 95M bytes. However, in general, message sizes should be kept small to improve performance and scalability. A review of our customers who use Web services shows that maximum message sizes tend to vary between about 1K bytes through to around 20M bytes.

Generating a large message can take a significant amount of time, depending on whether the message is generated from a large number of small objects or whether the main payload of the message is a string. For example, some applications use the Web service as a container to pass XML strings back and forth. In this case, the performance in generating and parsing the message will be insignificant, but this XML string will likely have to be subsequently parsed to obtain the information it contains.

Note that the strings representing these messages are stored on the object, so you will need to make sure that your transient cache is big enough to hold these messages.

It is, of course, not always possible to control the size of the message; for example, when you are consuming an external Web service where the requirements are dictated by the Web service provider. In this situation where large messages are involved, be prepared to have a big transient cache and possibly slow message generation.

If your transient cache is not large enough to hold the object in the cache, you will get a 1018 (*No memory for buffers*) exception. As discussed above, this will likely relate to the input message or the response message being too large. The input message is held on the **JadeWebServiceConsumer::soapRequest** property and is for the user to inquire upon. If you have no requirement to inspect this property in code, a possible workaround is to write a mapping method that nulls the value out, as shown in the following example.

```
soapRequest(set : Boolean; _value : String io) mapping, updating;
vars

begin
    if set then
        _value := null;
    endif;
end;
```

You could log other information, check its length, or even truncate the string in this method. Note that if you are using the statistics logging provided by the framework, the soap request value will be whatever value it was set to in this mapping method.

This technique *cannot* be applied to the response message that is stored in the **JadeWebServiceConsumer::soapResponse** property. This means that if the response messages from the Web service can be large, you will still need to increase your transient cache size.

Web Server Setup

The JADE Web applications framework currently supports the following Web servers.

- IIS on Windows
- Apache on Windows
- Apache on Linux

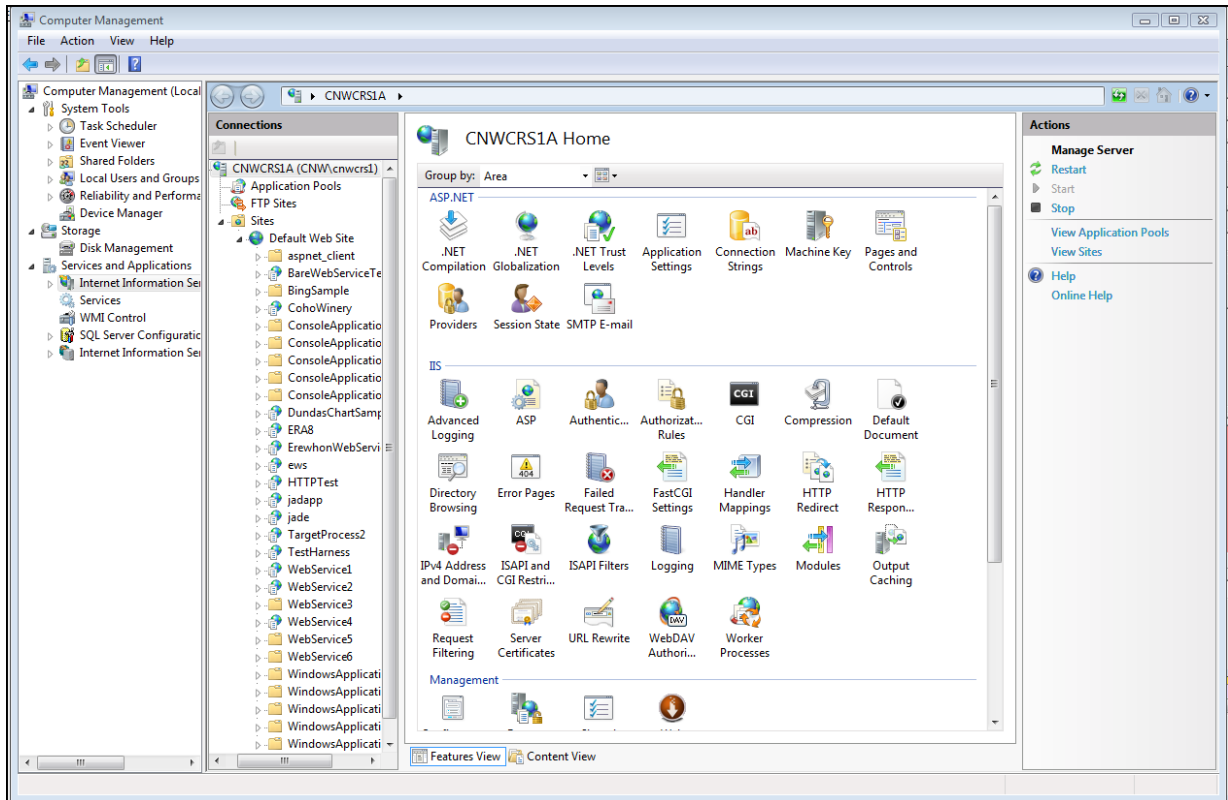
Configuring the Web servers to work with JADE Web applications used to be relatively straight-forward, but this is no longer the case. Certainly the different flavors of IIS have caused difficulty in attempting to define a single way of setting up the Web server. This section will endeavor to explain how to set up IIS 7.0 (available on Windows Vista and Windows 2008 systems).

The following example Web server setup assumes the following.

- IIS 7 is set up on your machine
- Use of Microsoft Management Console (MMC) is assumed

You can usually bring up MMC by right-clicking on **MyComputer** and then selecting **Manage**. If you are not in Features view, select the **Features View** tab.

Clicking on **Internet Information Services** under Services and Applications at the left shows you the display in the panes at the right. This will bring up the following window.



For the Web server to work with JADE Web applications, the following steps are required to be set up in IIS.

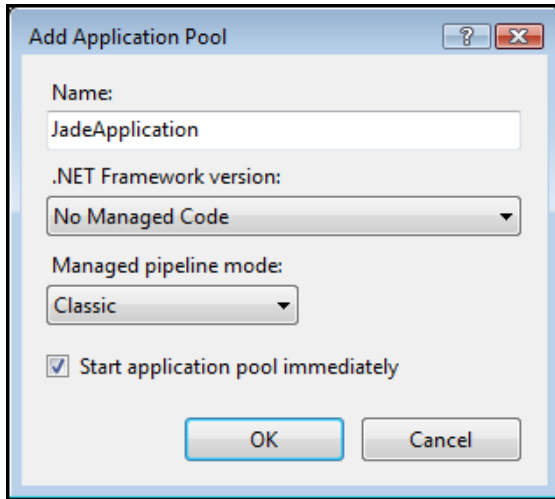
1. Create an application pool
2. Create a virtual directory
3. Set up the Handler Mappings
4. Set up ISAPI restrictions

Create an Application Pool

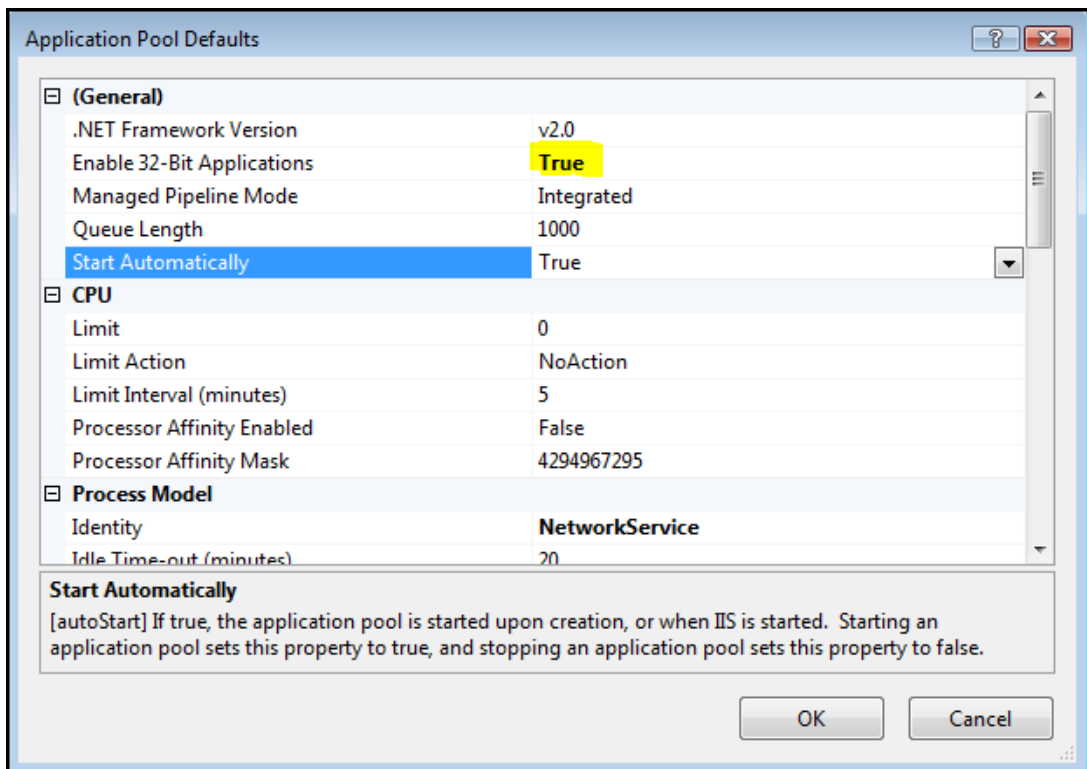
In the Computer Management window, click on **Application Pools** on the left (you might need to expand the list to see this) and then select **Add Application Pool** on the right.

This will bring up the Add Application Pool dialog.

Fill in the entries shown in the following dialog and then click **OK**.



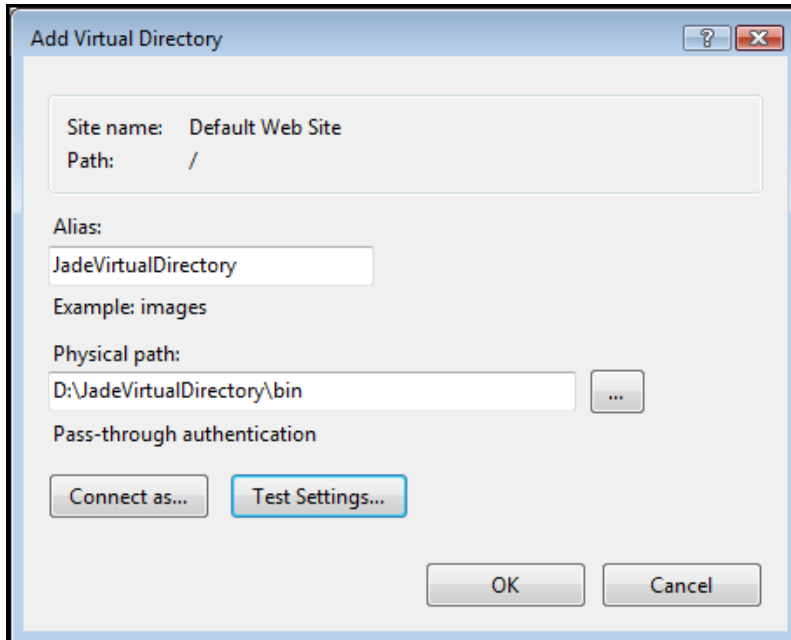
Now select **Set Application Pool Defaults** on the right of the Application Pool Defaults dialog, set Enable 32-Bit Applications to **True** if you are using 32-bit JADE, and then click **OK**.



Create a Virtual Directory

In the Computer Management window, click on **Default Web Site** on the left, right-click, and then select **Add Virtual Directory**.

Fill in the entries shown in the following dialog and then click **OK**.

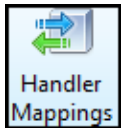


Make sure that the folder specified in the **Physical path** text box is present, or you can create it by using the ... button.

This directory must contain the **jadehttp.dll**.

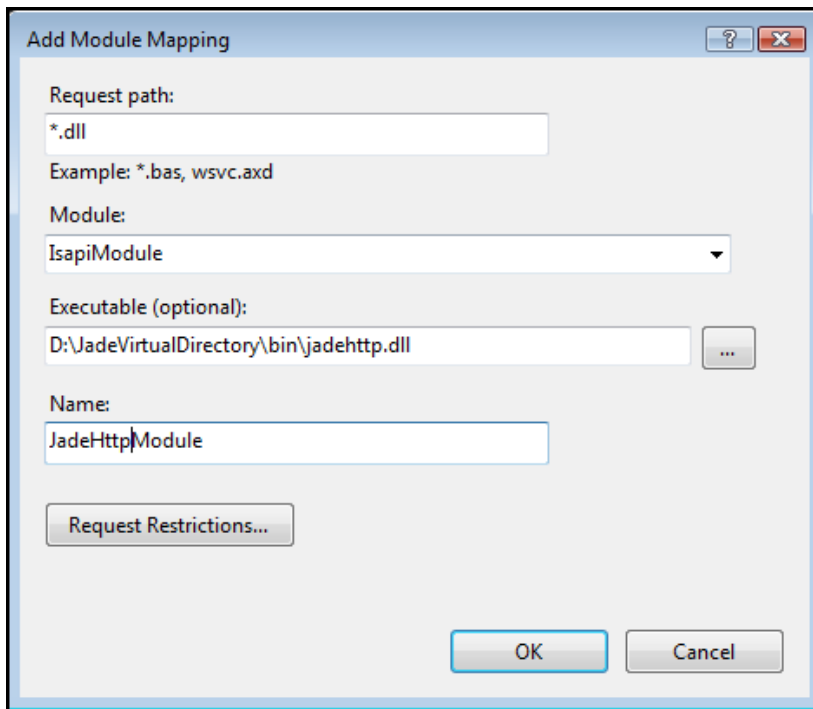
Set Up a Handler Mapping

In the Computer Management window, click on the virtual directory that you created in the left pane and then double-click on the **Handler Mappings** icon in the middle pane.

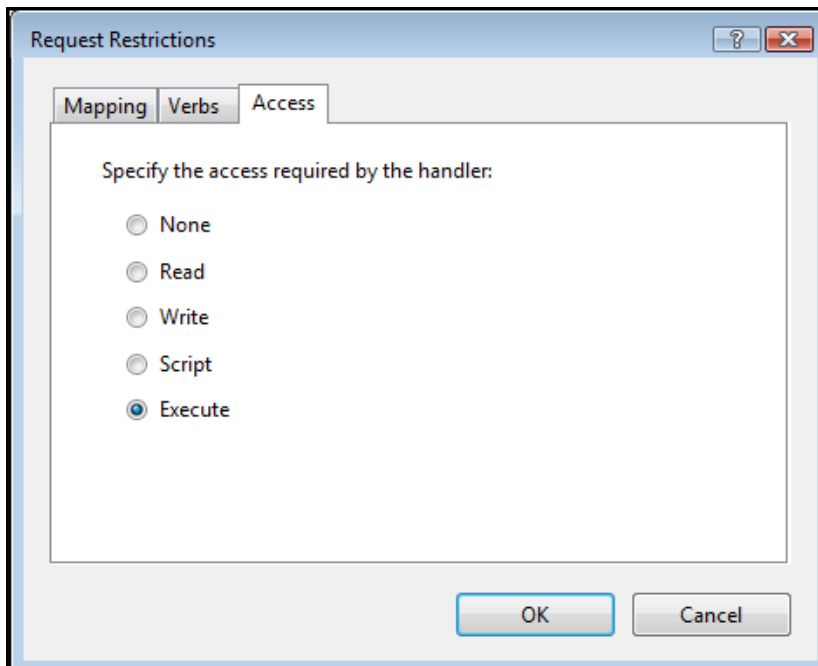


This will bring up a list of handler mappings. We need to add a new one for **jadehttp**.

Click on **Add Module Mapping** in the right pane and enter the following information in the dialog that is brought up.

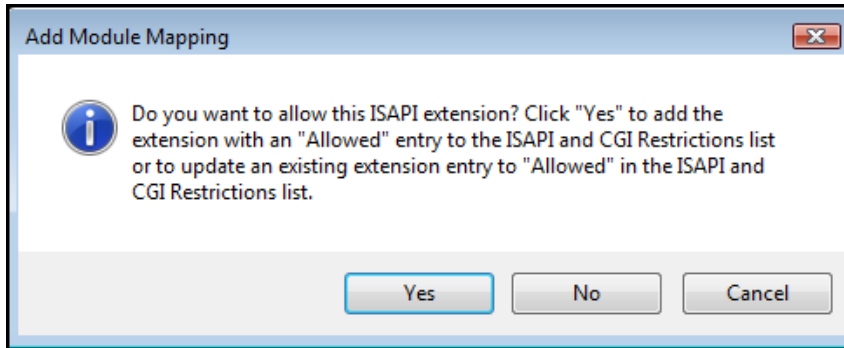


Click the **Request Restrictions** button, select **Execute** on the **Access** sheet of the Request Restrictions dialog, and then click **OK**.



Now click **OK** on the Add Module Mapping dialog.

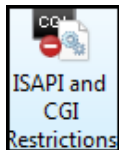
The following Add Module Mapping message box should be brought up.



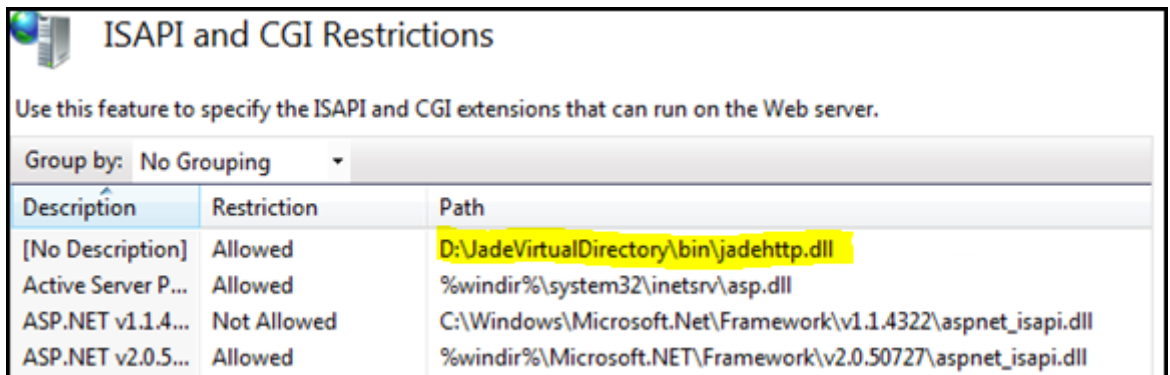
Click **Yes**.

Set Up ISAPI Restrictions

In the Computer Management window, click on the machine name (which is at the top of the tree) in the left pane and then double-click on the **ISAPI and CGI Restrictions** icon in the middle pane.

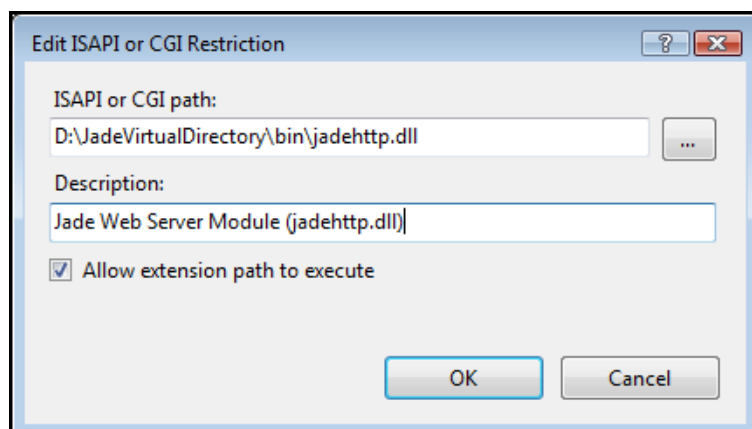


This will show a list of ISAPI and CGI extensions, including the one you created in the previous section when you clicked **Yes** on the Add Module Mapping message box; that is, the one with [No Description] in the **Description** column.

A screenshot of the 'ISAPI and CGI Restrictions' window. The title bar reads 'ISAPI and CGI Restrictions'. Below the title bar is a subtitle: 'Use this feature to specify the ISAPI and CGI extensions that can run on the Web server.' There is a 'Group by:' dropdown menu set to 'No Grouping'. Below this is a table with three columns: 'Description', 'Restriction', and 'Path'. The first row is highlighted in yellow.

Description	Restriction	Path
[No Description]	Allowed	D:\JadeVirtualDirectory\bin\jadehttp.dll
Active Server P...	Allowed	%windir%\system32\inetsrv\asp.dll
ASP.NET v1.1.4...	Not Allowed	C:\Windows\Microsoft.Net\Framework\v1.1.4322\aspnet_isapi.dll
ASP.NET v2.0.5...	Allowed	%windir%\Microsoft.NET\Framework\v2.0.50727\aspnet_isapi.dll

Double-click on this entry and then enter a suitable description in the Edit ISAPI or CGI Restriction dialog that is brought up.



Make sure that the **Allow extension path to execute** check box is checked and then click **OK**.

If you need to specify another virtual directory or physical directory, follow the same steps as above.

Usage Statistics

Web Service Client

A JADE Web service client can access information related to the last Web service call. As there is a performance overhead in gathering statistics, this information is gathered only when you specifically request it. When the **JadeWebServiceConsumer::logStatistics** property is set to **true**, this information is gathered.

At the end of a Web service call, you can request this information by calling the **JadeWebServiceConsumer::getLastStatistics** method. This method takes a Boolean parameter that indicates whether you want only the statistical information (**true**) or all information (**false**). Setting this parameter to **true** will return an XML string similar to the following.

```
<?xml version="1.0" encoding="utf-8"?>
<WebServiceStatistics>
  <name>ErewhonInvestmentsServiceAdmin</name>
  <operation>getClient</operation>
  <url>http://wilbur/jade/jadehttp.dll?WebServiceOverHttpApp&service
Name=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp
</url>
  <dateTime>11 September 2009, 14:44:08</dateTime>
  <requestTime>5</requestTime>
  <getResponseTime>692</getResponseTime>
  <responseTime>21</responseTime>
  <processingTime>26</processingTime>
  <errorCode>0</errorCode>
  <requestSize>452</requestSize>
  <responseSize>1781</responseSize>
</WebServiceStatistics>
```

Setting the parameter to **false** will return an XML string similar to the following.

```
<?xml version="1.0" encoding="utf-8"?>
<WebServiceStatistics>
  <name>ErewhonInvestmentsServiceAdmin</name>
  <operation>getClient</operation>
  <url>http://wilbur/jade/jadehttp.dll?WebServiceOverHttpApp&service
Name=ErewhonInvestmentsServiceAdmin&listName=WebServiceOverHttpApp
</url>
  <dateTime>11 September 2009, 14:44:40</dateTime>
  <requestTime>5</requestTime>
  <getResponseTime>29</getResponseTime>
  <responseTime>16</responseTime>
  <processingTime>21</processingTime>
  <errorCode>0</errorCode>
  <requestSize>452</requestSize>
  <responseSize>1781</responseSize>
  <requestHeaders><![CDATA[Accept: text/plain
Accept: text/html
Accept: text/xml
Content-Type: text/xml; charset=utf-8
Host: wilbur
Pragma: no-cache
Proxy-Connection: Keep-Alive
SOAPAction: "urn:JadeWebServices/WebServiceOverHttpApp/getClient"
User-Agent: Jade/6.3.04
]]></requestHeaders>
  <soapRequest><![CDATA[<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:tns="urn:JadeWebServices/WebServiceOverHttpApp/"
xmlns:s1="urn:JadeWebServices/WebServiceOverHttpApp/">
  <soap:Body>
    <s1:getClient>
      <s1:clientName>Brian Olsen</s1:clientName>
    </s1:getClient>
  </soap:Body>
</soap:Envelope>
]]></soapRequest>
  <responseHeaders><![CDATA[HTTP/1.1 200 OK
Content-Length: 1781
Content-Type: text/xml; charset=utf-8
Server: Microsoft-IIS/7.0
X-Powered-By: ASP.NET
Date: Fri, 11 Sep 2009 02:44:40 GMT
]]></responseHeaders>
  <soapResponse><![CDATA[<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getClientResponse
xmlns="urn:JadeWebServices/WebServiceOverHttpApp/">
      <getClientResult xsi:type="Client">
        <address1>2834 The Palace</address1>
        <address2>San Diego</address2>
        <address3>United States of America</address3>
        <email>bo@wol.com</email>
```

```

<fax>64 2 2930 9393</fax>
<name>Brian Olsen</name>
<phone>1 2 3848 8384</phone>
<webSite>www.wol/olsen.com</webSite>
<allRetailSales>
<RetailSale>
  <mySaleItem xsi:type="RetailSaleItem">
    <shortDescription>Jungle Hideaway</shortDescription>
  </mySaleItem>
  <price>6250000.00</price>
</RetailSale>
<RetailSale>
  <mySaleItem xsi:type="RetailSaleItem">
    <shortDescription>Wedding Ring</shortDescription>
  </mySaleItem>
  <price>8399.00</price>
</RetailSale>
</allRetailSales>
<allTenderSales>
<TenderSale>
  <mySaleItem xsi:type="TenderSaleItem">
    <shortDescription>Coffee Mill</shortDescription>
  </mySaleItem>
  <myTender>
    <offer>1305.00</offer>
    <timeStamp>1999-11-17T00:55:00.000-00:00</timeStamp>
  </myTender>
</TenderSale>
<TenderSale>
  <mySaleItem xsi:type="TenderSaleItem">
    <shortDescription>Oil Painting</shortDescription>
  </mySaleItem>
  <myTender>
    <offer>7732.00</offer>
    <timeStamp>2000-02-05T19:25:00.000-00:00</timeStamp>
  </myTender>
</TenderSale>
</allTenderSales>
</getClientResult>
</getClientResponse>
</soap:Body>
</soap:Envelope>

```

This call returns the HTTP headers and the request and response message. As the messages can be large, if you want only to capture statistical information, set the parameter to **true**. The following is a description of the tags.

Field	Appears As	Description
name	ErewhonInvestmentsServiceAdmin	The name of the Web service
operation	getClient	The name of the Web service operation
url	http://wilbur/jade/ ...	The URL of the request
dateTime	11 September 2009, 14:44:08	The date and time of the request
requestTime	5	Time to process and send the request

Field	Appears As	Description
getResponseTime	692	Time taken by the Web service call
response	21	Time taken to process the response
errorCode	0	Error code of error when sending/receiving
requestSize	452	Size of request message
responseSize	1781	Size of response message
requestHeaders	![CDATA[Accept: text/plain ...	HTTP Send headers
soapRequest	![CDATA[<?xml version="1.0" encoding="utf-8"?>	SOAP request message
responseHeaders	<![CDATA[HTTP/1.1 200 OK	HTTP response headers
soapResponse	<![CDATA[<?xml version="1.0" encoding="utf-8"?>	SOAP response message

Web Service Application

In the current release, there is no programmatic interface to obtain statistical information relating to a Web service application. Such information can be obtained by logging the Web application monitor output. The JADE Monitor has information relating to the Web service processes. An example of this is shown in the following diagram.

The screenshot shows the JADE Monitor application window titled "JADE Monitor (d:\jadesystems\jade6304\system : cnwcrs1) - [Web Performance]". The main area displays "Web Performance" data for the process "Node - CNWCRS1A1 [pid=5940] < server >".

Metric	Value	Delta	Rate
Node - CNWCRS1A1 [pid=5940] < server > :: Process - {17}			
Maximum Response Time	33	0	0
Minimum Response Time	1	0	0
Total Requests	651	108	177
Total Response Time	1,030	131	218
Rejected Requests	0	0	0
Queue Depth	3	2	3
Total Connections	5	0	0
Assigned Connections	0	0	0
Total Workers	5	0	0
Idle Workers	1	0	0
Node - CNWCRS1A1 [pid=5940] < server > :: Process - {18}			

Below the table is an "Overview" section showing a SOAP envelope: </soap:Envelope>. There are tabs for "Transformer", "Headers", "TextView", "SyntaxView", "ImageView", "HexView", "WebView", "JSON", "Auth", "Caching", and "Privacy".

The Web service information that is monitored is listed in the following table.

Row Name	Description
Maximum Response Time	Maximum time in ms. for a message response
Minimum Response Time	Fastest response time in ms. for a message response
Total Requests	Total requests
Total Response Time	Total response time
Rejected Requests	Number of requests rejected (no available connections)
Queue Depth	Number of requests in the queue
Total Connections	Number of total connections
Assigned Connections	Number of connections currently in use
Total Workers	Number of application copies
Idle Workers	Number of applications that are idle

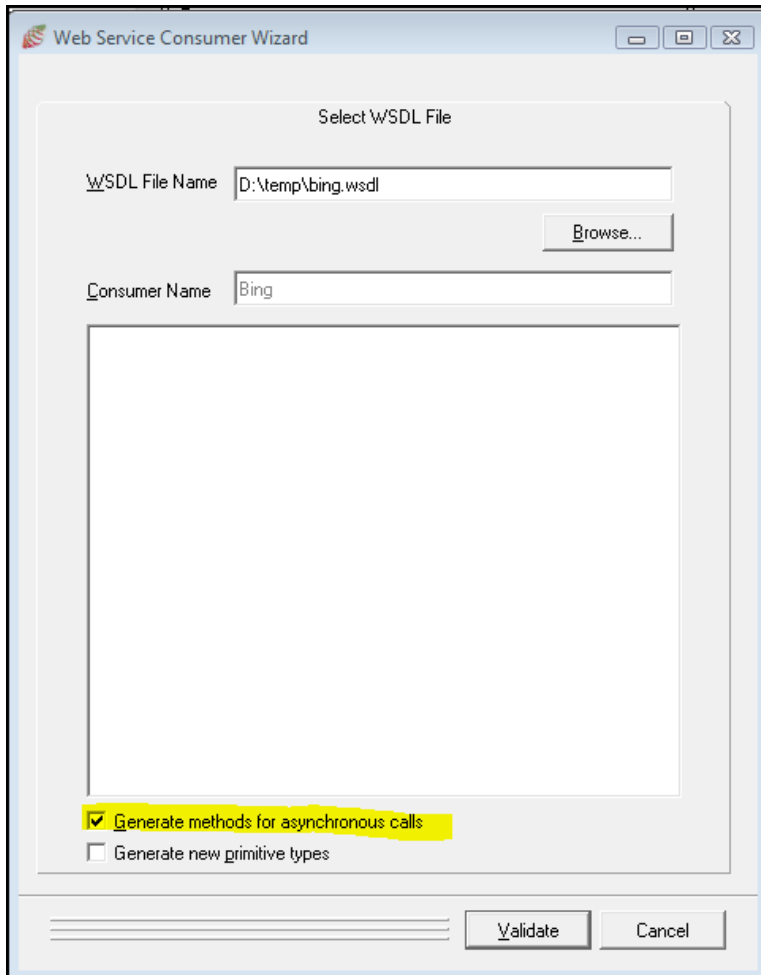
Consumer Asynchronous Calls

The steps involved in setting up a Web service to make asynchronous calls are as follows.

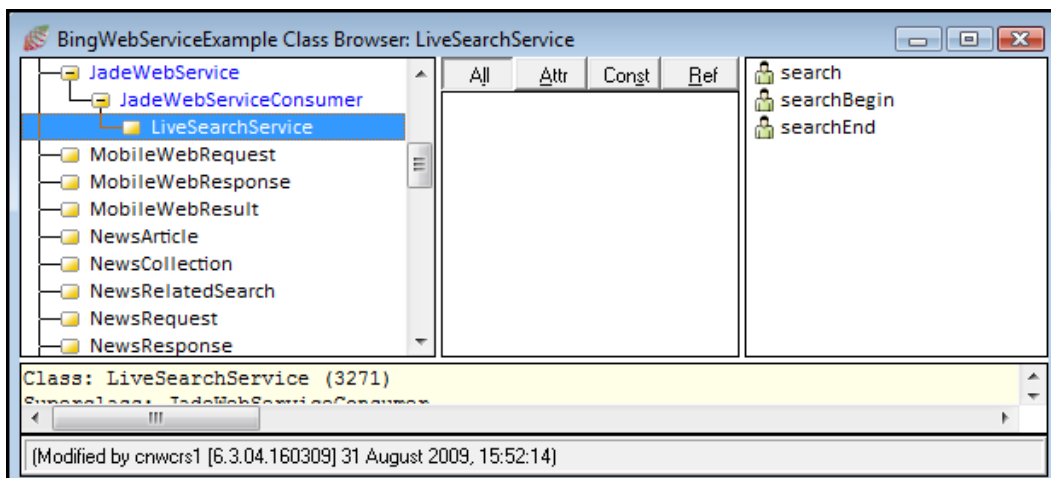
1. WSDL Import
2. Set up a worker application
3. Write code to handle asynchronous calls

WSDL Import

To generate the code for making asynchronous calls, check the **Generate methods for asynchronous calls** check box on the Web Service Consumer Wizard dialog, shown in the following diagram.



When you have done this, two additional methods will be generated for each Web service message defined in the WSDL, as shown in the following diagram.



The two additional methods have a **Begin** and **End** suffix. In this example, the **searchBegin** method is used to start the request and the **searchEnd** method is used to receive the response.

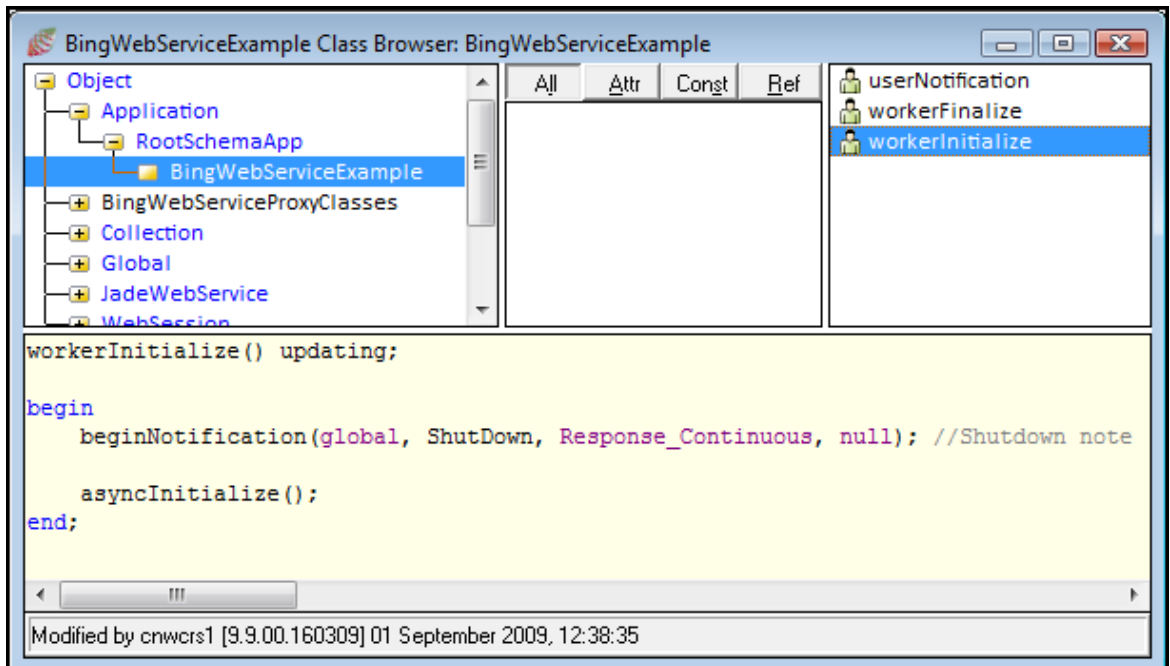
Setting up a Worker Application

To use asynchronous calls, set up a worker application that sends the request to the Web service.

In its simplest form, the worker application can be a non-GUI application that is required to only implement the **initialize** and **finalize** methods.

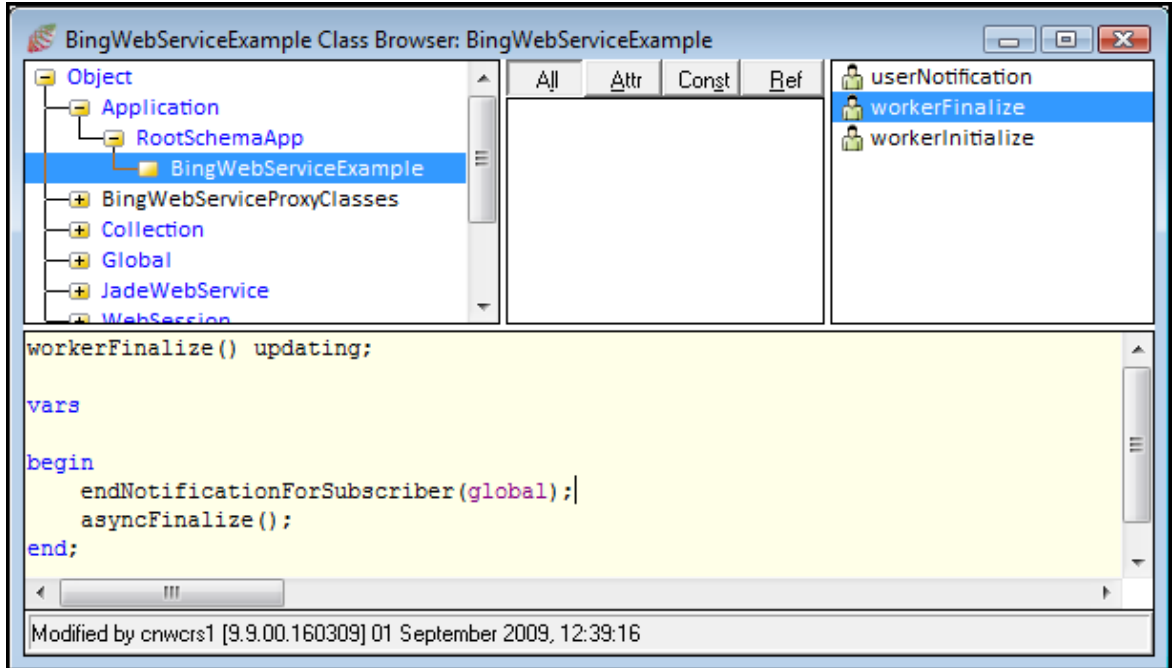
The initialize method must call the **Application::asyncInitialize** method and the **finalize** method must call the **Application::asyncFinalize** method.

In the supplied example schema, the **initialize** method is called **workerInitialize** and is implemented as follows.



The notification is so that the application can be shut down using a **causeEvent** call.

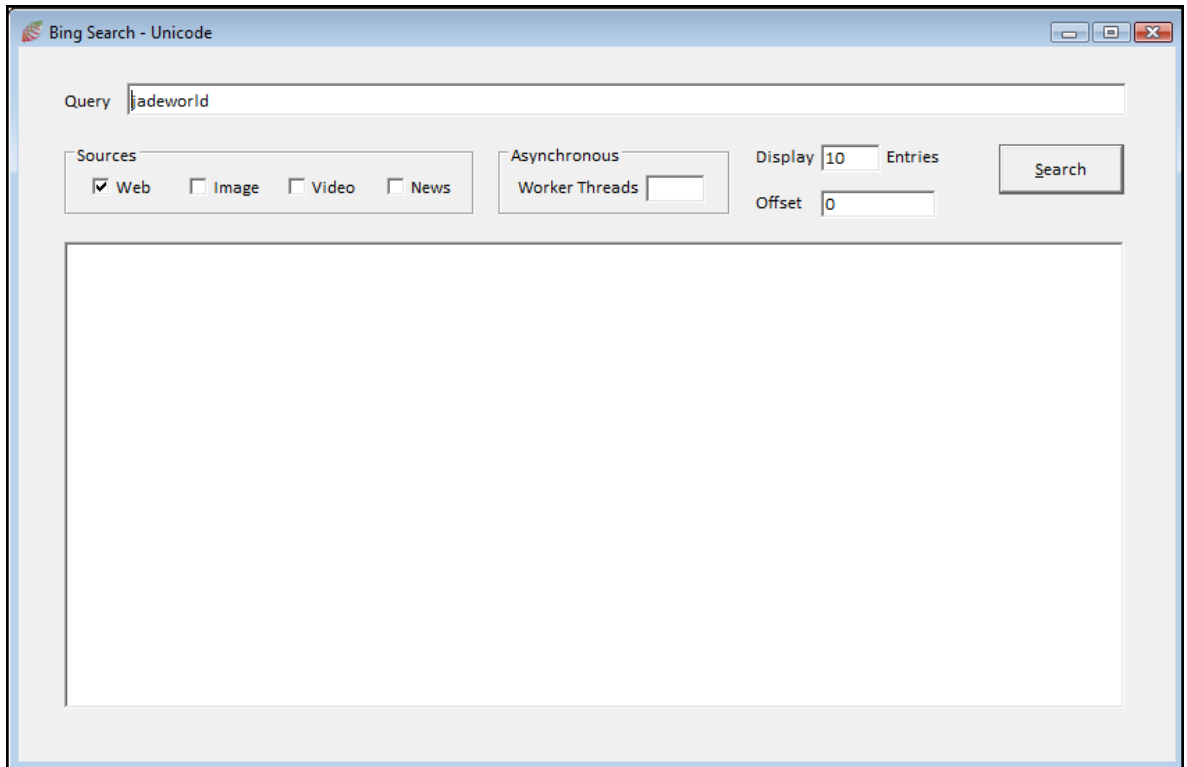
The **finalize** method is called **workerFinalize** and is implemented as follows.



The **userNotification** terminates the application when it receives a shutdown notification. We create a worker non-GUI application called **WebServiceWorkerApp**.

Coding Example

The sample application has the following (only) start up form.



This application has been set up to call the Web services synchronously or asynchronously. We will discuss only the asynchronous operation here. The Bing Search Web service provides several sources on which the search can occur. In this example, we are using four of these – Web, image, video, and news – and allowing one or more of these to be selected for the search. The form also allows you to enter the number of worker processes to start. If the current number of worker processes is less than this number, additional worker processes will be started, up to the specified number.

To demonstrate the benefit and use of asynchronous operations, we will set up four workers, one for each of the sources, so that if all four sources are selected, all four search requests can be done asynchronously. Note that the Bing API actually lets you specify all of the sources in a single call. However, we are not doing so in this example, so that we can demonstrate the use of asynchronous calls.

The **Form** class that we are using is called **BingSearch**. In this class, we define a **webService** property, to hold a reference to the Web service consumer instance (an instance of **LiveSearchService**). This is initialized in the **load** method of the **BingSearch** class.

The following discussion concentrates on a method called **runAsync** on the **BingSearch** class.

1. Tell the consumer instance the name of the worker application, as follows.

```
webService.workerApp := "WebServiceWorkerApp";
```

2. Start the required number of copies of the worker applications, as follows.

```
vars
  c: Integer;
begin
  // get a count of apps already running
  c := getAppCount(webService.workerApp);

  while c < txtWorkers.text.Integer do
    c := c + 1;
    app.startApplicationWithParameter(currentSchema.name,
      webService.workerApp, null);
  endwhile;
end;
```

3. We now need to set up the parameters to the Web service call, as follows (assuming that Web source is selected).

```
create searchRequestWeb;
create searchRequestWebParam;
searchRequestWeb.parameters := searchRequestWebParam;
searchRequestWebParam.appId := appID;
searchRequestWebParam.query := txtQuery.text;
searchRequestWebParam.sources.add("Web");
create webrequest transient;
searchRequestWebParam.web := webrequest;
webrequest.count := txtEntries.text.Integer;
webrequest.offset := txtOffset.text.Integer;
```

4. We now set up the required parameters for each source on which we want to search and then we initiate the asynchronous call by calling the **searchBegin** method on the Web service, passing the required parameter, as follows.

```
contextWeb := webService.searchBegin(searchRequestWeb);
objArray.add(contextWeb);
```

The **objArray** is local variable, defined as follows.

```
objArray: JadeMethodContextArray;
```

Note **JadeMethodContextArray** is defined in this schema as a subclass of **ObjectArray** with membership **JadeMethodContext**. The variable is used as the parameter to the **processForMethods** method call, as we will see later.

5. We write similar code for each of the other sources. When all of the required asynchronous calls have been initiated, we then do the following.

```
context := process.waitForMethods(objArray);
```

Tip The **waitForMethods** method takes a variable number of parameters that must be of type **JadeMethodContext** or an array of **JadeMethodContext**, and where this number can vary, it is easier to code using the array.

This method will now wait for one of the calls to complete. When a call is completed, the context variable will contain the **JadeMethodContext** of the call that was completed. The following code snippet assumes that the context returned was for the *web* source.

```
response := webService.searchEnd(searchRequestWeb, contextWeb);
webresponse := response.parameters.web;
text := null;
if webresponse <> null then
    staStatusLine.caption := staStatusLine.caption & "Web = " &
        webresponse.total.String & " ";
    jrtResults.setCharacterFormat(true, jrtResults.fontName,
        jrtResults.fontSize, Red, 0, 0, 0);
    foreach webresult in webresponse.results do
        text := text & webresult.title & CrLf & webresult.description &
            CrLf & webresult.url & CrLf & CrLf;
    endforeach;
    jrtResults.append(text);
endif;
```

The **searchEnd** method is now called on the Web service, to get the response object. The rest of the code in this snippet processes this response object.

6. We now need to call **waitForMethods** again, as follows.

```
context := process.waitForMethods(objArray);
```

When the context that is returned is null, all requests have been completed. The sample application is supplied with this white paper.

For more information, see the *Asynchronous Method Calls* white paper at <http://www.jadeworld.com/jade/whitepapers.htm>.

Note By default, the Web services framework allows you to have only two consecutive connections open (as defined by the HTTP 1.1 specification, or four if you are using HTTP 1.0). To increase the maximum number of allowed connections, you will need to change the Web service client's XML configuration file to a higher number. For details about configuring the Web service consumer, see Chapter 3 of the *JADE Web Application Guide (WebApps.pdf)*.

Using SOAP Headers

Even if you have done some fairly involved development of a JADE Web service, there is a good possibility that you have never bothered with SOAP headers. In fact, it is not uncommon to put information in the body of your SOAP message that really should be in the header section of your message.

We will look at what sort of information should go into the header, how you can read and write message headers in the JADE Web services framework, and how you can add to the current SOAP infrastructure by using SOAP headers.

The SOAP Header Element

The specifications for a SOAP header element differ slightly between SOAP 1.1 and SOAP 1.2. The following discussion is based on SOAP 1.1.

A SOAP message consists of three elements: the top level **Envelope** element and two of its children (the **Header** element and the **Body** element). The following is a SOAP message with all three of these elements.

```
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:tns="urn:JadeWebServices/WebServiceOverHttpApp/"
xmlns:s1="urn:JadeWebServices/WebServiceOverHttpApp/">
<soap:Header>
  <s1:erewhonHeader>
    <s1:priority>HIGH</s1:priority>
  </s1:erewhonHeader>
</soap:Header>
<soap:Body>
  <s1:getClient>
    <s1:clientName>Brian Olsen</s1:clientName>
  </s1:getClient>
</soap:Body>
</soap:Envelope>
```

The **Body** element is where the main data in the message lives. The **Header** element is where any metadata that might describe the body, details of how the body should be processed, or simply extra information about the message can live. The **Header** element is optional, but if it exists, it must be the first child of the **Envelope** element. The **Header** element itself consists of zero or more child elements, referred to as *header blocks*. Each header block needs to be namespace-qualified.

The SOAP 1.1 specifications define three attributes that can apply to header blocks: the **encodingStyle** attribute, the **actor** attribute, and the **mustUnderstand** attribute. All three of these attributes are optional and they can be used in addition to any other attributes that you may want to include.

The **encodingStyle** attribute is used to indicate how the encapsulated data is encoded. The SOAP specification includes a mechanism for encoding data that includes data type information in XML attributes. This is becoming less popular as more and more people define their headers using XML Schema.

The **actor** attribute is used to indicate which node should process this particular header block. A SOAP message can be passed through a sequence of nodes, and it is conceivable that a header block may apply to one node in the sequence and not to others. You might set the **actor** attribute to the endpoint of the node, which will process the header block so that other nodes in the sequence will know to ignore it. The absence of the **actor** attribute implies that the header block is targeted for the ultimate recipient of the SOAP message. Note that the **actor** attribute has been renamed the **role** attribute in the SOAP 1.2 specification.

The **mustUnderstand** attribute is the way that a header block indicates that it must be understood and processed in accordance with any specifications that may be defined for the specified qualified element name. If the recipient does not know how to process the header block, it must generate a MustUnderstand fault and not proceed with any further processing of the message. The **mustUnderstand** attribute is a Boolean value, and if it is not present, it is assumed to be **false**.

Information to Put in the Header

The SOAP specifications are fairly ambiguous concerning what information goes in the header. Those familiar with HTTP or MIME headers are probably used to seeing various sorts of metadata included with the main data in the message.

The focus of the SOAP header should be to help process the data in the body. It makes sense to include information about authentication or transactions, because this information will be involved in identifying the person or company who sent the body and in what context it will be processed. Expiration data could be included in the header to indicate when the data in the body may need to be refreshed. User account information could be included, in order to ensure that processing the message is performed only for a request that has been legitimately paid for.

Here's another factor in determining whether information should be included in SOAP headers.

- Will that information have broad application to a wide variety of SOAP messages?

If so, include it in the header. It makes more sense to define a single schema and insert it into the definition of one header element than to force inclusion of the same data into the body schemas of a large number of message definitions. For example, if you are defining several Web service methods that use common information, it may make sense to put these in the header rather than pass it in as a parameter. Authentication and routing are problems common to many XML Web services, so it makes sense that these specifications deal with information that lives in the Header element.

The stateless nature of Web services means that if you require state, SOAP headers can be used to relay this information. Within the JADE Web services framework, SOAP headers are used to relay session information when session handling is enabled.

The following SOAP header is generated when session handling is enabled.

```
<soap:Header>
  <JadeSessionHeader
xmlns="urn:JadeWebServices/WebServiceOverHttpApp/">
  <sessionId>91b17de375a04d9b</sessionId>
  </JadeSessionHeader>
</soap:Header>
```

Defining SOAP Headers

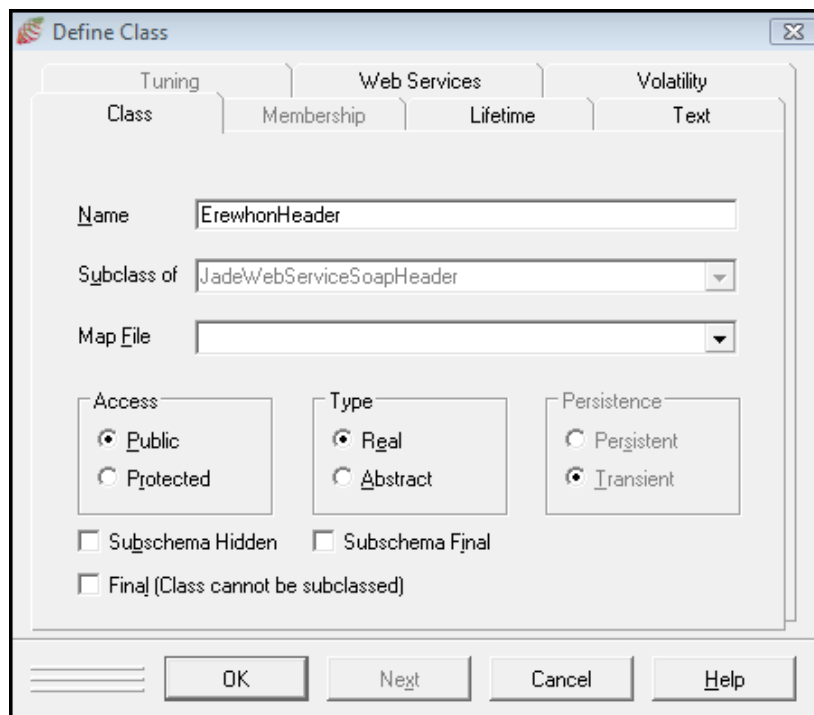
SOAP headers are defined by the Web service provider application. There are four simple steps to follow when defining these headers:

1. Create a subclass of the **JadeWebServiceSoapHeader** class.
2. Add properties to this class that you want to send in the SOAP header.
3. Add a property reference to this class in your Web service provider class.
4. Select the methods for which you want to generate the SOAP header.

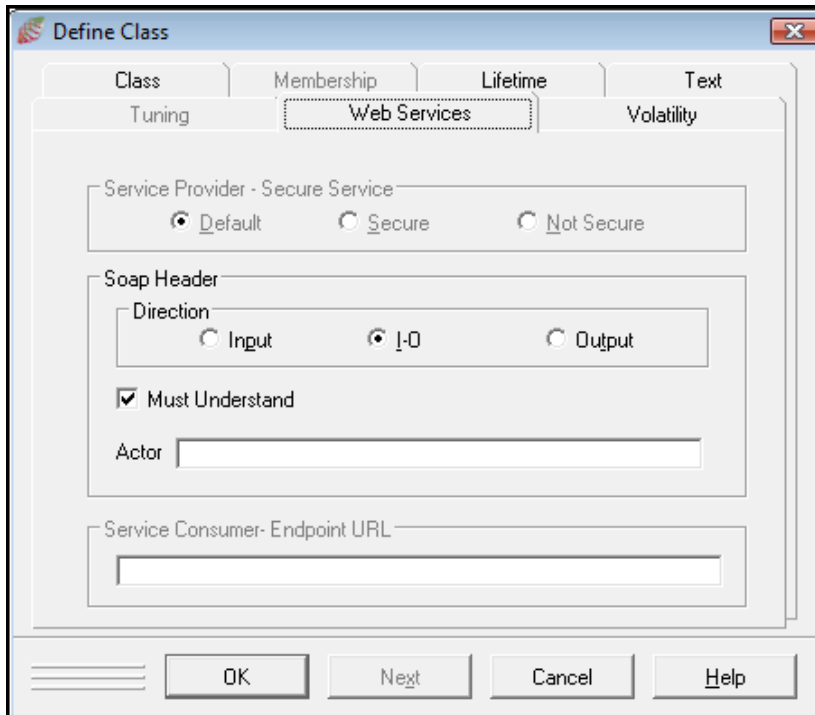
The following simple example demonstrates how to create and use a SOAP header in JADE. The example uses the Erewhon Investments sample system.

Step 1: Create a subclass of the JadeWebServiceSoapHeader class

We will call this class **ErewhonHeader**. The class can be defined as shown in the following diagram.



We will define the options for the header on the **Web Services** sheet, as shown in the following diagram.



We have defined the SOAP header direction as I-O (the default is Input). The following meanings apply to this setting.

- Input – This means that the SOAP header is provided by the Web service client and input to the Web service. The Web service itself does not send a SOAP header
- I-O – Both the Web service client and the Web service send this header.
- Output – Only the Web service sends this header. The Web service client receives this header but does not send it back.

The **Must Understand** attribute is set to **true**. (Its usage was explained in the previous section.)

Step 2: Add properties to the class

We will add the following single **String** property to this class.

```
Name: priority, type: String, length: 30, access: public
```

Step 3: Add a property reference to this class in your Web service provider class

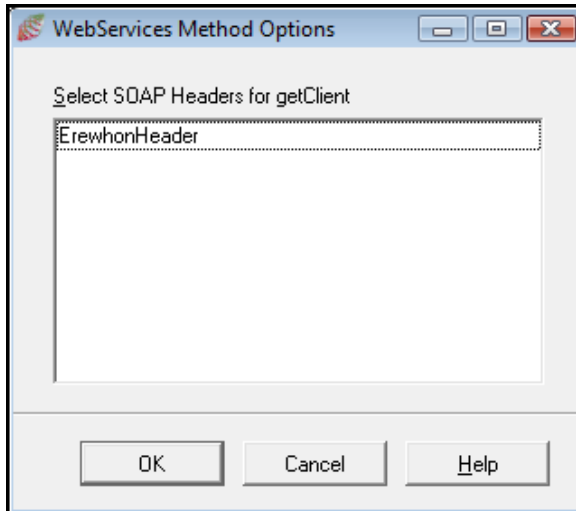
We will add the following property reference to the SOAP header class created in step 1 to the **ErewhonInvestmentsService** class.

```
Name: erewhonHeader, type: ErewhonHeader, access: public
```

Step 4: Select the methods for which you want to generate the SOAP header

We will add the header to the **ErewhonInvestmentsService** class **getClient** method. To do this, right-click on this method and then select the **Web Services Options** menu item.

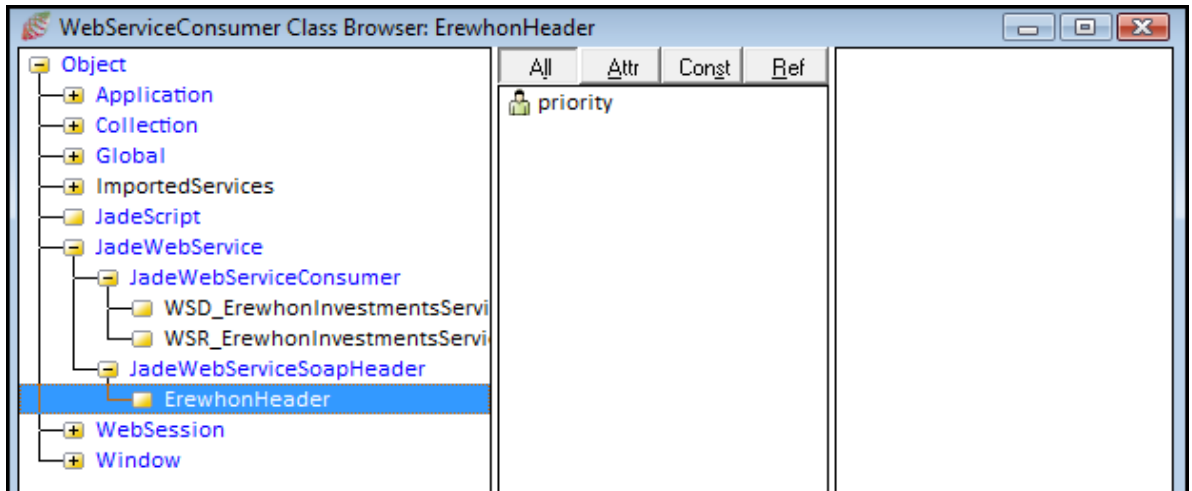
This will bring up the following dialog.



Select the **ErewhonHeader** entry and then click **OK**.

That's it for defining the header.

When the WSDL is generated, all of the required information for the SOAP header will be generated. Importing this Web service into a client now creates a subclass of the **JadeWebServiceSoapHeader** class on the client, as shown in the following diagram.



Now you can create an instance of this class and set its **priority** property to send to the Web service, as follows.

```
vars
    wsc      : WSD_ErewhonInvestmentsService;
    was      : GetClient;
    wasr     : GetClientResponse;
    header   : ErewhonHeader;
begin
    create wsc;
    create was;

    // create the header and assign this to the property
    create header;
    header.priority := 'HIGH';
    wsc.erewhonHeader := header;

    was.clientName := "Brian Olsen";
    wasr := wsc.getClient(was);
epilog
    delete was;
    delete wsc;
end;
```

The SOAP message that is generated from this call includes the SOAP header, as follows.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:tns="urn:JadeWebServices/WebServiceOverHttpApp/"
xmlns:s1="urn:JadeWebServices/WebServiceOverHttpApp/">
  <soap:Header>
    <s1:erewhonHeader>
      <s1:priority>HIGH</s1:priority>
    </s1:erewhonHeader>
  </soap:Header>
  <soap:Body>
    <s1:getClient>
      <s1:clientName>Brian Olsen</s1:clientName>
    </s1:getClient>
  </soap:Body>
</soap:Envelope>
```

The Web service can now check the header and decide whether to give this request a HIGH priority or not. If it does not, it can then set the priority value to MEDIUM, for example, and when the Web service client gets this header back, it knows that its request for HIGH priority was rejected.

Now if you had several of these methods where the header is applicable, you need only to hook up these methods with the header. If you did not have this header, this value would need to be passed as a parameter on each applicable method. If you then wanted to send additional parameters, you would need to add this to all affected methods. This can become cumbersome and error-prone. Another way of looking at this is that SOAP headers provide you with 'parameter refactoring' for Web services.

Inserting Authentication Headers

The JADE Web service WSDL import feature imports only headers that are defined in the WSDL. There are cases where the headers are not defined in the WSDL but are required by the Web service. An example of this is Web services security (WS-Security).

The following example shows how we can insert a header into the SOAP message before it is sent. The example demonstrates the use of the UsernameToken Profile. Refer to <http://www.oasis-open.org/committees/download.php/16782/wss-v1.1-spec-os-UsernameTokenProfile.pdf> for a detailed explanation on the use of this profile in Web services security.

First, we create an **insertSecurityHeader** method in the **JadeWebServiceConsumer** subclass, as follows.

```
insertSecurityHeader(userName, password: String) : String;

vars
  header: String;
begin
  header := '<soap:Header>' & CrLf;
  header := header & '<wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">' & CrLf;
  header := header & '<wsse:UsernameToken>' & CrLf;
  header := header & '<wsse:Username>' & userName & '</wsse:Username>' & CrLf;
  header := header & '<wsse:Password>' & password & '</wsse:Password>' & CrLf;
  header := header & '</wsse:UsernameToken>' & CrLf;
  header := header & '</wsse:Security>' & CrLf;
  header := header & "</soap:Header>" & CrLf;

  return header;
end;
```

We then re-implement the **JadeWebServiceConsumer** class **invoke** method, as follows.

```
invoke(inputMessage: String): String updating;

vars
  msg: String;
  p: Integer;
begin
  p := inputMessage.pos("<soap:Body>", 1);
  msg := inputMessage[1 : p - 1];
  msg := msg & insertSecurityHeader('fredbloggs', 'password');
  // put the appropriate username and password in here
  msg := msg & inputMessage[p : end];

  return inheritMethod(msg);
end;
```

This will then generate a SOAP message that looks like the following.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:tns="https://tpvs.hmrc.gov.uk/dpsauthentication">
  <soap:Header>
    <wsse:Security xmlns:wsse="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-
1.0.xsd">
      <wsse:UsernameToken>
        <wsse:Username>fredbloggs</wsse:Username>
        <wsse:Password>password</wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </soap:Header>
  <soap:Body>
    .....
  </soap:Body>
</soap:Envelope>
```

Denial of Service

Badly-Formed Messages

The issue we are going to deal with here is how we can best deal with the risk of non-malicious flooding leading to denial of service, resulting from badly-formed messages.

Badly-formed messages that are detected in the JADE Web service framework are of several types, as follows.

- Badly-formed XML (for example, no end tags)
- Badly-formed SOAP messages (for example, with no <Body> element)
- UTF-8 illegal byte sequence errors (ANSI systems or invalid UTF-8)
- Invalid data (for example, string too long)
- Client-side exceptions

When any of these exceptions occur, your code can trap these exceptions in a global exception handler, but the incoming data is not available to be inspected. We will discuss each of these in turn.

Badly-Formed XML

The XML parser can generate exceptions in the range 8901 through 8910, although only some of these apply to Web services. Specifically, mismatched tags, missing namespaces, and missing tags are some of the things that will cause exception 8901. This exception code is returned to the client, but the input message is not available to your exception handler.

Badly-Formed SOAP Messages

A SOAP message at a minimum must have an <Envelope> tag and a <Body> tag.

If the <Envelope> tag is missing, the SOAP parser will assume that the request is from the Test Harness and will respond with an HTML page. If the client is not a browser or it is not expecting HTML, this will cause an exception 11052 at the client.

If the <Body> tag is missing, the SOAP parser will not have the name of the Web service method to call and will return error 11002 (*Web Service method does not exist*). This exception code is returned to the client, but the input message is not available to your exception handler.

UTF-8 Illegal Byte Sequence Errors (Error Code 1418)

When this exception occurs while the message is being processed, the **Application** class **jadeWebServiceInputError** method is called. This method can be re-implemented in your **Application** subclass. The method is passed the incoming message as a binary value to be inspected or logged, or both inspected and logged. While further processing of this message cannot continue, you can define a suitable error message to be returned to the caller.

Invalid Data

When a message is being processed, transient objects are being created and property values being set on these transients. At this point, the following exceptions can occur because of invalid data.

- 1035 - String/Binary too long
- 4033 and 4043 - Result of expression overflows Decimal precision

When either of these exceptions occur, the Web services framework creates a SOAP exception with details of the property in error so that the Web service client knows which property value was invalid. The original exception that was raised is then passed back to the next exception handler.

Note Not all invalid data are handled. For example, an integer overflow does not create a special SOAP exception message so the Web service client will not know what caused the overflow problem.

Client-Side Exceptions

This category refers to exceptions that are raised only on the Web service client as a result of one of the following.

- Version number mismatch
- Invalid exposure list name
- Session timed out
- The requested service is a secure service
- Invalid response message

In these cases, no exception is raised in the Web service application.

Handling Badly-Formed Messages

Use Session Handling

It will be much easier to track within JADE where badly-formed messages are coming from if session handling is enabled.

With sessions, you can keep track of state information like the frequency of errors for a given session, types of errors, IP addresses, and other useful information. When an exception occurs, you can log this information using the **currentSession** system variable.

Use Web Server Logging

As discussed under "Logging", earlier in this document, Web servers can provide useful information like IP addresses, message sizes, and elapsed time. These can be analyzed to provide debugging information when users are experiencing problems.

Minimize Error Reporting

Logging detailed exception information is time-consuming, particularly if there is a continuous stream of badly-formed messages coming into the service. By keeping track of the message source and the frequency of the same types of errors, you can minimize detailed reporting of the same error over and over again.

Use Unicode JADE

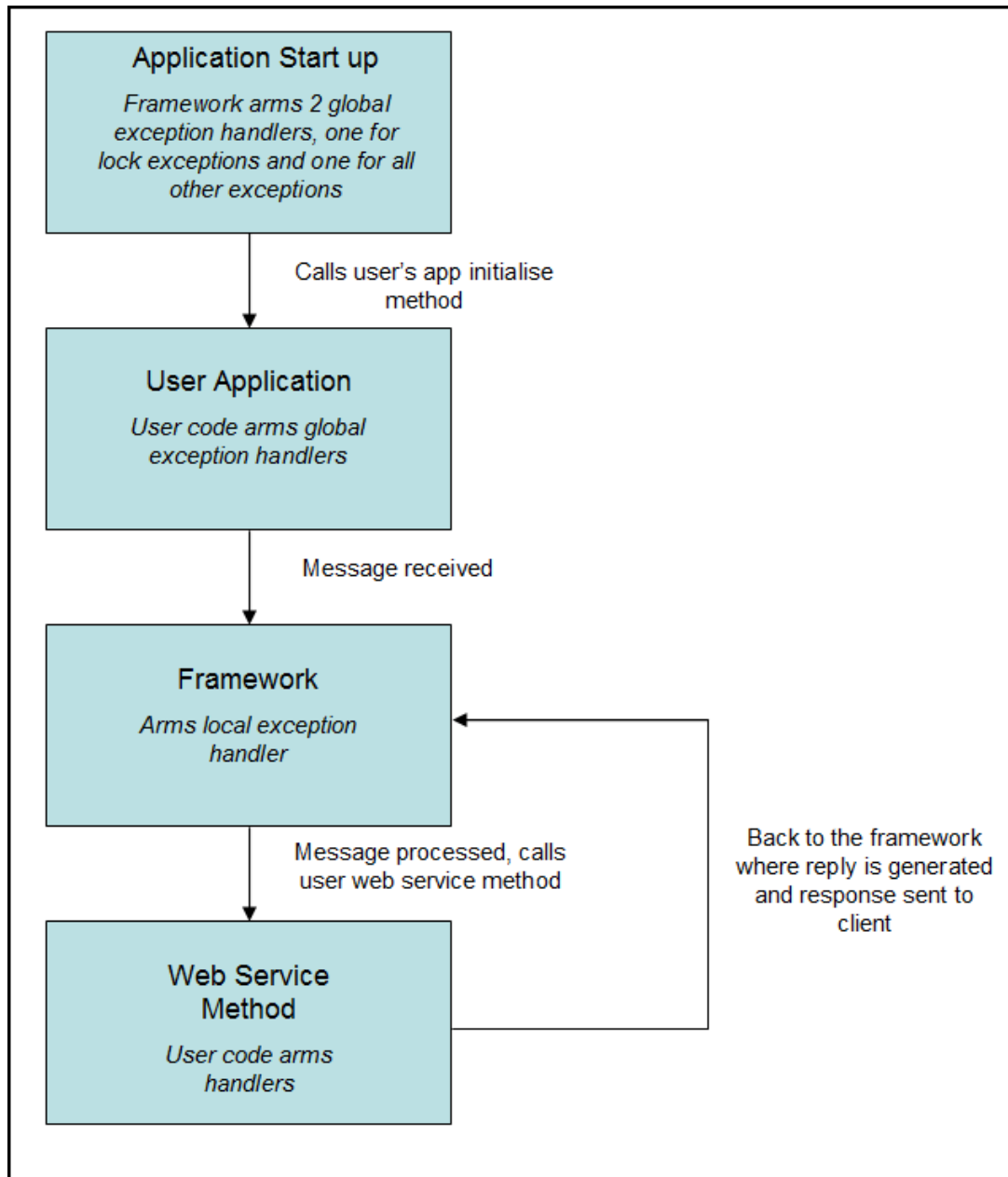
If you find that you are getting a high frequency of UTF-8 conversion errors, you should consider converting your system to use Unicode JADE. Refer to Chapter 4 of the *JADE Runtime Application Guide (RuntimeApps.pdf)*, for details about converting an ANSI system to Unicode.

Inform Users

Based on the information gathered, it should be possible to tell if the badly-formed messages are coming from a specific source. In this case, the user should be informed of the problem.

Exception Handling

A Web service application arms several exception handlers. The following diagram shows where the exception handlers are armed.



The exception handler stack will look like the following.

```

<Web Service Method>  user exception handlers, can be local or global
<Framework>          framework local exception handler
<User Application>   user global exception handlers
<Web Application>   global exception handlers in the framework
                    initialize method
  
```

The order of execution of the exception handlers is that local exception handlers are executed first, then the global ones. Within this, the most-recently armed handler is executed first. For example, if the exceptions were armed in the following order:

1. <framework global 1>
2. <framework global 2>
3. <user global 1>
4. <framework local 1>
5. <user global 2>
6. <user local 1>

The exception handler execution will be as follows, assuming **Ex_Pass_Back**.

1. <user local 1>
2. <framework local 1>
3. <user global 2>
4. <user global 1>
5. <framework global 2>
6. <framework global 1>

Note the change in order between the arming and the execution for <framework local 1> and <user global 2>.

Where your code arms an exception handler, it is safe to do an **Ex_Abort_Exception**, except for the following cases.

- Connection exceptions in the range 31000 through 31999
- Licence exceeded exceptions 5503 and 5504

These exception types must be handled by the Web application framework and your exception handlers must do an **Ex_Pass_Back**.

In addition, the framework also handles lock exceptions. When a lock exception is encountered, it will retry the lock based on the number of times specified in the **<lock_retries>** element in the configuration file. For details, see Chapter 3 of the *JADE Web Application Guide (WebApps.pdf)*. If you want to trap lock exceptions in your code, you will need to arm local exception handlers.

The framework's local exception handler guards against exceptions that occur when receiving and processing the input and when generating and sending the reply. The following exceptions are handled by the framework's local exception handler.

- Connection exceptions in the range 30000 through 32999
- Lock exceptions
- String/Binary too long (1035) or decimal precision errors (4033 and 4043). In these cases, the SOAP exception that is raised will contain details of the property in error.
- UTF-8 decoding exception (1418). When this occurs, you are given the option of inspecting the binary value that was sent, by re-implementing the **Application** class **jadeWebServiceInputError** method. The returned string from this method is sent back to the Web service client. The default message that is returned is "Input is not encoded as valid UTF-8".