

J2EE Web services in Geronimo, Part 1: JAX-RPC service endpoints, EJB endpoints, and client APIs

Skill Level: Intermediate

[Stefan Schmidt \(geronimo@stsmidia.net\)](mailto:geronimo@stsmidia.net)
Ph.D. candidate
Developer and Author

30 Aug 2005

Learn how to incorporate Web services into a classic Java™ 2 Platform, Enterprise Edition (J2EE)-based application deployed on the Apache Geronimo application server. Extending the BookShop example application, which lets customers use a Web browser to search a database for books by category, will provide insight into the configurations required to develop Web services-enabled applications for the J2EE 1.4 platform. And by incorporating two J2EE-compliant Web services into BookShop, you'll implement an expanded architecture that addresses both service-consumer and service-provider scenarios.

Section 1. Before you start

This tutorial is Part 1 of a two-part series that explains and demonstrates the integration of enterprise Web services into J2EE running on Geronimo.

About this series

This part of the series demonstrates the integration of service-consumer and service-provider Web services into the existing J2EE-based BookShop application. The second part of the series will demonstrate several additional aspects that are likely to be required when using enterprise Web services in Geronimo: security, attachments, registry access, and logging.

About this tutorial

This tutorial will introduce you to some of the APIs required for the integration of Web services into a classic J2EE application. BookShop is a classic J2EE-based book-selling application that is deployed on Apache Geronimo, a J2EE 1.4-compliant application server. The original BookShop application lets customers search for books based on categories and is exclusively targeted at customers using a Web browser. Search results include the book title, author names, ISBN, description, and price in U.S. dollars. The application consists of a JavaServer Pages (JSP) component, a servlet, one session Enterprise JavaBean (EJB), and three entity EJBs that access a relational database. You'll learn how to extend BookShop by incorporating two J2EE-compliant Web services into the application -- one that delivers current exchange rates to let BookShop calculate book prices in multiple currencies and one that enables business-to-business (B2B) access by nonbrowser clients. The examples you'll work through demonstrate both service-consumer and service-provider scenarios.

The tutorial is structured as follows:

- **Extending BookShop with JAX-RPC** presents an architectural overview of the BookShop example, clarifying the impact on the original application of extending it with Web services. It also gives a brief conceptual overview of the Java APIs for XML-Based Remote Procedure Call (JAX-RPC) and its integration into Geronimo.
- **Exposing Web services from BookShop** demonstrates the steps required on the server side of the BookShop application to expose two Web services.
- **Web services clients in BookShop** shows two ways to access the two external Web services from the J2EE-based BookShop application. It also shows a Java 2 Platform, Standard Edition (J2SE) client application that can access the Web services -- the JAX-RPC service endpoints (JSEs) and EJB endpoints -- exposed by the extended BookShop application.
- **Required deployment artifacts** describes the deployment artifacts necessary for successful implementation of the extended BookShop application.
- **BookShop in action** covers deployment and testing of the BookShop application on both the client and server sides.

Prerequisites

Java developers with prior experience in J2EE platform development are the primary

audience for this tutorial. General knowledge of distributed programming concepts and the role of Web services in this context is beneficial but not required.

System requirements

While this tutorial's example uses MySQL for the deployment steps, you can deploy the application using [Apache Derby](#) by converting the SQL file to make it Derby compliant. Part 2 of this series provides more detail on deployment with Apache Derby.

To run the example code in this tutorial, you need to perform the following steps:

1. Download and install the following applications and libraries:
 - The [J2SE 1.4.2_08 SDK](#)
 - [Apache Geronimo M4 or higher](#)
 - [Apache Ant](#)
 - [Apache Axis 1.2.1](#)
 - [MySQL and MySQL Connector/J driver](#)
 - [Sun Java Web Services Developer Pack \(Java WSDP\)](#)
2. Optionally, download and install the [Eclipse SDK](#).
3. Make sure that the environmental variables in Table 1 are set in your shell.

Table 1. Setting the environmental variables

Variable name	Required setting
GERONIMO_HOME	Set to the root folder of your Geronimo installation
ANT_HOME	Set to the root folder of your Ant installation
JAVA_HOME	Set to the root folder of your Java installation
AXIS_HOME	Set to the root folder of your Axis installation
AXIS_LIBRARIES	Set to all libraries in AXIS_HOME/lib
PROJECT_HOME	Set to the root folder of the BookShop application
PATH	Include JAVA_HOME/bin,

```
ANT_HOME/bin, JWSDP_HOME
and JWSDP_HOME/jaxrpc/bin
```

4. Create a database called `bookshopdb` in MySQL, and then run the supplied database script (`etc/bookshopdb.sql`) to create and populate the required relational-database tables.
5. Create a folder in `%GERONIMO_HOME%/repository/mysql/jars`, and copy the `mysql-connector-java-your_version_number.jar` driver to this directory.
6. Edit the `geronimo-ra.xml` database deployment plan found in `%PROJECT_HOME%/etc` by updating your exact MySQL Connector/J driver name, username, password, and server name as required.
7. Edit the `build.xml` project to set the correct paths for your Geronimo and Axis installations. Also, look into `%GERONIMO_HOME%/repository/tranql/rars` and adjust the TranQL file name in the `build.xml` file accordingly.

Section 2. Extending BookShop with JAX-RPC

This section presents an architectural overview of the BookShop example and the impact of its extension on the original application. You'll also get a brief conceptual overview of JAX-RPC and its integration into Geronimo.

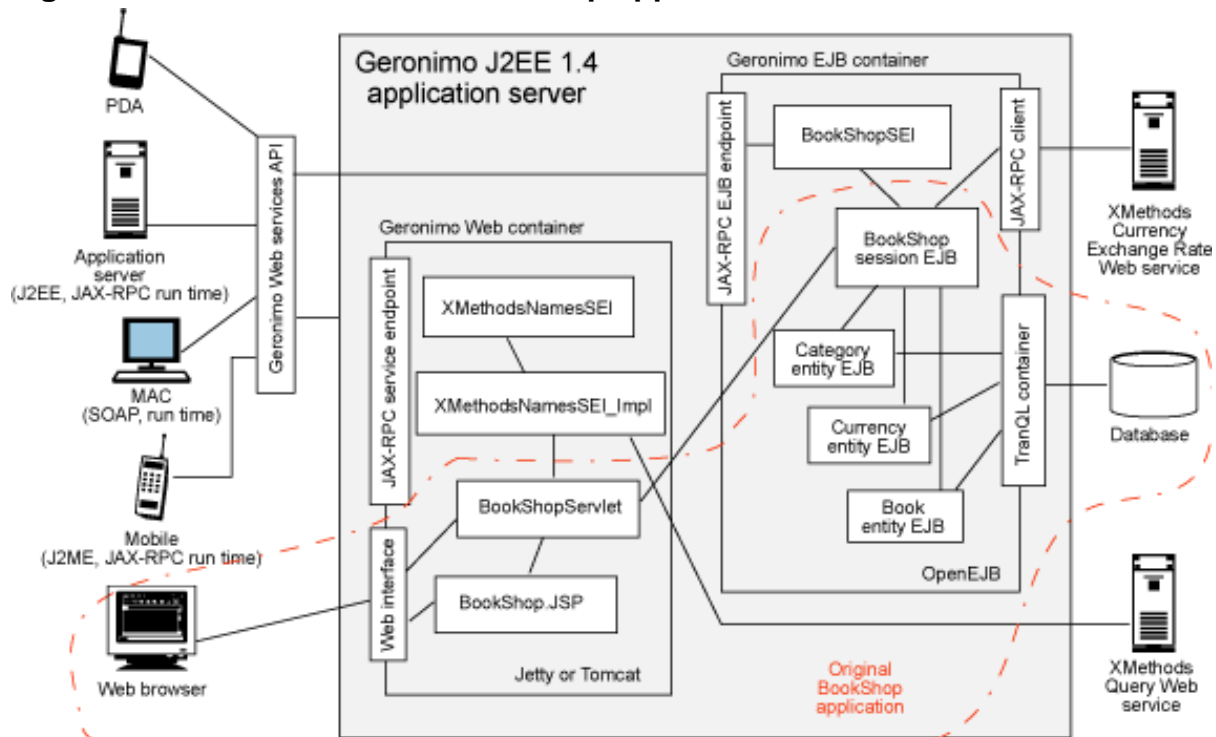
The new BookShop architecture

The original BookShop application lets customers search for books using a Web browser. Suppose your client, the BookShop owner, asks you to extend the J2EE-based application in two ways:

- Offer books to an international audience by dynamically calculating book prices for selected currencies, with currency rates delivered by an external Web service.
- Create an alternative interface to the BookShop application to let business partners use non-browser-based client applications to access the BookShop application.

Figure 1 depicts the BookShop application's architecture. The broken red line indicates the configuration for the original application.

Figure 1. Architecture of the BookShop application



This tutorial introduces you to the APIs required to integrate Web services into a classic J2EE application. You'll expand the BookShop application by adding two JAX-RPC 1.1 Web services endpoints and two JAX-RPC Web service clients. Including JAX-RPC Web services in the BookShop application extends the ways this application can be used in numerous scenarios:

- The public Web services interfaces increase connectivity options to the store dramatically. Customers of the BookShop are not limited to using a Web browser to access the store. They can use practically any device running on a platform with Web services capability to easily connect to the standardized JAX-RPC Web service endpoint interfaces.
- Other businesses can now connect to the BookShop Web service in a business-to-business (B2B) scenario.
- BookShop can use the JAX-RPC client APIs to benefit from other Web services, such as the XMethods.net Currency Exchange Rate Web service and the Query Web service (see [Resources](#) for a link to the Web site).

BookShop demonstrates JAX-RPC service endpoints, EJB endpoints, and client

APIs. The examples include in-depth explanations of the required deployment descriptors, mapping files, and other configuration-related issues.

Introduction to JAX-RPC

If you're already familiar with J2EE Web services specifications -- in particular, JAX-RPC concepts and their implications for J2EE application development -- you can skip to the next section, [Exposing Web services from the BookShop application](#).

The BookShop example exposes and consumes a number of fully J2EE 1.4 platform-compliant Web services. The J2EE 1.4 platform specification integrates various APIs required for the development and deployment of interoperable, standards-compliant Web services and clients. JAX-RPC 1.1, an integral part of the J2EE Web service specification, is an API that enables the creation and execution of remote procedure calls (RPCs) between client applications and remote servers in a distributed architecture. This API supports Web service endpoints running in the Web container (similar to servlets) as JSEs and Web service endpoints based on EJBs running in the EJB container.

JAX-RPC ensures a unified and interoperable model for the integration of Web services technologies, letting you create and process standards-compliant Simple Object Access Protocol (SOAP) messages. This API also helps you create XML-based remote procedure calls and supports Web Service Description Language (WSDL) documents, which describe the published operations, message formats, and service endpoints.

In this tutorial's example, the BookShop application acts as the client to the XMethods.net Web services (see [Resources](#)) and as a remote Web service provider for external clients.

Behind the scenes of JAX-RPC

JAX-RPC reduces complexity for developers by hiding code-intensive Document Object Model (DOM), Simple API for XML (SAX), and Extensible Stylesheet Language Transformations (XSLT) document-parsing procedures. Instead, it conveniently maps Java objects to and from SOAP messages by using XML Schema bindings extracted from WSDL documents. In particular, it performs the following tasks:

- Behind-the-scenes mapping for different scenarios:
 - Java-to-WSDL, WSDL-to-Java
 - Java-to-SOAP, SOAP-to-Java

- Eliminating the need for developers to perform SOAP marshalling and unmarshalling tasks by using XML Schema extracted from WSDL documents
- Standardizing SOAP messages
- Standardizing the marshalling and unmarshalling of parameters and other runtime- and deployment-specific details

JAX-RPC type-mapping capability

Another important design aspect for the BookShop application is the JAX-RPC type-mapping capability. JAX-RPC enables the automatic mapping of a standard set of Java classes and primitive types. However, it's often desirable to pass more complex objects, such as user-defined classes and JavaBeans, to and from the Web service client application. In the BookShop example, you should transport `Book` and `Category` JavaBeans using the Web service. When using complex objects for JAX-RPC type mapping, you must comply with the following rules:

- A public default constructor must exist.
- Complex objects are not allowed to implement the `java.rmi.Remote` interface directly or indirectly. However, any other interface is allowed.
- If the complex object is a JavaBean, it must provide getter and setter methods defining access for each of its private attributes.
- All complex objects must consist of the primitive data types or classes supported by JAX-RPC.
- Only public nonfinal attributes or methods are accessible by the Web service. However, it can contain public, private, protected, and package-level fields.

You can also pass these complex objects as arrays or in collections. The JAX-RPC API does not directly support use of the Java Collections Framework. Instead, you need to supply individual serializers and deserializers -- a topic that is beyond this tutorial's scope.

To this point, you have learned the basics required to understand the integration of Web services into the J2EE platform and the accompanying APIs -- in particular the JAX-RPC API. The next section focuses on two scenarios for the use of JAX-RPC in the BookShop example.

Section 3. Exposing Web services from BookShop

This section defines two scenarios for the use of JAX-RPC to expose a Web service from the BookShop J2EE application:

- Expose the business functionality through a JAX-RPC JSE. This option provides an interface to the Web tier where plain old Java objects (POJOs) serve as an implementation for the Web service interface.
- Expose the business functionality through a JAX-RPC EJB endpoint. This option provides an interface to the EJB tier where a session EJB serves as the implementation for the Web service interface.

Choosing your Web service endpoint

When you create a Web service you need to decide where to place the interface: either in the Web tier or in the EJB tier. The choice is simple. When you're developing a new application, place the Web service endpoint where most of your processing takes place. For example, use a JSE when most of your processing occurs in the Web tier. Or use an EJB endpoint if processing takes place only within the EJB tier.

The decision is slightly more complicated if you're designing a Web service interface for existing applications. When exposing business logic through a Web service in existing applications, place the Web service endpoint in the Web tier if the preprocessing happens there. Place the Web service endpoint in the EJB tier if the preprocessing and processing occurs in this tier.

Other considerations depend on the requirements for your application. For example, if multithreaded access, transactional integrity, or security for method-level access is important to you, an EJB endpoint is probably the better choice.

Generally, Web services are stateless; their underlying business logic should be as well. Still, keep in mind that in some cases it's useful to maintain sessions -- for example, if you require a short conversational interaction with clients. In this case, you should choose a JSE, because it's located in the Web tier and has full access to the `HTTPSession` object.

Now, what about the BookShop example application? Although in the real world you would likely choose only one endpoint, this tutorial demonstrates both. All Web service endpoints require a WSDL document that describes their characteristics. Because you want to provide a Web service interface in front of the existing

BookShop application, use the Java-to-WSDL design approach. This approach generates a WSDL document from an existing Java interface as opposed to the WSDL-to-Java method whereby a Java interface is created from an existing WSDL file.

JAX-RPC service endpoints

If you decide to preprocess incoming or outgoing RPCs before passing the request to the BookShop business logic, and your business logic is integrated into the Web tier, then expose your Web service interface as a JSE. This example exposes one method, `getXMethodsWebServiceNames`, through the `XMethodsNamesSEI` Web service endpoint interface. To accomplish this, add a simple interface class to the original BookShop application code, as shown in [Listing 1](#).

Listing 1. The `XMethodsNamesSEI.java` interface

```
package com.ibm.dw.bookshop.web;

import java.rmi.Remote;
import java.rmi.RemoteException;

/**
 * The service endpoint interface for the XMethodsNames Web service.
 * This interface is required by the JAX-RPC specification
 */
public interface XMethodsNamesSEI extends Remote{

    public String[] getXMethodsWebServiceNames() throws RemoteException;

}
```

As you can see, there's nothing special about this interface. The exposed method must extend the `java.rmi.Remote` interface and throw `java.rmi.RemoteException` for each exposed method, just like implementing `Remote` interfaces in EJBs.

Apart from the interface, you need a class that implements the interface's methods, providing the business logic for the BookShop example. The business logic residing in BookShop's Web tier is implemented in the `com.ibm.dw.bookshop.web.XMethodsNamesSEI_Impl` class, shown in [Listing 2](#).

Listing 2. The `XMethodsNamesSEI_Impl.java` class (abbreviated)

```
package com.ibm.dw.bookshop.web;

import java.rmi.RemoteException;

import javax.naming.InitialContext;
import javax.xml.namespace.QName;
import javax.xml.rpc.Service;

import net.xmethods.query.IDNamePair;
```

```
import net.xmethods.query.XMethodsQuerySoapPortType;

public class XMethodsNamesSEI_Impl implements XMethodsNamesSEI {

    public String[] getXMethodsWebServiceNames() throws RemoteException {
        IDNamePair[] names = getQueryPort().getServiceNamesByPublisher(
            "xmethods.net");
        String[] service_names = new String[names.length];
        for (int i = 0; i < names.length; i++)
            service_names[i] = names[i].getName();
        return service_names;
    }

    private static XMethodsQuerySoapPortType getQueryPort() {
        ... //implementation omitted
    }
}
```

JAX-RPC EJB endpoints

If the Web service that the BookShop application exposes requires multithreaded access, transactional integrity, and security for method-level access, then expose your Web service interface as an EJB endpoint. The technique for developing an EJB endpoint interface, shown in [Listing 3](#), is identical to that used for the XMethodsNamesSEI interface class in [Listing 1](#).

Listing 3. The BookShopSEI.java interface

```
package com.ibm.dw.bookshop.ejb;

import java.rmi.Remote;
import java.rmi.RemoteException;

/**
 * The service endpoint interface for the BookShop Web service.
 * This interface is required by the JAX-RPC specification.
 */
public interface BookShopSEI extends Remote {

    public float getBookPrice(String isbn, String currency) throws RemoteException;
    public Book[] getBooks(String category, String currency) throws RemoteException;
    public Category[] getCategories() throws RemoteException;
    public String[] getCurrencies() throws RemoteException;
}
```

In contrast to the previous JSE example in [Listing 2](#), the implementation class for the EJB endpoint interface is a stateless session EJB. The BookShop session EJB doesn't need to implement the BookShopSEI interface itself (although it doesn't hurt); nevertheless, the method signatures need to be identical. There is one exception: an implementation class for Web service interfaces or normal EJB remote interface classes must never throw the `java.rmi.RemoteException`; however, every other exception must be thrown or handled by the implementation class in convergence with its interface class.

The JAX-RPC run time accomplishes mappings between the XMethodsNamesSEI

interface and incoming and outgoing SOAP messages. The JAX-RPC API supports a standard set of Java classes and primitive types, such as `String` by default, so this parameter is bound implicitly. For custom objects, such as the `BookShop Book` and `Category` JavaBeans, you need to supply mappings that take care of the binding between JavaBeans and SOAP messages. As you'll see in the [Required deployment artifacts](#) section later in this tutorial, you can use the `wsc` tool (see [Resources](#) for a link to the Sun site) to create these JAX-RPC mapping documents.

EJB implementation class

[Listing 4](#) shows the `BookShop` session EJB implementation class, which serves both the `BookShopSEI` and `BookShopServlet` Web service interfaces.

Listing 4. The `BookShopBean.java` class (abbreviated)

```
package com.ibm.dw.bookshop.ejb;

import java.rmi.RemoteException;
import java.util.Collection;
import java.util.Iterator;

import javax.ejb.FinderException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import javax.xml.namespace.QName;
import javax.xml.rpc.Service;

import com.ibm.dw.bookshop.ejb.cmp.*;
import net.xmethods.currencyexchange.CurrencyExchangePortType;

/**
 * Bean implementation class for the BookShop Session Bean
 * and the BookShop EJB service endpoint
 */
public class BookShopBean implements BookShopSEI, SessionBean {

    private String defaultLanguage = "United States";
    private SessionContext ctx;

    public void setSessionContext(SessionContext ctx) {
        this.ctx = ctx;
    }

    public void ejbCreate() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}

    public float getBookPrice(String isbn, String currency) throws RemoteException {
        ... //implementation omitted
    }

    public Book[] getBooks(String catId, String currency) throws RemoteException {
        ... //implementation omitted
    }

    public Category[] getCategories() throws RemoteException {
```

```
        ... //implementation omitted
    }

    public String[] getCurrencies() throws RemoteException {
        ... //implementation omitted
    }
    ... //implementation omitted
}
```

If the BookShop session EJB's only purpose were to provide an implementation for the EJB endpoint interface class, then you wouldn't need to implement the normally required home-interface and either local- or remote-interface classes. But if you want to expose the EJB to a servlet (as in the BookShop example) or a J2EE client directly, you need to implement those interfaces as specified for the J2EE 1.4 platform.

You should think about the optimal granularity of your published operations: their parameters, return values, and thrown exceptions. SOAP roundtrips are quite costly because of SOAP's reliance on XML to carry the payload, so you should reduce the number of roundtrips. Design your interfaces as coarse-grained as possible to reduce expensive remote method calls on your EJB, especially when calling the EJB through a Web service.

Section 4. Web service clients in BookShop

This section outlines how to consume external Web services from within and outside the BookShop server-side implementation. It demonstrates two separate client-side concepts:

- Integration of the XMethods.net Web services into the BookShop application, showing a J2EE client using the JAX-RPC client API
- Consumption of the BookShop Web services from a Swing-based J2SE client application using the Apache Axis API

First you'll get a description of the BookShop example in the greater context of client-side Web services consumption. If you have advanced knowledge of Web services access from both J2EE and J2SE clients, you can skip to the [J2EE Web service clients](#) section.

Client-side Web services

The introduction of Web service concepts to distributed systems helped overcome

many boundaries by significantly improving interoperability between operating systems and programming languages. As [Figure 1](#) shows, Web service clients can take many forms, from mobile clients running on J2ME or Microsoft® .NET to other application servers running on J2EE, ColdFusion, or the .NET Framework. Web services completely decouple BookShop business logic residing on the Geronimo application server from client-side implementations.

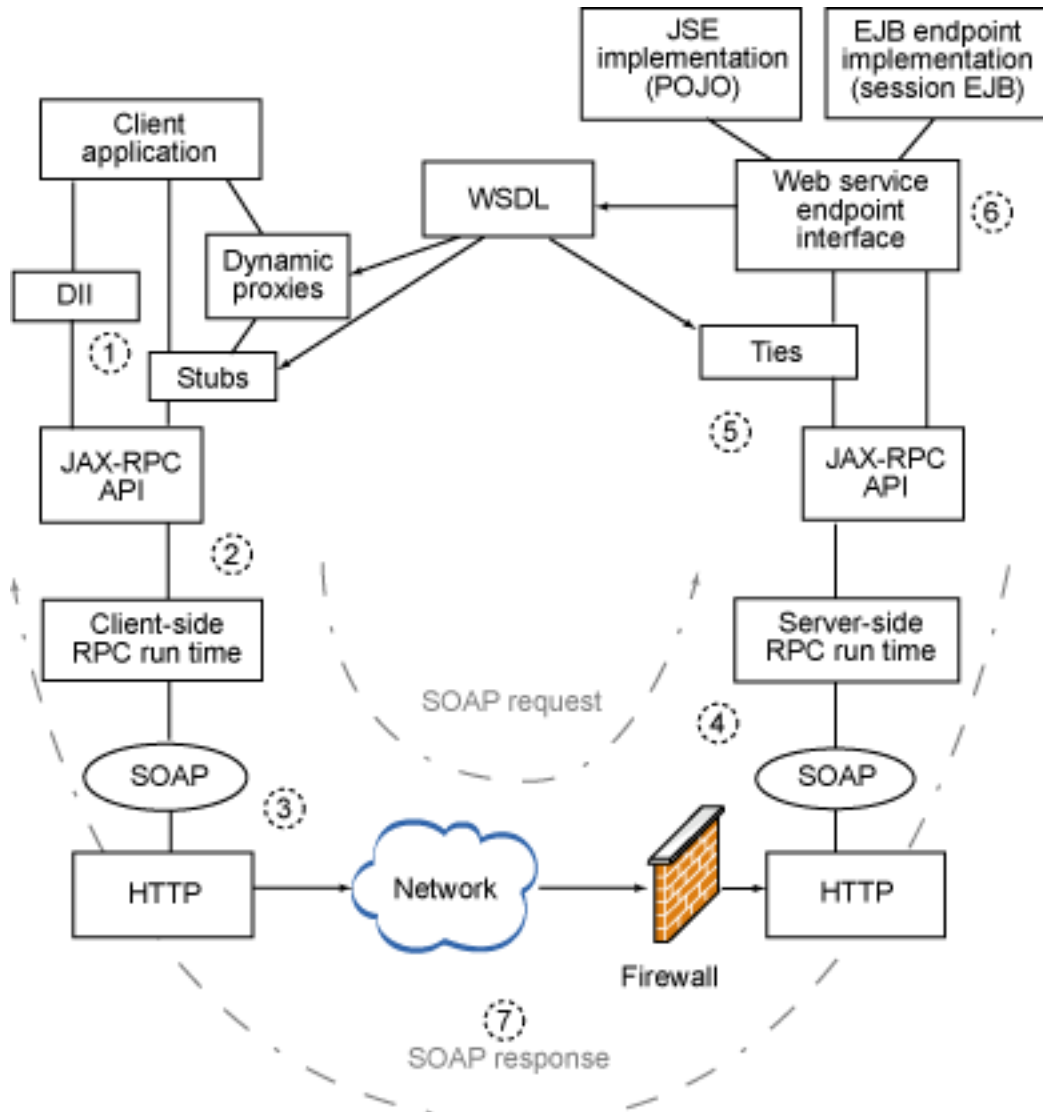
To facilitate the consumption and exposure of Web services from Java applications, several free Web service toolkits are available. This tutorial uses the Geronimo JAX-RPC implementation and the Apache Axis toolkit. You have three options for accessing the JAX-RPC or Axis run time from an application like BookShop:

1. Use *stubs*, which are local objects and interfaces on the client side that represent remote procedures. Typically stubs are automatically generated by tools included in the various Web services developer packs. For this tutorial, you'll use the `wscmpile` tool included in Sun's WSDP and the `WSDL2Java` tool included in the Apache Axis distribution. Short examples demonstrating both tools are covered later in this tutorial.
2. Use a *dynamic proxy* -- for example, to access the `XMethods.net` Web service from the BookShop application. A dynamic proxy alleviates the need for manual generation of stubs at deployment time by creating the necessary objects at run time. Apart from this, the Web service is accessed -- just as with stubs -- through a service endpoint interface (SEI).
3. Use the Dynamic Invocation Interfaces (DII) API, which doesn't require any pregenerated code artifacts, such as stubs and/or service interfaces. Generally, DII is code-intensive, error-prone, and slow. It's mainly used behind the scenes by some tool vendors. However, it provides more flexibility because it doesn't require a WSDL document and it enables access to nonstandard Web services. (This option is not demonstrated in this tutorial.)

High-level JAX-RPC in a J2EE context

To illustrate the three options outlined in the [Client-side Web services](#) section, [Figure 2](#) shows a high-level view of the role of JAX-RPC in the J2EE context.

Figure 2. High-level JAX-RPC architecture



Because this tutorial doesn't cover the use of the Java API for XML Registries (JAXR), assume that the J2EE client is aware of the Web service locations (or at least the location of the WSDL document). The RPC process from the point of view of a J2EE client follows these steps:

1. The client instantiates the dynamic proxy and calls the desired business method that represents the remote procedure. (Other possibilities for accessing a Web service from the client side are the use of stub objects and DII.)
2. The stub uses the JAX-RPC API to initiate and execute all necessary routines for the creation of the SOAP-based RPC.
3. The RPC run time transmits the message to the server as an HTTP

request.

4. The SOAP message is transported over the network through the server-side firewall to the server-side RPC run time.
5. Geronimo's JAX-RPC run time implementation maps the SOAP call to the tie object (server-side equivalent of a stub object) and invokes the corresponding method on the tie object.
6. The tie object calls the method on the Web service SEI, which invokes its implementation class and contains the actual business logic.
7. The response to the RPC is sent in the same manner back to the client application.

The scenario for accessing a Web service from the BookShop J2SE-based client application is similar to the process depicted in [Figure 2](#). The only difference is that this application uses the Axis API and run time instead of the JAX-RPC API and run time on the client side.

J2EE Web service clients

Now you'll build one of the two BookShop Web service clients. The XMethods.net Currency Exchange Web service is accessed from the EJB tier through a dynamic proxy. Dynamic proxies are the most convenient method of invoking a Web service from a J2EE application. They have an advantage over stubs in that they are independent of a specific JAX-RPC run time. When you use dynamic proxies, the JAX-RPC run time creates the stubs dynamically to gain access to the Web service.

Start with the J2EE application client. First you need to generate the required proxy classes as defined in the XMethods.net Currency Exchange Rate Web service's WSDL document. These proxy classes serve as a client-side endpoint interface to the XMethods.net Web services. A number of client-side classes (JavaBeans) that represent the Java implementation of complex types delivered by the Web service must also be generated. Fortunately, the wscompile tool generates these artifacts for the published XMethods.net Currency Exchange Rate Web service's WSDL document.

Generating client-side artifacts

The artifacts required by the JAX-RPC client include a JAX-RPC mapping file, Java interfaces, and stub objects. First you must create a config.xml file, as shown in [Listing 5](#), which specifies the location of the WSDL file:

Listing 5. config.xml file identifying WSDL file location

```

        <configuration
xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <wsdl
location="http://www.xmethods.net/sd/2001/CurrencyExchangeService.wsdl"
  packageName="net.xmethods.currencyexchange"/>
</configuration>

```

Using this configuration file, you can then use `wscompile` to generate the desired client-side artifacts:

```

% wscompile -cp %PROJECT_HOME%/classes -d
%PROJECT_HOME%/classes -gen:client
-f:strict -mapping
%PROJECT_HOME%/meta/ejb/client-jaxrpc-mapping.xml
%PROJECT_HOME%/etc/currency-client-config.xml

```

Note: The `wscompile` command's `-d` option specifies where the Java interfaces and stub classes used by the client application are to be created. Because the Java WSDP creates only precompiled Java class files and not source code, I decided to pack these auto-generated classes into a separate Java archive and pack it within the application. This means that you need to place them into the `WEB-INF/lib` directory if you access the Web service within a Web archive (WAR) application. When accessing the Web service from an EJB, include the prepacked library in your enterprise archive (EAR) and reference the location in the manifest file within your archive as a classpath element.

You also need to include a JAX-RPC mapping file and add a `service-ref` element (see [Listing 6](#)) to the deployment descriptor (`ejb-jar.xml` or `web.xml`).

Listing 6. service-ref element (ejb-jar.xml example)

```

...
<service-ref>
  ...
  <service-ref-name>service/Service</service-ref-name>
  <service-interface>javax.xml.rpc.Service</service-interface>
  <!-- The WSDL document must be packed within the J2EE
application! -->
  <wsdl-file>META-INF/wsdl/CurrencyExchangeService.wsdl</wsdl-file>
  <!-- A JAX-RPC mapping file must be packed within the J2EE
application! -->
  <jaxrpc-mapping-file>META-INF/client-jaxrpc-mapping.xml</jaxrpc-mapping-file>
  <service-qname
xmlns:ns="http://www.xmethods.net/sd/CurrencyExchangeService.wsdl">
    ns:CurrencyExchangeService
  </service-qname>
</service-ref>
...

```

The `service-ref` element maps the Web service-related artifacts to a Java Naming and Directory Interface (JNDI) resource name. This makes it easy to use a Web service within a J2EE application.

Invoking the Currency Exchange Rate Web service

Now that all artifacts required by the JAX-RPC client API are in place, you're ready to invoke the Currency Exchange Rate Web service from the BookShop EJB. The Web service returns the requested currency rate, which is then multiplied by a book price. The result is delivered either to a servlet or the EJB Web service endpoint. You can see this in [Listing 7](#), which shows the code for a session EJB client.

Listing 7. BookShopBean.java (abbreviated) -- a session EJB JAX-RPC client example

```
package com.ibm.dw.bookshop.ejb;

//imports omitted

/**
 * Bean implementation class for the BookShop Session Bean
 * and the BookShop EJB service endpoint
 */
public class BookShopBean implements BookShopSEI, SessionBean {

    private String defaultCurrency = "USD";
    private SessionContext ctx;
    public void setSessionContext(SessionContext ctx) {
        this.ctx = ctx;
    }
    public void ejbCreate() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}

    /**
     * Exposed business logic accessible via a servlet or Web service
     */
    public float getBookPrice(String isbn, String currency) {
        try {
            BookRemote book = getBookRef().findByPrimaryKey(isbn.trim());
            return convertCurrency(book.getPrice().floatValue(), currency);
        } catch (Exception re) {
            re.printStackTrace();
        }
        return 0f;
    }

    /**
     * Exposed business logic accessible via a servlet or Web service
     */
    public Book[] getBooks(String catId, String currency) {
        Book[] books = null;
        try {
            Collection coll = getBookRef().findByCategory(catId);
            Iterator it = coll.iterator();
            books = new Book[coll.size()];

            int i = 0;
            while (it.hasNext()) {
                BookRemote book = (BookRemote) it.next();

                books[i++] = new Book(book.getIsbn(), book.getTitle(), book.getAuthor(),
                    book.getDescription(),
                    convertCurrency(book.getPrice().floatValue(), currency));
            }
        } catch (Exception re) {
```

```

        re.printStackTrace();
    }
    return books;
}

/**
 * Access the XMethods Currency Exchange Rate Web service and convert the currency
 */
private float convertCurrency(float basePrice, String currAbbreviation) {
    float currencyFactor = 1.0f;
    float price = 0f;
    String currencyName = "";
    try {
        if (!currAbbreviation.equalsIgnoreCase("USD")) {
            Iterator i = getCurrencyRef().findByAbbreviation(currAbbreviation).iterator();

            while (i.hasNext())
                currencyName = ((CurrencyRemote) i.next()).getName();

            xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance">currency
Factor =
                getCurrencyServiceProxy().getRate("United States", currencyName);
        }
        return Math.round(basePrice * currencyFactor * 100) / 100f;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return 0f;
}

/**
 * Encapsulation of JNDI lookup and dynamic proxy generation
 * for the Exchange Rate Web service
 *
 * @return CurrencyExchangePortType
 */
private static CurrencyExchangePortType getCurrencyServiceProxy() {
    CurrencyExchangePortType proxy = null;
    try {
        xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance">InitialContext jndi =
new InitialContext();
        Service service = (Service) jndi.lookup("java:comp/env/service/Service");
        QName portName = new QName
            ("http://www.xmethods.net/sd/CurrencyExchange
Service.wsdl",
                "CurrencyExchangePort");
        proxy = (CurrencyExchangePortType) service.getPort(portName,
            CurrencyExchangePortType.class);
    } catch (Exception e) {e.printStackTrace();}
    return proxy;
}
... //implementation omitted
}

```

A J2SE client

To enable access to the BookShop Web services through a J2SE client, you'll create a simple Swing application. This application accesses the BookShop Web service and the XMethodsNames Web service using stubs generated by Apache Axis, as shown in [Listing 8](#).

Listing 8. Ant task for J2SE stub generation (in build.xml)

```

        <!-- declare the 'axis-wsd12java' task within Ant -->
<taskdef resource="axis-tasks.properties" classpathref="compile.class.path" />

<target name="wsdl2java" description="Create client side stubs for the J2SE Client
using AXIS">
  <!-- Create stub for BookShop Web service stubs and interfaces -->
  <axis-wsd12java output="${src}" testcase="false" verbose="true"
    url="meta/wsdl/BookShop.wsdl">
    <mapping namespace="http://ibm.com.dw.bookshop.webservice"
      package="com.ibm.dw.bookshop.j2seclient.ejb.generated" />
  </axis-wsd12java>

  <!-- Create stub for XMethodsNames Web service stubs and interfaces -->
  <axis-wsd12java output="${src}" testcase="false" verbose="true"
    url="meta/wsdl/XMethodsNames.wsdl">
    <mapping namespace="http://ibm.com.dw.bookshop.webservice"
      package="com.ibm.dw.bookshop.j2seclient.servlet.generated" />
  </axis-wsd12java>
</target>

```

Within the client-application class, you can then use the generated stubs to invoke both Web services exposed by the BookShop server-side implementation, as shown in [Listing 9](#).

Listing 9. BookShopClientFrame.java (abbreviated)

```

        package com.ibm.dw.bookshop.j2seclient;

...
import com.ibm.dw.bookshop.j2seclient.ejb.generated.*;
import com.ibm.dw.bookshop.j2seclient.servlet.generated.*;

public class BookShopClientFrame extends JFrame implements ActionListener {
    ... //Swing implementation omitted
    /**
     * invoke getCategories() and getCurrencies() methods on the EJB
     * Web service and fill the results in JComboBoxes
     */
    private void executeStep1() {
        try {
            catBox.removeAllItems();
            Category[] cat = getBookShopPort().getCategories();

            for (int i = 0; i < cat.length; i++) catBox.addItem(cat[i]);

            catBox.setEnabled(true);
            currBox.removeAllItems();

            String[] curr = getBookShopPort().getCurrencies();

            for (int i = 0; i < curr.length; i++) currBox.addItem(curr[i].toUpperCase());

            currBox.setEnabled(true);
        } catch (Exception e) {e.printStackTrace();}
        step1.setEnabled(false);
    }

    /**
     * invoke getBooks(String category, String currency) method on the EJB
     * Web service and fill the results into a JTextArea
     */
    private void executeStep2() {
        try {
            Book[] books = getBookShopPort().getBooks(
                ((Category) catBox.getSelectedItem()).getCatId(),

```

```

        (String) currBox.getSelectedItem());

    StringBuffer text = new StringBuffer();
    for (int i = 0; i < books.length; i++)
        text.append("Title: "+books[i].getTitle()
            +"\nAuthor: "+books[i].getAuthor()
            +"\nISBN: "+books[i].getIsbn()
            +"\nDescription: "+books[i].getDescription()
            +"\nPrice: "+books[i].getPrice()
            +"\n-----\n");

    bookResults.setText(text.toString());
    bookResults.setEnabled(true);
    bookResults.setCaretPosition(0);
    step2.setText("Try Again!");
} catch (Exception e) {e.printStackTrace();}
}

/**
 * invoke getXMethodsWebServiceNames() method on the Query Web service
 * and fill the results into a JTextArea
 */
private void invokeServletSEI(){
    String serviceNamesBuffer = new String();
    try{
        String [] names = getXMethodsNamesPort().getXMethodsWebServiceNames();

        for (int i = 0; i < names.length; i++) serviceNamesBuffer +=
            names[i]+"\\n";
    }catch(Exception e){e.printStackTrace();}
    serviceNames.setText(serviceNamesBuffer);
}

/**
 * This is a little helper method which locates and initializes the
 * generated BookShop Web service stub
 */
private BookShopSEI getBookShopPort() {
    BookShopLocator locator = new BookShopLocator();
    try {
        return locator.getBookShopSEIPort();
    } catch (Exception e) {e.printStackTrace();}
    return null;
}

/**
 * This is a little helper method which locates and initializes the
 * generated BookShopServlet Web service stub
 */
private BookShopServletSEI getBookShopServletWebServicePort() {
    XMethodsNamesLocator locator = new XMethodsNamesLocator();
    try {
        return locator.getXMethodsNamesSEIPort();
    } catch (Exception e) {e.printStackTrace();}
    return null;
}
}

```

The J2SE example application demonstrates how simple it is to generate and use stubs to access Web services using the Apache Axis API. Three lines of code in Listing 9 provide access to a Web service: the instantiation of the locator object, the retrieval of the generated Web service interface object from the locator, and the invocation of the Web service operations on the interface.

The next section outlines the configuration files necessary for integrating Web

services into the BookShop application as required by the J2EE platform.

Section 5. Required deployment artifacts

In this section you'll learn about the configurations required for successful deployment of the BookShop application on the Geronimo application server. You'll start with JAX-RPC mapping files, learning how you can use an external tool to generate them. Next, you'll examine the Web service deployment descriptor for the `XMethodsNames` Web service for the BookShop application. Finally, you'll find out how to make the necessary adjustments to deployment descriptors and how to package all the necessary elements to deploy the BookShop example successfully.

JAX-RPC mapping files

Geronimo's JAX-RPC run time requires a JAX-RPC mapping file. The mapping file specifies the exact associations between the BookShop interfaces that embody the `BookShopSEI` and `XMethodsNamesSEI` Web service endpoints and their corresponding WSDL documents. The following code examples demonstrate the generation of the server-side mapping document.

Geronimo complies with the J2EE platform specifications, which state that one JAX-RPC mapping file must exist for each Web service (consumed or exposed) WSDL file. You might be wondering why the WSDL document is not descriptive enough in its own right. In most cases the mappings are straightforward, which would make the mapping file redundant. But in some cases the requirement for the mapping file is justified -- for example, for custom errors, some complex types, and serializers.

JAX-RPC mapping files are probably the most complex of the artifacts you've seen so far. The good news is that the Java WSDP's `wscompile` command-line tool, which you first encountered in the [Exposing Web services from BookShop](#) section, takes care of generating the JAX-RPC mapping documents for the BookShop application.

The `wscompile` tool itself requires an XML-based configuration file that specifies the settings that `wscompile` needs. [Listing 10](#) shows the server-side configuration file.

Listing 10. bookshop-server-config.xml

```
<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <service name="BookShop"
    targetNamespace="http://ibm.com.dw.bookshop.webservice"
    typeNamespace="http://ibm.com.dw.bookshop.webservice"
    packageName="com.ibm.dw.bookshop.ejb">
```

```

<interface
  name="com.ibm.dw.bookshop.ejb.BookShopSEI"
  servantName="com.ibm.dw.bookshop.ejb.BookShopBean">
</interface>
<typeMappingRegistry>
  <additionalTypes>
    <class name="com.ibm.dw.bookshop.ejb.Book"/>
    <class name="com.ibm.dw.bookshop.ejb.Category"/>
  </additionalTypes>
</typeMappingRegistry>
</service>
</configuration>

```

As you can see, this configuration file is not too complex. The wscompile tool requires the following pieces of information in order to generate the required artifacts:

- A service name
- Namespaces
- Endpoint interface and implementation classes

Optionally, the configuration file can also include complex types that are transported through SOAP.

The following command uses information in the bookshop-server-config.xml file to generate a mapping file for the EJB-tier Web service and a WSDL file:

```

% wscompile -cp %PROJECT_HOME%/classes -gen:server -f:strict -mapping
%PROJECT_HOME%/meta/ejb/server-jaxrpc-mapping.xml
%PROJECT_HOME%/etc/bookshop-server-config.xml

```

A second command uses information in the bookshop-server-config.xml file to generate a mapping file for the Web-tier Web service and a WSDL file:

```

% wscompile -cp %PROJECT_HOME%/classes -gen:server -f:strict -mapping
%PROJECT_HOME%/meta/war/server-jaxrpc-mapping.xml
%PROJECT_HOME%/etc/xmethods-server-config.xml

```

Understanding WSDL documents

Now you can take a look at the generated WSDL document. [Listing 11](#) shows only the XMethodsNames.wsdl document. Go to the sample code's meta/wSDL/ directory to find the BookShop.wsdl document.

Listing 11. XMethodsNames.wsdl (abbreviated)

```

<types>
    <definitions name="XMethodsNames" ...>

```

```

    <schema targetNamespace="http://ibm.com.dw.bookshop.webservice" ...>
      <import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
      <complexType name="ArrayOfstring">
        <complexContent>
          <restriction base="soap11-enc:Array">
            <attribute ref="soap11-enc:arrayType" wsdl:arrayType="string[]" />
          </restriction>
        </complexContent>
      </complexType>
    </schema>
  </types>
  <message name="XMethodsNamesSEI_getXMethodsWebServiceNames" />
  <message name="XMethodsNamesSEI_getXMethodsWebServiceNamesResponse">
    <part name="result" type="tns:ArrayOfstring" />
  </message>
  <portType name="XMethodsNamesSEI">
    <operation name="getXMethodsWebServiceNames">
      <input message="tns:XMethodsNamesSEI_getXMethodsWebServiceNames" />
      <output message="tns:XMethodsNamesSEI_getXMethodsWebServiceNamesResponse" />
    </operation>
  </portType>
  <binding name="XMethodsNamesSEIBinding" type="tns:XMethodsNamesSEI">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getXMethodsWebServiceNames">
      <soap:operation
        soapAction="http://ibm.com.dw.bookshop.webservice/getXMethodsWebServiceNames" />
      <input>
        <soap:body use="encoded" namespace="http://ibm.com.dw.bookshop.webservice"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding" />
      </input>
      <output>
        <soap:body use="encoded" namespace="http://ibm.com.dw.bookshop.webservice"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding" />
      </output>
    </operation>
  </binding>
  <service name="XMethodsNames">
    <port name="XMethodsNamesSEIPort" binding="tns:XMethodsNamesSEIBinding">
      <soap:address location="REPLACE_WITH_ACTUAL_URL" />
    </port>
  </service>
</definitions>

```

By analyzing the WSDL document in [Listing 11](#), a client application can bind to the exposed `XMethodsNames` Web service. The `types` element defines XML Schema-based objects that correspond to the exposed JavaBeans. (For example, the `BookShop.wsdl` document defines two complex types -- the `Book` and `Category` JavaBeans.) The `portType` element specifies the service interface, and the `port`, `binding`, and `message` elements define details required to generate stubs. Finally, the `service` element provides information about the protocols supported by the Web service and the URL that must be used to invoke the Web service.

The generated WSDL document contains a wild card for the URL where the Web service will be published. Replace this wild card with an actual URL for the `BookShop` Web services:

Listing 12. Generated WSDL document

```
<!-- This is the URL wild card: -->
```

```
<soap:address location="REPLACE_WITH_ACTUAL_URL" />

<!-- change this to the following for the JSE Web service -->
<soap:address location="http://localhost:8080/DWBookShop/XMethodsNames" />

<!-- change this to the following for the EJB endpoint Web service -->
<soap:address location="http://localhost:8080/DWBookShop/BookShopWebService" />
```

Web service deployment descriptors

The Web service deployment descriptor document specifies details about the BookShop Web service that will be deployed on Geronimo. These details include dependencies in relation to the BookShop resources, such as a link to XMethodsNamesWS servlet mapping or the BookShop session EJB.

All necessary links to mapping files, deployment descriptors, and interface classes are referenced within the Web service deployment descriptor. In detail, the descriptor contains:

- A reference to the WSDL document: `<wsdl-file>`
- A reference to the JAX-RPC mapping file: `<jaxrpc-mapping-file>`
- A unique logical name for each port (as defined in the WSDL document): `<port-component-name>`
- A QName for the port (as defined in the WSDL document): `<wsdl-port>`
- A service endpoint interface for the port: `<service-endpoint-interface>`
- A reference to the implementation class of the service endpoint interface: `<service-impl-bean>` -- either an `<ejb-link>` element (for EJB endpoints) or a `<servlet-link>` element (for JSE)
- Optional references to message handlers (which will be covered in Part 2 of this tutorial series)

[Listing 13](#) shows the BookShop application's webservices.xml deployment descriptor document as integrated into the WAR file.

Listing 13. webservices.xml deployment descriptor

```
    <webservices xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://www.ibm.com/webservices/xsd/j2ee_web_services_1_1.xsd"
version="1.1">
<web-service-description>
<web-service-description-name>XMethodsNames Web Service
```

```
                </webservice-description-name>
<wsdl-file>WEB-INF/wsdl/XMethodsNames.wsdl</wsdl-file>
<jaxrpc-mapping-file>WEB-INF/server-jaxrpc-mapping.xml</jaxrpc-mapping-file>
<port-component>
  <port-component-name>XMethodsNames</port-component-name>
  <wsdl-port>XMethodsNamesSEIPort</wsdl-port>
  <service-endpoint-interface>
    com.ibm.dw.bookshop.web.XMethodsNamesSEI</service-endpoint-interface>
  <service-impl-bean>
    <servlet-link>XMethodsNamesWS</servlet-link>
  </service-impl-bean>
</port-component>
</webservice-description>
</webservices>
```

The `service-impl-bean` element links either to the BookShop session EJB endpoint (defined in the `ejb-jar.xml` deployment descriptor) or JSE (defined in the `web.xml` deployment descriptor). This link element connects the Web service port, which is described in the WSDL file, to the actual endpoint implementation within your J2EE application.

Now that you've created all necessary Java classes, deployment descriptors, and mapping files, you can package all archives and assemble them into one J2EE application archive (`DWBookShop.ear`).

J2EE deployment descriptors

In the previous sections you created Web service endpoint interfaces and their implementations for both the BookShop Web-tier application and the BookShop EJB-tier application. So far in this section, you've generated WSDL documents and their associated mapping files using the `wsc` tool. To deploy these applications, you now need to package the Java classes with several general deployment descriptors required by the J2EE platform and with Geronimo-specific deployment descriptors.

In addition to the usual descriptors (see the [Deployment descriptors from the original BookShop application](#) section), you need to include the following JAX-RPC-specific documents:

- A **WSDL document** (`BookShop.wsdl` for the EJB endpoint and `XMethodsNames.wsdl` for the JSE) describing the exposed operations as a set of endpoints containing either document-oriented or procedure-oriented information. The WSDL file that the J2EE application publishes can then be used by client applications to either generate stubs or dynamic proxies or to invoke the Web service dynamically using DII.
- A **JAX-RPC mapping file** (two `server-jaxrpc-mapping.xml` documents for the published Web services and two `client-jaxrpc-mapping.xml` documents for the consumed Web services). These documents, required

by the JAX-RPC run time, are responsible for the mapping between Web service endpoint interfaces, such as EJB endpoints and JSE, and the WSDL document.

- A **Web service descriptor file** (two webservices.xml files for both Web tier and EJB tier). This is an additional document that outsources part of the Web service-specific information required by J2EE and therefore by Geronimo. In short, the Web service deployment descriptor links the BookShop WSDL ports' information to their implementation classes.

Deployment descriptors from the original BookShop application

At this point, you've added several interfaces, deployment descriptors, and configuration files to the original BookShop application. Now you need to look at the standard deployment descriptors in the original BookShop application. These deployment descriptors are the web.xml document (see [Listing 14](#)) within the Web-tier component and the ejb-jar.xml document within the EJB-tier component of the BookShop. You must add references in these documents to the new Web services so that the Geronimo application server can recognize them.

For the JSE Web service endpoint in BookShop, you need to divert Web service calls to the JSE endpoint implementation class (XMethodsNamesSEI_Impl) and direct Web interface calls directly to the servlet (BookShopServlet), as shown in [Listing 14](#).

Listing 14. The web.xml deployment descriptor (abbreviated)

```

        <?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
        "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <!-- Mapping used for Web requests -->
  <servlet>
    <servlet-name>BookShop</servlet-name>
    <servlet-class>com.ibm.dw.bookshop.web.BookShopServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>BookShop</servlet-name>
    <url-pattern>/Web</url-pattern>
  </servlet-mapping>

  <!-- Mapping used for Web service (SOAP) requests -->
  <servlet>
    <servlet-name>XMethodsNamesWS</servlet-name>
    <servlet-class>com.ibm.dw.bookshop.web.XMethodsNamesSEI_Impl</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>XMethodsNamesWS</servlet-name>
    <url-pattern>/XMethodsNames</url-pattern>
  </servlet-mapping>
  . . .
</web-app>

```

The procedure is different for the BookShop session EJB endpoint. You need to

include a `service-endpoint` element in the `ejb-jar.xml` deployment descriptor (see [Listing 15](#)). This element points to the EJB endpoint interface `BookShopSEI`, as opposed to the implementation class in the JSE case (`XMethodsNamesSEI_Impl`).

Listing 15. The `ejb-jar.xml` deployment descriptor (abbreviated)

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar ...>
  <enterprise-beans>
    <session>
      <ejb-name>BookShopEJB</ejb-name>
      <local-home>com.ibm.dw.bookshop.ejb.BookShopLocalHome</local-home>
      <local>com.ibm.dw.bookshop.ejb.BookShopLocal</local>
      <!-- This service endpoint interface definition is required by JAX-RPC
           to expose the BookShop session EJB as a J2EE Web service -->
      <service-endpoint>com.ibm.dw.bookshop.ejb.BookShopSEI</service-endpoint>
      <ejb-class>com.ibm.dw.bookshop.ejb.BookShopBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    ...
  </ejb-jar>
```

Section 6. BookShop in action

In this section, you'll finalize the extended BookShop application by packaging it, deploying it, and testing the successful implementation on both the server side and the client side.

Packaging the archives

The BookShop application is composed of three main components:

- An archive containing all container-managed entity EJBs (packaged in `DWBookShop-cmp-ejb.jar`)
- An archive containing the BookShop session EJB and its Web service endpoint interface (packaged in `DWBookShop-ejb.jar`)
- The Web archive containing the JSP, the servlet, and the `XMethodsNames` Web service interface and its implementation object (packaged in `DWBookShop.war`).

You can use the Ant build task named `ear`, provided with the sample code, to package the three archives into a single J2EE archive (`DWBookShop.ear`) to facilitate deployment to the Geronimo J2EE application server. The directory trees

shown in Figure 3 and Figure 4 illustrate the packaging of these archives.

Figure 3. DWBookShop-cmp-ejb.jar (left) and DWBookShop-ejb.jar (right)

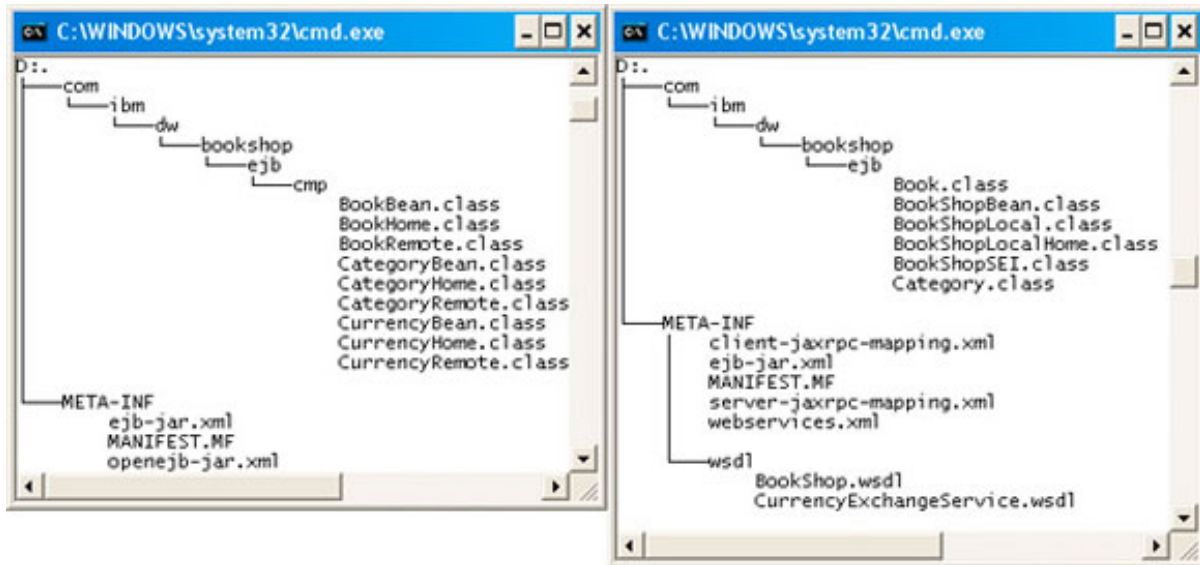
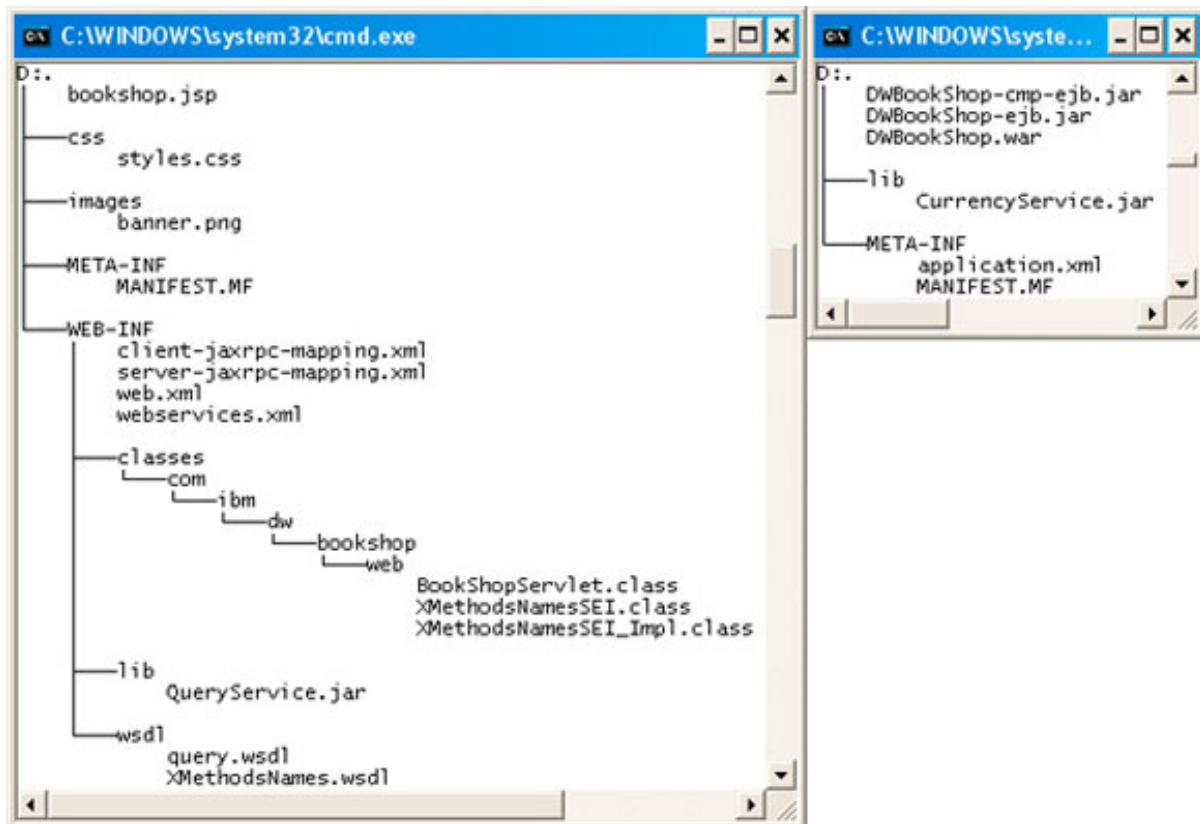


Figure 4. DWBookShop.war (left) and DWBookShop.ear (right)



Deploying BookShop

Now that you've packaged the BookShop application, start Geronimo with the following command:

```
% %GERONIMO_HOME% java - jar bin\server.jar
```

You also must deploy the Geronimo-specific deployment plan (geronimo-ra.xml, supplied in the sample code), which uses the TranQL JDBC connector included in Geronimo to load the MySQL Connector/J driver. You can do this by either running the supplied Ant task (deploy-database) or the following command in a new shell (adjust the TranQL file name accordingly):

```
D:\Geronimo> java - jar bin\deployer.jar --user system --password  
system deploy repository/tranql/rars/tranql-connector-1.0-20050716.rar geronimo-ra.xml
```

Finally, you can deploy the BookShop application (BookShop.ear) in Geronimo:

```
D:\Geronimo> java - jar bin\deployer.jar --user system  
--password system deploy ejb-app.jar
```

Alternatively, you can integrate the deployment command as an Ant task in your project's build file, as shown in [Listing 16](#).

Listing 16. Integrating the deployment command as an Ant task

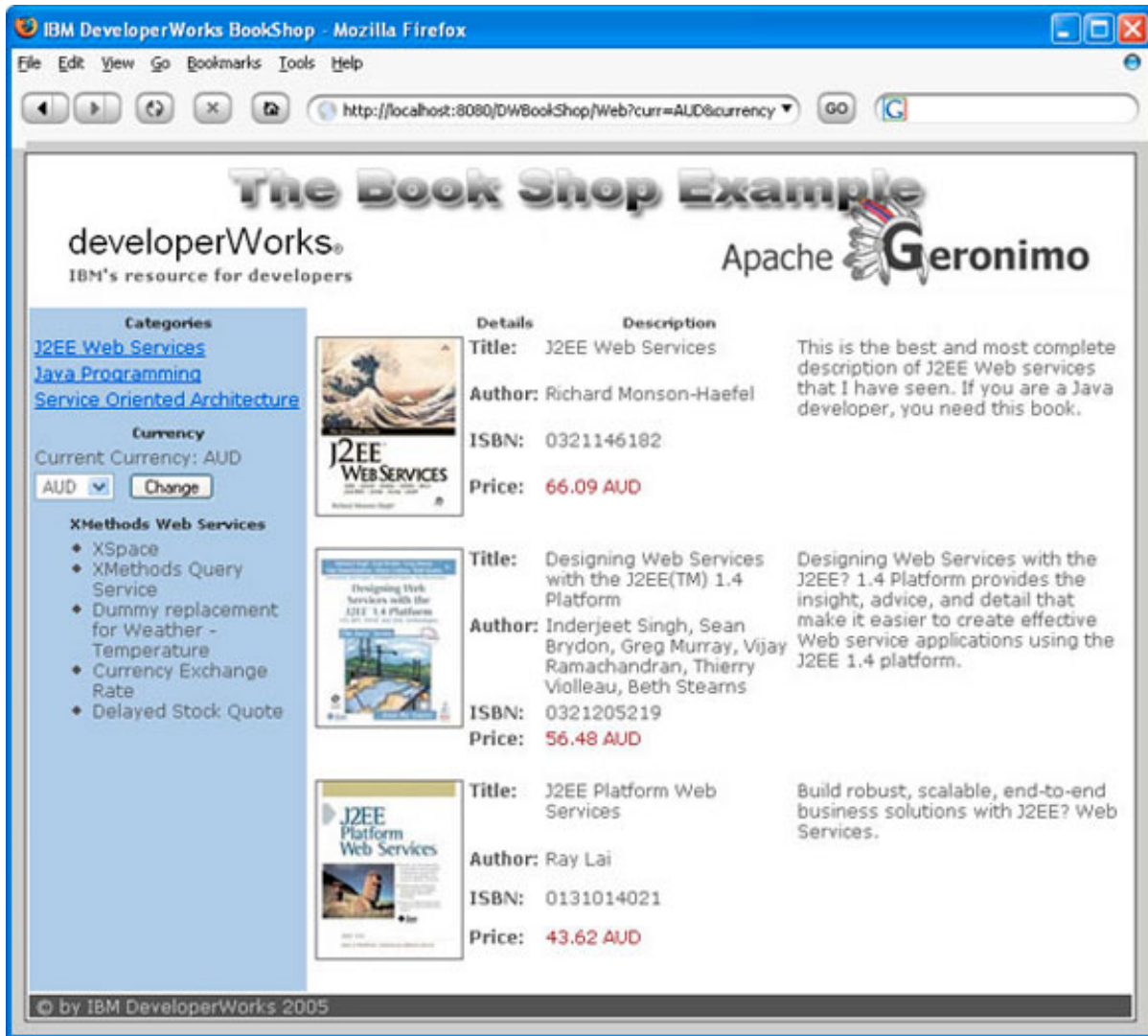
```
<target name="deploy-ejb" depends="ejbjar,undeploy-ejb">  
  <java jar="${geronimo.home}/bin/deployer.jar" fork="true">  
    <arg value="--user" />  
    <arg value="${geronimo.user}" />  
    <arg value="--password" />  
    <arg value="${geronimo.password}" />  
    <arg value="deploy" />  
    <arg value="${build.out}/${app.ejb.name}" />  
  </java>  
</target>
```

To confirm a successful deployment, enter the following URL in your Web browser:

```
http://localhost:8080/DWBookShop/Web
```

You should be able to see the BookShop storefront, as shown in [Figure 5](#).

Figure 5. The BookShop storefront



The JSP page in [Figure 5](#) displays the query results from the local database as well as book prices converted to the selected currency. A list of XMethods.net Web service names is displayed underneath the menu. These results confirm that the JAX-RPC client Web service implementation works correctly.

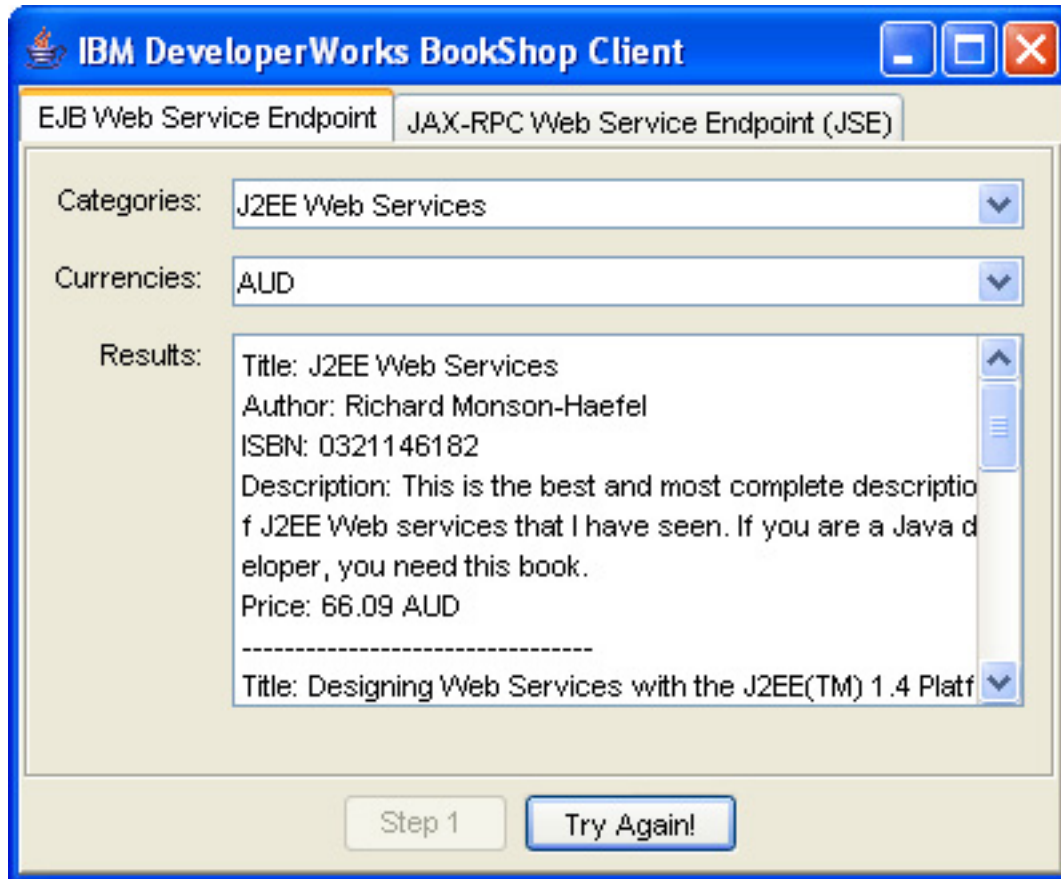
Testing the Web services

Now you want to test the Web services exposed by the BookShop application. To launch the client application, use the Ant task included in the build.xml file. The J2SE client application provides access to both Web services that are exposed by the BookShop application. The EJB Web Service Endpoint tab demonstrates how to access the BookShop Web service in two steps (see [Figure 6](#)):

1. Retrieve available categories and currencies from the book database.

2. Retrieve all books for a selected category with book prices already converted into the selected currency.

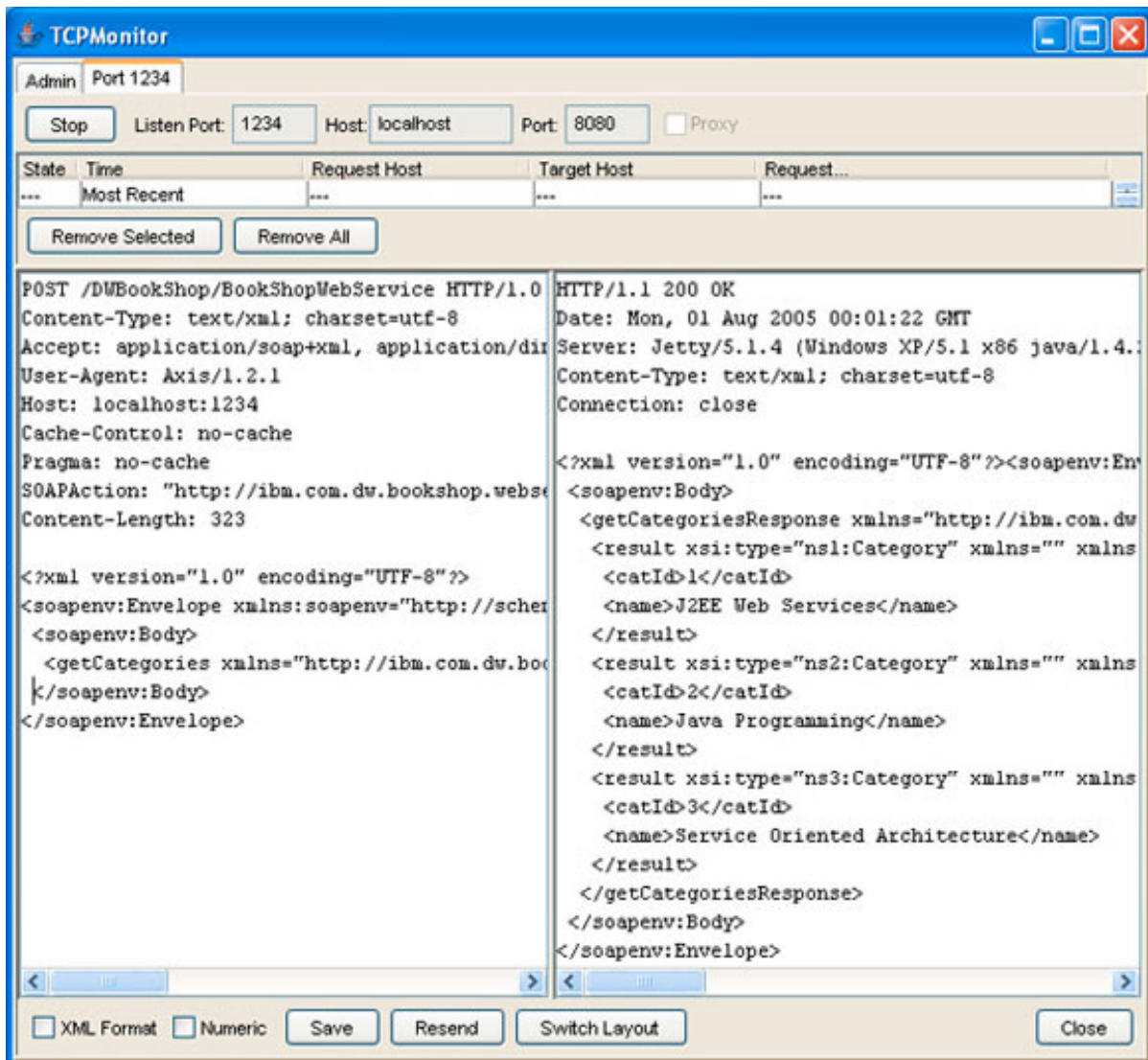
Figure 6. J2SE client application



The JAX-RPC Web Service Endpoint (JSE) tab simply presents all Web service names hosted on XMethods.net upon user request.

To get some insight into what is happening behind the scenes, you can use the TCPMonitor application included in the Axis distribution. This application acts as a proxy between the client application and the BookShop Web services. All SOAP requests and responses are displayed in two panels (see [Figure 7](#)).

Figure 7. Axis TCPMonitor application



In order to use this application for monitoring the traffic between the J2SE client and the BookShop Web service, you need to change the port number in the `**Locator.java` classes, which you generated earlier using the Apache Axis distribution (see [Listing 9](#)). The new Web service URLs should now look as they do in [Listing 17](#).

Listing 17. New Web service URL

```
//change this line in the class \
//com.ibm.dw.bookshop.j2seclient.servlet.generated.XMethodsNamesLocator
private java.lang.String XMethodsNamesSEIPort_address =
    "http://localhost:1234/DWBookShop/XMethodsNames";

//change this line in the class
//com.ibm.dw.bookshop.j2seclient.ejb.generated.BookShopLocator
private java.lang.String BookShopSEIPort_address =
    "http://localhost:1234/DWBookShop/BookShopWebService";
```

Starting the TCPMonitor

Now you can start the TCPMonitor application either by using the `tcpmon` Ant task, provided in the BookShop example build file, or by typing the following command:

```
% java org.apache.axis.utils.tcpmon 1234 localhost 8080
```

Section 7. Summary

This tutorial has given you an in-depth look into enterprise Web services demonstrated on the Apache Geronimo application server. You should now be familiar with JAX-RPC-related concepts and their integration into the J2EE 1.4 platform. You've learned about several options for accessing external Web services and the exposure of your own Web services from both the Web tier and the EJB tier of your J2EE applications. With the help of the BookShop example application, this tutorial has shed some light on the configuration jungle you must navigate when developing Web services-enabled applications for the J2EE 1.4 platform.

A brief look ahead

Part 2 of this tutorial series will concentrate on registry access, SOAP attachments, message handlers, and Web services security. It will discuss and demonstrate the following aspects of enterprise Java Web services in detail:

- **The Java API for XML Registries (JAXR):** JAXR defines a standard Java API for accessing and programmatically interacting with various types of metadata registries, such as Universal Description, Discovery, and Integration (UDDI) and Electronic Business XML (ebXML).
- **The SOAP with Attachments API for Java (SAAJ):** SAAJ provides convenient ways to access the various parts of SOAP messages. This includes message headers and the option to include binary message attachments.
- **Message handlers:** Learn several ways to implement message IDs for conversational Web services, the implementation of appropriate message-encryption mechanisms following the WS-Security standard, and suitable logging strategies for Web service-related activities.

- **SOAP faults:** Gain insight into exception handling for Web services, including standard SOAP faults, Web services-related exception types, and checked user exceptions.
- **The future of J2EE Web services:** Get a preview of Web service-related plans for the upcoming J2EE 5 platform. You'll see that many of the pitfalls developers have encountered in the J2EE 1.4 platform have been addressed.

Acknowledgement

The author would like to express his sincere thanks to the Geronimo community, which kindly helped during the development of this tutorial through mailing lists and IRC chats. Special thanks to David Jencks, who spent considerable time explaining Geronimo-specific Web services handling and fixing several bugs encountered during the development of the BookShop example code.

Downloads

Description	Name	Size	Download method
Source code for the BookShop application	DWBookShop.zip	493 KB	HTTP

[Information about download methods](#)

Resources

Learn

- Check out the [Apache Geronimo project](#), which strives to develop a certified world-class J2EE server.
- Read Sing Li's articles "[Geronimo! Part 1: The J2EE 1.4 engine that could](#)" and "[Geronimo! Part 2: Tame this J2EE 1.4 bronco](#)" (developerWorks, May 2005) for a good introduction to the Geronimo J2EE server.
- Read "[Developer's introduction to JAX-RPC, Part 1: Learn the ins and outs of the JAX-RPC type-mapping system](#)" (developerWorks, November 2002) in which Joshy Joseph takes you to the heart of JAX-RPC.
- Learn how to achieve the next level of Web service interoperability using the JAX-RPC standard's client- and server-side interface definitions and message-processing model by reading "[A developer's introduction to JAX-RPC, Part 2: Mine the JAX-RPC specification to improve Web service interoperability](#)" (developerWorks, January 2003).
- See "[Understanding your JAX-RPC SI Environment, Part 1](#)" for a white paper that explains how to unleash the power of the wscompile and wsdeploy tools to develop, deploy, and invoke a Web service using JAX-RPC.
- Visit the [IBM developerWorks SOA and Web services zone](#) for more articles on Web services.
- See [J2EE Web Services](#) by Richard Monson-Haefel for more in-depth information on J2EE Web services.
- Read [Designing Web Services with the J2EE Platform](#) by Inderjeet Singh et al., an excellent introduction to J2EE Web services. The entire book is freely available online in PDF and HTML formats.
- Visit the developerWorks [Apache Geronimo project area](#) to access resources for Geronimo developers.
- Visit the [developerWorks Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.
- Find a collection of resources you'll need for development using Apache Derby at the [developerWorks Apache Derby resource center](#).

Get products and technologies

- Check out the [Eclipse Platform](#), an open, extensible IDE. There are heaps of useful plug-ins and extensions available, making this platform extremely versatile.

- Download the [J2SE 1.4.2_08 SDK](#) from Sun.
- Download [Apache Geronimo](#) from the project site.
- Get the [Apache Ant Project](#), which provides a standard Java-based build tool.
- Download a Java-based platform for creating and deploying Web services applications from the [Apache Axis Project](#) site.
- Get MySQL, a popular open source relational database, and the MySQL Connector/J driver from the [MySQL](#) project site.
- Download Sun's [Java Web Services Developer Pack](#) (Java WSDP), a free integrated toolkit you can use to build, test, and deploy XML applications, Web services, and Web applications with the latest Web service technologies and standards implementations.
- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

Discuss

- [Participate in the discussion forum for this content.](#)

About the author

Stefan Schmidt

Stefan Schmidt is a Ph.D. student specializing in enterprise development and conception, distributed systems, and Web services technologies. He works as a tutor on subjects such as advanced Internet technologies and distributed computing architecture.