WS-Notification in WebSphere Application Server V7: Part 1: Writing JAX-WS applications for WS-Notification

Lucas Partridge (lucas_partridge@uk.ibm.com)
Software Engineer
IBM

James M. Siddle
Software Engineer
IBM

12 November 2008

Learn how to develop JAX-WS applications for use with the updated WS-Notification functionality included with IBM® WebSphere® Application Server V7.0. Readers are expected to have some familiarity with Java™-based Web services, the WS-Notification 1.3 family of specifications, and to have had some experience using the administration console for WebSphere Application Server.

Introduction

The WS-Notification 1.3 family of specifications collectively define a publish-subscribe messaging pattern for Web services. IBM WebSphere Application Server V6.1 provides an implementation of WS-Notification using the Java API for XML-based RPC (JAX-RPC) Web services. WebSphere Application Server V7 continues to support this JAX-RPC implementation. However, Version 7.0 also supports a new JAX-WS 2.1 implementation. When you create a new WS-Notification service in WebSphere Application Server V7, you must decide which of these implementations you want to use; either a “Version 6.1” (JAX-RPC) type of service, or a ”Version 7.0” (JAX-WS) type of service.

How do you decide? If you choose the Version 7.0 type of service, then you will be able to use policy sets to compose your WS-Notification service with other open WS- standards, such as WS-ReliableMessaging and WS-Security. This would then enable you to send your messages more reliably or securely, or both. You could also apply JAX-WS handlers to your WS-Notification service. If, however, you don’t want to compose WS-Notification with other WS- standards, or you still want to apply JAX-RPC handlers to your service, then you can create a Version 6.1 type of service instead.

This article explains how to write JAX-WS-based WS-Notification clients and services to work with the new Version 7.0 type of service. Know that you can also write JAX-WS clients and services to work with the Version 6.1 type of service. Conversely, you can also write JAX-RPC clients to work with the Version 7.0 type of service. However, if you want to make full use of composition with other Web service standards from when your message is published until the time it is received by your consumers, then you should write JAX-WS clients and services against a Version 7.0 type of service.

About this series

This article is the first in a series that describes how to compose JAX-WS based WS-Notification applications with other WS- standards. Subsequent articles will describe how to compose WS-Notification applications with the WS-ReliableMessaging standard to achieve reliable transmission of notification messages, how to configure the new WS-Notification implementation in a clustered environment, and how to compose WS-Notification with WS-Security to securely transmit messages.

Before reading further, you might want to refer to the IBM developerWorks article WS-Notification in WebSphere Application Server Version 6.1, which provides an introduction to the basic concepts and the Version 6.1 implementation of WS-Notification. This introductory article will be particularly helpful to readers new to WS-Notification.

The remainder of this article is in three parts. First, the basic configuration of WS-Notification in WebSphere Application Server V7 is described. This lays the foundation for the next section, which describes the steps required to develop JAX-WS-based publisher, consumer, and subscriber components to interact with the WS-Notification implementation. The article concludes with a walkthrough of an example scenario, using sample enterprise application archive files and scripts that are included with this article.

Configure your application server for WS-Notification

Before writing components to interact with the new JAX-WS-based WS-Notification Web services, you need to configure an application server to expose those Web services. The following steps provide an outline of how to achieve this in WebSphere Application Server V7, and will be most useful to those already familiar with WS-Notification configuration. For detailed instructions on configuration, refer to the WebSphere Application Server V7 Information Center (or to Part 2 of this series, when available). Alternatively, you can download the sample script included with this article, WAS7_WSN_JAXWS_Part1_Setup.py, to execute these configuration steps.

These instructions assume you have already created a single server profile in your application server installation. Unless otherwise indicated, defaults should be selected for configuration options.

1. **Create a Service Integration Bus.**
   Include your single server instance as a bus member. The Service Integration Bus provides the underlying messaging infrastructure for the WS-Notification implementation.
2. **Create a new Version 7.0 WS-Notification service.**
   The second step in the wizard for creating WS-Notification services asks you to choose which type of service to create. As mentioned earlier, to make use of new functionality, such as

reliable messaging, you must choose **Version 7.0** at this step. In this example, the service is called WSNService1 and is referred to hereafter as such.

3. **Create a service point associated with the previously created service.**
   This is a mandatory step when creating a WS-Notification service. The service point in this example is called WSNServicePt1. You do not need to create a permanent topic namespace provided the default option to enable dynamic topic namespaces is selected when you create the new service.
4. **Start the deployed service point enterprise application.**
   Each Version 7.0 service point is associated with a specially deployed enterprise application that must be started before applications can make use of the service point. The name of the enterprise application is derived automatically from the names you give your service and service point. Here, it is called WSN_WSNService1_WSNServicePt1.
   Use the administration console to start the deployed enterprise application, or, alternatively, restart the application server to achieve the same result.

Having created a Version 7.0 WS-Notification service and service point, you've laid the groundwork for writing applications to publish and consume messages in a reliable, secure way using Web services.

Write publisher, consumer, and subscriber components

In this section, you'll learn how to write applications using three key components: a publisher, a consumer, and a subscriber. The order in which you write these three components doesn't matter. For the sake of simplicity, however, we'll start with a publisher. You could also choose to implement a combined subscriber and consumer, but they are kept separate here for clarity.

In this example, all three of these components are implemented as Java EE Web applications. The core functionality of each application is implemented within a single bean. A servlet is also provided for each bean so that you can control the associated bean's behaviour (in the case of the publisher and subscriber), or to query its status (in the case of the consumer). You could equally have implemented your publisher and subscriber as Java EE application clients or EJBs, and your consumer as an EJB component, but the key steps for implementing the WS-Notification functionality would be the same as described below. (Details of the servlets are omitted here; if you want to know more about them you are invited to browse the downloadable code.)

A. Write a publisher client to publish notification messages

To write a publisher client:

1. **Obtain WSDL files for the notification broker service.**
   First, obtain the WSDL for the NotificationBroker Web service that is exposed by the Version 7.0 WS-Notification service point. This is the Web service through which notification messages are published.
   The easiest way to do this is through the WebSphere Application Server administration console. From the console, navigate to **WS-Notification services => WSNService1 => WS-Notification service points => WSNServicePt1 => Publish WSDL files**. At this point, you

will in fact have access to all the WSDL files for a particular service point, which will prove useful later in this article.

Extract the NotificationBroker.wsdl file from the resulting .zip file, and move (or copy) it to a temporary directory, ready for the next step.

2. **Run wsimport against NotificationBroker.wsdl to generate client stubs.**

Next, run the `wsimport` command that is provided with the application server to generate client stubs for your publisher to use. This command uses the information within a WSDL file to generate JAX-WS portable artifacts for client and service development. You can learn more about these artifacts from this WebSphere Application Server V7 Information Center page.

The following is an example of how to run the command in a Microsoft® Windows® environment. *<WAS>* refers to the root directory for your installation of WebSphere Application Server (for example, C:\was). This assumes that you have opened a command prompt in the directory which contains your copy of NotificationBroker.wsdl.

```
<WAS>\bin\wsimport -keep -b <WAS>\util\ibm-wsn-jaxb.xml -wsdllocation "WEB-INF/
wsdl/NBModule/NotificationBroker.wsdl" NotificationBroker.wsdl
```

Table 1 describes these arguments:

Argument	Description
`-keep`	Retain the generated source code (*.java) files. These can be useful for reference.
`-b <WAS>\util\ibm-wsn-jaxb.xml`	Use IBM helper classes for certain WSDL elements instead of the default classes (see below).
`-wsdllocation "WEB-INF/wsdl/NBModule/NotificationBroker.wsdl"`	Tells the generated Web service client class where to find the corresponding WSDL (see below).
`NotificationBroker.wsdl`	The WSDL to be parsed.

The second argument tells `wsimport` to use IBM helper classes for some of the WS-Notification schema elements referred to by the WSDL, instead of generating the default JAXB classes. This is a crucial step to work around certain limitations that arise when the JAXB standard is applied to interpret the WS-Notification WSDL. By including this option, you will be able to make use of full WS-Notification functionality in your JAX-WS publisher, consumer, and subscriber components.

The `wsdlLocation` argument specifies the value for the wsdlLocation attribute of the @WebServiceClient annotation in one of the generated files (more about this annotation later). It should refer to the location of the WSDL file describing the service that the Web service client will interact with, and this WSDL file must be accessible at this location when the Web service client runs. Possible values include a file system location or a relative location within an enterprise application archive (EAR) file. If this argument is omitted from the `wsimport` command, then the generated artifact will refer to the full path of the WSDL file on your hard disk. If you want to run your publisher on another machine, or the location of the WSDL file is liable to change, then package the WSDL file within your EAR file and refer to it with a relative URL as was done here. Your publisher will be much more portable as a result.

Notice that the `wsimport` command is also supported from within application development tools, such as IBM Rational® Application Developer. In Rational Application Developer V7, remember to specify the *<WAS>*\util\ibm-wsn-jaxb.xml JAXB bindings file when using the

Web Service Client wizard to ensure generated artifacts use the IBM helper classes. (See the Rational Application Developer V7 Information Center topic "Generating a Web service client from a WSDL document using the IBM WebSphere JAX-WS runtime environment" for details on how to specify the bindings file.)

3. **Ensure generated artifacts are in your development class path.**
   When `wsimport` has finished, you will find many .class and .java files in packages under the directory containing the WSDL file. Some of these artifacts merit a few words:
   - **com.ibm.websphere.wsn.notification_broker.NotificationBroker** -- This interface, annotated with @WebService, represents the service endpoint interface (SEI) for the NotificationBroker Web service. It describes all the operations exposed by the Web service, such as Notify and Subscribe, and any parameters that they require when invoked.
   - **com.ibm.websphere.wsn.notification_broker.WSNService1WSNServicePt1NB** -- This class gets its name from the name attribute of your service element in your NotificationBroker.wsdl file, which in turn was derived from the names you gave your WS-Notification service and service point. This class extends javax.xml.ws.Service and provides convenience wrapper methods, which clients can use to get a stub for the NotificationBroker Web service. The returned stub will implement the NotificationBroker service endpoint interface already mentioned above. Remember the `wsdlLocation` argument for `wsimport` discussed earlier? That argument specified the value for the wsdlLocation attribute in the @WebServiceClient annotation for this class.
   - In some packages, **org.oasis_open.docs.wsn.b_2** for example, you may notice a class called ObjectFactory. These classes provide convenience factory methods for creating the numerous JAXB objects that have been generated by `wsimport`. You don't have to use these classes, but they shield you from having to specify namespace URIs and javax.xml.namespace.QName objects.

   To make use of the generated artifacts, you must make sure they are present in your development class path. For example, when using Rational Application Developer, you must import the generated artifacts. When compiling your publisher code from the command line, simply ensure the generated artifacts appear somewhere in the class path used by the compiler.
   Do not be tempted to omit the generated class that is annotated with @WebServiceClient from your project, even when your Java code doesn't make direct use of it. In this example, this class is called WSNService1WSNServicePt1NB. If you omit it from your EAR file, or specifically the Web application module, the application server's container won't be able to detect your JAX-WS Web service client, and you won't be able to attach policy sets to your client.
4. **Ensure the JAX-WS thin client for WebSphere Application Server V7 is in your development class path.**
   Similar to the previous step, you must make sure that the JAX-WS thin client JAR file is in your development class path. This JAR file contains the IBM helper classes mentioned earlier. The full filename of the JAR file as packaged in WebSphere Application Server V7 on Microsoft Windows is:

`<WAS>\runtimes\com.ibm.jaxws.thinclient_7.0.0.jar`

Development tools such as Rational Application Developer also provide ways to include the JAR file in the class path. For example, Rational Application Developer V7 provides a way of adding a specific application server runtime library into the class path for a project via the **Build Path** dialog.

Having set up your development environment with the necessary artifacts, it's now time to write the publisher component. WS-Notification makes it possible for publishers to register with a broker before they are allowed to publish messages. However, publisher registration is beyond the scope of this article and you don't need to do it here, provided you made sure that the **Requires registration** option was disabled, as it is by default, when you created your WS-Notification service in the administration console.

5. **Look up the notification broker service and port.**

The first step in writing a notification message publisher is to look up the notification broker service, then to obtain a port from the service to which notification messages can be sent.

The code segment shown in Listing 1, which includes WS-Addressing functionality, shows how this is achieved with JAX-WS 2.1 artifacts and APIs. Throughout this article, the relevant import statement is included before each code listing the first time we refer to a new class.

Listing 1

```
import java.net.MalformedURLException;
import java.net.URL;

import javax.xml.namespace.QName;
import javax.xml.soap.SOAPElement;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.Service;
import javax.xml.ws.soap.AddressingFeature;

import com.ibm.websphere.sib.wsn.jaxb.base.TopicExpressionType;
import com.ibm.websphere.wsn.notification_broker.NotificationBroker;

/**
 * Publishes one WS-Notification notify message.
 *
 * @param brokerPortAddress - URL of WSN broker port.
 * @param brokerWSDLTargetNamespace - target namespace value
 * for the broker's NotificationBroker WSDL.
 * @param topicNamespace - URI of topic namespace.
 * @param topicNamespacePrefix - prefix for topic namespace.
 * @param topicDialect - topic expression dialect.
 * @param topicExpression - topic expression which describes
 * the topic to publish on.
 * @param messageContents - message to be published.
 */
public void publishOneMessage(String brokerPortAddress,
  String brokerWSDLTargetNamespace, String brokerServiceName,
  String brokerPortName, String topicNamespace,
  String topicNamespacePrefix, String topicDialect,
            String topicExpression, SOAPElement messageContents) {
// Get hold of a stub for the broker web service:
Service broker = null;
try {
 broker = Service.create(
new URL(brokerPortAddress + "?wsdl"),
new QName(brokerWSDLTargetNamespace, brokerServiceName));
} catch (MalformedURLException e) {
 e.printStackTrace();
}

// When we get the port object, remember to enable WS-Addressing:
NotificationBroker port = (NotificationBroker) broker.getPort(
  new QName(brokerWSDLTargetNamespace, brokerPortName),
  NotificationBroker.class,
  new AddressingFeature());
```

6. **Create a notification message.**

   Next, create a notification message holder type object. This maps directly to a single notification message and is used to hold a message, topic, subscription reference, and publisher reference. Only the message is required; the other three components are optional. In Listing 2, a message and a topic are included.

   ## Listing 2

```
import org.oasis_open.docs.wsn.b_2.NotificationMessageHolderType.Message;
import org.oasis_open.docs.wsn.b_2.NotificationMessageHolderType;

// Create the message and put the contents in its 'any' element:
Message message = new Message();
message.setAny(messageContents);

NotificationMessageHolderType notificationHolder =
new NotificationMessageHolderType();

// Put the message in the holder:
notificationHolder.setMessage(message);
```

   Note that the message payload (messageContents) is a javax.xml.soap.SOAPElement. (Strictly, the message payload passed into the Message.setAny() method must implement the org.w3c.dom.Element interface; javax.xml.soap.SOAPElement extends this interface.) Listing 3 shows one way of creating a SOAPElement.

   ## Listing 3

```
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPFactory;

/**
 * Returns a SOAPElement containing some message data for
 * the specified namespace.
 *
 * @param nsPrefix - namespace prefix.
 * @param elementName - name of the element.
 * @param nsURI - URI for the namespace.
 * @param data - message data. Must be valid XML.
 * @return the SOAPElement containing the message data.
 * @throws SOAPException -
 *             if the SOAPElement cannot be created properly.
 */
public SOAPElement createSOAPElement(String nsPrefix, String elementName,
String nsURI, String data) throws SOAPException {
 SOAPFactory soapFactory = SOAPFactory.newInstance();
 SOAPElement element = soapFactory.createElement(elementName, nsPrefix,
 nsURI);
 element.addTextNode(data);
 return element;
}
```

7. **Associate the message with a topic.**

A key part of message publication with WS-Notification is to associate messages with particular topics so that consumers are able to receive messages that are of interest to them. Listing 4 shows how to specify a topic for publication.

Listing 4

```
import com.ibm.websphere.sib.wsn.jaxb.base.TopicExpressionType;

// Prepare a topic expression to describe the topic we want to
// publish on:
topicExpression = topicNamespacePrefix + ":" + topicExpression;
// - this is the actual topic!

// The topic expression, its associated dialect and its namespace
// definition are all held within a TopicExpressionType object. This
// object maps directly to the Topic child element of the
// NotificationMessage element in the message that' transmitted.
TopicExpressionType topicExpressionType = new TopicExpressionType();

// Set the topic expression:
topicExpressionType.setExpression(topicExpression);

// Specify the mapping from the namespace prefix to the topic
// namespace URI:
topicExpressionType.addPrefixMapping(
 topicNamespacePrefix,
 topicNamespace);

// Specify the TopicExpression dialect to be used:
topicExpressionType.setDialect(topicDialect);

// Set the topic in the notification message holder:
notificationHolder.setTopic(topicExpressionType);
```

Note the use of the IBM helper class (from the com.ibm.websphere.sib.wsn.jaxb.base package) to associate the prefix that is used in the topic expression with a particular XML namespace. Making this association is very difficult when using the default artifacts generated from the WS-Notification WSDL.

8. **Issue a notify request.**
The final step in publishing a message is to issue the request by passing your created notification message objects to the port that you previously obtained (Listing 5)

Listing 5

```
import org.oasis_open.docs.wsn.b_2.Notify;

// Add the holder to the list of notifications to be sent with this
// Notify request. Remember a single Notify request may contain
// more than one NotificationMessage.
Notify notify = new Notify();
notify.getNotificationMessage().add(notificationHolder);

// Send the Notify request to the broker:
port.notify(notify);
```

That’s it! You have seen how to publish a notification message into the WS-Notification Web service provided by the application server.

A fully working example publisher is provided with this article. You can try it out by downloading and installing WSNPublisher.ear. Further instructions can be found in the example scenario section later in this article.

B. Write a consumer service to receive notification messages

The second component required for a WS-Notification application is a consumer to receive and process messages. Here are the key steps to writing a consumer using JAX-WS:

1. **Write WSDL to describe your consumer service.**
   Here, you will learn how to write a Web service that implements the Notify operation on the NotificationConsumer interface of the NotificationBroker port type. (Raw consumers and pullpoint-style consumers are beyond the scope of this article.)
   The first step in achieving this is to write the Web service description, using WSDL. Listing 6 is an example of a complete WSDL for a WS-Notification consumer. You can use this as the basis for your consumer; the key fields or elements you are free to change appear in bold type.

Listing 6

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:tns="http://wsn.test/PushNotificationConsumer/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://wsn.test/PushNotificationConsumer/"
 xmlns:bw2="http://docs.oasis-open.org/wsn/bw-2">

 <!-- Import the 'base notification' WSDL definitions; this
  includes the port type (a.k.a. interface definition)
  that the consumer will implement -->
 <wsdl:import namespace="http://docs.oasis-open.org/wsn/bw-2"
  location="http://docs.oasis-open.org/wsn/bw-2.wsdl">
 </wsdl:import>

 <!-- Define a binding from the NotificationConsumer port type
  to SOAP over HTTP, document literal style -->
 <wsdl:binding name="PushNotificationConsumerSOAP"
  type="bw2:NotificationConsumer">
  <soap:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="Notify">
   <soap:operation soapAction="" />
   <wsdl:input>
    <soap:body use="literal" />
   </wsdl:input>
  </wsdl:operation>
 </wsdl:binding>

 <!-- Define the web service and associated ports; only one port
  in this case - the SOAP/HTTP port defined above -->
 <wsdl:service name="PushNotificationConsumer">
  <wsdl:port name="PushNotificationConsumerSOAP"
   binding="tns:PushNotificationConsumerSOAP">
   <soap:address location=
"http://localhost:9080/ctx/svcs/consumerSOAP" />
  </wsdl:port>
 </wsdl:service>
```

```
</wsdl:definitions>
```

Notice that the Notify operation and its associated message type are already defined by the WS-Notification specification at http://docs.oasis-open.org/wsn/bw-2.wsdl. All you have to do is define a service, port, and binding to implement the Notify operation. Therefore, to customize the above WSDL to your needs, simply modify the namespace, service, and port information. You might also need to update the address for the port, although this is not required for consumers deployed in WebSphere Application Server V7, since the address will actually be replaced with the correct value whenever the WSDL is queried.

Finally, notice that SOAP/HTTP was used for the transport protocol. Both SOAP/HTTP and SOAP/JMS transport protocols are supported for the Version 6.1 (JAX-RPC) type of WS-Notification service in WebSphere Application Server V7. However, only SOAP/HTTP is supported for the Version 7.0 (JAX-WS) type of service.

2. **Run wsimport to generate service implementation classes.**

   Similar to your publisher, you need to generate JAX-WS artifacts by running the `wsimport` command against the consumer WSDL, as exemplified by this following Windows command line:

   ```
   <WAS>\bin\wsimport -keep -b <WAS>\util\ibm-wsn-jaxb.xml -wsdllocation "WEB-INF/
   wsdl/PushNotificationConsumer.wsdl" PushNotificationConsumer.wsdl
   ```

   Two of the artifacts that wsimport will have generated are:

   - **test.wsn.pushnotificationconsumer.NotificationConsumer** -- This is the SEI for the PushNotificationConsumer Web service.
   - **test.wsn.pushnotificationconsumer.PushNotificationConsumer** --This is the class which any clients of the consumer Web service could use to obtain a stub for subsequently invoking the Notify operation against.

3. **Ensure the generated artifacts and the JAX-WS thin client JAR are in your development class path.**

   Similar to your publisher, your consumer code must be compiled in an environment where the generated artifacts and JAX-WS thin client JAR are available.

4. **Implement the notify method to receive and process notification messages.**

   You now reach the point where you can write the code to implement the WS-Notification consumer. To receive notification messages in a consumer, you need to write a Java class that has the @WebService annotation. This annotation informs the application server that the associated class should be treated as a Web service when it is detected in deployed Web modules. Here, the implementation class is called PushNotificationConsumerSOAPImpl and you can see the most important parts of it here:

Listing 7

```
@javax.jws.WebService(
 endpointInterface =
        "test.wsn.pushnotificationconsumer.NotificationConsumer",
 targetNamespace = "http://wsn.test/PushNotificationConsumer/",
 serviceName = "PushNotificationConsumer",
 portName = "PushNotificationConsumerSOAP",
 wsdlLocation = "WEB-INF/wsdl/PushNotificationConsumer.wsdl")
public class PushNotificationConsumerSOAPImpl
```

```
    public void notify(Notify notify)
    {
        List<NotificationMessageHolderType> notificationMessages =
         notify.getNotificationMessage();

        for (NotificationMessageHolderType notificationMessage :
                notificationMessages)
        {
         // Your code here
        }
    }
    }
```

There are a few things to note about the above code.

- First, you must make sure the JAX-WS @WebService annotation's fields are completed with the corresponding values taken from your consumer's WSDL and `wsimport` command line values. The relevant fields appear in bold type, above.
- Second, the above code sample doesn't really do anything when notify is invoked, apart from looping through the list of notification messages provided. The next steps are completely optional, but they describe some of the things that you might want to do with notification messages.

5. **Obtain topic dialect, expression, and namespace mappings**
When processing notification messages, you might want to check under which topic the message was received. Listing 8 shows how to obtain dialect, expression, and namespace mapping information for the topic associated with a particular notification message.

Listing 8

```
import com.ibm.websphere.sib.wsn.jaxb.base.TopicExpressionType;
import java.util.Map;

// List topic dialect and expression
TopicExpressionType topicExpressionType =
notificationMessage.getTopic();
System.out.println("-- Topic dialect: "
        + topicExpressionType.getDialect());

String topicExpression = topicExpressionType.getStringExpression();
System.out.println("-- Topic expression, including the namespace prefix: "
 + topicExpression);

// List all the mappings between the prefixes that can appear in
// topic expressions and their corresponding namespace URIs
Map<String, String> namespaceMappings =
topicExpressionType.getNamespacePrefixesMap();

System.out.println(
"-- Number of namespace mappings in the topic expression: "
      + namespaceMappings.size());

for (String topicNamespacePrefix : namespaceMappings.keySet())
{
    System.out.println("---- Prefix '"
            + topicNamespacePrefix
            + "' maps to namespace '"
            + topicExpressionType
                    .getNamespaceForPrefix(topicNamespacePrefix)
            + "'");
}
```

Notice that the IBM helper class from the com.ibm.websphere.sib.wsn.jaxb.base package was used again, this time to obtain prefix information for any namespaces used in topic expressions. As before, this would be very hard to do using the default artifacts generated by `wsimport`.

6. **Obtain the topic expression.**
   For scenarios where topic expressions include namespaces, you might also want to strip out the namespace prefix (Listing 9).

Listing 9

```
// The topic expression (in our example) will be of the form
// topicNamespacePrefix:topicExpression, so we need to extract the
// topicExpression part:
int colonPos = topicExpression.indexOf(':');
if (colonPos > -1)
{
  topicExpression = topicExpression.substring(colonPos + 1);
}
System.out.println("-- Topic expression, without the prefix: "
    + topicExpression);
```

7. **Query the notification message payload.**
   The most important part of a notification message is the payload. Listing 10 shows how you obtain the payload, along with the namespace, prefix, and name for the payload’s root element.

Listing 10

```
import org.oasis_open.docs.wsn.b_2.NotificationMessageHolderType.Message;
import org.w3c.dom.Element;

// Get the message content as a DOM Element.
Message message = notificationMessage.getMessage();
Element messageContents = (Element) (message.getAny());

String messagePayload = messageContents.getTextContent();
System.out.println("-- Message payload: " + messagePayload);

System.out.println("-- Payload namespace: "
        + messageContents.getNamespaceURI());

System.out.println("-- Payload namespace prefix: "
        + messageContents.getPrefix());

System.out.println("-- Payload element name: "
        + messageContents.getLocalName());
```

You should now have a working, JAX-WS-based WS-Notification consumer. A fully working example consumer, WSNConsumer.ear, is also provided with this article.

The final step is to write a JAX-WS based subscriber.

C. Write a subscriber client to subscribe the consumer to the publisher

So far, you have written a publisher to publish messages via a WS-Notification service, and a consumer to consume messages from a WS-Notification service. But without issuing a subscription request, none of the messages you publish will be transmitted to your consumer. The next steps describe how to issue a subscription request on behalf of your consumer.

1. **Obtain WSDL files for the notification broker and subscription manager services.**
   In fact, you will have already done this when publishing WSDL files from the WS-Notification service point, created earlier.
   In addition to the NotificationBroker.wsdl file, you'll also need the SubscriptionManager.wsdl file. This is because in WebSphere Application Server, subscriptions are requested from the notification broker Web service, whereas unsubscriptions are requested from the subscription manager service. Therefore, your subscriber must be a client of both these Web services.
2. **Run wsimport to generate client stubs.**
   You should already have client stubs for the notification broker service from when you wrote your publisher, so we won't reproduce the command here. However, you will need to run `wsimport` to generate client stubs for the subscription manager service, as the following example shows:
   `<WAS>\bin\wsimport -keep -b <WAS>\util\ibm-wsn-jaxb.xml -wsdllocation "WEB-INF/wsdl/SMModule/SubscriptionManager.wsdl" SubscriptionManager.wsdl`
3. **Ensure the generated artifacts and the JAX-WS thin client JAR are in your development class path.**
   As with your publisher and consumer, your subscriber code will also require the generated artifacts and JAX-WS thin client JAR.
4. **Look up the notification broker service.**
   To issue a subscription request, your subscriber will need to obtain a reference to the notification broker service and port. Step 5 of the instructions for creating a publisher described how to do this.
5. **Create a subscription request object and set the consumer reference.**
   Before issuing a subscription request, you need to create a Subscribe object to contain details about the subscription, such as a reference to the consumer to which notification messages should be sent. Listing 11 shows how to achieve this. (Exception handling has been omitted from subscription-related code snippets for clarity.)

   ## Listing 11

   ```
   import org.oasis_open.docs.wsn.b_2.Subscribe;
   import javax.xml.ws.wsaddressing.W3CEndpointReference;
   import javax.xml.ws.wsaddressing.W3CEndpointReferenceBuilder;

   // Create the subscription request object which maps to the Subscribe
   // request. This MUST contain a ConsumerReference and MAY also contain
   // a Filter, an InitialTerminationTime and a SubscriptionPolicy.
   Subscribe subscribeRequest = new Subscribe();

   // Tell the broker to whom the notifications are to be sent:
   W3CEndpointReference consumerReference =
       new W3CEndpointReferenceBuilder()
   .address(consumerURI).build();
   subscribeRequest.setConsumerReference(consumerReference);
   ```

   The consumer endpoint reference object above is constructed using the URI of your consumer Web service's endpoint (denoted by the string consumerURI in Listing 11). For consumers deployed in WebSphere Application Server, a URI can be obtained by publishing WSDL files for the Web module that the consumer service appears in. The WSDL file for your consumer service will include a valid URI for the Web service. Note that this URI should only be used when the consumer is directly accessible from the application server (or servers) where WS-Notification has been configured. If the consumer service will be accessed via an indirect route, such as a proxy, the URI should be updated accordingly.

6. **Define a topic expression as a filter on the subscription.**

   In addition to setting the consumer reference, you might also want to associate a Filter object with the subscribe request. The filter tells the notification broker which messages should be forwarded to the consumer. Messages can be filtered based on topic, message content, or both. This example shows you how to filter messages by topic. In Listing 12, a topic expression, prefix mappings, dialect, and Filter object are defined before being associated with the subscription:

   ## Listing 12

```
import com.ibm.websphere.sib.wsn.jaxb.base.FilterType;
import com.ibm.websphere.sib.wsn.jaxb.base.TopicExpressionType;

// Prepare a topic expression to describe the topic to which we wish to
// subscribe (topicExpression was a String passed into this method):
topicExpression = topicNamespacePrefix + ":" + topicExpression;
TopicExpressionType topicExpressionType = new TopicExpressionType();
topicExpressionType.setExpression(topicExpression);

// Specify the mapping from the namespace prefix to the topic namespace
// URI:
topicExpressionType.addPrefixMapping(
 topicNamespacePrefix,
 topicNamespace);

// Specify the TopicExpression dialect to be used:
topicExpressionType.setDialect(topicDialect);

// Create the filter. This tells the broker which messages are to be
// sent to the consumer.
FilterType filter = new FilterType();

// Add the topic expression to the filter, and set the filter on the
// subscribe request
filter.addTopicExpression(topicExpressionType);
subscribeRequest.setFilter(filter);
```

   Notice that once again you’re using IBM helper classes to simplify your work.

7. **Specify subscription duration.**

   Another important characteristic of a subscription is how long it should last. Your subscription request can indicate this to the notification broker as shown in Listing 13.

   ## Listing 13

```
import javax.xml.bind.JAXBElement;
import javax.xml.datatype.DatatypeFactory;
import javax.xml.datatype.Duration;

// Specify the duration of the subscription from the current time (one
// year hence in this case). It is also possible to specify an absolute
// termination time using javax.xml.datatype.XMLGregorianCalendar
// instead.
DatatypeFactory factory = DatatypeFactory.newInstance();

// See section 3.2.6.1 in XML Schema 1.0 at
// http://www.w3.org/TR/xmlschema-2/#duration
// for an explanation of string representations of duration.
// "P1Y" means one year.
Duration duration = factory.newDuration(”1Y”;

// You can either: (1) new up a JAXBElement directly, like so...
JAXBElement<String> initialTerminationTime =
  new JAXBElement<String>(
     new QName("http://docs.oasis-open.org/wsn/b-2",
     "InitialTerminationTime"), String.class, duration.toString());

// Or: (2) if you prefer to avoid namespace URIs and QNames, you can
// use the relevant JAXB ObjectFactory artefact that was generated by
// wsimport to create the appropriate JAXBElement:
org.oasis_open.docs.wsn.b_2.ObjectFactory objectFactory = new
org.oasis_open.docs.wsn.b_2.ObjectFactory();
initialTerminationTime = objectFactory
 .createSubscribeInitialTerminationTime(duration.toString());

// Whichever way you created 'initialTerminationTime', you can now pass
// it in to the subscribe request object:
subscribeRequest.setInitialTerminationTime(initialTerminationTime);
```

In Listing 13, you see two different ways of defining the initial termination time for the subscription; that is, the time at which your subscriber client wishes the subscription to expire. If you’re not comfortable working directly with namespace URIs and QName objects (option 1 in Listing 13), then feel free to use the convenience factory method provided by the appropriate JAXB ObjectFactory instead (option 2).

8. **Issue the subscribe request.**

   Finally, having built the subscription request, it can be issued to the Web service via the notification broker port:

   ```
   // Send the SubscribeRequest to the broker:
   org.oasis_open.docs.wsn.b_2.SubscribeResponse subscribeResponse =
   port.subscribe(subscribeRequest);
   ```

   You should now have JAX-WS publisher, consumer, and subscriber components, which together enable the publication and consumption of notification messages on particular topics, via the WS-Notification broker Web services provided in WebSphere Application Server V7.

   One final step remains. You could rely on the subscription expiring to prevent messages being delivered to your consumer indefinitely. This approach assumes you specified an initial termination time in the original subscribe request. Alternatively, it is a better practice to terminate the subscription explicitly once your consumer has received all the messages it wants.

9. **Issue an unsubscribe request to stop the delivery of notification messages.**

   To clean up subscription related resources, you simply issue an unsubscribe request for the subscribed consumer to the subscription manager service exposed by the application server. Listing 14 shows how:

Listing 14

```
import com.ibm.websphere.wsn.subscription_manager.SubscriptionManager;
import org.oasis_open.docs.wsn.b_2.UnsubscribeResponse;

// Get the port, remembering to enable WS-Addressing.
SubscriptionManager port = subscribeResponse.getSubscriptionReference()
        .getPort(SubscriptionManager.class, new AddressingFeature());

// Issue the unsubscribe request:
UnsubscribeResponse unsubscribeResponse =
 port.unsubscribe(new Unsubscribe());
```

One thing to notice is that to unsubscribe, you don't look up the subscription manager service in the same way that you looked up the notification broker service to subscribe. Instead, you simply extract the endpoint reference of the subscription manager service from the response that was received when issuing the original subscribe request. (Do make sure you have retained the response object, otherwise you won't be able to use it to unsubscribe!) This endpoint reference has (within its reference parameters) the details of the specific subscription to be destroyed. Once you have obtained the endpoint reference, you can obtain the port for the subscription manager Web service and simply issue the unsubscribe request against it.

By now, you should have a fully working JAX-WS based WS-Notification application, comprising of a publisher to publish notification messages, a consumer to receive them, and a subscriber to create and destroy a subscription on behalf of your consumer. As with the other components, you can find a fully working example subscriber called WSNSubscriber.ear in the download file.

Example scenario: Subscribe, publish, consume, and unsubscribe

The final part of this article describes an example scenario where the sample publisher, consumer, and subscriber provided with this article are used to publish and consume notification messages.

Prepare your system

To use the sample application you'll need to run through the following preparation steps:

1. Install WebSphere Application Server V7 with a single server profile, then start the server.
2. Unzip the download file included with this article and run the example Version 7.0 WS-Notification configuration script by issuing this command:
   `<WAS>\bin\wsadmin -f WAS7_WSN_JAXWS_Part1_Setup.py SIBus WSNService1 WSNServicePt1`
   Remember that configuring a Version 7.0 WS-Notification service point also creates and deploys a Web service enterprise application that must be started before WS-Notification Web services can be invoked. The script above starts this application.
3. Install the three EAR files, also found in the download file, into your application server. Simply accept the default options when installing the applications.
   These files contain the sample publisher, consumer, and subscriber components, along with HTTP servlets that enable you to compose and invoke WS-Notification requests.

4. Start the three enterprise applications.
5. Open a Web browser and create three browser tabs (or open three Web browsers), and direct them to load these URLs:
   `http://localhost:9080/WSNSubscriberWeb/SubscriberServlet`
   `http://localhost:9080/WSNPublisherWeb/PublisherServlet`
   `http://localhost:9080/WSNConsumerWeb/ConsumerServlet`
   The Web pages show input fields and buttons for invoking WS-Notification functionality, as well as the results of any requests, and messages that are published and consumed.

You're now ready to run through the example scenario, which is presented in the next section.

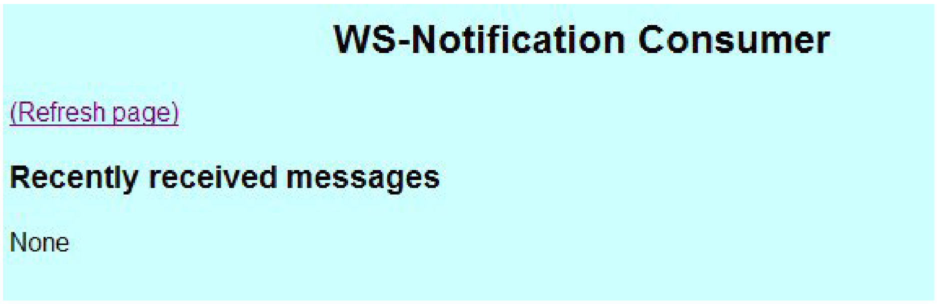
Running the example scenario

The example scenario describes the key steps required to subscribe the sample consumer service to a topic, publish a notification message via the sample publisher, and check that the consumer received the notification. The scenario finishes by describing how to unsubscribe the consumer.

The instructions below include images of the Web pages exposed by the publisher, consumer, and subscriber components of the sample application, the administration console, and captured output from the system output log for an application server. The scenario also includes several optional steps that are included to illustrate the behaviour of the system. In addition to showing some of the details for how the Version 7.0 WS-Notification implementation works, these optional steps might also prove useful when testing your own WS-Notification applications.

1. Open the browser tab with the title **WS-Notification Consumer**, and check that the list of recently received messages says **None** (optional), as shown in Figure 1.

   Figure 1. WS-Notification consumer showing no messages have been received yet

   
   WS-Notification Consumer

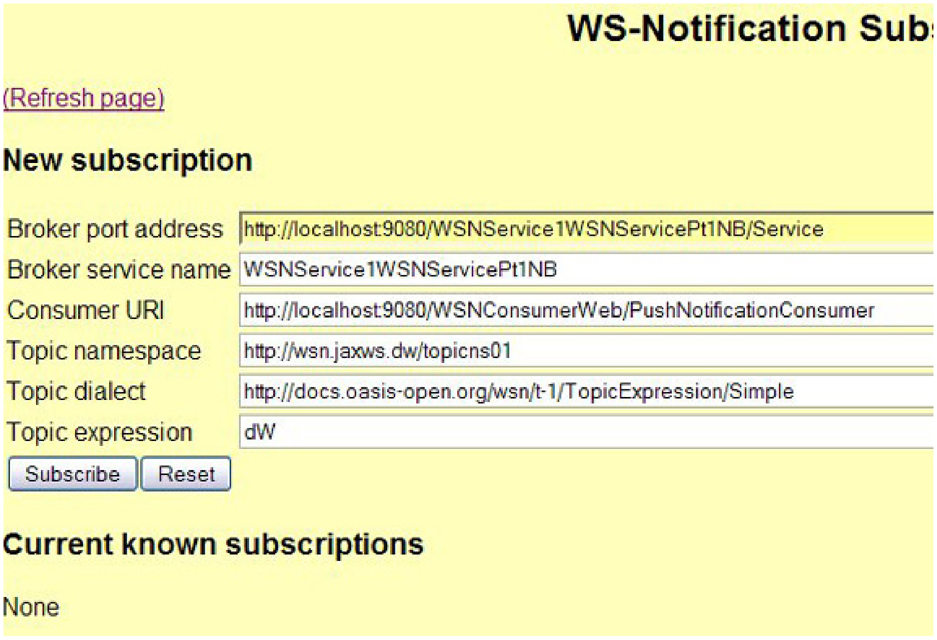
   (Refresh page)

   Recently received messages

   None

   This confirms that the consumer has not received any messages yet, as expected.
2. Open the browser tab titled **WS-Notification Subscriber**, enter values for the port and service related fields, then click the **Subscribe** button. Most subscription-related fields will be populated with valid default values, but you will need to complete the Broker port address and Broker service name fields according to your WS-Notification configuration. Assuming that you've run the script described in the previous section, enter the values shown in Figure 2.

Figure 2. WS-Notification Subscriber prior to issuing a subscription request



WS-Notification Sub

New subscription

Broker port address	http://localhost:9080/WSNService1WSNServicePt1NB/Service
Broker service name	WSNService1WSNServicePt1NB
Consumer URI	http://localhost:9080/WSNConsumerWeb/PushNotificationConsumer
Topic namespace	http://wsn.jaxws.dw/topicns01
Topic dialect	http://docs.oasis-open.org/wsn/t-1/TopicExpression/Simple
Topic expression	dW

Current known subscriptions

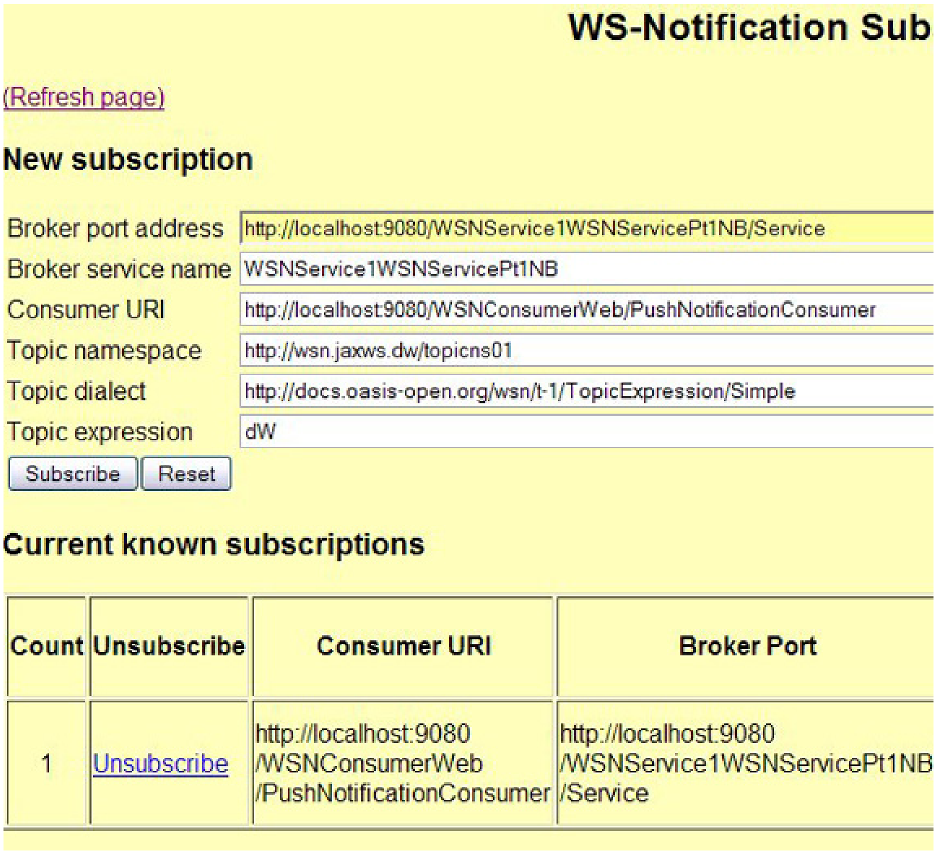
None

You should also check the host and port values shown in the Consumer URI field. However, the default values of localhost and 9080 will be valid for most configurations.

By clicking the **Subscribe** button, you issue a subscription request to the notification broker service and port indicated by the first two fields in the form. This request will include the consumer and topic information provided by the other fields.

3. Examine the result of the subscribe request (optional). Figure 3 shows the **WS-Notification Subscriber** browser tab after a subscribe request has been issued.

Figure 3. WS-Notification Subscriber after issuing a subscription request


WS-Notification Sub

(Refresh page)

New subscription

Broker port address	http://localhost:9080/WSNService1WSNServicePt1NB/Service
Broker service name	WSNService1WSNServicePt1NB
Consumer URI	http://localhost:9080/WSNConsumerWeb/PushNotificationConsumer
Topic namespace	http://wsn.jaxws.dw/topicns01
Topic dialect	http://docs.oasis-open.org/wsn/t-1/TopicExpression/Simple
Topic expression	dW

Current known subscriptions

Count	Unsubscribe	Consumer URI	Broker Port
1	Unsubscribe	http://localhost:9080/WSNConsumerWeb/PushNotificationConsumer	http://localhost:9080/WSNService1WSNServicePt1NB/Service

The **Current known subscriptions** section displays details of the subscription that was made, along with a link through which an unsubscribe request can be issued. (The subscriber component only lists subscriptions that were created through it during the current session. It will not be aware, for example, of any subscriptions created by any other applications.)

4. Examine the system output log for subscription related messages (optional).

   The sample subscriber outputs several diagnostic messages into the system output log, which you might wish to examine. Open the SystemOut.log file, which can be found in the logs \server1 directory of your single application server profile.

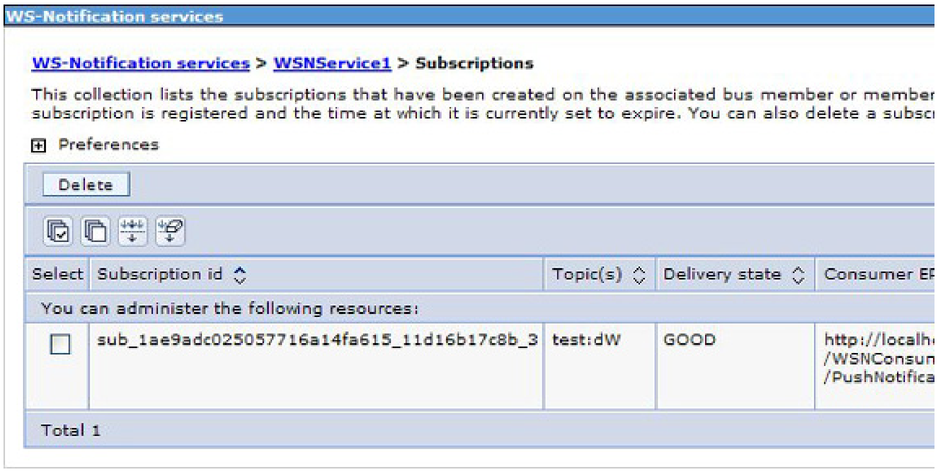
Listing 15

```
[19/10/08 21:00:09:977 BST] 00000025 SystemOut     O Subscribe response details:
[19/10/08 21:00:09:997 BST] 00000025 SystemOut     O Address of SubscriptionManager
port: http://vespa.hursley.ibm.com:9080/WSNService1WSNServicePt1SM/Service
[19/10/08 21:00:09:997 BST] 00000025 SystemOut     O Subscription ID:
sub_1ae9adc025057716a14fa615_11d16b17c8b_3
[19/10/08 21:00:10:007 BST] 00000025 SystemOut     O Current time: 19-10-2008
21:00:09
[19/10/08 21:00:10:007 BST] 00000025 SystemOut     O Termination time: 19-10-2009
21:00:09
```

5. Verify that the subscription exists in the Version 7.0 WS-Notification service (optional).

   Open the administration console for the application server and navigate to **WS-Notification services => WSNService1 => Subscriptions** to see the list of active subscriptions made against the WS-Notification service. (If you can't see the Subscriptions link, make sure you click on the **Runtime** tab after navigating to the page for your WS-Notification service.) You should see one subscription (Figure 4), the details for which should match those of the subscription listed in the **WS-Notification Subscriber** browser tab (Figure 3):

Figure 4. Viewing a subscription in the administration console



WS-Notification services

WS-Notification services > WSNService1 > Subscriptions

This collection lists the subscriptions that have been created on the associated bus member or member
subscription is registered and the time at which it is currently set to expire. You can also delete a subsc

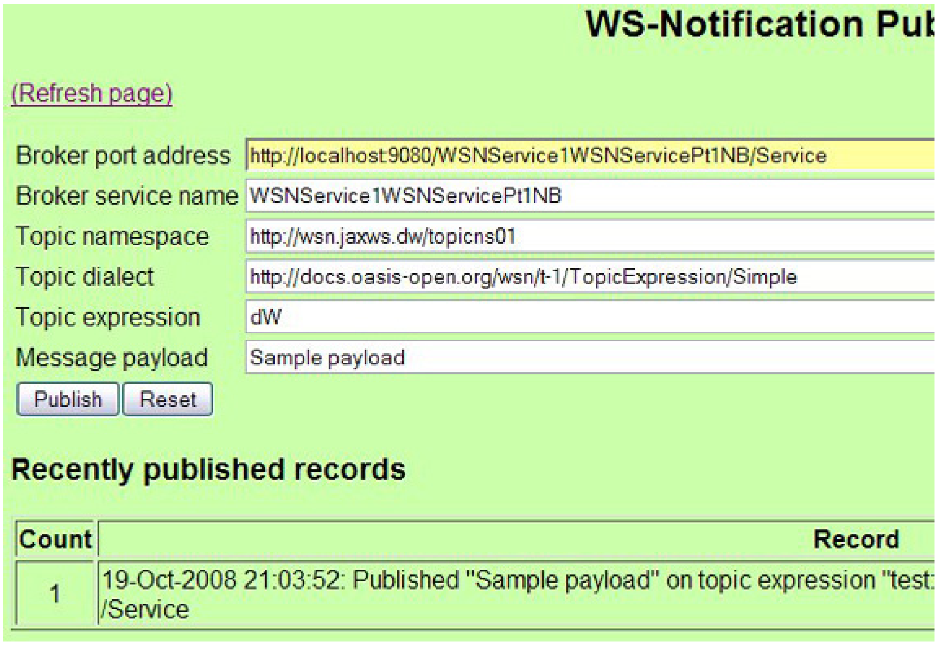
⊞ Preferences

Delete

Select	Subscription id	Topic(s)	Delivery state	Consumer EP
You can administer the following resources:				
☐	sub_1ae9adc025057716a14fa615_11d16b17c8b_3	test:dW	GOOD	http://localh /WSNConsun /PushNotifica
Total 1				

6. Open the **WS-Notification Publisher** tab, enter values for the **Broker port address** and **Broker service name** fields, then click **Publish**.
   This step issues a publication request to the WS-Notification service indicated by the first two fields, using topic and payload information provided by the remaining fields. The topic namespace value must match the corresponding value you used for your subscription request, and the values for your topic dialect and topic expression must also be consistent with the corresponding values used in your subscription request. If you want to send a different payload, simply enter a different string into the Message payload field. Figure 5 shows the WS-Notification Publisher tab after a publication request has been issued.

Figure 5. WS-Notification Publisher following a publication request



WS-Notification Pub

(Refresh page)

Broker port address	http://localhost:9080/WSNService1WSNServicePt1NB/Service
Broker service name	WSNService1WSNServicePt1NB
Topic namespace	http://wsn.jaxws.dw/topicns01
Topic dialect	http://docs.oasis-open.org/wsn/t-1/TopicExpression/Simple
Topic expression	dW
Message payload	Sample payload

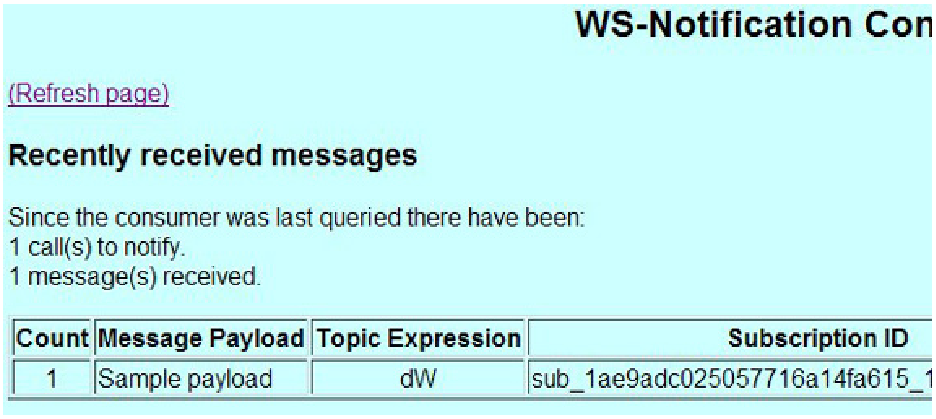
Publish Reset

Recently published records

Count	Record
1	19-Oct-2008 21:03:52: Published "Sample payload" on topic expression "test: /Service

7. Open the **WS-Notification Consumer** tab to verify that the consumer received the message. You will need to refresh the browser to ensure that the latest consumer status is displayed (Figure 6)

Figure 6. WS-Notification Consumer showing the received message



WS-Notification Con

(Refresh page)

Recently received messages

Since the consumer was last queried there have been:
1 call(s) to notify.
1 message(s) received.

Count	Message Payload	Topic Expression	Subscription ID
1	Sample payload	dW	sub_1ae9adc025057716a14fa615_1

Success! The panel shows the details of the notification message that was received, including the payload entered via the WS-Notification Publisher tab.

8. Examine the system output log to verify that the consumer's notify method was invoked (optional). The sample consumer also outputs diagnostic messages to the system output log (Listing 16).

Listing 16

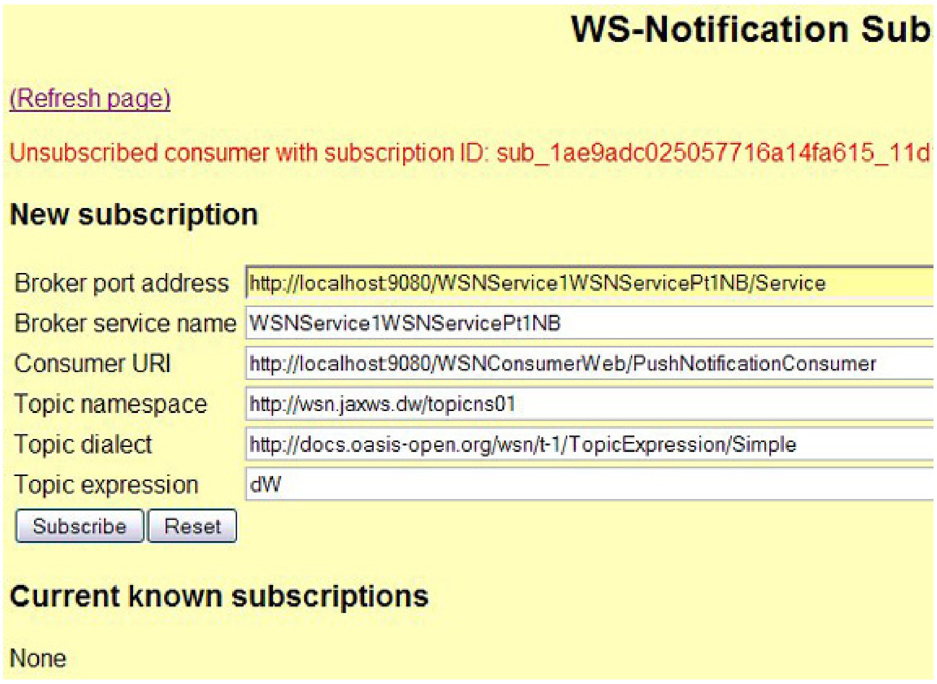
```
[19/10/08 21:03:52:878 BST] 00000035 SystemOut     O The consumer has received 1
message(s).
[19/10/08 21:03:52:878 BST] 00000035 SystemOut     O Details for message 1:
```

```
[19/10/08 21:03:52:878 BST] 00000035 SystemOut     O -- Topic dialect: http://docs.
oasis-open.org/wsn/t-1/TopicExpression/Simple
[19/10/08 21:03:52:878 BST] 00000035 SystemOut     O -- Topic expression, including
the namespace prefix: wsntopprefix:dW
[19/10/08 21:03:52:878 BST] 00000035 SystemOut     O -- Number of namespace mappings
in the topic expression: 2
[19/10/08 21:03:52:878 BST] 00000035 SystemOut     O ---- Prefix 'b2' maps to
namespace 'http://docs.oasis-open.org/wsn/b-2'
[19/10/08 21:03:52:878 BST] 00000035 SystemOut     O ---- Prefix 'wsntopprefix' maps
to namespace 'http://wsn.jaxws.dw/topicns01'
[19/10/08 21:03:52:878 BST] 00000035 SystemOut     O -- Topic expression, without
the prefix: dW
[19/10/08 21:03:52:878 BST] 00000035 SystemOut     O -- Message payload: Sample
payload
[19/10/08 21:03:52:878 BST] 00000035 SystemOut     O -- Payload namespace: http://
wsn.jaxws.dw.ibm.com
[19/10/08 21:03:52:878 BST] 00000035 SystemOut     O -- Payload namespace prefix: dW
[19/10/08 21:03:52:878 BST] 00000035 SystemOut     O -- Payload element name: dWMsg
[19/10/08 21:03:53:288 BST] 00000035 SystemOut     O -- Subscription ID:
sub_1ae9adc025057716a14fa615_11d16b17c8b_3
```

9. To clean up subscription resources, issue an unsubscribe request via the **WS-Notification Subscriber** browser tab.

   Return to the WS-Notification Subscriber browser tab, and then click on the **Unsubscribe** link displayed next to the subscription information. This issues an unsubscribe request to the subscription manager Web service for the previously created subscription. Figure 7 shows the confirmation message that is displayed.

   Figure 7. WS-Notification Subscriber following a successful unsubscribe request



WS-Notification Sub

(Refresh page)

Unsubscribed consumer with subscription ID: sub_1ae9adc025057716a14fa615_11d

New subscription

Broker port address	http://localhost:9080/WSNService1WSNServicePt1NB/Service
Broker service name	WSNService1WSNServicePt1NB
Consumer URI	http://localhost:9080/WSNConsumerWeb/PushNotificationConsumer
Topic namespace	http://wsn.jaxws.dw/topicns01
Topic dialect	http://docs.oasis-open.org/wsn/t-1/TopicExpression/Simple
Topic expression	dW

Current known subscriptions

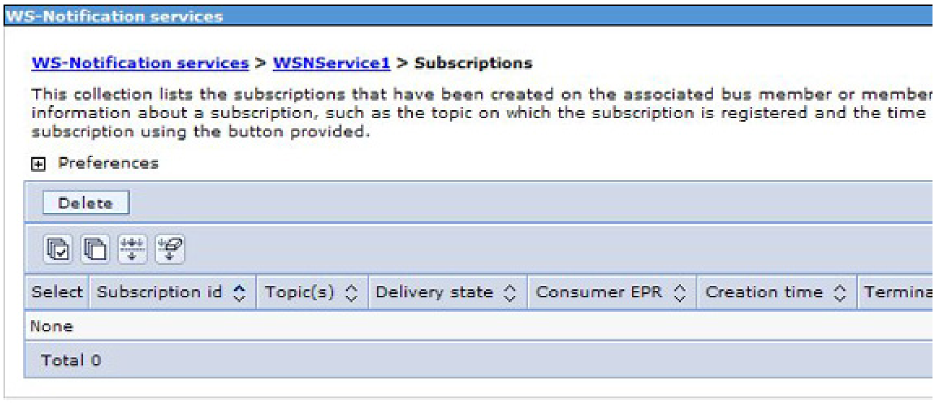
None

10. Examine the system output log to verify that the unsubscription took place (optional).

```
[19/10/08 21:07:53:924 BST] 00000023 SystemOut O Unsubscribed
subscription ID: sub_1ae9adc025057716a14fa615_11d16b17c8b_3
```

11. Verify that the subscription no longer exists in the Version 7.0 WS-Notification service (optional).
    By returning to the administration console and revisiting the Subscriptions panel, you can confirm that the unsubscribe request has been received and processed by the Version 7.0 WS-Notification service (Figure 8).

    Figure 8. Administration console showing that there are no subscriptions



WS-Notification services

WS-Notification services > WSNService1 > Subscriptions

This collection lists the subscriptions that have been created on the associated bus member or member information about a subscription, such as the topic on which the subscription is registered and the time subscription using the button provided.

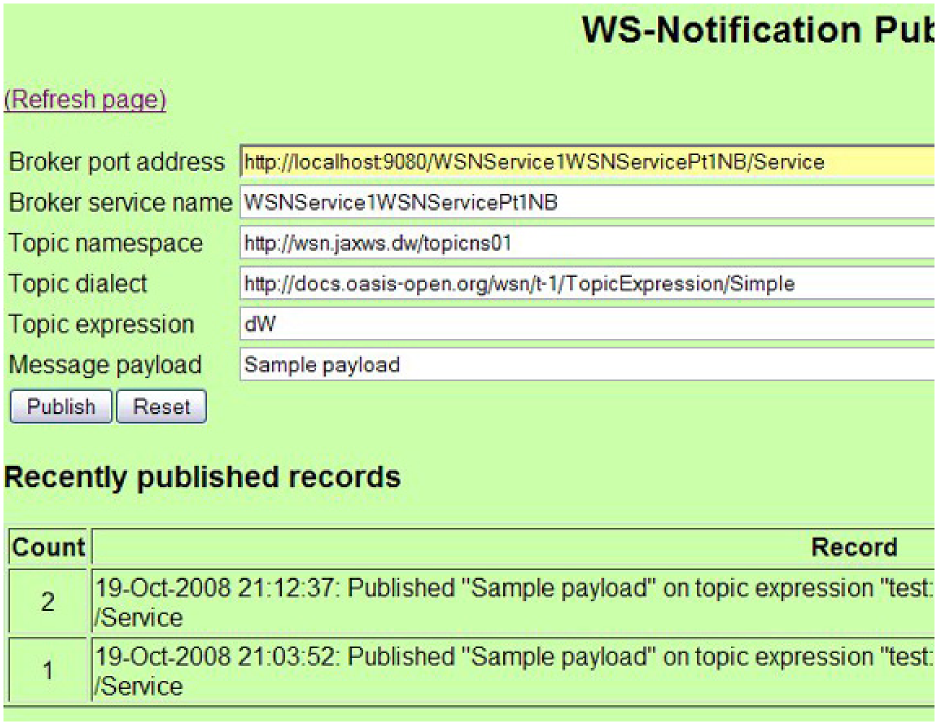
⊞ Preferences

Delete

Select	Subscription id	Topic(s)	Delivery state	Consumer EPR	Creation time	Termina
None						
Total 0						

12. Publish again and check that the consumer does not receive the message (optional).
    As a final verification that the previously created subscription has been destroyed, return to the **WS-Notification Publisher** tab, click **Publish**, then check that the notification message you sent is not shown in the **WS-Notification Consumer** tab (after a refresh), as shown in Figures 9 and 10.

    Figure 9. WS-Notification Publisher following a second publication request



WS-Notification Pub

(Refresh page)

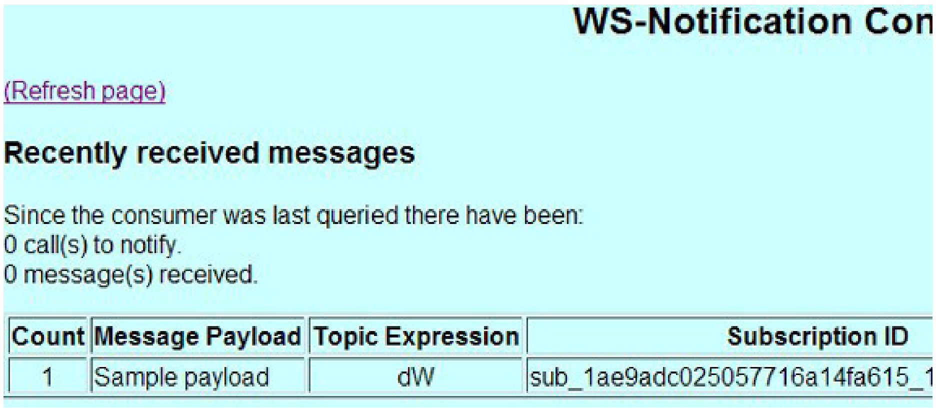
Broker port address	http://localhost:9080/WSNService1WSNServicePt1NB/Service
Broker service name	WSNService1WSNServicePt1NB
Topic namespace	http://wsn.jaxws.dw/topicns01
Topic dialect	http://docs.oasis-open.org/wsn/t-1/TopicExpression/Simple
Topic expression	dW
Message payload	Sample payload

Publish Reset

Recently published records

Count	Record
2	19-Oct-2008 21:12:37: Published "Sample payload" on topic expression "test: /Service
1	19-Oct-2008 21:03:52: Published "Sample payload" on topic expression "test: /Service

Figure 10. WS-Notification Consumer showing that publications are no longer received



WS-Notification Con

(Refresh page)

Recently received messages

Since the consumer was last queried there have been:
0 call(s) to notify.
0 message(s) received.

Count	Message Payload	Topic Expression	Subscription ID
1	Sample payload	dW	sub_1ae9adc025057716a14fa615_1

Well done! You've worked through the whole scenario. Feel free to experiment with the sample components; for example, try different topic dialects and expressions. By looking through the code for the appropriate servlets, you should see that it would be straightforward to add further fields to the input forms if you want to manipulate more of the parameters within the publish or subscribe processes.

Summary

This article explained how to write JAX-WS-based client and service components for use with the new JAX-WS implementation of WS-Notification that is provided in IBM WebSphere Application Server V7.

First, the key steps required to create and configure a JAX-WS, or Version 7.0, type of WS-Notification service were described. Then instructions were provided to help you write JAX-WS based publisher, consumer, and subscriber components. Finally, an example scenario, using a sample script and enterprise application archive files included with this article, was described. This walkthrough scenario showed you how to use the sample application to perform the key WS-Notification operations of subscription, publication, message consumption, and unsubscription.

Having written your JAX-WS Web service clients and services, you're now ready to configure your system for reliable notification. This will be covered in Part 2 of this series.

Acknowledgements

We would like to thank Brian De Pradine, Peter Niblett, and David Illsley for reviewing this article, Chris Whyley for providing sample Jython scripts, and Emily Jiang for sample code that uses the IBM helper classes. David and Brian also provided valuable advice when Lucas wrote the code.

Downloads

Description	Name	Size
Sample application	0811_partridge_JAXWSsampleapp.zip	606 KB

Resources

- WebSphere Application Server V7 Information Center
- Rational Application Developer V7 Information Center
- Developing a JAX-WS client created from a WSDL file
- Developing and Deploying JAX-WS Web services clients
- Generating a Web service client from a WSDL document using the IBM WebSphere JAX-WS runtime environment
- WS-Notification in WebSphere Application Server Version 6.1
- OASIS specifications for WS-Notification
- IBM developerWorks WebSphere Application Server zone

About the authors

Lucas Partridge

Lucas Partridge is a software engineer at IBM Hursley Park in the United Kingdom. He works on WebSphere Application Server.

James M. Siddle

James Siddle is a software engineer at IBM Hursley Park in the United Kingdom. He works on WebSphere brand products such as WebSphere Application Server and WebSphere Business Events.