

## CHAPTER 8

# J2EE and Web Services

This book has discussed in detail how Java and web services fit together. Web services use standards-based frameworks to extend an application's reach. However, a web service isn't the application itself. The web service must still be implemented on a proven application infrastructure—one that supports reliability, availability, serviceability, transactions, security, and other critical enterprise needs. Ultimately, J2EE tries to define just such an infrastructure. Thus, if web services and Java can fit together, and J2EE is the Java form of application infrastructure, the question of how web services fit together with J2EE comes straight to the forefront.

This chapter discusses different approaches of integrating J2EE and web services. How does a web service map into an EJB, a servlet, or J2EE Connector Architecture (CA) adapter? This chapter discusses these topics, looks at some existing standards initiatives, and speculates on what might happen over the next few years.

## The SOAP-J2EE Way

Since SOAP is the cornerstone of interoperability and web services, understanding how J2EE and web services work together comes down to analyzing how SOAP and J2EE can work together. SOAP is a wire protocol that can be layered upon other wire protocols such as HTTP, FTP, and SMTP. J2EE supports these Internet protocols through servlets. Therefore, it makes sense that servlets and JSP technology will become the entry point into a J2EE framework for web services. Let's look at how this occurs.

Within J2EE, servlets, JSPs, EJBs, JMS resources, JDBC drivers, and J2EE CA adapters provide access to the business logic and enterprise resources that a web service needs. Servlets and JSPs are designed to encapsulate page-based flow and logic and can also work with numerous Internet protocols. It makes sense that servlets will become the entry point for web services and an automatic bridge between a web service message and the other J2EE services contained within an application server (see Figure 8-1).

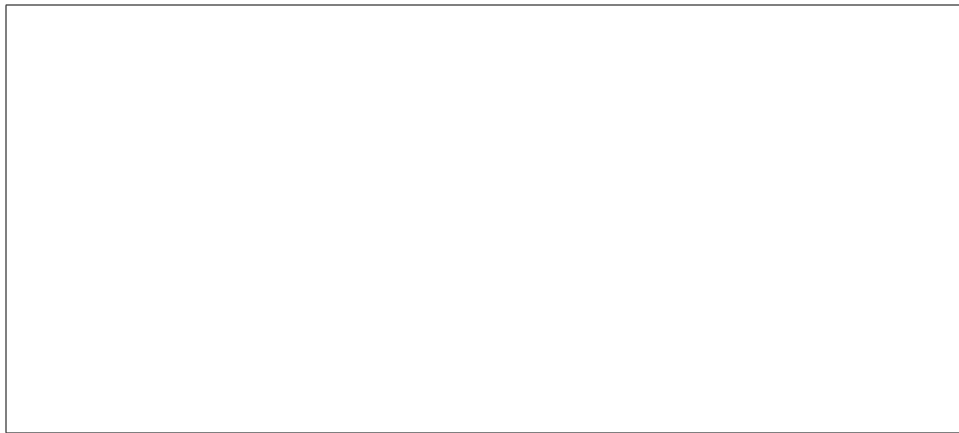


Figure 8-1. SOAP servlet integration

Even though Figure 8-1 presents a simplistic view of SOAP and J2EE integration, a servlet must work hard to make this integration possible. Let's look at some of the issues involved.

## SOAP Parsing

First, servlets are responsible for extracting the SOAP contents from another wire packet. The SOAP contents must then be parsed so the servlet can acquire access to the elements and attributes contained within the SOAP document. A servlet must contain the logic for:

### *Envelope parsing*

The servlet must gain access to the SOAP envelope. The servlet must be able to extract the SOAP header and body portions of the envelope separately.

### *Parsing attachments*

SOAP messages may use attachments to transport payload information. Therefore, the servlet must be able to access all attachments.

### *Validating message format*

If the servlet is the entry point for an endpoint represented by another WSDL file, it must confirm that the format of the SOAP message conforms to the abstract message defined in the corresponding WSDL file. If the incoming message does not conform to the constraints as defined in the corresponding WSDL file, the servlet must create a SOAP Fault message and deliver it to the client.

### *Validating XML*

The SOAP message is in XML format and may be bound to an external XML Schema. The servlet must determine the different namespaces for the elements contained in the SOAP packet and validate the contents of the XML payload against those namespace schema definitions.

### *Rapid XML parsing*

An XML parser that does full validation of elements, attributes, and datatypes won't be the fastest parser on the market. Since many J2EE applications are designed for performance, a servlet has a responsibility to find unique ways to parse the XML payload rapidly. For example, a vendor could implement a parser that specializes in small XML packets if it expects that most SOAP messages will be tiny. Or, if the servlet is invoked by SOAP clients that are trusted to deliver prevalidated messages, the servlet may have a SOAP parser that does only partial validation

### *XML-Java binding*

After an XML packet is parsed and its elements are understood, some data contained within the SOAP payload may need to be converted to a Java object. This conversion might be necessary if the SOAP message is an RPC-style message that requires invocation of an EJB method. EJB interfaces are entirely in Java, so any SOAP payload information that must be delivered as an input parameter needs to be converted to a Java object. The XML-Java binding can be custom binding from an application server provider, or it can take a standards-based approach such as one proposed by JAXB.

### *Payload conversion*

If the SOAP message is a message-style invocation, the XML data contained within the SOAP packet is placed into a JMS destination. The XML data contained within the SOAP message must be converted to a message type that is valid to JMS. This message type could simply be a `BytesMessage`, `TextMessage`, or `ObjectMessage`—or it could be a specialized extended type designed to handle XML and SOAP with Attachments payloads.

### *Explicit versus implicit servlet processing*

The mapping and translation between a SOAP-over-HTTP message and a back-end J2EE component such as an EJB or JMS destination may not be exposed explicitly in the servlet layer. Higher-level layering may implicitly hide that information from the programmer. Or, the servlet API may actually be extended across an RMI or JMS infrastructure and exposed to the service at the ultimate remote destination. We will refer to the servlet layer abstractly, mainly because that layer is easily identifiable in the J2EE architecture diagrams.

## **Behavior Handling**

Based upon how WSDL, JAXM, and JAX-RPC eventually define the behavior of web services, four fundamental types of messages can be transported over SOAP:

- Request/response
- Solicit/response

- One-way
- Notification

The format and behavior of these transmission primitives are described in more detail in Chapter 5. These primitives provide an intriguing technical challenge for application server architects, however, since servlets were primarily designed to handle client-invoked, request-response behavior. A servlet that supports web services obviously needs to support familiar request-response behavior; however, it must also support server-invoked, solicit-response behavior and other server-initiated invocations.

This topic is technically challenging because a servlet must be engineered to receive asynchronous notifications from other resources located in the same application server. Servlets are invoked synchronously by a wire protocol, but to support asynchronous web services, they need to be engineered to receive asynchronous messages while they execute for inclusion on any outgoing response. Additionally, a servlet is generally thought of as something that reacts to inbound requests passively. It must now also be able to actively generate an outbound SOAP-over-HTTP solicitation. Your SOAP provider is responsible for implementing this servlet behavior. Vendor offerings such as BEA WebLogic Server, SonicXQ, CapeClear Studio, and Systinet WASP all provide these implementations to developers transparently.

## Figuring Out What to Invoke

After a servlet parses the SOAP message, it needs to either do an RPC-style invocation or deliver the XML message to another resource, such as a JMS destination. Before the servlet can pass processing on to the next resource, though, it must determine what that resource is. How does it do this? A servlet has some options for determining the next step in the process:

### *SOAPAction header field*

The SOAPAction header field value can contain information that indicates which JMS destination or EJB needs to be invoked. Or, the value of this field can contain the information needed to perform a lookup in a web services routing table that an application server supports.

### *Determine through WSDL*

If the application server knows which WSDL file the incoming message belongs to, it may contain an internal mapping indicating how EJBs and JMS destinations map to SOAP messages. An application server can provide tools for mapping message definitions in a WSDL file to the resources available within an application server.

### *Contained within the SOAP message*

The SOAP header and body may contain proprietary information that helps the servlet determine how to route the message. However, any vendor that decides to use custom tags for routing probably makes its service nonportable as a result.

### Mapping of header information

Converting HTTP header information or SOAP Header elements into JMS properties for server-side filtering may be desirable, particularly if a response to a request will be handled asynchronously or at a later time.

One of the interesting problems posed by this scenario is how an application server should handle conversational web services. SOAP, in its basic form, does not have a standardized way to track session tokens. Session tokens are needed to associate a message with an established conversation. For servlets, conversations are associated with a `sessionID` variable that accompanies every HTTP request. SOAP messages can carry these session tokens, but since no standardized way exists for them to be incorporated into the message, any session enablement of a SOAP message is nonportable. Despite this factor, conversational web services are very important and many vendors are building the extensions needed to support them.

If web services are going to be conversational, then the servlet SOAP handler needs to handle conversations, too. This means that a servlet implementation must figure out whether a message is part of a conversation, where that conversation is managed, and whether any special routing needs to occur as a result of the conversational dynamics.

## RPC-Style Invocations

Mapping web service invocations to EJBs is important because EJBs provide the component framework necessary for implementations of reliable and highly available business logic. EJBs have access to a range of enterprise technologies and provide a portable way to encapsulate the business logic necessary to interact with them. Mapping SOAP to EJBs is important because it allows application developers to continue to develop portal logic solutions for their systems while leveraging the benefits offered by web services. Figure 8-2 depicts the steps involved in generating an EJB invocation from a SOAP message:



Figure 8-2. SOAP-EJB invocation model

1. An inbound SOAP message arrives at a SOAP protocol handler. The SOAP protocol handler parses the message and determines which EJB instance needs to be invoked. This process may or may not involve a JNDI lookup. The reference to the EJB may already be cached within the handler.

2. The SOAP handler invokes the EJB with the appropriate input parameters. The EJB can be a stateless session, a stateful session, or an entity EJB. If the EJB is a stateful session or an entity EJB, the SOAP message must have some way to maintain the session token or primary key of the entity EJB.
3. The EJB can invoke any number of backend resources, including other EJBs, databases, J2EE CA adapters, or JMS destinations. The EJB should contain the meat of the web service implementation.
4. The EJB sends its response to the SOAP handler. The SOAP handler must convert the response from the EJB to XML to be part of the SOAP response payload. If the EJB throws a system exception, then the SOAP handler needs to create a SOAP Fault for delivery to the client.
5. The SOAP handler creates the SOAP envelope for the response and delivers the message to the client.

This process is pretty straightforward. It's so simple, in fact, that most vendors now have toolkits that, given an EJB, perform most of these steps automatically. If you have a stateless session EJB, a SOAP handler generator creates the servlet that serves as the SOAP handler, creates the necessary deployment descriptors, and generates any WSDL necessary to map SOAP messages to an EJB. The code generator also creates any Java-XML binding that needs to occur. This process allows EJB developers to develop EJBs to implement their business logic without having to worry about the semantics of SOAP development. It provides a seamless transition from one paradigm to the other. In the next section, we provide pseudocode that shows how a servlet might integrate SOAP and JMS. That pseudocode can be extended to do the same for integrating SOAP and EJBs.

## Message-Style Invocations

A very important notion for B2B, workflow, and system connectivity is the idea of mapping a WSDL endpoint to a JMS Topic or Queue. The Topic or Queue may have either a Message-Driven Bean or a standalone JMS client registered as a listener.

The interface between a web service and JMS is important because the world out there is “usually connected.” An organization may have many occasionally connected business partners. Legacy systems, which may be designed to move data around via bulk nightly batch processes, need to connect to more modern infrastructures designed to communicate in near-real time. JMS provides reliable asynchronous communication, and thus allows for a loosely coupled distributed architecture in which all parts of a distributed system don't have to run constantly for the whole system to remain healthy. If you recall, one of the defining characteristics of web services is that they are also loosely coupled. Therefore, mapping web services to JMS is natural and important. Figure 8-3 shows the steps required to receive a SOAP message and, as a result, send a JMS message to a JMS destination.



Figure 8-3. SOAP-JMS invocation model

In Figure 8-3, the SOAP-JMS invocation can begin in two ways. First, a SOAP message sent by a client starts a request/response scenario or a one-way invocation that causes a JMS message to be placed onto a destination. In the second model, a message is placed onto a JMS destination monitored by an outbound SOAP handler. Any messages initiated by a JMS client are converted into SOAP messages and delivered using a solicit-response model.

1. An inbound SOAP message arrives at a SOAP protocol handler. The SOAP protocol handler parses the message and determines which JMS destination the message should be delivered to. This step may or may not involve a JNDI lookup.
2. The XML payload is converted to a JMS message and placed onto the appropriate JMS destination. This destination can be a JMS Topic or Queue. The conversion may be a thorough deep mapping into a natively supported JMS message type such as `BytesMessage`, `TextMessage`, or `ObjectMessage`. Alternatively, it may use a vendor-supplied extension that allows the SOAP envelope to be retained in its natural form and dealt with as SOAP throughout its lifespan in the system. HTTP header fields and SOAP header elements need to be preserved as part of the mapping in the event that the message response, or a related descendant of it, is eventually reconstructed as an outbound SOAP-over-HTTP message.
3. A JMS consumer consumes the JMS message and performs any related business logic. Depending on what type of mapping is used to create the message, the consumer may use either JMS APIs directly or API extensions that allow a DOM interface, JAXM APIs, or Apache SOAP APIs.
4. If the SOAP invocation is a one-way, asynchronous invocation, the Message-Driven Bean does not need to deliver a response. However, if the transmission behavior is request-response, the JMS client needs to place the response message onto the JMS destination indicated by the `ReplyTo` field of the original JMS request message.

5. The SOAP protocol handler acts as a message consumer on the response JMS destination. It waits for a response message. When the response is delivered, the SOAP protocol handler converts the JMS response into a SOAP response or fault message.
6. The SOAP protocol handler delivers the message to the SOAP client that initiated the process.
7. If an error occurs during the request/response processing, a SOAP Fault may be generated and sent back to the SOAP client originating the request.
8. If an error occurs during the processing of an asynchronous invocation, a SOAP Fault may be generated and sent to a known Fault destination that is monitored administratively and dealt with at the application level.

The same process occurs for the solicit-response model, except that it is initiated by a backend resource that places a JMS message onto a queue monitored by a SOAP protocol handler. The protocol handler must then wait for responses from the clients to which the SOAP message is delivered. In the case of a one-way or notification model, no response is necessary—the rest of the steps are identical. In fact, a truly asynchronous environment may consist entirely of one-way and notification operations.

## A Simple Example

To demonstrate how integration between web services and J2EE works in practice, we have provided pseudocode that shows how an application server or JMS vendor might create a servlet that receives SOAP messages, parses their content, creates a JMS message, and places that message onto a JMS destination. Since most vendors provide this type of implementation, you will never have to write this code yourself. We provide it to show how things work internally and to give you a better understanding of how web services and J2EE interact:

```
// JAVA PSEUDO-CODE FOR INTEGRATING A SERVLET WITH SOAP & JMS
// This code is designed to receive a SOAP message and then
// to parse the message and to place that message onto a JMS Queue
//
// NOTE: This class is not compilable and does not have proper
//       Exception handling. It is just pseudo-code to demonstrate
//       how Servlet / JMS / SOAP can work together.
//
// This servlet is NOT responsible for receiving JMS messages.
// Web Services are loosely coupled, so a separate servlet is
// responsible for checking to see if a response message has been
// delivered onto the response queue. That receive servlet has to
// be invoked separately to check for the correct contents.
// If the vendor supports asynchronous outbound notifications,
// then polling the servlet for response is not necessary.
//
import javax.jms.*;
import com.vendor.specific.SOAPMessage;
import com.vendor.specific.SOAPMessageFactory;
```

```
public class SoapJMSSendServlet extends HttpServlet {

    static String SEND_QUEUE_NAME = "SomeQueueName";
    static String QUEUE_CONNECTION_FACTORY = "SomeConnectionFactoryName";
    static String SOAP_MESSAGE_FACTORY = "SomeSoapMessageFactoryName";
    private Destination destination;
    private ConnectionFactory factory;
    private QueueConnection qConnect;
    private QueueSession qSession;
    private QueueSender qSender;

    private SOAPMessageFactory smf;
    private SOAPMessage soapMessage, responseMessage;
    private long requestTimeout = 180000; // 30 seconds

    public void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException {

        // Parse SOAP input from HTTP Stream.
        org.w3c.dom.Document doc
            = SOAPParser.getDocument(
                new InputSource(request.getInputStream()));

        // This representative "SOAPMessage"
        // could be a JAXM-based message, and ApacheSoap kind of message, or an
        // Axis message. The important thing to note is that it contains a SOAP
        // envelope along with any other runtime-specific information that allows
        // it to be routed appropriately. The runtime-specific information could
        // be derived from HTTP headers, from SOAP headers, WSDL, or deployment
        // descriptors
        soapMessage = smf.createSOAPMessage(doc, request);

        if (soapMessage.isRPC())
            responseMessage = callEJB(soapMessage);
        else if (soapMessage.isJMSSender())
            responseMessage = sendToQueue(soapMessage);

        // We have to send a SOAP response to the client informing them
        // of the success.
        if (responseMessage != null)
            responseMessage.write(response.getOutputStream());
    }

    // this method converts the SOAPMessage into a JMS message and places it onto
    // a JMS queue. If the message is supposed to be participating in a
    // synchronous request/reply, then it blocks and waits for the reply
    // to be received, or a timer expires - whatever comes first.
    private SOAPMessage sendToQueue(SOAPMessage soapMessage)
        throws ServletException {
        SOAPMessage responseMessage = null;

        // This hypothetical SOAPMessage knows how to marshall itself
        // into a JMS Message
        qSender.send(soapMessage.createJMSMessage());
    }
}
```

```

// This could be a request/response
if (soapMessage.needsReply()){
    // receive blocks until either a message is received,
    // or timeout occurs. If timeout occurs, msg is null
    javax.jms.Message msg = qReceiver.receive(requestTimeout);

    if (msg == null)
        // generate fault. This hypothetical SOAPMessage has a
        // static method for doing that
        responseMessage
            = SOAPMessage.createSOAPFault("Request Timed out");
    else
        // The SOAPMessage can also create a SOAP envelope
        // from a JMS message
        responseMessage = SOAPMessage.createSOAPMessage(msg);
}
else
    responseMessage
        = SOAPMessage.createSOAPMessage("Message Delivered");
return responseMessage;
}

public void init() throws ServletException {
    // Set up Queue -- only needs to occur once.
    InitialContext ctx = new InitialContext();
    factory
        = (ConnectionFactory)ctx.lookup(QueueConnectionFactory);
    destination = (Destination)ctx.lookup(SEND_QUEUE_NAME);
    smf = (SoapMessageFactory)ctx.lookup(SOAP_MESSAGE_FACTORY);
    qConnect = factory.createQueueConnection(username, passwd);
    qSession
        = qConnect.createQueueSession(false,
            javax.jms.Session.AUTO_ACKNOWLEDGE);
    qSender = qs.createSender(destination);
}
}

```

The code provided here is a standard servlet that acts as a JMS message producer. We'll look at a few of the more interesting pieces in detail. Since SOAP over HTTP has just the SOAP payload as part of the HTTP request, the servlet accesses the SOAP envelope by using the `HttpServletRequest` object and the `InputStream` by using the `getInputStream()` method. The contents of this method are then passed into a SOAP parser. In this example, a `SOAPParser` object represents the SOAP parser:

```

// Parse SOAP input from HTTP Stream.
org.w3c.dom.Document doc
    = SOAPParser.getDocument(
        new InputSource(httpRequest.getInputStream()));
soapMessage = smf.createSOAPMessage(doc, request);

```

This representative SOAP message could be based on JAXM, JAX-RPC, Apache SOAP, or Axis. The important thing to note is that it contains a SOAP envelope along with any other runtime-specific information that allows it to be routed appropriately. Since the XML in the SOAP message could represent a message that will be

placed on the JMS queue or topic, an RPC invocation, or something else, our hypothetical SOAP message has an `isRPC()` and an `isJMSSender()` method for determining how to dispatch the message. The dispatching information could be derived from HTTP headers, SOAP headers, WSDL, or deployment descriptors.

```

if (soapMessage.isRPC())
    responseMessage = callEJB(soapMessage);
else if (soapMessage.isJMSSender())
    responseMessage = sendToQueue(soapMessage);

```

The `callEJB()` method simply deserializes the method name and its parameters, invokes the appropriate EJB, and then sends the response back. In J2EE 1.4, this method will be based on the rules defined by JAX-RPC.

The `sendToQueue()` method needs more explanation since it is a bit more complicated. It can send the message and optionally wait for a synchronous reply. We will see the JMS replier side in a moment. For the purpose of our pseudocode example, let's assume that the `createJMSMessage()` creates a JMS message and marshals the SOAP content into it. It doesn't matter what the type is, as long as sufficient APIs are on either side (for accessing the message by using JMS APIs and accessing the XML content by using JAXP or SOAP Envelope APIs). The JMS message is created and sent to its destination:

```

private SOAPMessage sendToQueue(SOAPMessage soapMessage)
    throws ServletException {
    SOAPMessage responseMessage = null;

    // This fictitious SOAPMessage knows how to marshall itself
    // into a JMS Message
    qSender.send(soapMessage.createJMSMessage());
}

```

Next, the SOAP message also knows enough about itself to determine whether it should be an asynchronous `send()` or a synchronous request/reply operation. If a reply is expected, the `QueueReceiver.receive()` blocks until a message is returned or a timeout occurs. The timeout is important because we must satisfy the HTTP request with either a response message or a Fault message before the HTTP request itself times out:

```

// This could be a request/response
if (soapMessage.needsReply()){
    // receive blocks until either a message is received,
    // or timeout occurs. If timeout occurs, msg is null
    javax.jms.Message msg = qReceiver.receive(requestTimeout);

    if (msg == null)
        // generate fault. This hypothetical SOAPMessage has a
        // static method for doing that
        responseMessage
            = SOAPMessage.createSOAPFault("Request Timed out");
    else
        // The SOAPMessage can also create a SOAP envelope
        // from a JMS message
        responseMessage = SOAPMessage.createSOAPMessage(msg);
}

```

```

else
    responseMessage
        = SOAPMessage.createSOAPMessage("Message Delivered");
    return responseMessage;
}

```

In JMS, an asynchronous response might result from this message. However, the “response” may not happen immediately. It may occur hours or days after the request, or the “response” may be just another message delivered to a new destination that is not related to the SOAP client that initiated the request. This servlet is responsible only for retrieving any JMS response messages that might be created for an immediate synchronous response. To handle the other situations, there are other possibilities:

- A separate servlet could be established to act as a holding mechanism for responses. In this scenario, a SOAP client sends to one servlet and queries another servlet at a later point to pick up any responses.
- If the vendor supports asynchronous outbound notifications, then polling the servlet for response messages is not necessary. The responses are converted to SOAP-over-HTTP messages as they are created, and delivered in real time to the receiving party.

### The JMS replier

The JMS replier also uses our hypothetical SOAP message.\* It first looks to see if the message contains an attachment. If so, it uses JavaMail Multipart API’s to get at the first part of the message, which is the SOAP envelope:

```

public class PsuedoJMSSoapReplier
    implements javax.jms.MessageListener
{
    // ...
    public void onMessage( javax.jms.Message aMessage)
    {
        if (aMessage instanceof SOAPMessage)
        {
            SOAPMessage sMsg = (SOAPMessage)aMessage;

            int partCount = sMsg.getPartCount();
            //Display number of parts in multipart message
            System.out.println("MultipartMessage received with : "
                + partCount + " parts");

            Part part = null;
            for(int i = 0; i < partCount; i++)
            {

```

\* This example is derived from Sonic Software’s SonicXQ. The original code is included with the online example archive and is available with the SonicXQ product.

```

        part = sMsg.getPart(i);
        System.out.println("Multipart message part(" + i
            + ") has content type: "
            + part.getHeader().getContentType());
        System.out.println("Multipart message part(" + i
            + ") has size: "
            + part.getContentBytes().length + " bytes");
    }
    org.w3c.dom.Document doc
        = SOAPParser.getDocument(
            new InputSource(sMsg.getPart(0).getInputStream()));

    if (doc == null)
    {
        throw new org.apache.soap.SOAPException (
            org.apache.soap.Constants.FAULT_CODE_CLIENT, "parsing error");
    }

    //Validate the XMLMessage content using Apache SOAP
    org.apache.soap.Envelope msgEnv =
        org.apache.soap.Envelope.unmarshall (doc.getDocumentElement());

    //... Do SOAP Envelope things....

    System.out.println("Successfully processed MultipartMessage");
}

```

The destination could be either a response or another destination where one could forward the message. Any number of mechanisms can be used—the JMSReplyTo destination, for example, or a WSDL endpoint—to determine which destination to go to next.

```

        // Check for a ReplyTo Queue and send one if necessary
        javax.jms.Queue replyQueue = (javax.jms.Queue) aMessage.getJMSReplyTo();
        if (replyQueue != null){
            // create reply message and send it.
        }else{
            // create a new SOAP message to go to another destination
            String urlEndpoint = sMsg.getNextURLEndpoint();
            ...
        }
    }
}

```

## Content-Based Routing, Data Transformation, and the J2EE Connector Architecture

In an enterprise environment, the processing of SOAP requests often doesn't fit a simple request/response model involving a simple servlet interaction; the servlet model doesn't necessarily work well for a multistep workflow web service. In this section, we will describe how J2EE technologies such as servlets, JMS, and EJB can

be used together with XML-related technologies to orchestrate a multistep complex business process that supports the implementation of a web service.

Consider a company that exposes its business interface to its customers as a web service. It regularly receives purchase orders for its goods from any number of prospective buyers. Figure 8-4 shows the complex chain of events that occurs when a purchase order is received. The purchase order may come into the system as a SOAP message over the HTTP protocol. Upon arriving at the servlet or the SOAP processor, the message may be sent immediately to a routing service that uses an XPATH expression such as `/PurchaseOrder/Items` to identify that it is indeed a purchase order. If the routing service uses a parser that supports JavaScript extensions, then additional logic may be plugged into the routing service to do some real-time expression evaluation.

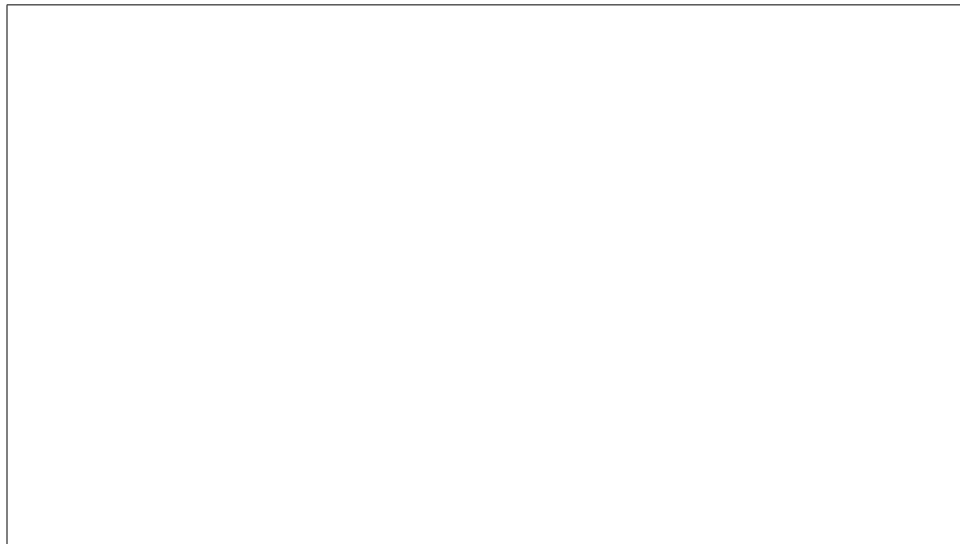


Figure 8-4. Content-based routing, XSLT transformation, and J2EE connectors

The routing service may then determine that this purchase order is valid and needs to proceed to the next step in the process—performing a credit authorization. It sends the purchase order on to the next step by placing a message onto a JMS queue. The credit authorization service listening on that queue may run some business logic and choose to delegate its credit checking responsibility to another external web service. To accomplish this delegation, it may make a direct SOAP call over an HTTP connection or choose to create a new JMS message and place it on a queue destination that is mapped to an outbound SOAP-over-HTTP handler. If that request for credit approval is denied, then a “credit denied” message is sent back to the sender, either synchronously or asynchronously, depending on how the whole service is described in WSDL.

If the credit check is successful, the next step may be to execute some business logic by invoking an EJB—directly as a synchronous call or asynchronously by sending a message to a message-driven bean listening on a queue. The business logic might break the purchase order into its respective line items and send each item to the legacy fulfillment application (EIS) that resides at the remote warehouse. In the process, several things need to happen: the purchase order’s line items must be broken into separate requests, the data must be converted from XML into the proprietary format that is used by the EIS, and the logic must communicate with the EIS in a standardized way.

The breaking of `<Items>` into multiple entries and the transformation from XML into the legacy format required by the EIS are both accomplished by routing the purchase order to a special XSLT-based translation service. Given the appropriate style sheet, we can rely on the parser to do most of the work.

The next step, communicating with the EIS, is accomplished by using a connector, as defined by the J2EE Connector Architecture. In Figure 8-4, the implementation of the connector may reside in the local container, at the remote location, or it may be spread across both places. After the processing is complete, a final SOAP response message may be generated to travel back to the sender, using similar processes.

## JSR109: Industry in Flux

Unfortunately, although integration between J2EE and web services has matured among many of the providers, the Java Community Process (JCP) has not yet standardized it. JSR109, “Implementing Enterprise Web Services,” is intended to provide a standardized approach for J2EE/web services integration. Unfortunately, as of early 2002, JSR109 has stalled, and may not be delivered to the public until the middle of 2002. This means that any vendor implementations following the standard probably won’t be fully realized until 2003 or 2004. That is a very long time, given the pace at which technology tends to mature.

One positive, note, however, is the expectation that the contents of JSR109 will be incorporated into J2EE 1.4, forcing application server vendors to provide a standardized level of web services support by 2003.

JSR109 is expected to cover several things. Everything is in flux at this point, but some of the highlights appear to be:

### *Mapping rules*

The main portion of the specification will focus on rules for mapping SOAP and WSDL onto J2EE 1.3 components such as EJBs and JMS destinations.

### *JNDI lookup rules*

The specification will highlight how J2EE services bound into a JNDI tree map to services described via WSDL. This process will allow a SOAP protocol handler that has the WSDL of a web service and access to a J2EE JNDI directory to connect SOAP messages with J2EE services.

### *WSDL binding extensions for JMS and EJB*

Given a JMS destination or an EJB (and a WSDL file describing abstract messages), how should those messages be mapped onto the interfaces of J2EE components? For example, if a WSDL file has a `DeliverMe` message, is it supposed to map to a `deliverMe()` method on an EJB's remote interface, local interface, or the `DeliverMe` JMS destination?

### *WSDL generation rules*

Given the deployment descriptor of an EJB, what policies should be followed for the automatic generation of WSDL from the deployment descriptor?

### *JAX API integration*

The charter of the JSR should eventually have the specification define the official role of JAX-RPC, JAXR, and JAXM with J2EE.

### *Service publication*

What services developed with the J2EE framework will be published to a UDDI registry, and how will a J2EE framework automate the publication process?

### *Security, transactions, and management*

How do the web services standards that are trying to add security, transaction contexts, and management get mapped to the equivalent services offered in an application server? For example, if a SOAP message has a transaction context associated with it, what are the semantics for associating the SOAP message with an existing J2EE transaction or for starting a new J2EE transaction?

## **The Java Web Service (JWS) Standard**

Not to be confused with the topic of this book, a newly proposed standard called the Java web service (JWS) standard is currently in development. It is spearheaded by BEA Systems, which also has a reference implementation.

The JWS is a format designed to integrate non-Java developers with J2EE. Sounds ambitious, doesn't it? BEA has actually designed a technology that might work. At the core of the JWS specification is the idea that developers don't create J2EE components. Rather, developers create a web service, and a single Java class represents web service implementation. The Java class then has a number of simple, predefined JavaDoc tags that indicate different behavioral implementations of the web service. Based on the values of the JavaDoc tags inserted into the Java class, a behind-the-scenes code generator then creates all necessary J2EE components required to implement the web service.

The JWS JavaDoc system has tags representing a full range of web service behaviors, including stateless methods, stateful methods, and asynchronous invocations. The challenge left to JWS implementations is to take the definition of the JavaDoc tags and generate J2EE components that implement this behavior in a reliable and available manner.

The JWS proposal is appealing because tool vendors can support BEA's prototype implementation quickly. It comes with a nice IDE that ties together design, coding, and testing. The concept of deployment is completely hidden from the developer. The goal is to have a framework for developing web services with J2EE that is similar to working in Visual Basic.

Let's look at an example. The following listing, *HelloWorld.jws*, shows all of the code necessary to create a complete web service:

```
import com.bea.jws.*;
import com.bea.jws.control.*;

/**
 * A simple web service that returns a string.
 */
public class HelloWorld extends Service
{
    /**
     * @operation
     * @conversation stateless
     */
    public String getHelloWorld()
    {
        return "Hello World";
    }
}
```

That's it. A JWS file is simply a Java class that implements the *Service* interface. The methods of the class implement the web service; the *JavaDoc* extensions tell the behind-the-scenes code generator what type of web service to produce. The methods in the Java class may not be exposed as part of the WSDL of the web service. If the method you create should be part of the external WSDL, then the *@operation* tag should be placed in the *JavaDoc* before the method. This tag causes the code generator to create necessary definitions in the WSDL file.

The *@conversation* tag defines the behavior of this method as part of the web service. The *@conversation* tag takes a single parameter that can be *stateless*, *start*, *continue*, or *finish*. The *stateless* parameter means that this method supports only the request/response paradigm. If the *@conversation* parameter is *start*, *continue*, or *finish*, then the underlying generated infrastructure is responsible for associating incoming messages with a conversation handler for each client. If the *start* parameter is used, the infrastructure needs to create a new session when the method is invoked. If the *continue* parameter is used, the infrastructure needs to determine which existing session this client belongs to and make sure the method invocation occurs in that context. Finally, if the *finish* parameter is used, the infrastructure executes the method in the appropriate session context and ends the session afterwards.

JWS can be slightly more complicated. JWS has JavaDoc tags for defining different aspects of parameters, methods, and the class itself. These parameters include:

*Asynchronicity*

A method can be made asynchronous with a simple JavaDoc tag. Clients can implement a polling model to get a result or the web service can do a notification.

*Buffering*

If the @buffer tag is used on any method, a JMS destination is inserted between any SOAP messages and the web service implementation. Buffering the messages in JMS provides better scalability and is transparent to the developer.

*Controls*

The JWS specification has a simple way to represent EJBs, JMS destinations, J2EE CA adapters, databases, and other web services through a simple Java interface, called a Java Web Interface (JWI) file. An enterprise developer who creates one of these resources is expected to provide a JWI file that is merely a Java interface with JavaDoc tags. JWS implementations can then access any resource exposed through JWI either visually or via a straightforward Java invocation. JWI files allow tools to hide all J2EE semantics from developers and expose these components and services as simple Java interfaces. Given this factor, any developer with a cursory knowledge of Java can reuse complex services.

*XML maps*

JWS provides an extension to ECMAScript that maps the fields of Java objects to XML (and vice versa). A developer with a cursory understanding of JavaScript or VBScript can take the XML format of parameters and map them into the fields of the Java objects used as input parameters on the web service without knowing the details of how the Java objects are structured.

The simplicity of the JWS standard enables a large body of developers to use J2EE, including many who do not have enough experience with Java or J2EE to be successful on their own terms. The number of companies that choose to adopt the standard will certainly factor into the success of JWS. However, this standard doesn't necessarily have to be adopted by an application server vendor. A JWS code engine must create J2EE-compliant code, so the generated web service can operate in any J2EE-certified application server. Adoption of JWS will probably occur at the tool level and will be driven by whether other tool vendors support the standard or add extensions to it.