Building a JMS Web service using SOAP over JMS and WebSphere Studio

Helen Du
Jeffrey Liu

February 01, 2004

SOAP over JMS offers an alternative messaging mechanism to SOAP over HTTP. While it is not yet standardized and hence may not be interoperable across platforms, SOAP over JMS offers more reliable and scalable messaging support than SOAP over HTTP. As JAX-RPC and JSR-109 become integral parts of the J2EE standard, enterprise messaging in Web services using SOAP over JMS will become well-established. This article describes the basics of JMS Web services and shows you how to develop a two-way request and response JMS Web service using WebSphere Studio V5.1.

Introduction

Web services provide a powerful architecture for integrating applications running in heterogeneous, distributed environments. Open standards make Web services interoperable across these environments. For example, the Simple Object Access Protocol (SOAP) and HTTP transport standards allow for an interoperable messaging mechanism. Although standards compliance and the associated interoperability remain the most important objectives of Web services, they are not the only ones. Consider a mission-critical system that needs to achieve zero down time and zero data loss. Using SOAP over HTTP messaging is not sufficient because HTTP does not guarantee message delivery, so a more reliable and scalable messaging mechanism is required. In the Java™ world, Java Message Service (JMS) can be used to achieve such messaging support. This article will discuss the motivations behind JMS Web services, present a use case scenario to demonstrate the use of SOAP over JMS, and then show you in detail how to create a JMS Web service using IBM® WebSphere® Studio V5.1.

Understanding JMS Web services

In a distributed computing environment, data exchange between applications is governed by the underlying messaging protocol. To satisfy Java applications that require enterprise messaging support, Sun® Microsystems introduced JMS, which is a common set of APIs and frameworks that support reliable, scalable, portable, and asynchronous messaging among J2EE applications. As JAX-RPC and JSR-109 become integral parts of the J2EE 1.4 specification in order to form the basis of enterprise Web services, support for JMS-based Web services is expected. IBM has recognized this fact and introduced a Web service implementation in WebSphere Application Server V5 that supports JMS messaging.
JMS supports two basic styles of message-based communications: point-to-point and publish-subscribe (pub-sub). Point-to-point lets multiple receivers listen for messages that arrive in a JMS queue, and the arrived message can be retrieved by only one receiver. Once the message is retrieved, it is deleted from the JMS queue and is no longer available to other receivers. Pub-sub communication lets multiple subscribers listen for messages that get published in a JMS topic. Unlike point-to-point communication, a message published in a JMS topic does not get deleted once it is retrieved. It stays on the topic and is made available for all subscribers.

Building on top of JMS, JMS Web services support two messaging styles: one-way request messaging and two-way request and response messaging. One-way request messaging lets a Web service client unblock when the request message reaches a JMS queue or topic. Two-way request and response messaging blocks a Web service client until the response message is received.

**SOAP over JMS workflow diagrams for WebSphere Application Server V5**

This section will show you how JMS Web services work in WebSphere Application Server V5. Figure 1 below shows a workflow diagram for one-way request messaging and two-way request and response messaging to a single receiver. Both JMS queue or JMS topic can be used for one-way request messaging, but JMS queue is the recommended approach because it results in better performance. For two-way request and response messaging, a JMS queue must be used, because JMS topic does not support point-to-point communication.

**Figure 1. One-way request messaging, and two-way request and response messaging to a single receiver**

Figure 2 shows a workflow diagram for one-way request messaging to multiple receivers. This messaging style resembles the JMS pub-sub structure, so a JMS topic must be used in this case.
JMS sender is an internal handler that is a part of the SOAP engine's pluggable transport mechanism. It is responsible for delivering the SOAP request message to the destination queue or topic, and receiving the SOAP response message, if applicable. JMS listener is basically a message-driven bean that dispatches an incoming request to the SOAP engine for processing. It is also responsible for sending the response message to the reply queue. WebSphere Studio supports both of the above messaging styles. The following section presents a use-case scenario on the use of JMS Web services.

Use-case scenario

Since SOAP is transport-independent and can be bound to any protocol, SOAP over JMS is an alternative messaging mechanism to the standard SOAP over HTTP messaging. Although both of them serve as a communication channel between a Web service provider and a Web service client, they are very different. When interoperability is the driving factor, use SOAP over HTTP because it is standardized by the Web Service Interoperability (WS-I) organization. When reliability, scalability, and asynchronous messaging are the key factors, consider SOAP over JMS.

SOAP over JMS ensures reliability because message delivery is guaranteed. Messages sent by a Web services provider or Web services client are placed onto a queue or topic and stored in persistent storage. In case of a communication failure, the failing message is retrieved from the persistent storage and re-sent until transmission is successful, which is extremely useful in transaction- and data- critical systems. Businesses that use Enterprise Application Integration (EAI) should find SOAP over JMS appealing because it boosts confidence when exchanging critical data between client and server.

Scalability is another advantage with SOAP over JMS. Unlike HTTP, JMS can support high-volume connections, even for services that get tens of thousands of connections per second.

Asynchronous messaging lets a client invoke a service without waiting for the response. Asynchronous invocation can be implemented by both synchronous and asynchronous transport. JMS as an asynchronous transport can provide a correlator mechanism for associating response messages with request messages. It also lets a client query the status of its requests, and retrieve the responses independently. These features, which HTTP lacks, make the implementation of asynchronous invocation easier for both service requestor and service provider.

The following scenario illustrates a possible configuration that lets businesses take advantage of the interoperability aspect of SOAP over HTTP messaging, as well as the reliability aspect of
SOAP over JMS messaging. Figure 3 shows the configuration of a Web services gateway that uses SOAP over HTTP as the communication channel between external businesses, and uses SOAP over JMS within the company's intranet. The use of SOAP over HTTP is for the sake of interoperability, since external businesses may be running on different platforms. Conversely, since a company controls applications running within its intranet, interoperability becomes less of an issue, and other features such as reliability may be deemed more important. The use of SOAP over JMS can compensate for the possible downtime of the systems running within the intranet, keeping data losses to a minimum.

**Figure 3. Configuration of a Web services gateway**

**Building a JMS Web service**

1. Begin by creating a WebSphere V5.0 test environment server and configuring its JMS provider. Bring up the wizard selection dialog. Select **File => New => Other**.
2. Launch the Server Creation wizard. Select **Server** from the menu on the left and **Server and Server Configuration** from the list on the right. Click **Next**.
3. Enter `WebSphere_5.0_Test_Environment` as the server name and select **Test Environment** from the WebSphere V5.0 folder. Click **Finish**.
4. Switch to the Server perspective. Select **Windows => Open Perspective => Other**.
5. In the Perspective selection dialog, select **Server** and click **OK**.

**Figure 5. Perspective selection dialog**
6. Use the Server Configuration Editor to configure the JMS provider. To open the
   Server Configuration Editor, double-click on the server configuration labeled
   **WebSphere_5.0_Test_Environment** in the server configuration view at the bottom left corner
   of the workbench window.

7. Click on the **JMS** tab of the editor.

8. By default, the JMS provider is disabled. To enable it, click on the **MQ Simulator for Java
   Developers** radio button.

9. We need to create a queue in the JMS server to maintain the incoming requests to our
   Web service. Under the JMS Server Properties section, click on the **Add** button beside the
   Queue Names table. In the Add Queue Name dialog, enter **HelloWorld** and click **OK**. The
   WebSphere JMS Provider Options page should look like Figure 6:

   **Figure 6. WebSphere JMS Provider Options page**

10. The next step is to create and associate a JMS destination with the HelloWorld queue we
    created earlier. A JMS destination lets you configure the queue's settings, for example, its
    JNDI name. In this tutorial, the JMS Web service will use two-way request and response
    messaging. Under the JMS Destinations section, click on the **Add** button beside the
    WASQueue entries table. In the Add WASQueue dialog shown in Figure 7, enter **HelloWorld**
    as the name (this name must match the queue name you added to the JMS server in the
    previous step) and **jms/HelloWorld** as the JNDI name. Click **OK**.
11. Now that you have a JMS destination for the Web service, you need to create a JMS connection factory to control how connections are made to the JMS destination, for example, the connection timeout period. Under the JMS Connection Factories section, click on the Add button beside the WASQueueConnectionFactory entries table. In the Add WASQueueConnectionFactory dialog shown in Figure 8, enter `HelloWorldQCF` as the name and `jms/HelloWorldQCF` as the JNDI name. Click OK.
12. Since our JMS Web service uses two-way request and response messaging, we need to create a reply queue for receiving the response message. Create another WASQueueConnectionFactory entry. Use WebServicesReplyQCF as the name and jms/WebServicesReplyQCF as the JNDI name. You cannot modify these names because they are fixed by the WebSphere Web service implementation. The WebSphere JMS Provider Options page should look like Figure 9:
13. Click on the **EJB** tab of the Server Configuration Editor.
14. Click on the **Add** button beside the Listener Ports table. In the Add Listener Port dialog shown in Figure 10, enter **HelloWorldPort** as the name, select **jms/HelloWorldQCF** as the connection factory JNDI name, and **jms/HelloWorld** as the destination JNDI name. Click **OK**. This listener port will be used by a message-driven bean (MDB) to retrieve request messages from the defined queue.
15. Click on the **Environment** tab of the editor.
16. Expand the **Classpath** section and click on the **Add External JARs** button. Add `<WebSphere Studio installation directory>runtimes\base_v5\lib\urlprotocols.jar` to the server's classpath:
17. Save and close the Server Configuration Editor. You have successfully configured the JMS provider for our Web service. Next, you will create the actual JMS Web service from a HelloWorld EJB.
18. Go to the Download section of this article and download the HelloWorld EJB to your file system.
19. Import the HelloWorld EJB into WebSphere Studio. Select File => Import, then select EJB JAR file and click Next.
20. In the EJB import wizard, click Browse and navigate to the location where you downloaded the HelloWorld EJB. Click Finish.
21. Generate deployment code for the HelloWorld EJB. Right-click on the HelloWorldEJB project and select **Generate => Deployment and RMIC Code**. Click **Select all** and then click **Finish**.
22. Create an EJB router project for the HelloWorld JMS Web service. The router project must be created before running through the Web service wizard. This project is used to maintain the MDB. Select **File => New => Other**.
   23. Select **EJB** from the menu on the left and **EJB Project** from the list on the right. Click **Next**.
   24. In the first page of the new EJB project wizard, accept the default setting, that is to create a version 2.0 EJB project. Click **Next**.
25. Enter **HelloWorldEJBRouter** as the project name and choose **HelloWorldEJBEAR** as the EAR project. Click **Finish**.
26. We are all set to expose the HelloWorld EJB as a JMS Web service. Launch the Web service creation wizard: select **File => New => Other**. Select **Web services** from the left and **Web service** from the right, then click **Next**.
27. Figure 13 shows the Web service creation wizard. Choose **EJB Web service** as the Web service type. Select the **Generate a proxy**, **Test the generated proxy** and **Overwrite files without warning** checkboxes. Click **Next**.
28. In the Service Deployment Configuration page, make sure **WebSphere_5.0_Test_Environment** is selected as the server. Choose **HelloWorldEJBRouter** as the router project and **HelloWorldEJB** as the EJB project. Click **Next**.
29. In the Web service EJB selection page, click on Browse EJB beans. In the browse EJB beans dialog, choose HelloWorldEJBEAR as the EAR project and select HelloWorldEJB in the EJB bean table. Click OK.

30. Click on the SOAP over JMS radio button and enter the information as shown in Figure 15. Click Finish.
31. In the sample JSP, click on the `getHelloWorld(java.lang.String)` method. Enter your name in the input text field and click **Invoke**. The result from the HelloWorld JMS Web service is displayed in the result pane:
The use of the JMS transport is transparent to a Web service client, in other words, the client does not have to know the transport protocol that the Web service uses. When generating a client using WebSphere Studio, the tools will handle the selection of the JMS transport based on the information defined in the WSDL file. Figure 17 shows an XML fragment from the WSDL file describing the HelloWorld JMS Web service. The transport attribute of the WSDL binding element points to http://schemas.xmlsoap.org/soap/jms and the location of the endpoint address uses the JMS protocol. These are the signs that indicate the JMS transport protocol is being used.
Conclusion

SOAP over JMS offers an alternative messaging mechanism to SOAP over HTTP. While it is not yet standardized and hence may not be interoperable across platforms, SOAP over JMS offers more reliable and scalable messaging support than SOAP over HTTP. As JAX-RPC and JSR-109 become integral parts of the J2EE standard, enterprise messaging in Web services using SOAP over JMS will become well-established. This article has described the basics of JMS Web services and showed you how to develop a two-way request and response JMS Web service using WebSphere Studio V5.1.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HelloWorldEJB.zip (HTTP</td>
<td>FTP)</td>
</tr>
</tbody>
</table>
Related topics

- Hypertext Transfer Protocol (HTTP)
- Java 2 Enterprise Edition (J2EE)
- Java Message Service (JMS)
- Simple Object Access Protocol (SOAP)
- WebSphere Application Server
- Web Service Interoperability (WS-I)

© Copyright IBM Corporation 2004
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)