Basic SOA Using REST

In this chapter, I describe the basic tools and techniques for implementing SOA components using the REST paradigm. REST stands for Representational State Transfer. It was first introduced by Roy Fielding[1] in his 2000 doctoral dissertation [Fielding]. For the past several years, a great debate has been going on about the merits of the REST versus SOAP architectural styles for Web Services. It is not my intention, in this book, to weigh in on either side of that debate. My feeling is that both approaches are useful for implementing SOA components. For simple applications, REST is an easy way to get started.

If you are an advanced Java programmer, you might find the first half of this chapter to be very basic. I have intentionally started out with simplistic examples, using only HTTP and servlets, so that readers who are not advanced in Java can come up the learning curve and get a sense of the basics before introducing the Java Web Services (JWS) APIs. If you have a good grounding in HTTP and Java servlets, please feel free to skip the introductory material and focus on the sections dealing with JAX-WS.

3.1   Why REST?

Some readers may wonder why this book starts with REST before discussing SOAP and WSDL-based Web Services. The reason is that REST is easy to understand. By starting with REST, I can describe some of the basic SOA Web Services concepts without getting into the complexities of SOAP and WSDL. Also, the limitations of REST provide the motivation for introducing SOAP and WSDL in Chapter 4. If you are not interested in REST, feel free to skip ahead to Chapter 4.

---

1. Fielding is one of the principal authors of the HTTP specification and a co-founder of the Apache HTTP Server project.

3.1.1 What Is REST?

REST-style services (i.e., RESTful services) adhere to a set of constraints and architectural principles that include the following:

- RESTful services are *stateless*. As Fielding writes in Section 5.1.3 of his thesis, "each request from client to server must contain all the information necessary to understand the request, and cannot take advantage of any stored context on the server."
- RESTful services have a uniform interface. This constraint is usually taken to mean that the only allowed operations are the HTTP operations: GET, POST, PUT, and DELETE.
- REST-based architectures are built from *resources* (pieces of information) that are uniquely identified by URIs. For example, in a RESTful purchasing system, each purchase order has a unique URI.
- REST components manipulate resources by exchanging *representations* of the resources. For example, a purchase order resource can be represented by an XML document. Within a RESTful purchasing system, a purchase order might be updated by posting an XML document containing the changed purchase order to its URI.

Fielding writes that "REST-based architectures communicate primarily through the transfer of representations of resources" (Section 5.3.3). This is fundamentally different from the Remote Procedure Call (RPC) approach that encapsulates the notion of invoking a procedure on the remote server. Hence, RPC messages typically contain information about the procedure to be invoked or action to be taken. This information is referred to as a *verb* in a Web service request. In the REST model, the only verbs allowed are GET, POST, PUT, and DELETE. In the RPC approach, typically many operations are invoked at the same URI. This is to be contrasted with the REST approach of having a unique URI for each resource.

These are the basic principles behind REST. However, when people talk about the benefits of RESTful systems today, they usually are not strictly applying these principles. For example, among REST advocates, keeping shopping cart data on the server and maintaining a session related to the shopping process that is using the cart is acceptable.[2] In fact, the XML/HTTP Binding provided by JAX-WS for implementing RESTful

---

2. Storing session information or shopping cart data on the server is a clear violation of Fielding's original REST concept since it violates the requirement that a service be stateless.

services provides for session management capabilities using cookies, URL rewriting, and SSL session IDs.

More significant deviations from Fielding's definition of REST involve getting around the "uniform interface" constraint by embedding verbs and parameters inside URLs. The Amazon.com REST interface, for example, includes verbs in query strings and doesn't have unique URIs for each resource. Systems like this, although labeled as RESTful, are really starting to look very much like RPC using XML over HTTP without SOAP.

For the purposes of this book, I am not going to wade into a debate on what is or isn't RESTful. I simply define RESTful Web Services in contrast to SOAP Web Services. Table 3–1 illustrates the principal differences.

Table 3–1 RESTful Web Services versus SOAP Web Services

	REST	SOAP
Message Format	XML	XML inside a SOAP Envelope
Interface Definition	none[a]	WSDL
Transport	HTTP	HTTP, FTP, MIME, JMS, SMTP, etc.

a. Some would argue that XML Schema could be used as an interface definition for RESTful services. Not only is that approach possible, but it is used in many practical cases. However, it is not a complete interface solution because many, if not most, RESTful services incorporate HTTP parameters (e.g., URL query strings) in addition to XML as part of their invocation interface. Chapter 9 looks at the Yahoo! Shopping RESTful interface, which uses HTTP parameters in this manner.

This is consistent with common usage in the REST versus SOAP debates. REST uses simple XML over HTTP without a WSDL interface definition.

3.1.2 Topics Covered in This Chapter

In addition to introducing RESTful Web Services, this chapter introduces and reviews some basic techniques for integrating Enterprise Information Systems (EISs) using XML, XSLT, HTTP, and Java. For each example, I demonstrate how to implement it with and without JWS. The versions of the examples without JWS use basic Java HTTP and XML techniques. Both approaches are provided to give you a sense of what is really happening, under the covers, when a Web service is consumed or deployed using JWS. This should give you a better understanding of the mechanisms underlying JWS and when to use them. For simple Web services, often it is easier to

work with the basic Java tools than to pull out all the power of JWS. On the other hand, you will see from these examples how things can quickly get complicated and require the power of the JWS technologies.

Since one focus of this book is on SOA-style development for the enterprise, many of the examples deal with EIS—the basic infrastructure of most corporate computing environments. This chapter describes

- Structuring EIS Records as XML documents
- Getting EIS records from a REST service (with and without JWS)
- Posting EIS records to a REST service (with and without JWS)
- Basic SOA-style integration of REST services using XSLT for data transformation
- Deploying a REST service to be used for getting EIS records—in other words, an HTTP GET service (with and without JWS)
- Deploying a REST service to be used for posting EIS records—in other words, an HTTP POST service (with and without JWS)

3.2   XML Documents and Schema for EIS Records

The first step toward implementing an SOA component that consumes or provides EIS records involves formatting the EIS records that need to be exchanged as XML documents. This process is formalized by creating an XML Schema to represent the structure of an XML document for a particular EIS record. This section introduces some simple examples that are used throughout this chapter to illustrate the role of XML and XML Schema in SOA-style applications development based on Web Services. Understanding these examples requires a basic knowledge of XML and XML Schema. If you are new to XML, you should get an introductory text such as *Beginning XML* by David Hunter et al. [Hunter]. For the necessary background on XML Schema, I suggest *Definitive XML Schema* by Priscilla Walmsley [Walmsley]. Alternatively, if you know basic XML, but need to brush up on XML Schema, you can probably find all you need to know for this book by reading through the W3C's "XML Schema Part 0: Primer" [XSD Part 0].

To illustrate how XML is used, I employ an example based on the fictitious XYZ Corporation. The example illustrates real SOA challenges faced by many companies. XYZ Corporation has an Order Management System (OMS) that needs to be integrated with a Customer Service System (CSS). The OMS should be thought of as an EIS, such as SAP, for taking customer

orders and tracking them through delivery. The CSS should be thought of as an EIS, such as Oracle's Siebel Customer Relationship Management Applications, that is used by customer service employees as a tool for handling customer inquiries.

XYZ Corporation would like to build an SOA application bridging the OMS and the CSS. Every time a new order is entered in the OMS (or an existing order is updated), the new SOA application should transfer that information to the CSS and add it to the relevant customer's history log. The purpose of this SOA application is to ensure that customer service representatives have fast access, through the CSS, to basic customer order information. If customer service representatives need access to more detailed order information from the OMS, the CSS will contain the keys within the customer history log (updated via the SOA application) to query the OMS and access that detailed information.

Figure 3–1 illustrates what an OMS order record looks like as it might appear on a user interface.

Figure 3–1 An OMS order record as it appears in the user interface.

The structure displayed in the user interface provides a guide to constructing an XML document for the EIS order record. Note that the record is divided into four sections that contain data: Order Number, Order Header, Order Items, and Other Information. Example 3–1 illustrates how this record can be represented as an XML document.

Example 3–1 An XML Representation of the Order Record Appearing in Figure 3–1

```
 4  <Order xmlns="http://www.example.com/oms"
 5    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 6    xsi:schemaLocation="http://www.example.com/oms
 7    http://soabook.com/example/oms/orders.xsd">
 8    <OrderKey>ENT1234567</OrderKey>
 9    <OrderHeader>
10      <SALES_ORG>NE</SALES_ORG>
11      <PURCH_DATE>2005-12-09</PURCH_DATE>
12      <CUST_NO>ENT0072123</CUST_NO>
13      <PYMT_METH>PO</PYMT_METH>
14      <PURCH_ORD_NO>PO-72123-0007</PURCH_ORD_NO>
15      <WAR_DEL_DATE>2005-12-16</WAR_DEL_DATE>
16    </OrderHeader>
17    <OrderItems>
18      <item>
19        <ITM_NUMBER>012345</ITM_NUMBER>
20        <STORAGE_LOC>NE02</STORAGE_LOC>
21        <TARGET_QTY>50</TARGET_QTY>
22        <TARGET_UOM>CNT</TARGET_UOM>
23        <PRICE_PER_UOM>7.95</PRICE_PER_UOM>
24        <SHORT_TEXT>7 mm Teflon Gasket</SHORT_TEXT>
25      </item>
26      <item>
27        <ITM_NUMBER>543210</ITM_NUMBER>
28        <TARGET_QTY>5</TARGET_QTY>
29        <TARGET_UOM>KG</TARGET_UOM>
30        <PRICE_PER_UOM>12.58</PRICE_PER_UOM>
31        <SHORT_TEXT>Lithium grease with PTFE/Teflon</SHORT_TEXT>
32      </item>
33    </OrderItems>
34    <OrderText>This order is a rush.</OrderText>
35  </Order>
```

book-code/chap03/eisrecords/src/xml/order.xml

Note the use of namespaces in this example. The `Order` element is from the namespace `http://www.example.com/oms`. Note that `http://www.example.com` is the URL used by XYZ Corporation as the base part of its corporate namespaces. The `/oms` indicates, more specifically, the namespace associated with the OMS. When developing SOA systems with XML, it is important to use namespaces, because documents originating from different systems may use the same tags (e.g., "item"), and it is important to interpret the tag in the proper context. For more information on namespaces and how they are used, see the World Wide Web Consortium's (W3C) Recommendation [Namespaces in XML].

In addition to namespaces, when developing SOA systems based on XML, it is important to employ XML Schema to validate documents. Just as a relational database management system allows you to impose constraints on data values and format within the database schema, so XML Schema can be used to validate the integrity of XML documents. XML Schema is important for maintaining data quality and integrity when sharing information among multiple systems.

Notice that the `Order` element of `order.xml` contains the attribute:

```
xsi:schemaLocation="http://www.example.com/oms
  http://soabook.com/example/oms/orders.xsd"
```

This attribute associates `order.xml` with an XML Schema and contains two references. First, `http://www.example.com/oms` gives the namespace to be used for interpreting the schema. Second, `http://soabook.com/example/oms/orders.xsd` is a location where the schema can be found.

Example 3–2 shows just a fragment[3] of the schema used to validate `order.xml`. As indicated by the file reference printed at the bottom of the example, the entire schema document can be found at `com/javector/chap4/eisrecords/order.xsd`. This schema (`order.xsd`) and its instance document (the `order.xml` file) are simplified examples of the SAP XML interface for the business object `SalesOrder` within the Logistics Module.

Although this example is simplified, it illustrates the major issues faced when creating an SOA application that accesses SAP or another EIS.

---

3. Because fragments published in this book correspond directly to the source files in the accompanying download package, sometimes—for XML documents—the closing tags get cut off. Although that can sometimes make the structure look confusing or "off balance," I decided that was better than including the entire XML file in cases where the length could run on for several pages.

Example 3–2 A Fragment of the XML Schema for Validating an Order Document

```
4   <schema targetNamespace="http://www.example.com/oms"
5     xmlns="http://www.w3.org/2001/XMLSchema"
6     xmlns:oms="http://www.example.com/oms" version="1.0"
7     elementFormDefault="qualified">
8     <element name="Orders" type="oms:OrdersType"/>
9     <element name="Order" type="oms:OrderType"/>
10    <complexType name="OrdersType">
11      <sequence>
12        <element ref="oms:Order" maxOccurs="unbounded"/>
13      </sequence>
14    </complexType>
15    <complexType name="OrderType">
16      <annotation>
17        <documentation>A Customer Order</documentation>
18      </annotation>
19      <sequence>
20        <element name="OrderKey">
21          <annotation>
22            <documentation>
23    Unique Sales Document Identifier
24            </documentation>
25          </annotation>
26          <simpleType>
27            <restriction base="string">
28              <maxLength value="10"/>
29            </restriction>
30          </simpleType>
31        </element>
32        <element name="OrderHeader" type="oms:BUSOBJ_HEADER">
33          <annotation>
34            <documentation>
35    Order Header referencing customer, payment, sale organization information.
36            </documentation>
37          </annotation>
38        </element>
39        <element name="OrderItems">
40          <annotation>
41            <documentation>Items in the Order</documentation>
42          </annotation>
43          <complexType>
44            <sequence>
45              <element name="item" type="oms:BUSOBJ_ITEM"
```

```
46                   maxOccurs="unbounded"/>
47              </sequence>
48           </complexType>
49        </element>
```

book–code/chap03/eisrecords/src/xml/orders.xsd

Notice that schemas allow you to restrict values and specify formats for data. For example, the element `OrderKey`, that is the unique identifier for sales documents, is restricted to being, at most, 10 characters in length. The restriction is accomplished using the `restriction` element in the simple type definition of `OrderKey`. Restrictions on simple types like this are known as facets. For further explanation of simple type facets, see [XSD Part 0]. Facets are an important data quality management tool in an SOA environment because they enable you to ensure that the data being shared across systems is properly formatted and can be interpreted by the receiving system.

Next, Figure 3–2 shows the Customer History Record from the Customer Service System (CSS). Consider how it relates to orders and how it is used within the CSS.
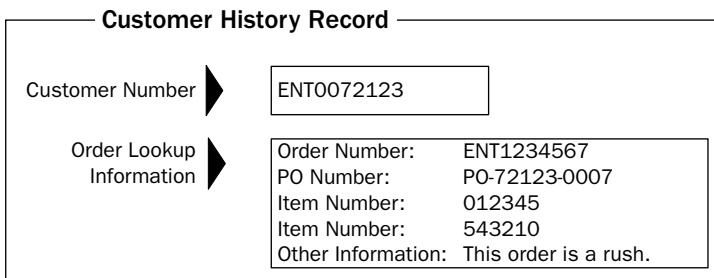

Figure 3–2 A Customer History Record as it appears on a CSS form.

The simple SOA application described in this chapter is responsible for linking the OMS to the CSS to ensure that each time an order is created or modified in the OMS, a corresponding Customer History Record is sent and entered in the Customer History Log within the CSS. The Customer History Log is a record of all transactions the customer has had with XYZ Corporation. It is important to note that not all of the order information is stored in the Customer History Log within the CSS. Only enough is stored so that if a customer calls with a question about an order, the customer service

representative can pull up the History Log and drill down to individual order records stored in the OMS to answer the customer's questions. Individual, detailed order records are retrieved from the OMS in real time using the keys stored in the CSS.

The architecture is designed this way to avoid storing lots of redundant data in the CSS and OMS. The problem with redundant data is that it takes up unnecessary disk space, and tends to get out of sync with the original data, creating data quality problems that can be quite difficult to debug and clean up.

The form in Figure 3–2 shows the minimal set of information that needs to be moved from the OMS to the CSS. Example 3–3 shows how that information is structured as an XML record. The OMS sends this type of XML to the CSS each time there is a new order.

Example 3–3 XML Representation of the Screen Pictured in Figure 3–2

```
 4  <css:CustomerHistoryEntry xmlns:css="http://www.example.com/css"
 5    xmlns="http://www.example.com/css"
 6    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 7    xsi:schemaLocation="http://www.example.com/css
 8    http://soabook.com/example/css/custhistentries.xsd">
 9    <CustomerNumber>ENT0072123</CustomerNumber>
10    <OrderLookupInfo>
11      <OrderNumber>ENT1234567</OrderNumber>
12      <PURCH_ORD_NO>PO-72123-0007</PURCH_ORD_NO>
13      <ITM_NUMBER>012345</ITM_NUMBER>
14      <ITM_NUMBER>543210</ITM_NUMBER>
15      <OrderText>This order is a rush.</OrderText>
16    </OrderLookupInfo>
17  </css:CustomerHistoryEntry>
```

```
book-code/chap03/eisrecords/src/xml/custhistentry.xml
```

In this example, the `CustomerNumber` element uniquely identifies the customer and is referenced by the `CUST_NO` element, inside the `Order-Header` element illustrated in Figure 3–1. Likewise, the `OrderNumber` element inside the `OrderLookupInfo` element referenced the `OrderKey` element illustrated in Figure 3–1. These constraints could be enforced within the schema by using the `unique`, `key`, and `keyref` XML Schema Elements (see Section 5 of [XSD Part 0]). However, for simplicity, those types of constraints are left out for now.

Example 3–4 shows a fragment of the schema for validating the Customer History Record. This schema is important for validating the quality of the data being reformatted and exchanged by the SOA application bridge.

Example 3–4 XML Schema for an Entry in the CSS Customer History Log

```
4   <schema targetNamespace="http://www.example.com/css"
5     xmlns="http://www.w3.org/2001/XMLSchema"
6     xmlns:css="http://www.example.com/css" version="1.0"
7     elementFormDefault="qualified">
8     <element name="CustomerHistoryEntries"
        type="css:CustomerHistoryEntriesType"/>
9     <element name="CustomerHistoryEntry" type="css:CustomerHistoryEntryType"/>
10    <complexType name="CustomerHistoryEntriesType">
11      <sequence>
12        <element ref="css:CustomerHistoryEntry" maxOccurs="unbounded"/>
13      </sequence>
14    </complexType>
15    <complexType name="CustomerHistoryEntryType">
16      <sequence>
17        <element name="CustomerNumber">
18          <annotation>
19            <documentation>Unique Customer Identifier</documentation>
20          </annotation>
21          <simpleType>
22            <restriction base="string">
23              <maxLength value="10"/>
24            </restriction>
25          </simpleType>
26        </element>
27        <element name="OrderLookupInfo">
28          <annotation>
29            <documentation>Keys and searchable text that can be used to look
30              up additional order information from the OMS</documentation>
31          </annotation>
32          <complexType>
33            <sequence>
34              <element name="OrderNumber">
35                <annotation>
36                  <documentation>Unique Sales Order Identifier – Key for CSS
37                    lookup of order records</documentation>
38                </annotation>
39                <simpleType>
```

```
40                    <restriction base="string">
41                      <maxLength value="10"/>
42                    </restriction>
43                  </simpleType>
44                </element>
```

book–code/chap03/eisrecords/src/xml/custhistentries.xsd

As in Example 3–2, you can see the facets are used to restrict the values for `CustomerNumber` and `OrderNumber`. Notice that schemas allow us to restrict values and specify formats for data. For example, the element `OrderKey`, that is the unique identifier for sales documents, is restricted to being, at most, 10 characters in length. The restriction is accomplished using the restriction element in the simple type definition of `OrderKey`.

3.2.1 No WSDL Doesn't Necessarily Mean No Interfaces

The previous examples show how XML Schema can be used to define structure for XML documents. That structure is critical for defining how applications interact with each other in an SOA-style Web Services infrastructure. It is a fundamental principle of systems integration design that applications must interact across well-defined interfaces. Even in this simple example, as illustrated in Example 3–4, you can see how XML Schema can be used to define the structure of an update record to the CSS Customer History Log. The Customer History schema defines the interface to the Customer History Log. In this manner, any system that needs to update the CSS with customer activity can map whatever form their data is in to the `custhistentry.xsd` schema and send it as a message to the CSS. The following two concepts, illustrated by the simple examples in this section, provide the foundation for SOA-style integration using Web Services:

1. XML documents are used to exchange messages between applications.
2. XML Schema documents define the application interfaces.

Point (2) is important to bear in mind. It tells you that, even when using RESTful services (without WSDL and SOAP), the schema of the XML documents being exchanged between SOA components can be used to define interfaces for the services. Unfortunately, this use of XML Schema to formalize interfaces for RESTful services is not universally accepted as part of

the REST framework.[4] If you plan to use REST for your SOA applications, however, I strongly encourage you to provide XML Schema to define the message structure interface for each service.

This section gave you a quick introduction to how EIS records can be represented as XML documents. In the next section, we begin to look at the basics of messaging—sending XML over the Hypertext Transfer Protocol (HTTP).

3.3   REST Clients with and without JWS

A basic capability you often need when implementing an SOA Web service is easily downloading and uploading XML files from/to an HTTP server. For example, suppose that the OMS runs a nightly batch job and writes all new and changed orders from the previous day to a set of XML documents that can be accessed using a Web service named `NewOrders`. The CSS can then, each morning, retrieve those files and update its Customer History. This is a simple, but common and highly practical, form of SOA-style loosely coupled integration.

The next few sections focus on uploading/downloading XML documents with bare-bones RESTful Web Services. I show how to write clients for RESTful services with and without JWS. This material may seem very basic to advanced Java programmers, but it is always good to review the basics before diving into a complex subject like SOA with Java Web Services.

It is a common misconception that implementing Web Services with Java requires lots of heavy machinery like JAX-WS. This is not the case, as even J2SE 1.4 provides powerful tools for HTTP communication and XML processing that enable you to build and consume RESTful Web services. JAX-WS and the other JWS tools provide many advantages, of course, which we discuss later in this section and throughout the rest of this book.

Doing REST without JWS gives you a hands-on appreciation for what HTTP can and cannot do. It quickly becomes clear that, although it is easy to do simple things without the JWS machinery, as you get more ambitious, you start to need some more powerful tools to handle invocation, serialization, and the other components of an SOA Web Services infrastructure.

Since this is a book about Java, I start with the assumption that the EISs have Java APIs for accessing the needed records. The challenge addressed

---

4. Perhaps this is because such interface definitions complicate the REST model, and REST proponents like to position it as simpler than SOAP.

in the next few sections is to deploy a Java API as a Web service or to invoke a Web service using Java.

3.3.1 Getting EIS Records from a REST Service without Using JWS

This section briefly examines how to get an XML document from a REST-ful Web service. In this example, the Web service is accessed with an HTTP GET request. The client application needs to issue the HTTP GET, and process the HTTP response stream that contains the XML document. Instead of using the JWS APIs (e.g., JAX-WS), I simply use the `javax.net.HttpURLConnection` class to handle most of the work related to generating the HTTP GET and processing the response.
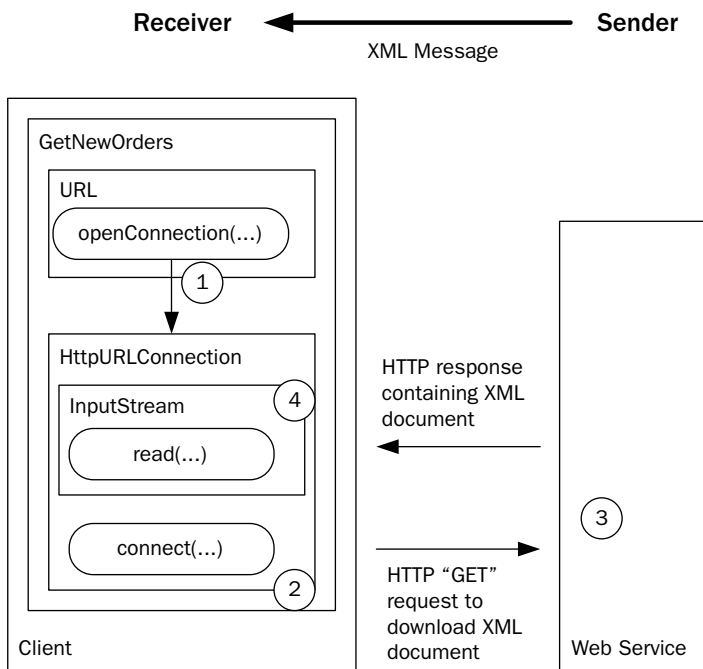

Figure 3–3 The client uses the `HttpURLConnection` class to make an HTTP GET request and receive an HTTP response.

Figure 3–3 illustrates the XML transfer process from the client's side. Note that the client is implemented as the class `GetNewOrders` and that it uses instances of the classes `URL` and `HttpURLConnection`. The following

steps show how these classes work together to implement the XML docu-
ment download protocol.

1. The client uses the `URL.openConnection()` method to create an
   instance of `HttpURLConnection` representing a connection to the
   Web service's `URL`.
2. `HttpURLConnection.connect()` sends the HTTP GET request that
   has been configured using the Web service's URL.
3. The Web service processes the request and writes the appropriate
   XML document to the HTTP response stream.
4. The `HttpURLConnection`'s `InputStream` is used to read the HTTP
   response's XML document.

In the implementation of this example, the client simply writes the
XML document to the console. You will see it print out on your console
when you run it (instructions follow). In a real SOA-style loosely coupled
application, the document might be parsed, transformed, and sent to
another component of the distributed application. An example of such pro-
cessing is provided in Section 3.4.

Example 3–5 shows the client-side code for issuing the HTTP GET
request and receiving the XML document via the HTTP response. Notice
that the `String` used to construct the URL instance is passed to the client as
`args[0]`. The `HttpULRConnection`—con—doesn't send the HTTP request
until its `connect()` method gets invoked. Before this happens, the `setRe-`
`questMethod()` is invoked to specify that a GET request should be sent.

Example 3–5 Implementing a Java Client to Download an XML Document from
a RESTful Web Service

```
27    public static void main(String[] args) throws Exception {

28

29      if (args.length != 1) {
30        System.err.println
31        ("Usage: java GetNewOrders <Web Service URL>");
32        System.exit(1);
33      }
34      // Create the HTTP connection to the URL
35      URL url = new URL(args[0]);
36      HttpURLConnection con =
37        (HttpURLConnection) url.openConnection();
38      con.setRequestMethod("GET");
39      con.connect();
```

```
40        // write the XML from the input stream to standard out
41        InputStream in = con.getInputStream();
42        byte[] b = new byte[1024];  // 1K buffer
43        int result = in.read(b);
44        while (result != -1) {
45          System.out.write(b,0,result);
46          result =in.read(b);
47        }
48        in.close();
49        con.disconnect();
50      }
```

book-code/chap03/rest-get/client-http/src/java/samples/GetNewOrders.java

To run this example, do the following:

1. Start GlassFish (if it is not already running).
2. Go to `<book-code>/chap03/rest-get/endpoint-servlet`.
3. To build and deploy the Web service enter:

```
mvn install 5
```

... and when that command finishes, enter:

```
ant deploy
```

4. Go to `<book-code>/chap03/rest-get/client-http`.
5. To run the client enter:

```
mvn install
```

6. To undeploy the Web service, go back to `<book-code>/chap03/rest-get/endpoint-servlet` and enter:

```
ant undeploy
```

In this example, the `HttpURLConnection` class does all the work. It sends the HTTP GET request to the Web service URL[6] and provides access to the response as an `InputStream`. Now, let's look at how this is done using JWS.

---

5. `mvn` is the command to run Maven, the build tool used throughout this book. See Appendix B, Software Configuration Guide, for details about installing and running the examples in this book.

6. In this example, the URL where the RESTful service deployed is http://localhost:8080/rest-get-servlet/NewOrders (assuming your Java EE Web container is running on localhost:8080). You can always see what parameters are used to invoke code in the examples by examining the build.xml file from the directory where you are running the example.

3.3.2 Getting EIS Records from a REST Service with JWS

When using the JAX-WS 2.0 API to create a client that consumes a REST-ful Web service, the `javax.xml.ws.Dispatch<T>`[7] interface does most of the work—performing a role similar to `HttpURLConnection`.
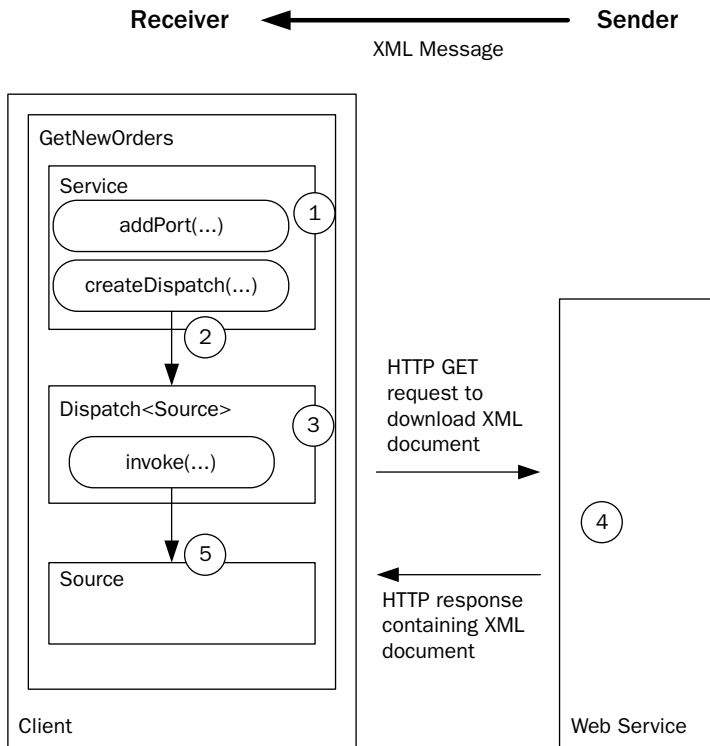

Figure 3–4 The JAX-WS-based client uses an instance of `Dispatch<Source>` to make an HTTP GET request and receive an HTTP response.

Figure 3–4 shows how the JAX-WS-based client works. This time, the class `GetNewOrders` is implemented using the `javax.ws.xml.Service` class. `Service` is an abstraction that provides a client interface to a Web service. First introduced in JAX-RPC 1.0, `Service` is designed to represent a WSDL defined service. Since RESTful services do not have WSDL

---

7. A detailed discussion of the `javax.xml.ws.Dispatch<T>` interface is provided in Chapter 6, Section 6.2, where the JAX-WS client API is explained in depth.

representations, the `Service` API is a little awkward for our purposes here (as you can see in the discussion of the code). However, that is how the JAX-WS API is designed.

In this example, `Service` is used to create an instance of `javax.xml.ws.Dispatch<Source>`, which enables XML message-level interaction with the target Web service. `Dispatch` is the low-level JAX-WS 2.0 API that requires clients to construct messages by working directly with the XML, rather than with a higher-level binding such as JAXB 2.0 schema-derived program elements. For many REST proponents, however, this is exactly the programming paradigm they want—direct access to the XML request and response messages.

The following steps trace the execution of the JAX-WS version of the `GetNewOrders` client illustrated in Figure 3–4.

1. The client uses the `Service.addPort()` method to create a port within the `Service` instance that can be used to access the RESTful Web service.
2. Next, the `Service.createDispatch()` method is invoked to create an instance of `Dispatch<Source>`—a `Dispatch` instance that enables you to work with XML request/response messages as instances of `javax.xml.transform.Source`.
3. The `Dispatch.invoke()` method then packages the XML request—per the JAX-WS 2.0 HTTP Binding—and sends it to the RESTful service. The `invoke()` method waits for the response before returning.
4. The service processes the HTTP GET and sends an HTTP response that includes the XML.
5. The `invoke()` method returns the response XML message as an instance of `Source`.

Example 3–6 shows the code used to implement the JAX-WS version of `GetNewOrders`. Browsing through this code, you can see some of the awkwardness that comes from applying the WSDL-oriented `Service` API in a REST context. First, notice that you have to create `QName` instances for the `Service` instance and the "port" that corresponds to the RESTful Web service. In a SOAP scenario, these qualified names would correspond to the WSDL definitions for the `wsdl:service` and `wsdl:port`. Since there is no WSDL when invoking a RESTful service, these `QName` instances are gratuitous in this example. They are required by the API, but not used to invoke the RESTful service.

Example 3–6 The `GetNewOrders` Client As Implemented with JAX-WS

```
35    public static void main(String[] args) throws Exception {
36      if (args.length != 1) {
37        System.err.println
38        ("Usage: java GetNewOrders <Web Service URL>");
39        System.exit(1);
40      }
41      QName svcQName = new QName("http://sample", "svc");
42      QName portQName = new QName("http://sample", "port");
43      Service svc = Service.create(svcQName);
44      svc.addPort(portQName, HTTPBinding.HTTP_BINDING, args[0]);
45      Dispatch<Source> dis =
46        svc.createDispatch(portQName, Source.class, Service.Mode.PAYLOAD);
47      Map<String, Object> requestContext = dis.getRequestContext();
48      requestContext.put(MessageContext.HTTP_REQUEST_METHOD, "GET");
49      Source result = dis.invoke(null);
50      try {
51        TransformerFactory.newInstance().newTransformer()
52        .transform(result, new StreamResult(System.out));
53      } catch (Exception e) {
54        throw new IOException(e.getMessage());
55      }
56    }
```

book-code/chap03/rest-get/client-jaxws/src/java/samples/GetNewOrders.java

To run this example, do the following:

1. Start GlassFish (if it is not already running).

2. Go to `<book-code>/chap03/rest-get/endpoint-servlet`.

3. To build and deploy the Web service enter:

```
mvn install
```

... and when that command finishes, then enter:

```
ant deploy
```

4. Go to `<book-code>/chap03/rest-get/client-jaxws`.

5. To run the client enter:

```
mvn install
```

6. To undeploy the Web service, go back to `<book-code>/chap03/rest-get/endpoint-servlet` and enter:

```
ant undeploy
```

Looking at the code in Example 3–6, you can also see that the `addPort()` method takes a URI parameter that defines the transport binding. The default is SOAP over HTTP, but in this case, the `HTTPBinding.HTTP_BINDING` URI is used to specify the JAX-WS 2.0 HTTP Binding. The final parameter passed to the `addPort()` method is the URL of the RESTful Web service—in this case, `args[0]`.

Once the port for the RESTful service has been added by the `addPort()` method, it can be used to create an instance of `Dispatch<Source>`. The type parameter—in this case, `Source`—is passed as a parameter to the `createDispatch()` method. The type of payload is specified as well. Here, I have specified `Service.Mode.PAYLOAD` (as opposed to `Service.Mode.MESSAGE`). When working with SOAP, the MESSAGE mode indicates that you want to work with the entire SOAP envelope as opposed to just the SOAP body (or payload). In the REST scenario, there is no envelope, so PAYLOAD is the option that makes sense.

Besides `Source`, the other valid type parameters for `Dispatch` are `JAXB` objects, `java.xml.soap.SOAPMessage`, and `javax.activation.DataSource`. In Chapter 6,[8] I look at examples using `JAXB` and `SOAPMessage`. `DataSource` enables clients to work with MIME-typed messages—a scenario I don't cover in this book.

How does this implementation compare with the `HttpURLConnection` version illustrated in Example 3–5? Table 3–2 illustrates some similarities and differences.

Table 3–2 `HttpURLConnection` versus JAX-WS

	HttpURLConnection Version	**JAX-WS Version**
Representation of the RESTful Web service	`java.net.URL`	`javax.xml.ws.Service`
Invocation Object	`java.net.HttpURLConnection`	`javax.xml.ws.Dispatch`
Message Form	`java.io.InputStream`	`javax.xml.transform.Source`

As you can see, the JAX-WS version gives us a much richer interface, although in a simple REST scenario like this, it is not always that useful. A URL is an adequate representation of a RESTful service since there is no

---

8. Chapter 6 is a detailed overview of the JAX-WS client-side API.

associated WSDL to provide further definition anyway. About the only other information that is needed is whether to use HTTP POST or HTTP GET. As we have seen, `Service` is really designed to be a Java representation of WSDL, so it's not particularly helpful here.

The `Dispatch` interface, on the other hand, is better suited for working with XML request/response than `HttpURLConnection`. First, its `invoke()` method captures the request/response semantics of the HTTP Binding better than the `HttpURLConnection.connect()` method. Second, rather than reading and writing streams, the `Dispatch` interface enables us to work directly with XML representations such as `Source`. This is much more natural, as we will see when we start linking RESTful services together and using XSLT for data transformation (Section 3.4).

Having looked at clients that get XML from a RESTful Web service, the next two sections show how to send XML to such a service. These sections also demonstrate how to pass parameters to the RESTful service as part of the URL.

3.3.3 Sending EIS Records to a REST Service without Using JWS

In Section 3.3.1, I used `HttpURLConnection` to implement a "pull" architecture for XML document messaging—in other words, the document is "pulled" by the client from the Web service. In a "pull" architecture, the receiver of the XML document initiates the transfer. In the code example provided, we used the HTTP GET method to serve as the request mechanism.

Sending XML to a RESTful Web service is not much different from getting XML. The main differences for the "push" architecture are that the sender initiates the transfer and the HTTP POST method is used. This architecture is used, for example, when the OMS wants to upload all new and changed orders to the CSS on a regular basis. To implement such an upload process, the CSS would need to provide a Web service where the OMS could post XML documents.

In this example, I implement a "push" architecture using `HttpURLConnection`.

Figure 3–5 illustrates push messaging. As you can see, it is similar to the previous example, except that the client is now the sender of messages and the Web service is the receiver. The following steps are illustrated in the figure:

1. The client uses the `URL.openConnection()` method to create an instance of `HttpURLConnection` representing a connection to the

RESTful Web service's URL. In this example, the URL has a parameter: `SourceSystem=OMS`. This parameter indicates that the XML document being sent comes from the "OMS" system.

2. `HttpURLConnection.connect()` begins the HTTP POST request to the Web service's URL.

3. The client writes the XML document to the HTTP request stream, essentially appending it to the POST request that has been started.

4. The RESTful service processes the HTTP POST.

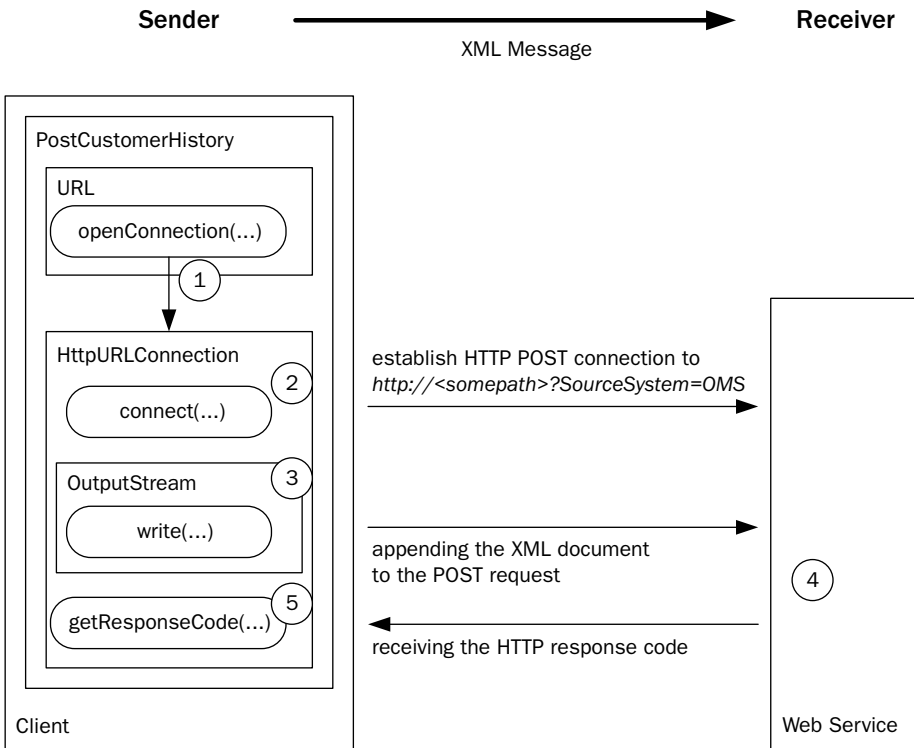5. The response—a simple "200 OK"—is sent back to the client indicating that the document has been received.

Figure 3–5 Push messaging with HTTP POST.

Example 3–7 shows the client-side code for implementing the HTTP POST with an `HttpULRConnection`. Notice that we use `setRequest-Method("POST")` to configure the HTTP request as a POST. After that, the `connect()` method initiates the HTTP request, and the remaining code writes the XML document (specified by the filename `args[0]`) to the output stream.

Example 3–7 Client Uses POST to Upload an XML Document to the Web Service

```
28   public static void main(String[] args) throws Exception {
29     if (args.length != 2) {
30       System.err.println
31         ("Usage: java PostCustomerHistory <XML file name> "
32          + "<Web Service URL>");
33       System.exit(1);
34     }
35     FileInputStream in = new FileInputStream(args[0]);
36     URL url = new URL(args[1]);
37     HttpURLConnection con =
38       (HttpURLConnection) url.openConnection();
39     con.setDoOutput(true);
40     con.setRequestMethod("POST");
41     con.connect();
42     OutputStream out = con.getOutputStream();
43     // write the XML doc from file to the HTTP connection
44     byte[] b = new byte[1024];  // 1K buffer
45     int result = in.read(b);
46     while (result != -1) {
47       out.write(b,0,result);
48       result = in.read(b);
49     }
50     out.close();
51     in.close();
52     // write HTTP response to console
53     System.out.println(con.getResponseCode() +
54                        " " + con.getResponseMessage());
55   }
```

book-code/chap03/rest-post/client-http/src/java/samples/
  PostCustomerHistory.java

To run this example, do the following:

1. Start GlassFish (if it is not already running).
2. Go to `<book-code>/chap03/rest-post/endpoint-servlet`.
3. To build and deploy the Web service enter:

   `mvn install`

   ... and when that command finishes, then enter:

   `ant deploy`

4. Go to `<book-code>/chap03/rest-post/client-http`.

5. To run the client enter:

    ```
    mvn install
    ```

6. To undeploy the Web service, go back to `<book-code>/chap03/rest-post/endpoint-servlet` and enter:

    ```
    ant undeploy
    ```

What you can't see here is the form of the URL that is passed in as `args[1]`. To see the URL being used, you can look at the <book-code>/chap03/rest-post/endpoint-servlet/build.xml file containing the goal used to invoke this service. You will see that the URL has the form:

```
http://<somepath>?SourceSystem=OMS
```

The parameter `SourceSystem` specifies where the XML document (i.e., the customer history entry) is coming from. In this example, the only value for `SourceSystem` that the RESTful Web service accepts is "OMS." Try changing the URL inside build.xml to specify `SourceSystem=XYZ` and see what happens. You will get an error message indicating the source is not supported yet.

The URL parameter `SourceSystem` is a parameter of the RESTful Web service. That is one way parameters are passed to RESTful services. Chapter 4 discusses how SOAP services get parameters—they are embedded in the SOAP message itself. You can also design a RESTful service that receives parameters in the XML document, but this is kind of like reinventing SOAP.

The REST approach of using URL parameter passing is simple, but it also has drawbacks. The primary drawback is that there is no interface description of a RESTful service, so there is no way to determine—without some other form of documentation—what URL parameters are required. Some REST purists handle that objection by pointing out that URL parameters are not needed for proper REST systems where resources are uniquely defined by URIs. In this example, the URL could instead have the form:

```
http://<somepath>/OMS
```

In this case, the convention is that you post customer histories from the OMS to the `.../OMS` URI, and customer histories from the XYZ system to the `.../XYZ` URI, and so on.

Other REST advocates, who are slightly less purist, argue that URL parameters are fine as long as they are *nouns* rather than *verbs*. An example of a verb parameter would be something like:

```
http://<somepath>/ShoppingCart?action=clear
```

In this case, the `action` parameter specifies an operation to be carried out on the resource—clearing the shopping cart. Specifying verbs like this is a big REST no-no, but you can still find lots of so-called RESTful services out there that are implemented this way.

My perspective on this debate is that, even if we follow the REST purists and do away with URL parameters, we have just changed the syntax, not the semantics. The underlying semantics (and therefore the implementation) defines a resource (Customer History System) that can receive updates from various sources (e.g., OMS, XYZ), and needs to know what the source is.

If you implement that semantics by embedding parameters in the URL path—rather than by using URL parameters—you have only made the system's interface even harder to understand. For example, when you use the URL parameter form (e.g., `http://<somepath>?SourceSystem=OMS`), at least you can tell that the OMS is a parameter designating the source system. However, when you use the normalized version without parameters (e.g., `http://<somepath>/OMS`), you don't get any clues as to the meaning of the "OMS."

But, in either case, REST still provides you with no way to document your interface—in other words, no WSDL. In my opinion, this is the primary reason why SOAP is more appropriate than REST for SOA-style systems integration. Doing systems integration is all about defining the interfaces between systems. If you don't have a language in which to express the interfaces (i.e., no WSDL), it is very hard to be rigorous about defining the interfaces. As indicated in Section 3.2.1, you can try to work around this REST limitation by using XML Schema to define the interface. That approach works, but in addition to not being standard practice, it has other limitations. For example, in the case just discussed, the parameter (`SourceSystem=OMS`) is not part of the XML message received by the RESTful service. So, to define an interface that specifies this parameter, you would have to refactor the RESTful service to accept a parameter inside the XML message that indicates the source system. The basic problem here is that URL parameters, since they are not part of the XML message, cannot be specified in an XML Schema-based interface definition.

This example has shown how to develop an `HttpURLConnection`-based client for sending XML documents to a RESTful service that requires URL parameters. The next section shows you how to do the same thing using JAX-WS 2.0.

3.3.4 Sending EIS Records to a REST Service with JWS

As in Section 3.3.2, the client in this section uses `javax.xml.ws.Service` and `javax.xml.ws.Dispatch` rather than `java.net.URL` and `HttpURLConnection`. The major difference from that section is that here, the XML document is being pushed to the service. To do that, the XML document needs to be stored as an instance of a Java class that can be used by the `Dispatch<Source>` instance. In this case, the type parameter is `Source`, so a `Source` instance must be created from the XML document that is to be sent to the RESTful service.

This example also illustrates how to get HTTP-related information from the `Dispatch` object by accessing its response context. As demonstrated here, a bit more work is needed to get the HTTP status code than with the simple `HttpURLConnection.getResponseCode()` method used in the previous example.

Figure 3–6 illustrates push messaging as implemented with JAX-WS 2.0. There is a little more detail here than shown in the `HttpURLConnection` example from Figure 3–5. The steps are as follows:

1. The client uses the `Service.addPort()` method to create a port within the `Service` instance that can be used to access the RESTful Web service.
2. Next, the `Service.createDispatch()` method is invoked to create an instance of `Dispatch<Source>`—a `Dispatch` instance that enables you to work with XML request/response messages as instances of `javax.xml.transform.Source`.
3. The XML file to be posted to the RESTful service is wrapped in an instance of `javax.xml.transform.stream.StreamSource`. `StreamSource` implements the `Source` type parameter required by `Dispatch<Source>`.
4. The `Dispatch.invoke()` method then packages the XML document into an HTTP POST request—per the JAX-WS 2.0 HTTP Binding—and sends it to the RESTful service. The `invoke()` method waits for the response before returning.
5. The service processes the HTTP POST and sends an HTTP response that includes an HTTP response code.
6. Because the HTTP response code is part of the HTTP message (transport level), and not part of the XML payload, to examine it the client invokes `Dispatch.getResponseContext()` to get the HTTP context for the response.
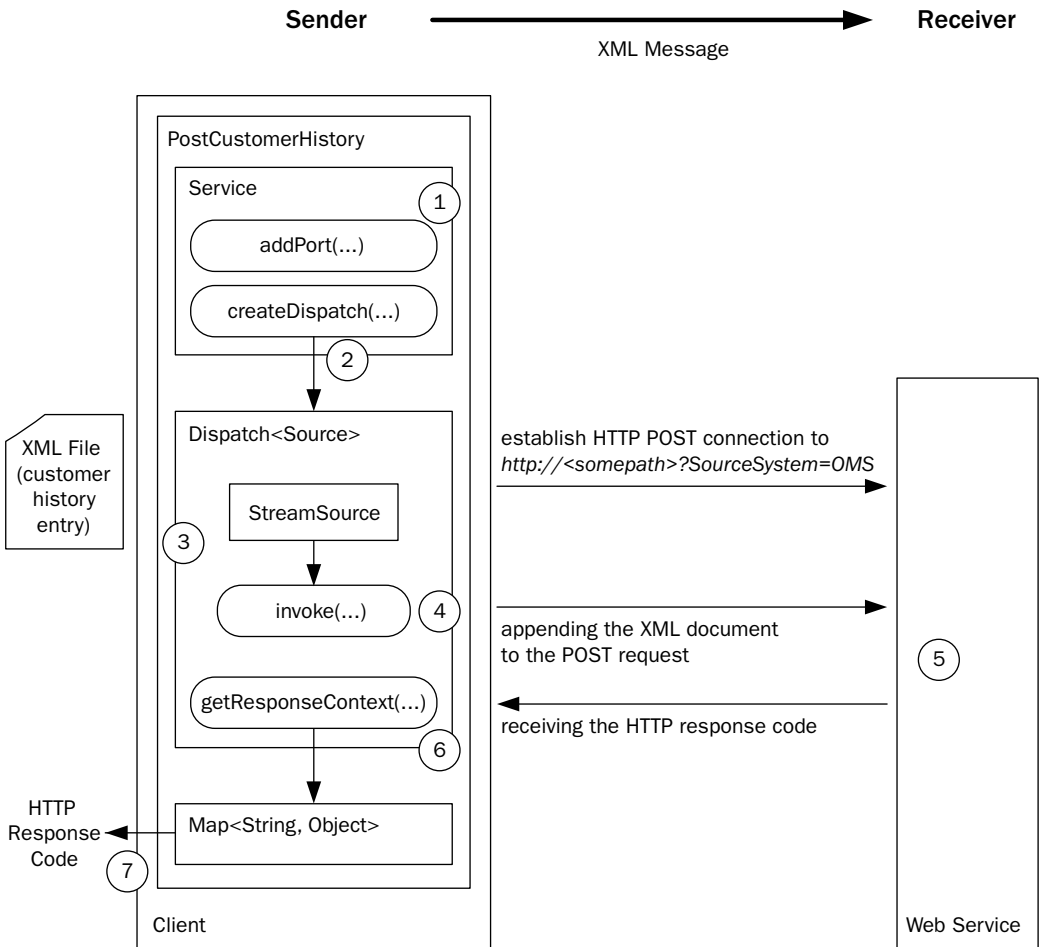
Figure 3–6 Push messaging with HTTP POST and JAX-WS 2.0.

7. The HTTP context is represented as a `Map<String, Object>` instance. This map provides access to the HTTP headers and other information that is outside the XML payload. Here, it is used to access the HTTP response code (i.e., 200 for "OK," 500 for "Server Failure," etc.).

Example 3–8 shows the implementation of `PostCustomerHistory` using JAX-WS. It is similar to Example 3–6, and you should review the discussion of REST and JAX-WS given there.

The main difference here from the `HttpURLConnection` version (Example 3–7) is that the `Dispach.invoke()` method is invoked with a `StreamSource` parameter that is constructed from the XML file being posted to the RESTful Web service. Notice that there is no need to write the XML out to a stream as in the `HttpURLConnection` example. The `Dispatch<Source>` instance lets you deal with the XML request and response payloads as instances of `Source`.

Example 3–8 The `PostCustomerHistory` Client as Implemented with JAX-WS

```
33    public static void main(String[] args) throws Exception {
34      if (args.length != 2) {
35        System.err.println
36          ("Usage: java XMLUploadSender <XML file name> "
37          + "<Web Service URL>");
38        System.exit(1);
39      }
40      QName svcQName = new QName("http://sample", "svc");
41      QName portQName = new QName("http://sample", "port");
42      Service svc = Service.create(svcQName);
43      svc.addPort(portQName, HTTPBinding.HTTP_BINDING, args[1]);
44      Dispatch<Source> dis =
45            svc.createDispatch(portQName, Source.class, Service.Mode.PAYLOAD);
46      dis.invoke(new StreamSource(new File(args[0])));
47      Map<String, Object> respContext = dis.getResponseContext();
48      Integer respCode =
49        (Integer) respContext.get(MessageContext.HTTP_RESPONSE_CODE);
50      System.out.println("HTTP Response Code: "+respCode);
51    }
```

book–code/chap03/rest-post/client-jaxws/src/java/samples/PostCustomerHistory.java

To run this example, do the following. After the example is run, the results (customer history entries) are written by the application to a temporary file of the form `${user.home}/tmp/soabook*.xml`. So, you can look to your `${user.home}/tmp` directory to verify that the example ran properly.

1. Start GlassFish (if it is not already running).
2. Go to `<book–code>/chap03/rest-post/endpoint-servlet`.
3. To build and deploy the Web service enter:

```
mvn install
```

... and when that command finishes, then enter:

```
ant deploy
```

4. Go to `<book-code>/chap03/rest-post/client-jaxws`.

5. To run the client enter:

```
mvn install
```

6. To undeploy the Web service, go back to `<book-code>/chap03/rest-post/endpoint-servlet` and enter:

```
ant undeploy
```

As you can see from this example, getting the HTTP response code from the `Dispatch` instance is a little awkward. First, you need to request the response context. That is because `Dispatch` is not an HTTP-specific interface. So, it doesn't make sense for `Dispatch` to have a convenience method like `HttpURLConnection.getResponseCode()`. The JAX-WS Expert Group envisions scenarios where `Dispatch` is used with non-HTTP bindings. So, the way it works is that the `Dispatch.getResponseContext()` method provides an instance of `Map<String, Object>` that contains context information about the underlying protocol.

The `getResponseContext` method is inherited from the `javax.xml.ws.BindingProvider` interface (of which `Dispatch` is a sub-interface). `BindingProvider` provides a representation of the underlying protocol binding (e.g., XML/HTTP or SOAP/HTTP) being used for Web Services communication. When a `BindingProvider` does a request/response, the request and response messages are embedded in a context that is binding-specific. The message and its context move through a chain of handlers during the invocation process. All this is beyond the scope of our simple discussion here, but it is useful background (see Chapter 6 for a detailed discussion of JAX-WS client-side handlers). The response context represents the final state of the message context after the invocation is completed. So, access to the response context is provided through the JAX-WS handler framework APIs.

The way this handler framework manifests itself here is that the keys that are used to look up information in the response context are provided by `javax.ws.handler.MessageContext`. As shown in the code, `Message-Context.HTTP_RESPONSE_CODE` is the key used to access the HTTP response code.

In the `HttpURLConnection` case, there is no distinction between the message and its context. One works directly with the HTTP requests and responses. So, the processing model is simpler. However, the drawback

is that you have to create your own code to extract the XML messaging from the HTTP communications. In these simple examples, that doesn't seem like a big deal. The only manifestation of that extraction process so far has been reading and writing XML to the HTTP input and output streams. However, as the complexity of the processing increases, dealing with messages rather than streams becomes a valuable additional layer of abstraction. For example, when you want to introduce handlers to do Java/XML binding or reliable messaging, you don't want to have to create your own handler framework for pre- and post-processing of the HTTP streams.

That wraps up our basic discussion about how to invoke RESTful Web services. Next, the discussion turns to XSLT—the XML data transformation language—and how it can be used to implement basic SOA-style loosely coupled integration of multiple Web services.

3.4   SOA-Style Integration Using XSLT and JAXP for Data Transformation

Some readers may be wondering why a book about SOA with Java Web Services would include a section on XSLT. The reason is that XLST provides a powerful and efficient data transformation engine that can be used to translate messages from one format to another. When building SOA applications, developers commonly encounter problems where they need to integrate systems that require the same message type (e.g., purchase order) in different formats. XSLT provides a standard mechanism for translating between message formats when messages are represented as XML documents.

The perceding section introduced two Web services:

- The OMS "NewOrders" Web service that provides access to the new orders that have come in during the past day
- The CSS "CustomerHistory" Web service that receives updates to the customer history database

This section shows how to build a simple SOA application that links these two Web services together. I am going to walk through an example that gets the new orders, transforms them into customer history entries, and posts those entries to the CSS. This example introduces data transformation using XSLT—a cornerstone of SOA-style loosely coupled integration.

3.4.1 How and Why to Use XSLT for Data Transformation

A core component of any SOA system is its capability to transform data from one format to another. Often referred to as *data transformation*, this capability is most naturally addressed within the Web Services context using eXtensible Stylesheet Language Transformations (XSLT). We assume the reader is familiar with the basics of XSLT. Our focus is on the application of XSLT to SOA-style loosely coupled integration. To brush up on XSLT, see the W3C's Web site at www.w3.org/Style/XSL/. In addition to the specification (the examples in this book use Version 1.0—see [XSLT 1.0]), the site has links to a variety of helpful tutorials. Another good refresher can be found in Sun's Java/XML Tutorial (http://java.sun.com/webservices/jaxp/dist/1.1/docs/tutorial/xslt/index.html). Sun's tutorial is especially useful because it discusses the JAXP APIs along with XSLT.

Because XSLT is a sophisticated data transformation language, it can take a long time to learn in depth. Fortunately, the data transformations required by most SOA applications need to use only a small part of the XSLT language that does not take a long time to learn. If you are struggling to understand this section of the book, you should have no trouble once you review the Sun tutorial. There is no need to study XSLT in great depth at this point!

XSLT makes sense as the transformation tool of choice within SOA integration frameworks, because it is a universally accepted standard and the transformation engines that interpret XSLT to perform data transformations keep getting better and faster. Although it may be expedient to write some quick and dirty code to perform a simple transformation here and there, it does not make sense to leave the data transformation language and processing unstandardized when developing a framework for SOA integration that will be used across an organization.

To demonstrate how XSLT can be used for data transformation, we consider a scenario where the new orders, accessed from the OMS, are used to create customer history records that update the CSS customer history database. The business reason for doing this is so that users of the CSS have fast access to an up-to-date record of the transactions a customer has made with the company. Such information needs to be available nearly instantly when handling customer care telephone calls, for example. If the customer care representative needs to examine the details of any transaction in the customer history, the information stored there from the OMS provides important keys, such as `OrderKey` and `ITM_NUMBER`, that will enable detailed information to be retrieved from the OMS rapidly.

Figure 3–7 illustrates the data mapping that transforms an OMS order record into a CSS customer history record. The order record and customer history record are introduced in Figure 3–1 and Figure 3–2, respectively.
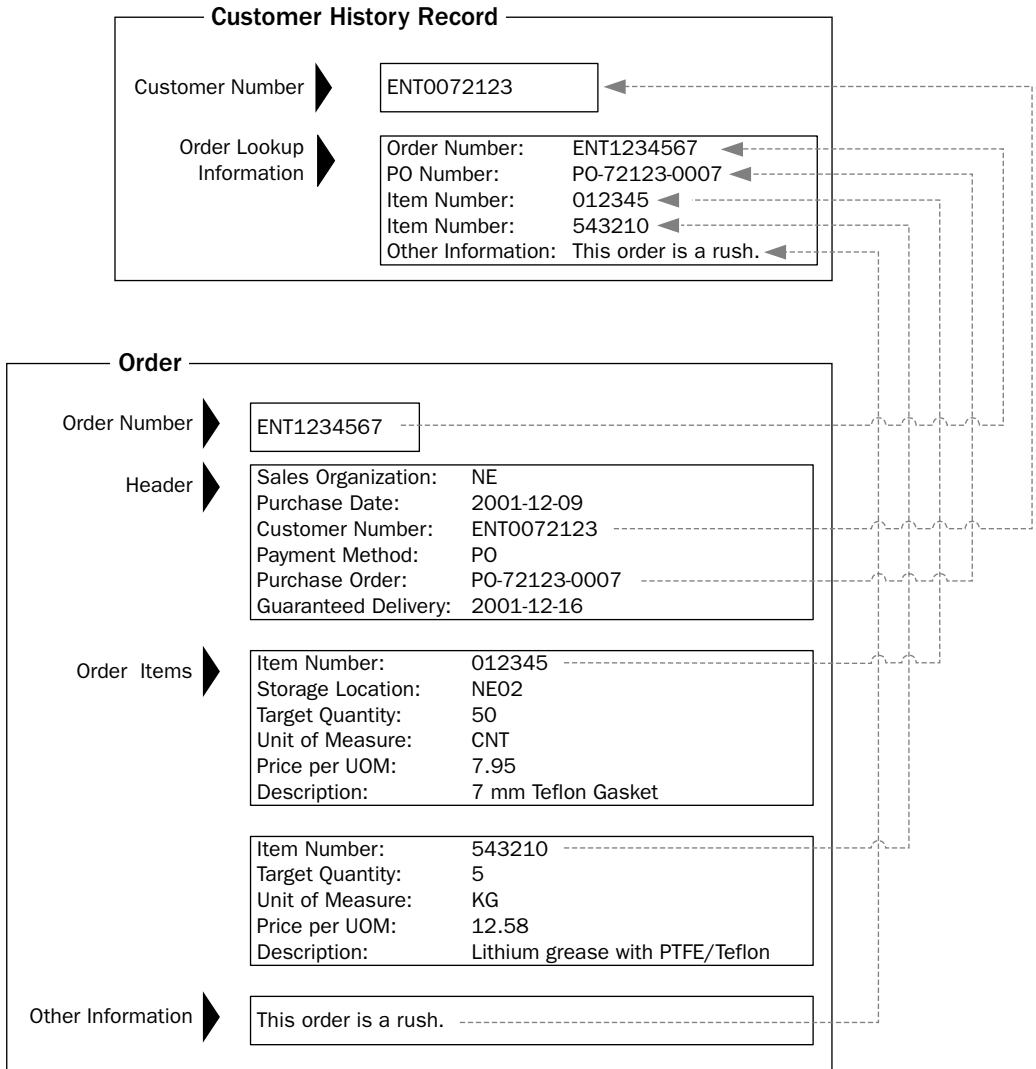

Figure 3–7 A data mapping for the transformation from a sales order to a customer history record.

As illustrated in Figure 3–7, the Customer Number in the Order becomes a foreign key in the Customer History Record, which links it back to other information about the customer. Order Number, PO Number, and Item Number are mapped over because having them in the CSS will enable additional SOA components to be built that provide quick lookups from a customer history record to detailed order, purchase order, and item information in the OMS and other systems. Note that there may be multiple instances of an item number in a customer history record, if an order includes more than one type of item.

The following examples review the XSLT for transforming a set of OMS orders into a set of CSS customer history entries. The set of orders is formatted as an `oms:Orders` element in the schema `http://soabook.com/example/oms/orders.xsd` (Example 3–2). The set of customer histories is formatted as a `css:CustomerHistoryEntries` element in the schema `http://soabook.com/example/css/custhistentries.xsd` (Example 3–4). The XSL transformation from an order to a customer history is represented pictorially in Figure 3–7.

The XSLT language is declarative. It defines transformations of source documents to target documents. An XSL transformation comprises a set of template rules—represented by instances of the `xsl:template` element—that are children of the root `xsl:stylesheet` element. Hence, an XSLT document is often referred to as a stylesheet. The template elements in the stylesheet define the structure of the target document that is created from the source.

XSLT uses the XPath language (see [XPath]) to identify chunks of data in the source document (e.g., `OrderKey`). Together with the template rules, the XPath expressions determine where to place the chunks of data in the target document.

Example 3–9 illustrates a stylesheet for transforming orders to customer histories. This discussion breaks the stylesheet into bite-size chunks. The example shows the beginning of the XSLT document, including the `xsl:stylesheet` namespace declarations and `xsl:output` element.

Example 3–9 XSLT for Customer History—Namespaces and Output Elements

```
4   <xsl:stylesheet version="1.0"
5     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
6     xmlns:oms="http://www.example.com/oms">
7     <xsl:output method="xml" version="1.0" encoding="UTF-8"/>
```

book-code/chap03/xslt/etc/order_to_history.xslt

As you can see, the prefix `oms` is used to denote the Order Management System namespace: `http://www.example.com/oms`. The `xsl:output` element controls the format of the stylesheet output. The attribute `method="xml"` indicates that the result should be output as XML. Note that this stylesheet does not specify that the output should be indented (i.e., it does not include the attribute `indent="yes"`). You should avoid specifying visual formatting such as indentation in the base-level transformation. I recommend using a separate XSLT stylesheet to format XML for human-readable output when necessary. Note that the encoding is specified (`encoding="UTF-8"`), as it is throughout this book, as UTF-8.

The next portion of the XSLT, shown in Example 3–10, provides the rules for processing the `oms:Orders` element.

Example 3–10 XSLT for Customer History—Creating the Customer History Entry

```
11    <xsl:template match="oms:Orders">
12      <CustomerHistoryEntries xmlns="http://www.example.com/css"
13        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
14        xsi:schemaLocation="http://www.example.com/css
15        http://soabook.com/example/css/custhistentries.xsd">
16        <xsl:apply-templates/>
17      </CustomerHistoryEntries>
18    </xsl:template>
19    <xsl:template match="oms:Order">
20      <CustomerHistoryEntry xmlns="http://www.example.com/css">
21        <CustomerNumber>
22          <xsl:apply-templates select="./oms:OrderHeader/oms:CUST_NO"/>
23        </CustomerNumber>
24        <OrderLookupInfo>
25          <xsl:apply-templates select="./oms:OrderKey"/>
26          <xsl:apply-templates
27            select="./oms:OrderHeader/oms:PURCH_ORD_NO"/>
28          <xsl:apply-templates
29            select="./oms:OrderItems/oms:item/oms:ITM_NUMBER"/>
30          <xsl:apply-templates select="./oms:OrderText"/>
31        </OrderLookupInfo>
32      </CustomerHistoryEntry>
33    </xsl:template>
```

book–code/chap03/xslt/etc/order_to_history.xslt

At the beginning of the block, you see a template being defined to match the pattern "oms:Orders"—in other words, it matches the `oms:Orders` element in the source document. Inside this template, you see the definition of a `CustomerHistoryEntries` element. The contents appearing inside the template form the output for the target document. So, this template provides the rules for transforming an `oms:Orders` element into a `css:CustomerHistoryEntries` element.

Now, notice that this template has the instruction `<xsl:apply-templates/>` inside it. Inside a template, XML in the `xsl` namespace is interpreted as instructions and not as content to be output to the target document. This particular instruction tells the XSLT processor to apply the other templates in the stylesheet to the children of the `oms:Orders` node and insert the results into the target document. So, that is how the `css:CustomerHistoryEntries` element gets constructed in the target document. Its opening and closing tags are specified in this template. Its children are defined by the results of the `<xsl:apply-templates>` instruction and are inserted between the opening and closing tags.

Continuing to examine Example 3–10, you can see that the bottom half defines another template matching the `oms:Order` element. So, if any children of `oms:Orders` are instances of `oms:Order`, these children will be processed by this template and the results will be inserted the `CustomerHistoryEntries` tags. Looking inside this template for `oms:Order`, you can see that it contains the contents for an element, `CustomerHistoryEntry`, and the two top-level elements `CustomerNumber` and `OrderLookupInfo`. Now, look inside the tags for `CustomerNumber` and you see another `xsl:apply-templates` instruction. However, this one has the attribute:

```
select="./oms:OrderHeader/oms:CUST_NO"
```

This template is providing the instructions for filling in the contents of the `CustomerNumber` element. And the XPath expression `./oms:OrderHeader/oms:CUST_NO` restricts the children of `oms:Order` that this template is applied to. That XPath expression tells the XSLT processor to only apply templates to `oms:CUST_NO` elements that are children of `oms:OrderHeader`. In this manner, the XPath expression reaches into the source document, pulls out the `oms:CUST_NO` element, processes it, and inserts the results inside the `CustomerNumber` tags. That is how `oms:CUST_NO` gets transformed into `css:CustomerNumber` and inserted into the right place in the target document.

Looking at some of the other `xsl:apply-templates` instructions occurring in Example 3–10, you can see that the `<OrderLookupInfo>` element is populated from the source elements specified by the XPath expressions: `./oms:OrderKey`, `./oms:OrderHeader/PURCH_ORD_NO`, `./oms:OrderItems/item/ITM_NUMBER`, and `./oms:OrderText`. Notice that these XPath expressions correspond to the dotted line mappings in Figure 3–7.

Continuing to review this stylesheet, now have a look at Example 3–11, which shows the templates that match these XPath expressions.

Example 3–11 XSLT for Customer History—Detail-Level Templates

```
37    <xsl:template match="oms:CUST_NO">
38      <xsl:value-of select="."/>
39    </xsl:template>
40    <xsl:template match="oms:OrderKey">
41      <OrderNumber xmlns="http://www.example.com/css">
42        <xsl:value-of select="."/>
43      </OrderNumber>
44    </xsl:template>
45    <xsl:template match="oms:PURCH_ORD_NO">
46      <PURCH_ORD_NO xmlns="http://www.example.com/css">
47        <xsl:value-of select="."/>
48      </PURCH_ORD_NO>
49    </xsl:template>
50    <xsl:template match="oms:ITM_NUMBER">
51      <ITM_NUMBER xmlns="http://www.example.com/css">
52        <xsl:value-of select="."/>
53      </ITM_NUMBER>
54    </xsl:template>
55    <xsl:template match="oms:OrderText">
56      <OrderText xmlns="http://www.example.com/css">
57        <xsl:value-of select="."/>
58      </OrderText>
59    </xsl:template>
```

book-code/chap03/xslt/etc/order_to_history.xslt

Here you can see, for example, that the template matching `oms:OrderKey` simply returns the value of that element (the instruction `<xsl:value-of  select="."/>` returns the string value of the current node). The net result is that this `stylesheet` maps the value of

`oms:OrderKey` to a subelement in the target document named `OrderNumber` that is a child of `CustomerHistoryEntry`.

Having walked through an example of an XSLT, the next section looks at how such transformations are applied using Java.

3.4.2 XSLT Processing Using JAXP

XSLT processing in Java is accomplished using the Java API of XML Processing (JAXP) [JSR 206]. Specifically, the JAXP `javax.xml.transform.Transformer` class can be used to convert a source document to a target document according to the rules specified in a stylesheet. JAXP provides the foundation from which all Java XML processing is built.

Figure 3–8 shows a simplified architecture diagram illustrating the role of the JAXP API. A variety of different types of Java applications can use the JAXP API, including servlets, JSPs, and EJBs. All of these use JAXP to access the various capabilities that are included in any JAXP implementation, such as a SAX parser, a DOM implementation, and an XSL processor that supports XSLT. The package `javax.xml.parsers` provides a common factory interface to access different implementations of SAX and DOM (e.g., Xerces) as well as XSLT (e.g., Xalan). The interfaces for SAX and DOM are found in the `org.xml.sax` and `org.w3c.dom` packages, respectively. The XSLT APIs are found in the `javax.xml.transform` packages.

As shown in Figure 3–8, JAXP isolates a Java application (e.g., client, servlets, JSP, EJB) from the implementation of the XSLT transformer, and the SAX and DOM parsers. JAXP defines factory classes that instantiate wrapper objects on the transformer and parser implementations. The transformer/parser implementation classes that are used at runtime are determined by system property and/or classpath settings.

JAXP is an important enabling standard making it feasible to use Java and Web Services for constructing SOA-style systems integration applications. Not only does it integrate the XML parsing and transformation standards with Java, but also it isolates the SOA application components from the SAX, DOM, and XSLT implementations. This is important, because as better and faster implementations come to market, SOA components will be able to take advantage of them to improve the performance without needing to be rewritten.

By using the JAXP architecture and XML for messaging, most of the data transformation work involved in integrating SOA components with Java boils down to writing XSLT. The example used to demonstrate this is illustrated in Figure 3–9. This application reads orders from an OMS Web service, transforms them into customer history updates, and writes these updates to a CSS Web service.
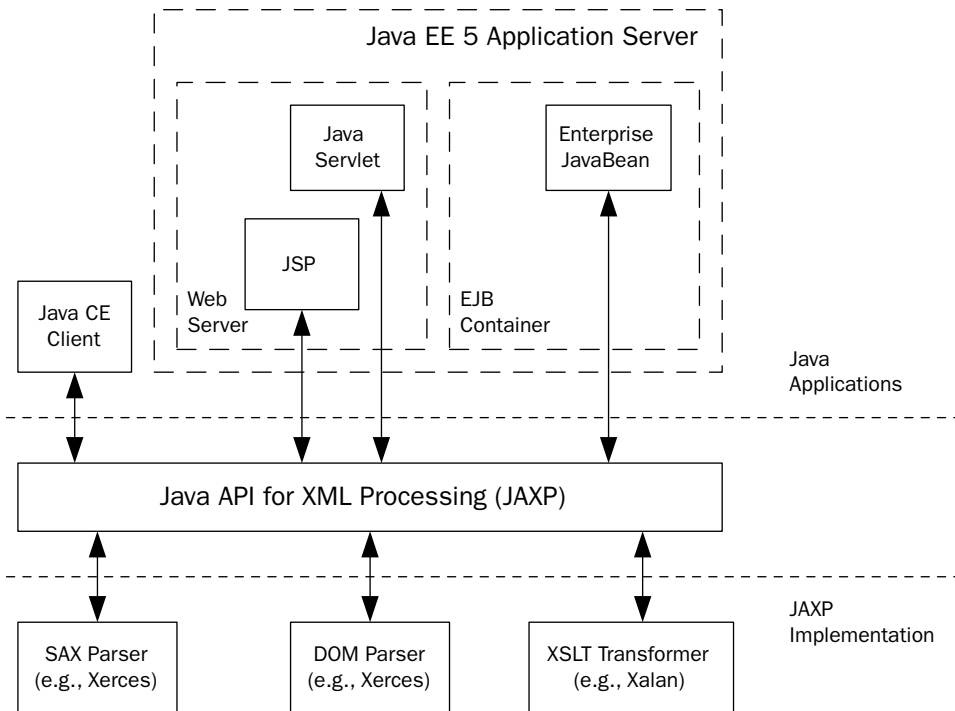
Figure 3–8 Architecture of the Java API for XML Processing (JAXP).

This example is constructed by tying together the examples from Sections 3.3.2 and 3.3.4 and using XSLT in the middle to transform the orders into customer histories. The steps in the process illustrated in Figure 3–9 are:

1. A `Service` instance is used to create two `Dispatch<Source>` instances—one to invoke the OMS Web service, and the other to invoke the CSS Web service.
2. The first `Dispatch<Source>` instance's `invoke` method is used to get the orders from the OMS Web service.
3. The orders (an XML document) are returned from `invoke()` as a `Source` instance.
4. The XSLT stylesheet file (`order_to_history.xslt`) is used, by a `TransformerFactory`, to construct a `Transformer` instance based on the stylesheet.
5. The `Transfomer.transform()` method is invoked to apply the stylesheet rules to the `Source` instance (`orders`). The resulting cus-
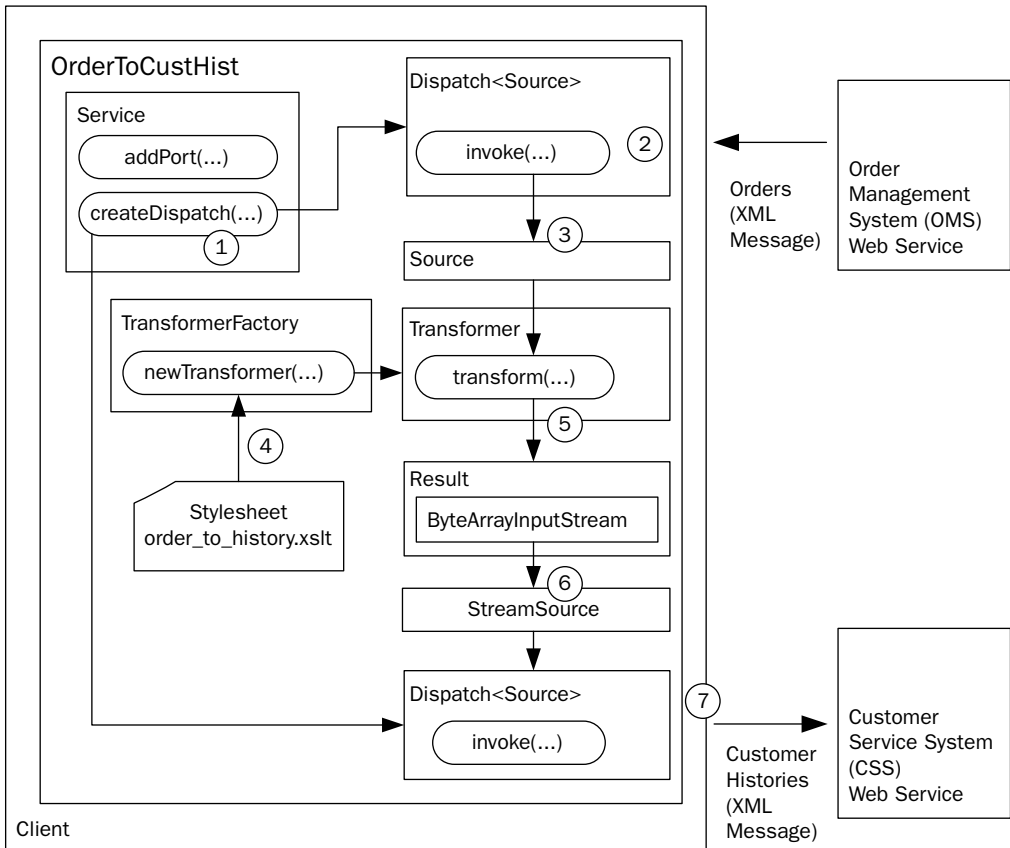
Figure 3–9 SOA-style integration with XSLT for data transformation.

tomer histories (an XML document—see Example 3–3) are written to a `Result` instance that has been created.

6. In this case, the `Result` instance is created as a wrapper from a `ByteArrayInputStream`. So, the XML is extracted from the underlying array and wrapped in a `StreamSource` object that can be consumed by the second `Dispatch` instance.

7. Lastly, as in Figure 3–6, the `Dispatch.invoke()` method is used to post the customer histories XML to the CSS Web service.

The code in Example 3–12 shows how the steps from Figure 3–9 are implemented. The Java used to create and invoke the `Dispatch` instances (to get and send the XML to the RESTful Web services) is the same as in

Example 3–6 and Example 3–8—please see those discussions for an overview of how `Dispatch` works in this scenario.

Example 3–12 Java Code That Applies the XSLT for Customer History

```
51      // Get the new orders
52      Service svc = Service.create(svcQName);
53      svc.addPort(orderQName, HTTPBinding.HTTP_BINDING, newOrdersUrl);
54      Dispatch<Source> getOrdersDispatch =
55        svc.createDispatch(orderQName, Source.class, Service.Mode.PAYLOAD);
56      Source newOrdersSource =
57        getOrdersDispatch.invoke(new StreamSource(new StringReader("<empty/>")));
58      // Instantiate a Transformer using our XSLT file
59      Transformer transformer =
60        TransformerFactory.newInstance().newTransformer
61        (new StreamSource(new File(xsltFile)));
62      // Transform the new orders into history entry files
63      ByteArrayOutputStream ba = new ByteArrayOutputStream();
64      transformer.transform(newOrdersSource, new StreamResult(ba));
65      // Update the customer histories
66      svc.addPort(histQName, HTTPBinding.HTTP_BINDING, addCustHistUrl);
67      Dispatch<Source> postCustomerHistoryDispatch =
68        svc.createDispatch(histQName, Source.class, Service.Mode.PAYLOAD);
69      postCustomerHistoryDispatch
70      .invoke(new StreamSource(new StringReader(ba.toString())));
```

book–code/chap03/xslt/src/java/samples/OrderToCustHist.java

To see how the XSLT is implemented, look to the middle of the example code where an instance of the default `TransformerFactory` is obtained using `TransformerFactory.newInstance()`. Using this factory, a `Transformer` is created by passing the XSLT file (the one discussed previously that implements the mapping illustrated in Figure 3–7) as a `StreamSource` to the `newTransformer()` method. The resulting `Transformer` then applies the XSLT to the `Source` instance obtained by invoking the `getOrdersDispatch` instance. As shown here, the second parameter used in the invocation of `transformer.transform()` is a `StreamResult` wrapping an underlying `ByteArrayOutputStream`. In this manner, the results of the XSL transformation are written to that byte array. The end of the example code shows how that byte array is wrapped inside a `StreamSource` that can be passed to the `postCustomerHistoryDispatch.invoke()` method to post the customer histories to the CSS Web service.

To run this example, do the following. After the example is run, the results (customer history entries) are written by the application to a temporary file of the form `${user.home}/tmp/soabook*.xml`. So, you can look to your `${user.home}/tmp` directory to verify that the example ran properly.

1. Start GlassFish (if it is not already running).
2. Go to `<book-code>/chap03/rest-post/endpoint-servlet`.
3. To build and deploy the Web service enter:

```
mvn install
```

... and when that command finishes, then enter:

```
ant deploy
```

4. Go to `<book-code>/chap03/rest-get/endpoint-servlet`.
5. To build and deploy the Web service enter:

```
mvn install
```

... and when that command finishes, then enter:

```
ant deploy
```

6. Go to `<book-code>/chap03/xslt`.
7. To run the client enter:

```
mvn install
```

8. To undeploy the Web service, go back to `<book-code>/chap03/rest-get/endpoint-servlet` and enter:

```
ant undeploy
```

9. Do the same in the directory `<book-code>/chap03/rest-post/endpoint-servlet`.

That concludes this brief introduction to data transformation using XSLT with JAXP. XML processing with JAXP is examined in more detail in Chapter 5 where data binding is introduced and JAXB is compared with SAX and DOM. The next two sections of this chapter look at how RESTful services are deployed—with and without the JWS.

3.5   RESTful Services with and without JWS

The focus now switches from client-side consumption of RESTful Web Services to development and deployment of such services themselves. As in Section 3.3, this section examines how to deploy such services both with and without JAX-WS. As before, the purpose here is to compare and contrast the JWS approach with the bare-bones approach of simply

working with Java's HTTP servlet tools to build and deploy a simple RESTful service.

Again, the example used to illustrate the concepts in this section is a basic building block of SOA Web Services—the deployment of a simple download service. In this case, it is the "New Orders" service discussed from the client perspective in previous sections. This section examines how to deploy a Java class that provides a `getNewOrders()` method both with and without JAX-WS.

3.5.1 Deploying a REST Service without Using JWS

This section provides the first example in the book of how to deploy a Java method as a Web service. It is the simplest RESTful Web service imaginable—the service consumed by the client in Section 3.3.2. This service simply returns an XML document containing the "new orders."

Example 3–13 shows a *mock object* implementation of the `getNewOrders()` method. This implementation of `OrderManager` is called a mock object because it is a "mock-up" of a real `OrderManager` class. It is a stub implementation of the server side. The `getNewOrders()` method returns the contents of a hard-coded file (`/orders.xml`) instead of providing a live interface to an Order Management System. Throughout this book, mock objects like this are used to illustrate the Java classes that are deployed as Web Services. This strategy provides realistic examples you can actually run, without requiring you to have any actual back-end systems to provide Order Management or Customer Service implementations.

Example 3–13 The `OrderManager.getNewOrders(...)` Method to Be Deployed As a Web Service

```
26  public class OrderManager {
27
28    public Source getNewOrders() throws FileNotFoundException {
29      // get the resource that provides the new orders
30      InputStream src = getClass().getResourceAsStream("/orders.xml");
31      if ( src == null ) {
32        throw new FileNotFoundException("/orders.xml");
33      }
34      return new StreamSource(src);
35    }
36
37  }
```

book–code/chap03/rest-get/endpoint-servlet/src/java/samples/OrderManager.java

Several questions arise when considering how to enable this method as a Web service:

1. How is the class instantiated?
2. How is the `Source` instance returned by `getNewOrders()` converted into an XML stream in an HTTP response message?
3. How is the class deployed as a Web service?

These three questions mirror the three components of a Web Services Platform Architecture outlined in Chapter 1, Section 1.2. Question #1 is about invocation—the first component of a Web Services Platform Architecture. To invoke a Web service, you must be able to get an instance of its implementation class. Question #2 is about serialization—the second component of a Web Services Platform Architecture. The Java object that is returned must be serialized to XML that can be written out "onto the wire." Question #3 is about deployment—the third component of a Web Services Platform Architecture.

This section gives simple answers to these questions, as the example shows how to deploy the Java method as a Web service using a dedicated servlet. However, what is interesting to note is that, even in this simple case, the solutions need to deal with all three of the Web Services Platform Architecture components.

In this example, the Web service is accessed with an HTTP GET request. The servlet needs to handle the HTTP GET request, instantiate the Java class, invoke the `getNewOrders()` method, and write the results out as XML in the HTTP GET response. Figure 3–10 shows the basic architecture.

As illustrated here, this simple architecture deploys a RESTful Web service using the `java.servlet.http.HttpServlet` class. The `GetNewOrders-Servlet` class acts as a front-end to the `OrderManager` that mediates between the HTTP request/response messages and invokes the Java method that implements the Web service. The steps illustrated in the figure are:

1. The client sends an HTTP GET request to the appropriate URL (a URL that has been associated, in the servlet container's configuration, with the `GetNewOrders` class).
2. The servlet container wraps the HTTP request stream in an instance of `HTTPServletRequest` and passes it to the `GetNewOrdersServlet`'s `doGet()` method.
3. The `doGet()` method creates an instance of the `OrderManager` class and invokes the `getNewOrders()` method.
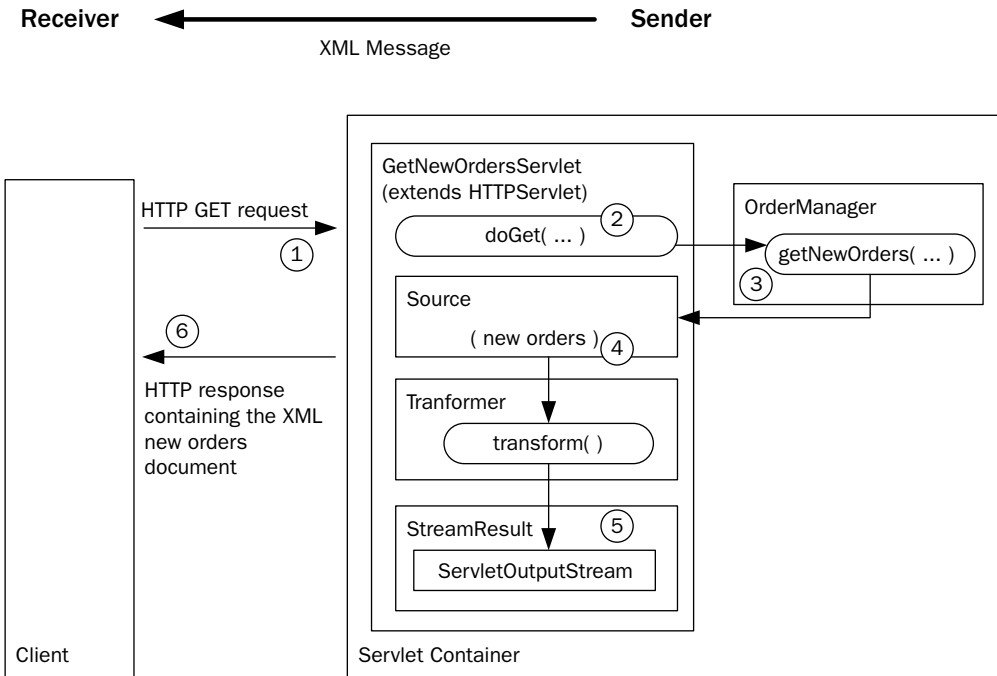
Receiver ◄─────────────────── Sender
                XML Message

Figure 3–10 RESTful service deployed using `HTTPServlet`.

4. The `getNewOrders()` method returns the new orders XML document as an instance of `javax.xml.transform.Source`.
5. An instance of `javax.xml.transform.Transformer` is used to write the new orders `Source` to the `ServletOutputStream` (wrapped in an instance of `java.xml.transform.stream.StreamResult`).
6. The `getNewOrders()` method returns and the HTTP response containing the new orders XML document is returned to the client.

Example 3–14 shows the code for the `GetNewOrdersServlet`'s `doGet()` method. This code is straightforward, but there are a few items to notice and think about. First, as you can see, the servlet has to instantiate the `OrderManager` class. This assumes that `OrderManager` is available to the class loader. In this example, I accomplish that by bundling the `OrderManager` class in the WAR that is deployed. This is the simplest way to get an instance of a class that needs to be deployed as a Web service, but it is not always feasible. For example, if the class is implemented as an EJB, you will need to request its interface from the EJB container, instead of instantiating an instance. Furthermore, suppose the class requires other container

services (e.g., JNDI and a database connection). In the real world, it is not so easy to just deploy a POJO by packaging its class definition into a WAR. Even in this example, using a mock object, the returned data (`orders.xml`) needs to be packaged into the WAR along with the deployed class. How you get an instance of a class being deployed as a Web services is a topic I cover in some detail when we explore the design of the SOA-J in Chapter 11.

Another item to notice is the use of the `HttpServletResponse.setContentType("text/xml")` method to set the content type of the HTTP response. This is important, because many REST clients (including early versions of the GlassFish implementation of `Dispatch`) will fail if the content type is not `"text/xml."` You need to be doubly careful with this, because some of the `HttpServletResponse` methods (e.g., `sendError(int sc, String msg)`), on some servlet containers, change the content type to `"text/xml"` since their error messages are implemented as HTML content.

Lastly, notice the use of the default instance of `Transformer` to simply write XML from a `Source` to a `Result`. Unlike in Section 3.4, here I am not doing any XSL transformation. I am just using the `Transformer` to write the XML returned by the `OrderManager` to the `ServletOutputStream`.

Example 3–14 The `GetNewOrdersServlet doGet(...)` Method

```
33    public void doGet(HttpServletRequest req,
34        HttpServletResponse res)
35    throws IOException, ServletException {
36      // invoke the Java method
37      OrderManager om = new OrderManager();
38      Source src = om.getNewOrders();
39      // write the file to the HTTP response stream
40      ServletOutputStream out = res.getOutputStream();
41      res.setContentType("text/xml");
42      StreamResult strRes = new StreamResult(out);
43      try {
44        TransformerFactory.newInstance().newTransformer()
45        .transform(src, strRes);
46      } catch (Exception e) {
47        throw new IOException(e.getMessage());
48      }
49      out.close();
50    }
```

book-code/chap03/rest-get/endpoint-servlet/src/java/samples
  /GetNewOrdersServlet.java

The instructions for deploying and invoking this servlet are included with Example 3–6.

Example 3–15 shows the deployment descriptor for the Web service implemented by the `GetNewOrdersServlet` class (together with `OrderManager`). This is the standard `web.xml` file, which is placed into the `WEB-INF` subdirectory of the WAR package used to deploy this Web service.

Example 3–15 The `web.xml` Deployment Descriptor Bundled in the `GetNewOrdersServlet` WAR

```
 9  <web-app>
10    <servlet>
11      <servlet-name>GetNewOrdersServlet</servlet-name>
12      <servlet-class> samples.GetNewOrdersServlet </servlet-class>
13    </servlet>
14    <servlet-mapping>
15      <servlet-name>GetNewOrdersServlet</servlet-name>
16      <url-pattern>/NewOrders</url-pattern>
17    </servlet-mapping>
18  </web-app>
```

`book-code/chap03/rest-get/endpoint-servlet/src/webapp/WEB-INF/web.xml`

Notice in Example 3–15 that the servlet `GetNewOrdersServlet` is mapped to the URL pattern `/NewOrders`. The deployment tool you use to deploy the WAR determines the context root for the Web application. In this case, the GlassFish default is being used—and it takes the context root from the name of the WAR file (`chap03-rest-get-endpoint-servlet-1.0.war`). And the base URL for the servlet container is http://localhost:8080 (unless you have customized the profile section of the `<book-code>/pom.xml` file when you installed the code example—see Appendix B, Section B.5). So, the URL for this Web service becomes:

`http://localhost:8080/chap03-rest-get-endpoint-servlet-1.0/NewOrders`

That pretty much wraps up the "how-to" discussion for creating and deploying a simple REST Web service using a servlet to invoke a Java method. As you can see, it is not hard to use servlets for deployment of basic RESTful Web services. The three questions posed at the beginning of the section have been answered as follows:

1. The Web service implementation class (i.e., `OrderManager`) is instantiated using the `no-arg` default constructor. This assumes that such a constructor exists and that the class definition is on the classpath. It also assumes that all resources required by the class are available.
2. The object returned by `getNewOrders()` is converted into an XML stream using a `Transformer`. This is a very simple scenario where the method being deployed returns a result that is already represented as an XML infoset. In most cases, the result will be a Java object that is not an XML infoset representation and requires serialization.
3. The class is deployed by bundling it with a dedicated servlet.

Problems are encountered, however, when you want to deploy multiple services. Using the architecture described in this section, you need to have a servlet for each Web service. That quickly becomes cumbersome and inefficient. What is needed is an invocation subsystem so that a single servlet can be used to invoke more than one service. However, to do that requires a method of mapping URLs to services at a finer granularity than provided by the `web.xml` file. As you can see in Example 3–15, the `web.xml` mapping is from a single URL to a single servlet. So, for a single servlet to handle multiple URLs (and multiple Web services), additional deployment meta-data must be added to this simple architecture that falls outside of the servlet processing model. One way to do this might be to map all base URLs of the form http://example.com/services/* to the servlet. Then, local paths such as `/getNewOrder`, `/addCustomerHistory`, and so on are mapped to individual Web services.

In fact, this approach is used by a variety of Web Services engines, including Apache Axis [AXIS] and the SOA-J engine introduced in Chapter 11. JWS also offers a variation on this approach, which I examine in detail in Chapter 7. For now, I'm going to defer further discussion of deployment issues and move on to the nuts and bolts of how to deploy this "New Orders" example Web service using JAX-WS and JSR-181 annotations.

3.5.2 Deploying a RESTful Service with JWS

This section illustrates how a Web service is deployed using JWS. This is the same service that was deployed in Section 3.5.1. However, the operation of the service—as deployed using JWS—is very different.

The primary difference here is that instead of using a servlet as in Example 3–14, the JWS version uses an instance of `Provider<Source>` to implement the RESTful service. Example 3–16 shows how it is implemented. The `@WebServiceProvider` annotation used in this example is defined by the JAX-WS 2.0 specification. It is used to declare a reference

to a Web Service that implements a `Provider<Source>` interface. The `@WebServiceProvider` annotation is required for deploying a RESTful Web service and is discussed in more detail in Chapter 7, which covers JAX-WS server-side development and deployment.

The `Provider<Source>` interface is basically the server-side version of `Dispatch<Source>` discussed in Section 3.3.2. It enables you to create a Web service that works directly with the XML message as an instance of `javax.xml.transform.Source`—rather than a JAXB 2.0 representation of the XML message. Along with `@WebServiceProvider`, the `javax.xml.ws.Provider` interface is explained in greater detail in Chapter 7. In this section, my goal is just to give you a quick example of how a RESTful service can be created and deployed with JAX-WS.

The `@BindingType` annotation (`javax.xml.ws.BindingType`) used in Example 3–16 is also defined by the JAX-WS 2.0 specification and is used to specify the binding that should be employed when publishing an endpoint. The property value indicates the actual binding. In this case, you can see that the value is specified as follow:

```
value=HTTPBinding.HTTP_BINDING
```

This indicates that the XML/HTTP binding should be used, rather than the default SOAP 1.1/HTTP. This is how REST endpoints are specified in JAX-WS 2.0—by setting the `@BindingType`. If one were to leave the `@BindingType` annotation off this example, the Java EE 5 container would deploy it as a service that expects to receive a SOAP envelope, rather than straight XML over HTTP (i.e., REST).

Example 3–16 `GetNewOrdersProvider` Implements `Provider<Source>` to Create a RESTful Web Service

```
21   import javax.xml.transform.Source;
22   import javax.xml.ws.BindingType;
23   import javax.xml.ws.Provider;
24   import javax.xml.ws.WebServiceProvider;
25   import javax.xml.ws.http.HTTPBinding;
26   import javax.xml.ws.http.HTTPException;
27
28   @WebServiceProvider
29   @BindingType(value=HTTPBinding.HTTP_BINDING)
30   public class GetNewOrdersProvider implements Provider<Source> {
31
```

```
32    public Source invoke(Source xml) {
33      OrderManager om = new OrderManager();
34      try {
35        return om.getNewOrders();
36      } catch (Throwable t) {
37        t.printStackTrace();
38        throw new HTTPException(500);
39      }
40    }
41
42  }
```

book-code/chap03/rest-get/endpoint-jaxws/src/java/samples
  /GetNewOrdersProvider.java

The `Provider<Source>` interface specifies the `invoke()` method, which receives and returns an instance of `Source`. As shown in this example, inside the `invoke()` message, the `OrderManager` class gets instantiated and the `OrderManager.getNewOrders()` method is invoked to implement the Web service functionality. So, instead of wrapping the `OrderManager` inside an `HttpServlet.doGet()` method, as in Example 3–14, this example wraps the service implementation class inside a `Provider.invoke()` method.

At this point, it is worth asking the same questions posed in Section 3.5.1. In particular:

1. How is the class instantiated?
2. How is the `Source` instance returned by `getNewOrders()` converted into an XML stream in an HTTP response message?
3. How is the class deployed as a Web service?

As you can see from the code, some of these questions get very different answers when a RESTful Web service is implemented as a `Provider<Source>` than when it is implemented using an `HttpServlet`. These differences serve to contrast the JAX-WS 2.0 approach to Web Services deployment with the straightforward servlet implementation of the preceding section.

The answer to the first question is the same in both cases—the class is instantiated each time the service is invoked. However, the answer to the second question is different in this case. Here, the `Source` instance can be returned directly. It does not need to be converted into a stream and written out to the HTTP response message. These details related to the binding of

the service to the HTTP transport are handled by the JAX-WS run-time implementation. Lastly, the answer to the third question is also very different. Java EE 5 supports many options for deploying Web services—these are all discussed in Chapter 8. A `web.xml` deployment descriptor can be used (even though this is not a servlet!), but is not required. In fact, it is possible to deploy a JWS Web service without any deployment descriptors. The Java EE 5 container can often deploy a service based entirely on its annotations.

Figure 3–11 shows the architecture supporting the RESTful Web service created and deployed here using JAX-WS 2.0.
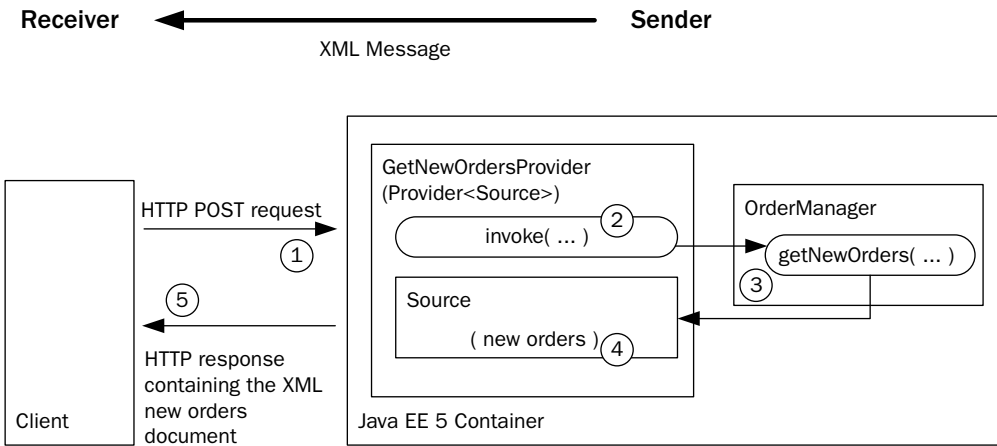

Figure 3–11 RESTful service deployed using `Provider<Source>`.

As illustrated here, this JWS architecture deploys the RESTful Web service using the `java.servlet.http.Provider<Source>` class. The `GetNewOrdersProvider` class acts as a front-end to the `OrderManager` that mediates between the XML request/response messages and invokes the Java method that implements the Web service. The steps illustrated in the figure are:

1. The client sends an HTTP POST request to the appropriate URL (a URL that is specified at deployment time—either in a deployment descriptor or by a deployment tool). Notice that a POST request is used here, rather than a GET request. That is because early implementations of JAX-WS allowed RESTful Web services only to accept POST requests. Recent versions support both POST and GET requests.

2. The JWS container extracts the XML message from the HTTP POST request and passes it to the `GetNewOrders.invoke()` method. This functionality is provided by the JAX-WS runtime.

3. The `invoke()` method creates an instance of the `OrderManager` class and invokes the `getNewOrders()` method.

4. The `getNewOrders()` method returns the new orders XML document as an instance of `javax.xml.transform.Source`.

5. The instance of `Source`—the Web service's return XML message—is inserted into the HTTP response message and returned to the caller. This functionality (i.e., wrapping the XML response message inside the HTTP response message) is provided by the JAX-WS runtime.

In some ways, the JWS implementation of this RESTful Web service is simpler than its servlet-based counterpart discussed in Section 3.5.1. The simplification comes from not having to translate between HTTP request/response messages and the XML request/response messages. JAX-WS handles that translation so that the developer can work directly with the XML messages. In other ways, however, the JWS implementation shown here seems more cumbersome. Two annotations are required—`@WebService-Provider` and `@BindingType`. If you are not used to annotations, these can make the example seem confusing.

To deploy and invoke this RESTful Web service example, do the following:

1. Start GlassFish (if it is not already running).

2. Go to `<book-code>/chap03/rest-get/endpoint-jaxws`.

3. To build and deploy the Web service enter:

```
mvn install
```

... and when that command finishes, then enter:

```
ant deploy
```

4. Go to `<book-code>/chap03/rest-get/client-jaxws`.

5. To run the client enter:

```
ant run-jaxws
```

6. To undeploy the Web service, go back to `<book-code>/chap03/rest-post/endpoint-jaxws` and enter:

```
ant undeploy
```

3.6   Conclusions

In this chapter, I provided a broad introduction to consuming, creating, and deploying RESTful Web Services—using standard `java.net.*` classes and servlets, as well as the JAX-WS 2.0 APIs and annotations. A primary goal of this chapter was to highlight the similarities and differences between traditional Java techniques and the JAX-WS 2.0 approach to Web Services. Another goal was to provide a grounding in some of the basic XML processing techniques used to implement SOA-style integration of RESTful Web services.

In the next chapter, I look at the Java/XML data binding problem and how it can be addressed using traditional JAXP approaches, as well as using JAXB. As I did in this chapter, I compare and contrast the approaches so that you can see the power available to you from the JAXB machinery, but also some of the drawbacks. I help you to determine in which situation JAXB provides the most value and where you are better off using other binding tools.