Beginning Java Web Services

by Henry Bequet

Chapter 11: Case Study – Application Integration

Overview

So far in this book we have covered the various technologies and tools that are used for implementing web services. In this chapter, we will be discussing a case study to illustrate the use of web services for integrating a set of disparate applications within an enterprise. The case study will focus on leveraging the services provided by an existing set of applications, with the web services technology, to perform Enterprise Application Integration (EAI).

The case study will be deployed on BEA WebLogic 7.0 server and WebLogic Workshop. An evaluation copy of the WebLogic server and Workshop can be downloaded from http://www.bea.com/.

In this case study we will use WebLogic Workshop to package three existing applications as web services. These applications will then be accessed through their web services interface to build a brand new web-based shopping system, which will leverage the services provided by these applications. The three applications that are integrated to build the online shopping system are:

- A database application that stores catalog information
- A J2EE application that processes credit card payment
- A proprietary messaging system that is used for order processing

In the course of this chapter we will learn how to expose these applications as web services with a minimum amount of effort, and how these applications can be easily accessed from J2EE web applications through their web services interface.

Internal and External Web Services

Before we delve into the various intricacies of our case study, we will have a quick look at the two scenarios in which web services are more popularly used. When the web services hype started gaining momentum, everyone was excited about companies implementing, describing, publishing, discovering, and consuming web services on the Internet and forming dynamic businesses. These types of web services, where the producers and consumers are different companies, are called **external web services.**

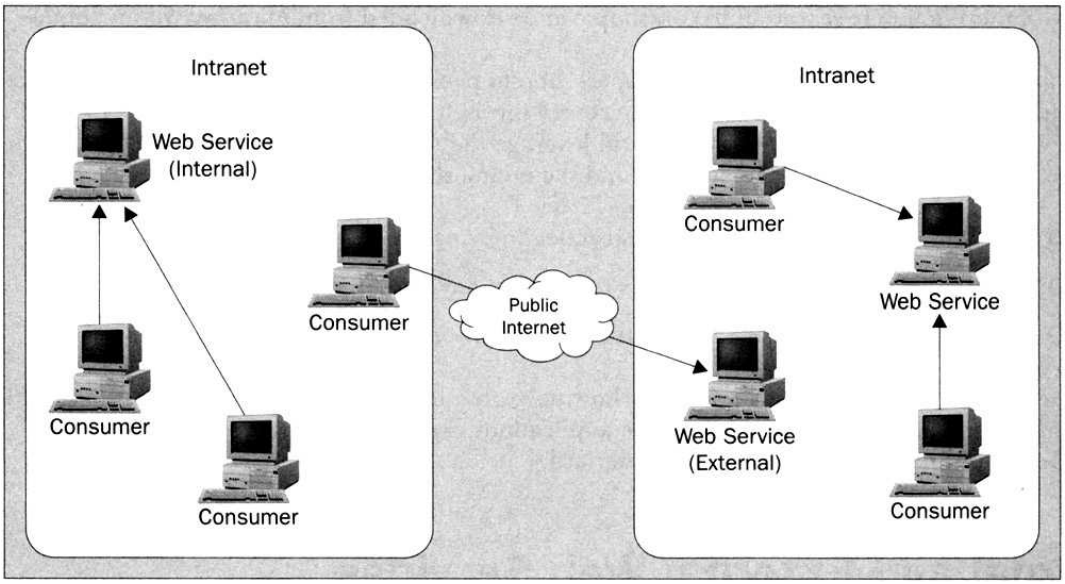
However, in most practical scenarios, forming business partnerships involves many formal procedures and processes. This realization leads to new business collaboration standards and specifications such as ebXML and BizTalk, which address issues not dealt by the basic web services technologies like XML, SOAP, WSDL, UDDI, etc.

External web services mainly provide services that are accessed by trading partners. For example, http://www.xmethods.com provides a public registry of external web services that can be accessed over the public network. Examples for external web services include services that provide stock quotes, weather forecasts, and so on. External web services may be free or commercial. Commercial web services can adopt different payment models, including payment per invocation, as well as a subscription-based model.

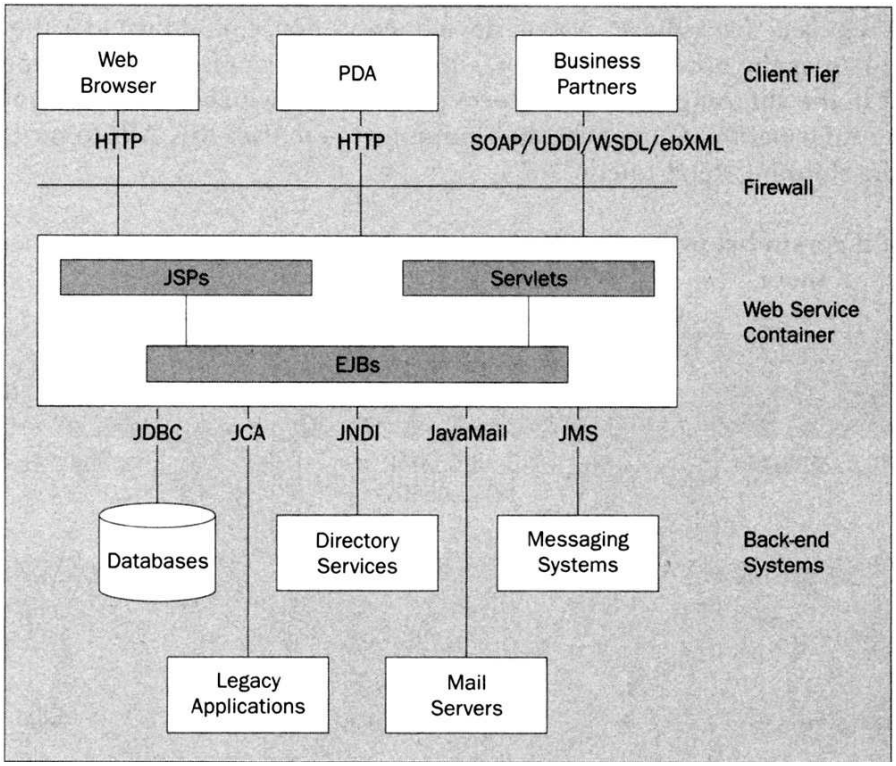
For example, if we are developing an online system for trading security instruments, we may enter a trading agreement with Reuters or FT to access their web services for providing the latest quote for the specified security.

However, at the time of writing, lots of products are coming into the market, which use web services technologies to provide end-to-end EAI solutions. In such scenarios, the consumers and producers of web services are the applications running within an enterprise. Such web services are called **internal web services**. For example, we may have an application that manages the production unit of our company interfacing with the inventory system, using web services.

The diagram below depicts the use of external and internal web services:



Since web services use platform-neutral, language-neutral, and vendor-neutral technologies such as XML, HTTP, SOAP, and WSDL, they are very well suited for integrating disparate, heterogeneous, and disconnected applications within an enterprise. On top of this, most of the mainstream J2EE server vendors offer support for web services and provide easy-to-use tools that can be used for exposing standard J2EE components like EJBs and JMS destinations via web services. The diagram below depicts how web services can fit into an existing J2EE infrastructure:



The seamless integration of web services with existing J2EE technology components, contributes highly towards leveraging the power of existing solutions and components to build a web services-based EAI solution.

Over the past years one of the most critical problems with EAI solutions has been the dependence on proprietary vendor API and plug-ins. Most of the time application developers were writing 'glue' code to get the EAI solutions from various vendors to work with their applications. Web services and associated technologies provide a vendor-neutral and open standard solution, which can leverage the existing infrastructure of enterprises to provide an elegant EAI solution.

Let's now move on to discuss our case study, which, as we mentioned earlier, will make use of web services to integrate a set of disparate internal applications and expose them as a web application.

Case Study Overview

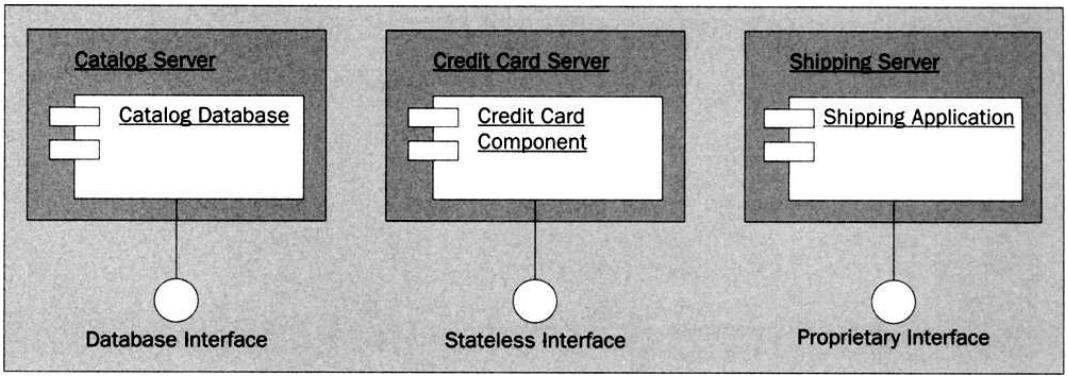
Acme Corporation, our fictitious case study corporation, currently provides catalog-based shopping services to its customers. It delivers printed copies of catalogs containing the products on offer to its customers, and the customers place orders over the telephone. The orders are processed by desk clerks who validate the payment details and use an in-house system to control the shipping of the order.

The company has an in-house J2EE-based system that connects to various credit card providers to process credit card payment. The system provides a simple stateless interface implemented using a J2EE session EJB component.

The products available in the catalog are stored in a MySQL database. The company has a system written in C that connects to the database and prints the catalog information.

The shipping system is a Visual Basic system developed in-house, that provides the desk clerks an easy-to-use interface to enter the order information, which is later processed by the shipping department. The system used in the shipping department provides an API, which can be used by external systems for pushing order information. Currently, the shipping system uses this API to push the order information to the shipping department.

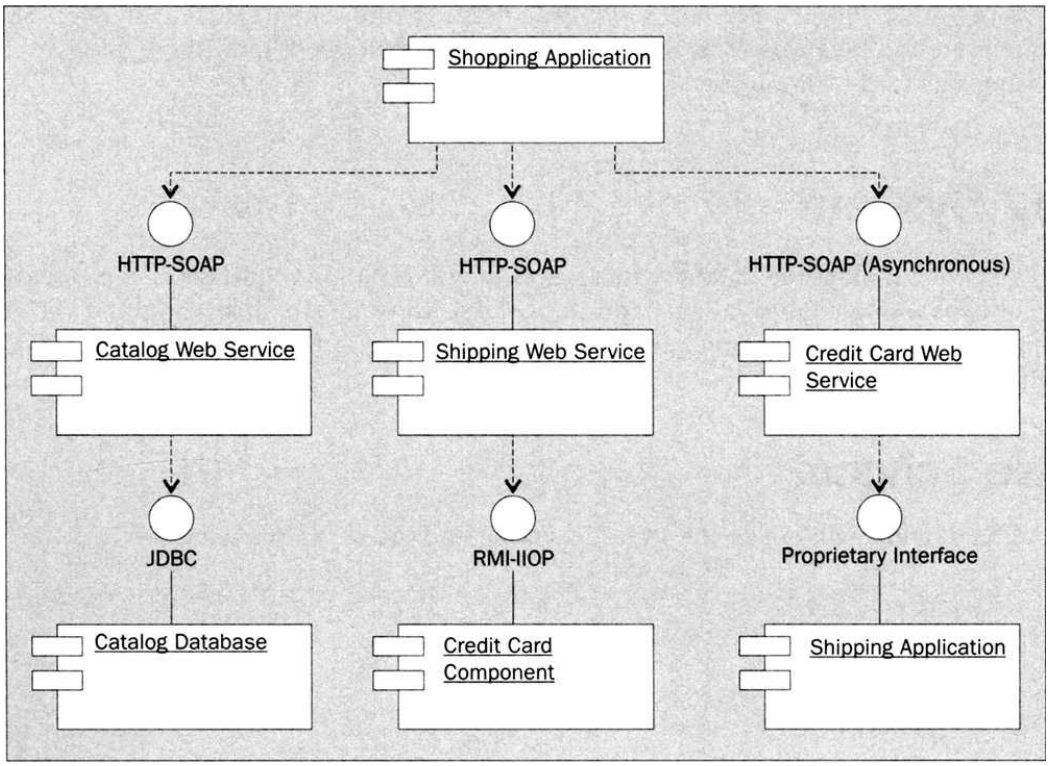
The deployment diagram below depicts the existing systems within the company and the interfaces provided by each of them:



The company has now made a decision to make some of its services available to the public over the Internet. It has also decided to integrate the existing disparate set of applications to provide a new end-to-end online shopping solution to its customers. It has also been agrred upon to use the Web Services technology for integrating the existing applications in a loosely coupled manner.

The component diagram opposite depicts how the existing applications may be integrated using the web services technology: The current interface to the shipping system is currently very slow and will not be acceptable in a web-based solution. Hence, as part of the new system, it has also been decided to improve the transaction throughput of the system by providing an asynchronous interface.

This component diagram depicts the proposed architecture of the new system:



In short, we can summarize the situation as:

- The online shopping application will access the catalog application through a web service using SOAP over HTTP. At the moment, the catalog information is stored in a database, but the SOAP-based web service that can be accessed using standard web services invocation technologies such as JAX-RPC and JAXM, will access the database using JDBC to provide the necessary catalog information.
- The credit card application provides a stateless session EJB interface that will be exposed as a standard SOAP-based web service that can be accessed using JAX-RPC or JAXM. The web service will use the standard RMI-IIOP protocol for accessing the EJB component.
- The shipping application currently provides a proprietary interface. This will also be accessed from a SOAP-based web service. The web service will provide message buffering and an asynchronous invocation facility to improve performance and transaction throughout.

Acme Corporation has decided to use the BEA Workshop and WebLogic Server for developing and deploying its web services and shopping application, as its existing applications currently run on WebLogic. BEA WebLogic Server is a proven platform for developing scalable and robust J2EE applications and also provides a robust environment for deploying web services.

BEA Workshop is an IDE for assembling, developing, debugging, deploying, and testing enterprise class web services. Workshop also allows you to expose standard J2EE components like EJBs, JMS destinations, and JDBC data sources as web services accessible through standard protocols like JAX-RPC and JAXM with the minimum amount of effort. For more details on BEA WebLogic, please refer to Chapter 10

WebLogic provides a variety of tools for exposing existing components such as stateless session EJBs, JMS destinations, and Java classes as web services that are described, published, discovered, and invoked using standard technologies.

Catalog System

First, we will take a look at the database that stores the information required for the catalog system. The database contains a single table called `Product`, which stores information about the various products available in the catalog. The section below shows the structure of the table used for storing the catalog information.

Database Schema

The script given below will create the `Product` table and insert sample data into it:

```
CREATE TABLE Product (
  code varchar (30),
  name varchar (30),
  description varchar (60),
  price double);
```

The table stores a unique code identifying the product, the name of the product, a description for the product, and the price of the product:

```
INSERT INTO Product VALUES ('PRO1', 'T-Shirt', 'Hooded T-Shirt', 123.45);
```

```
INSERT INTO Product VALUES ('PRO2', 'Jacket', 'Leather Jacket', 234.56);
INSERT INTO Product VALUES ('PRO3', 'Trousers', 'Denim Straights', 345.67);
INSERT INTO Product VALUES ('PRO4', 'Trainers', 'Cross Mountain Trainers', 456.78);
INSERT INTO Product VALUES ('PRO5', 'Tights', 'Denim Tights', 567.89);
```

Note We have used MySQL for storing the table and data shown above. However, you can make use of any relational database with a valid JDBC-compliant driver. By running the scripts given above, you can create the table and populate it with the sample data.

Develop the Catalog Web Service

In this section we will develop the catalog web service, which will connect to the database and provide the catalog information to the web service clients. In this section we will be performing the following tasks:

- Configure a datasource within WebLogic server to connect to the database
- Create a new web services project in WebLogic Workshop
- Create a new web service that will provide the catalog information
- Deploy and test the web service
- Generate the proxies that can be used by clients for accessing this web service

Configure the Datasource

Datasources are used in J2EE for getting connections to the database. They implement the `javax.sql.DataSource` interface. Normally either the JDBC provider or the J2EE server vendor provides classes that implement this interface. In most cases J2EE servers provide tools for configuring datasources and making them available for JNDI lookup. The clients that require connecting to databases can look up these datasources using JNDI calls and use them for creating connections to databases. Please refer to *Professional Java Server Programming* from Wrox Press (ISBN 1-86100-537-7) for an in-depth coverage of JDBC and JNDI.

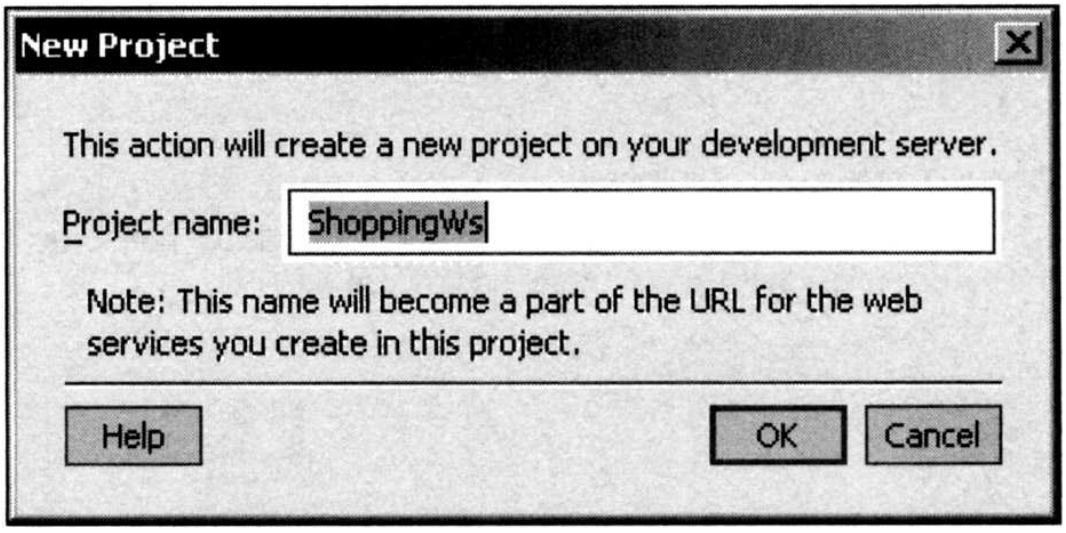
In Chapter 10, we have seen how to configure datasources within the WebLogic server, for this web service we need to configure a datasource called `productDataSource` to connect to the database containing our `Product` table (for more details on configuring datasources within WebLogic please refer to Chapter 10).

Create a New Web Services Project

In this section we will create a new web services project in Workshop that will contain all the three web services used in this case study. For this we need to perform the following steps:

1. Start the database server containing the `Product` table and data (in our case it will be MySQL)
2. Start the WebLogic server in the workshop domain by running the `%WLS_HOME%\WebLogic700\samples\workshop\startWebLogic.cmd` where `%WLS_HOME%` is the directory in which we have installed WebLogic server and Workshop
3. Start the Workshop IDE either using the shortcuts available on the start menu

Once the Workshop IDE is up and running, we will create our project. For this click on the `File` menu and select the `New Project` option. This will open a dialog box asking us to enter the name of the project. In this dialog box enter `ShoppingWS` and click `OK`:



This will create the following file structure:

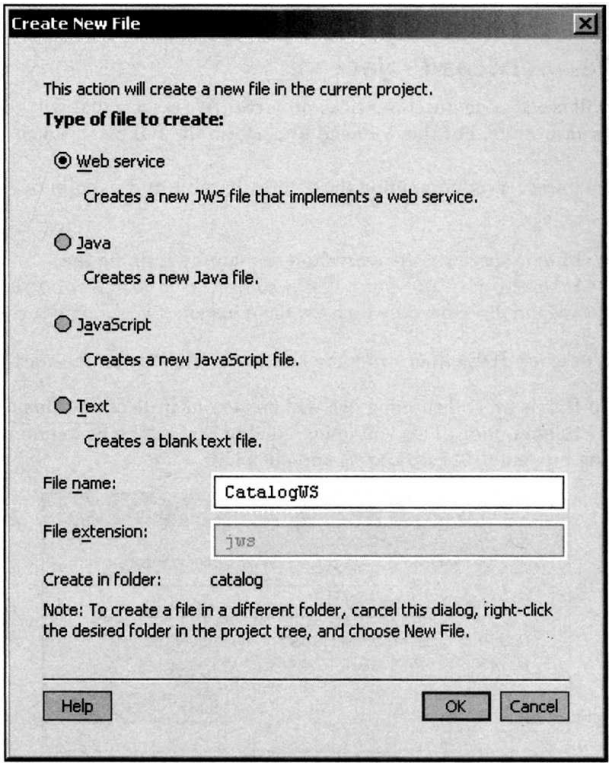
```
%WLS_HOME%\Weblogic7000\samples\workshop\applications\ShoppingWS
%WLS_HOME%\Weblogic7000\samples\workshop\applications\ShoppingWS\WEB-INF\lib
%WLS_HOME%\Weblogic7000\samples\workshop\applications\ShoppingWS\WEB-INF\classes
```

Now a dialog box will pop up, click the `Cancel` button of this dialog box. Then, in the tree on the lefthand side of the IDE right-click on the root item and select `New Folder` from the pop-up menu. Create a folder called `catalog`. In this folder we will be storing the required files for the catalog web service.

Repeat the process to create folders `shipping` and `creditCard` for the files implementing the shipping and credit card web services respectively.

Create the Catalog Web Service

In this section we will create the catalog web service. For this, right-click on the `catalog` folder in the tree on the left-hand side of the IDE, and then click on the `New File` item in the popup menu. This will display a dialog for entering details about the file we are going to create. The `Web Service` radio button will be selected by default. Enter the name of the file CatalogWS and click `OK`:



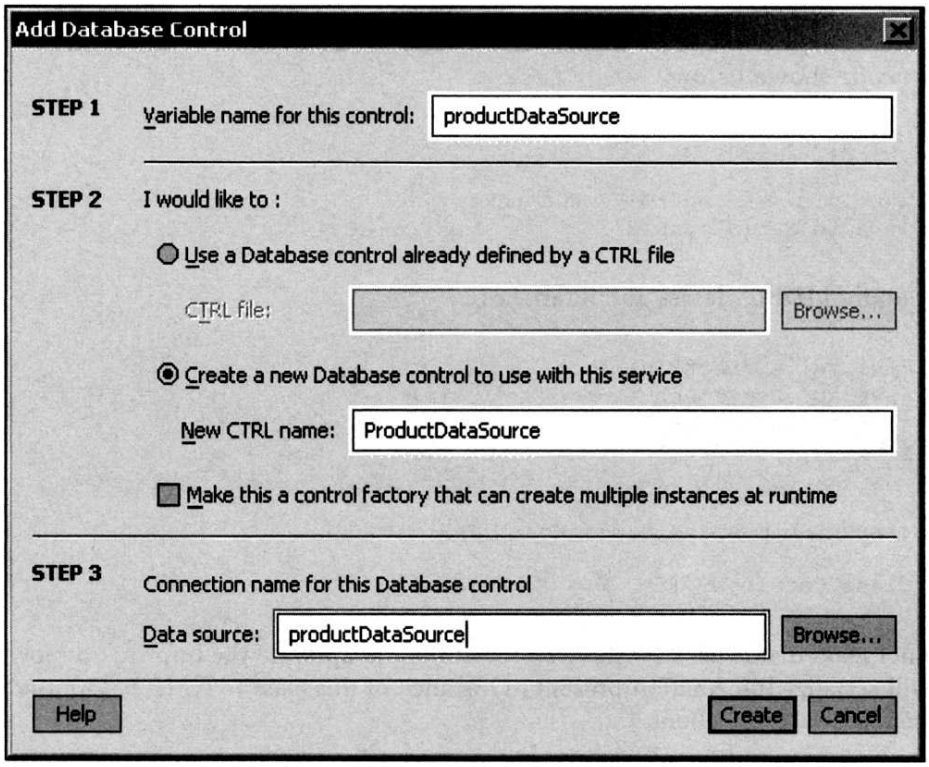
Now the IDE will display the design view of the web service in the middle pane. In the property pane on the right-hand side enter the target namespace of the web service as `catalog`.

Add the getCatalog() Method and the Database Control

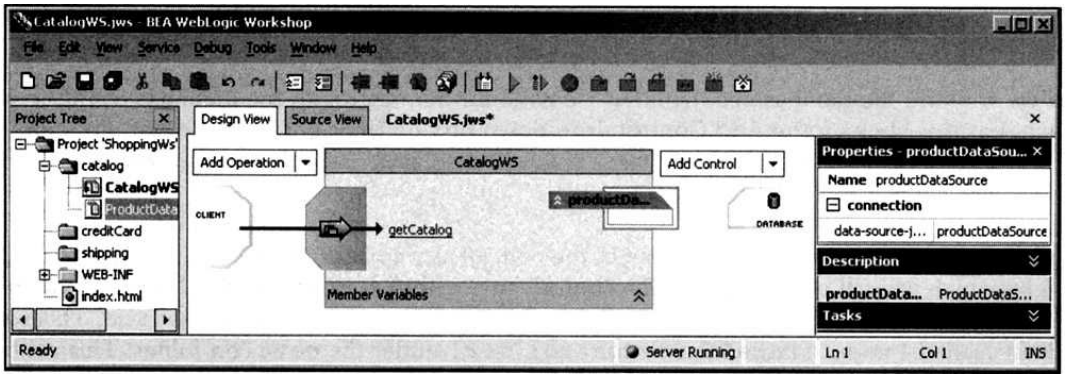
Now we will add the `getCatalog()` method, which will be exposed by the web service to its clients. For this click on the `Add Operation` drop-down box in the middle pane and select the `Add Method` item. Enter the name of the method as `getCatalog`.

Next we will add the database control that we need for connecting to the database from the web service. For this click on the `Add Control` drop-down box on the right-hand side of the middle pane and select the `Add Database Control` item. This will display a dialog box for entering the details of the database control.

In the dialog box enter the variable name of the control as `productDataSource`. This will be the name by which we will be referring to the control from within our web service. Select the radio button for creating a new control and enter the name of the control as `ProductDataSource`. This will create a new file called `ProductDataSourceControl.ctrl` under the `catalog` folder. This control can be reused in other web services as well. Select the `productDataSource` that was configured within WebLogic by clicking the `Browse` button. Now click `OK`:



The figure below shows the design view of our web service with the operation on the left side and the control on the right side:



Add Code to the Web Service

In this section, we will be adding code to our web service for connecting to the database and returning the catalog information to the clients. For this, click on the `Source View` tab of our catalog web service and enter the code shown below:

```
package catalog;

import weblogic.jws.control.JwsContext;
import java.io.Serializable;
```

Import the required JDBC classes and interfaces:

```
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;

/**
 * @jws:target-namespace namespace="catalog"
 */
public class CatalogWS {
```

This is an inner class that is used by the web service to encapsulate the information for each product. WebLogic will serialize information present in instance of this class to XML before the web service response is send back to the client:

```
      public static class Product implements Serializable {
```

This represents the unique code for each product:

```
      private String code;
      public String getCode() { return code; }
      public void setCode(String val) { code = val; }
```

This represents the name of the product:

```
      private String name;
      public String getName() { return name; }
      public void setName(String val) { name = val; }
```

This represents the description for the product:

```
      private String description;
      public String getDescription() { return description; }
      public void setDescription(String val) { description = val; }
```

This represents the price of the product:

```
      private double price;
      public double getPrice() { return price; }
      public void setPrice(double val) { price = val; }
    }
```

This is the datasource control we will use for connecting to the database. This line of code was automatically generated by the web service when we added the database control in design view:

```
      /**
      * @jws:control
      */
      private ProductDataSourceControl productDataSource;

      /** @jws:context */
      JwsContext context;
```

This is the operation exposed by the web service. WebLogic Workshop generated the shell of this method when we added the method in the design view:

```
      /**
      * @jws:operation
      */
      public Product[] getCatalog() throws SQLException {
```

Get a connection from the datasource and use the connection to create a statement:

```
      Connection con = productDataSource.getConnection();
      Statement stmt = con.createStatement();
```

Execute the SQL query to get the number of products in the database:

```
      ResultSet res = stmt.executeQuery("SELECT COUNT(*) FROM PRODUCT");
      res.next();
      Product products[] = new Product[res.getInt(1)];
```

Execute the SQL query to get all the product details from the database, loop through the result set, and create the array of products:

```
      res = stmt.executeQuery("SELECT * FROM PRODUCT");
      for(int i = 0; i < products.length && res.next(); i++) {
        Product product = new Product();
        product.setCode(res.getString(1));
        product.setName(res.getString(2));
        product.setDescription(res.getString(3));
        product.setPrice(res.getDouble(4));

        products[i] = product;
      }
```

Close the database resources:

```
      res.close();
      stmt.close();
      con.close();
```

Return the array of products:

```
        return products;
```

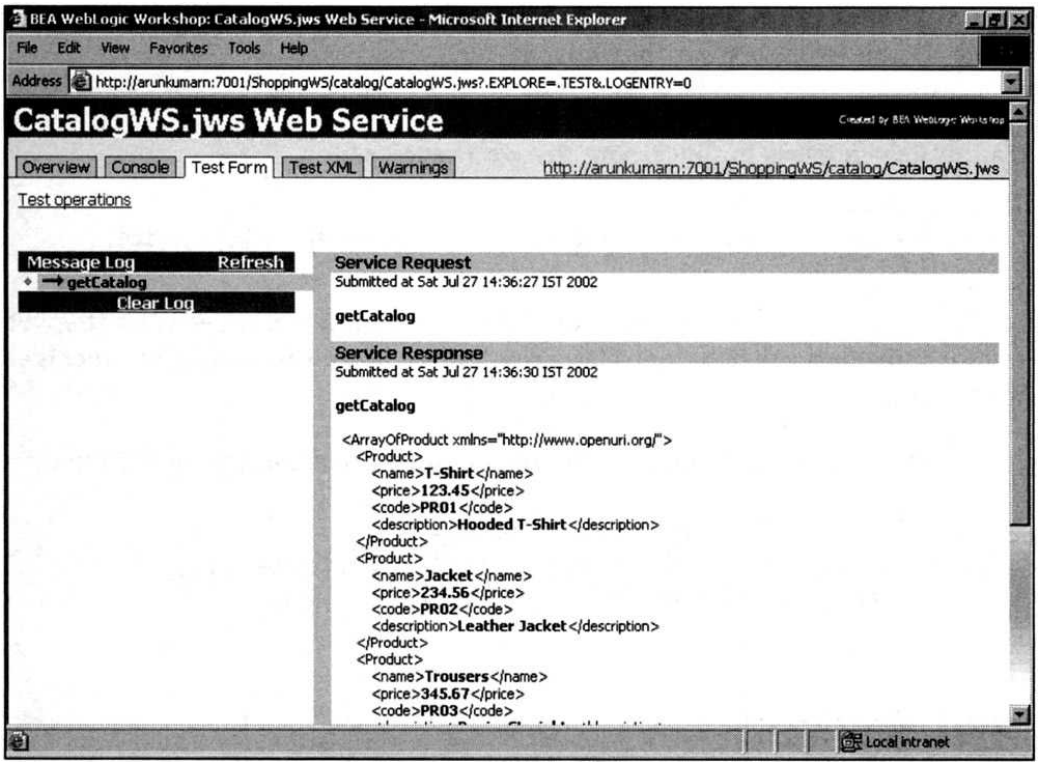
```
        }
    }
```

Test the Web Service

In this section we will test whether our web service is working properly. Click on the `Debug` menu and select the `Start` option. This will open the web service home page in a browser window. This page has four tabs:

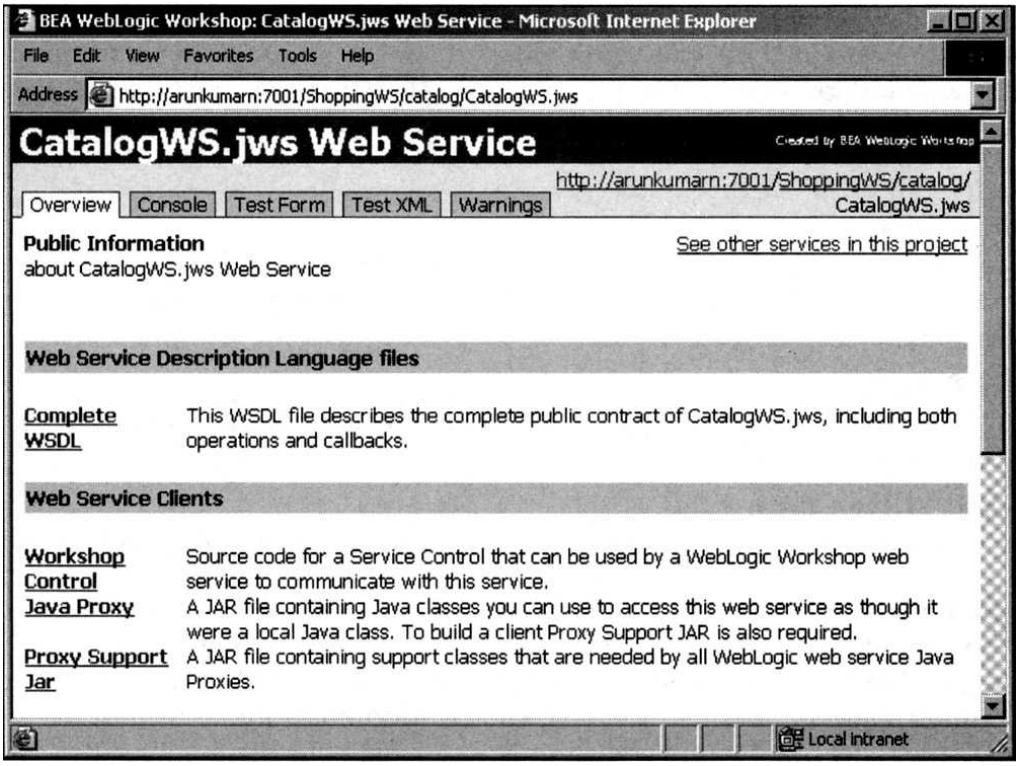
- `Overview`
  This tab provides links for viewing the WSDL for the web service, downloading client proxies, and so on
- `Console`
  For controlling the various logging and persistence aspects of the web service
- `Test Form`
  This will provide functionality for testing the web service using HTTP GET and POST
- `Test XML`
  This will provide functionality to test the web service by sending raw XML

We will test our web service using the `Test Form` tab. For this, click on the getCatalog link located on the tab. This will execute the web service and display the result in the browser as shown:



Generate Client Proxy

Now we will generate the client proxy JAR file for accessing the web service from clients. We will use this JAR file later in the shopping application for accessing the web service. For this click on the `Overview` tab and then on the `Java Proxy` link:



Save the generated JAR file under the name `CatalogClient.jar`. This JAR file contains three classes inside and is used for accessing the web service:

- `WebLogic.proxies.jws.CatalogWS`

  This is the service class for accessing the web service.

- `WebLogic.proxies.jws.CatalogWSSoap`

  This is the interface used for invoking the operation on the web service.

- `catalog.Product`

  This is the client-side representation of the product information sent by the web service. The client-side stubs and other helper classes will deserialize the XML data received from the web service to generate instances of this class.

The snippet below shows how the catalog web service can be accessed from the client:

```
    CatalogWS_Impl catalogWS = new CatalogWS_Impl();
    CatalogWSSoap catalogWSSoap = catalogWS.getCatalogWSSoap() ;
    Product[] products = catalogWSSoap.getCatalog();
```

Credit Card System

In this section we will be implementing the credit card application as a web service. First we will have a look at the backend component that implements the web service and then we will look at exposing this component as a SOAP-based web service.

Credit Card Session Bean

As we have already mentioned the existing credit card application is implemented as a J2EE application and provides a simple stateless interface using-session EJB. EJBs are part of the J2EE technology suite and are used for building transactional business components. The EJB specification provides three types of EJBs:

- **Session Bean**

  Session beans act as extensions to the clients that invoke the beans. Session beans can either be stateless or stateful. Stateful session beans remember their instance state across multiple invocations.

- **Entity Bean**

  Entity beans are components that can be shared by multiple clients and are used for modeling persistent domain objects.

- **Message-Driven Bean**

Message-driven beans are components activated by asynchronous JMS messages.

Both session and entity beans provide a client view that can be accessed by clients. This is achieved by using the home and component interface. The home interface provides methods that govern the life cycle of a bean like creating, finding, removing the bean, and so on. The component interface defines the business method provided by the bean. In addition to these two interfaces, an EJB needs a bean implementation class that implements the business logic for the methods defined in the component interface.

For our credit card session bean we need to write the component interface, home interface, and bean implementation class. Please note that a comprehensive coverage of EJB is beyond the scope of this chapter and the book. For a detailed coverage of EJBs refer to *Professional EJB* from *Wrox Press (ISBN 1-86100-508-3)* for more details.

Component Interface

The listing below shows the component interface for the session bean. The interface defines a single method specifying the credit card number to debit, and the amount that needs to be debited. It returns a Boolean value indicating whether the payment was processed successfully:

```
package com.acme.creditcard;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface CreditCardService extends EJBObject {
```

The component interface defines the business method used by the clients for authorizing credit card payments:

```
public boolean debit(String cardNo, double amount) throws RemoteException;
}
```

Home Interface

The code for the home interface is shown below:

```
package com.acme.creditcard;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

public interface CreditCardServiceHome extends EJBHome {
```

Define the lifecycle method to create an instance of the bean:

```
  public CreditCardService create() throws CreateException, RemoteException;
}
```

Since the catalog service is going to be implemented as a stateless component, the home interface defines a no-argument create() method.

Bean Implementation Class

The source file for the bean implementation class is shown below:

```
package com.acme.creditcard;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

import java.util.Random;

public class CreditCardServiceEJB implements SessionBean {
```

These are the lifecycle methods that need to be implemented from the SessionBean interface:

```
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void ejbRemove() {}
  public void setSessionContext(SessionContext ctx) {}

  public void ejbCreate() {}
```

The business method implementation:

```
  public boolean debit(String cardNo, double amount) {
    return new Random() .nextBoolean();
  }
}
```

Standard Deployment Descriptor

The code below lists the standard `ejb-jar.xml` deployment descriptor:

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC
  "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN"
  "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>

<enterprise-beans>
  <session>
```

The name of the EJB:

```
        <ejb-name>CreditCardService</ejb-name>
```

The home interface:

```
        <home>com.acme.creditcard.CreditCardServiceHome</home>
```

The component interface:

```
        <remote>com.acme.creditcard.CreditCardService</remote>
```

The bean class:

```
        <ejb-class>com.acme.creditcard.CreditCardServiceEJB</ejb-class>
```

Define the bean as stateless:

```
        <session-type>Stateless</session-type>
      </session>
  </enterprise-beans>

</ejb-jar>
```

WebLogic Deployment Descriptor

The code below lists the WebLogic-specific deployment descriptor `WebLogic-ejb-jar.xml` for the session EJB:

```
<?xml version="1.0"?>

<!DOCTYPE WebLogic-ejb-jar PUBLIC
  "-//BEA Systems, Inc.//DTD WebLogic 7.0.0 EJB//EN"
  "http://www.bea.com/servers/wls700/dtd/WebLogic700-ejb-jar.dtd">

<WebLogic-ejb-jar>

    <WebLogic-enterprise-bean>
        <ejb-name>CreditCardService</ejb-name>
```

Define the JNDI name for looking up the home interface:

```
        <jndi-name>CreditCardServiceHome</jndi-name>
    </WebLogic-enterprise-bean>

</WebLogic-ejb-jar>
```

Building and Deploying the EJB

In this section we will build and deploy our EJB on WebLogic server. For this, first we need to compile the component interface, home interface, and bean class. Please make sure that while compiling these classes, we have the `weblogic.jar` file in our classpath:

```
set classpath=%classpath%;%BEA_HOME%\server\lib\weblogic.jar
```

%BEA_HOME% is where we have installed WebLogic 7.0.

We will change to the `%jwsDirectory%\Chp11\CreditCardService\src` directory and compile the files:

```
javac -d ..\classes CreitCardSercice.java
javac -d ..\classes CreditCardServiceHome.java
javac -d ..\classes CreditCardServiceEJB.java
```

After compiling the classes, create the following directory structure inside the `\classes` directory:

```
/com/acme/creditcard/CreditCardService.class
```

```
/com/acme/creditcard/CreditCardServiceHome.class
/com/acme/creditcard/CreditCardServiceEJB.class
/META-INF/ejb-jar.xml
/META-INF/WebLogic-ejb-jar.xml
```

Then in the `%jwsDirectory%\Chp11\temp` run the `jar` command to create a JAR file with structure as shown above.

```
jar cf CreditCardService.jar *
```

Copy the JAR file to `%WLS_HOME%/Weblogic700/samples/workshop/applications` to deploy the EJB.

Develop the Credit Card Web Service

In this section we will develop the credit card web service, which will use the EJB to process credit card payments issued by the web service clients. We will perform the following tasks:

- Create a new web service that will process credit card information
- Deploy and test the web service
- Generate the proxies that can be used by clients for accessing this web service

Create the Credit Card Web Service

Now we will create the credit card web service. For this right-click on the `creditcard` folder in the tree on the left-hand side of the IDE and click on the `New File` item in the popup menu. This will display a dialog for entering details about the file we are going to create. The `Web Service` radio button will be selected by default. Enter the name of the file `CreditCardWS` and click `OK`. Now the `IDE` will display the design view of the web service in the middle pane. In the property pane on the right-hand side enter the target namespace of the web service as `creditCard`.

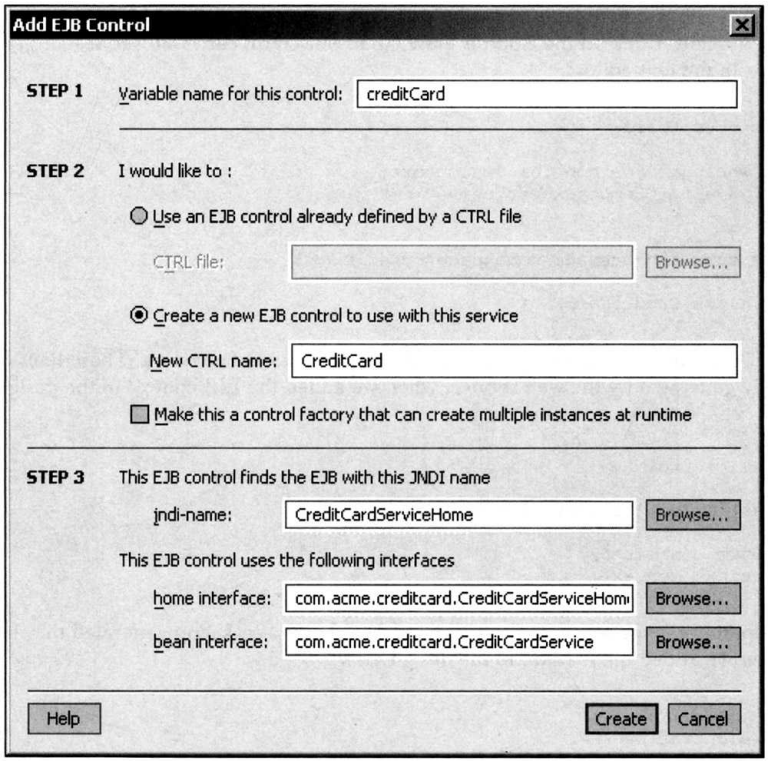
Add the debit() Method and the EJB Control

Now we will add the `debit()` method, which will be exposed by the web service to its clients. For this click on the `Add Operation` drop-down box in the middle pane and select the `Add Method` item. Enter the name of the method as `debit`.

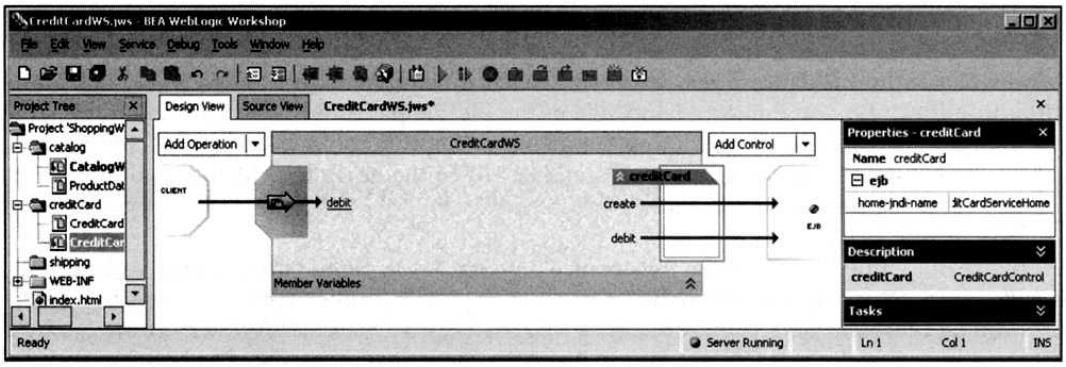
Before we can use the EJB control for accessing the credit card EJB we need to make the client view of the EJB available to our web services project. For this we need to copy the `CreditCardService.jar` file to `%WLS_HOME%/WebLogic700/samples/workshop/applications/ShoppingWS/WEB-INF/lib` directory. After this we will have to restart both WebLogic server and Workshop.

Now we will add the EJB control that we need to access the credit card EJB. For this, click on the `Add Control` drop-down box on the right-hand side of the middle pane and select the `Add EJB Control` item. This will display a dialog box for entering the details of the EJB control. In the dialog box enter the variable name of the control as `creditCard`. This will be the name by which we will be referring to the control from within our web service.

Select the radio button for creating a new control and enter the name of the control as `CreditCard`. This will create a new file called `CreditCardControl.ctrl` under the `creditcard` folder. This control can be reused in other web services as well. Select the `CreditCardServiceHome` by clicking the `Browse` button against the `JNDI` name. This will automatically populate the home and remote interfaces. Then click `OK`:



The figure below shows the design view of our web service with the operation on the left side and the control on the right side:



Add Code to Web Service

In this section we will add code to our web service, for using the credit card EJB for processing the credit card payment. Click on the `Source View` tab of our credit card web service and enter the code shown below in the text editor:

```
package creditcard;

import weblogic.jws.control.JwsContext;
import java.rmi.RemoteException;

/**
 * @jws:target-namespace namespace="creditCard"
 */
public class CreditCardWS {
```

This is the EJB control we will be using for processing credit card payment. These lines of code were automatically generated by the web service, when we added the EJB control in the design view:

```
  /**
  * @jws:control
  */
  private CreditCardControl creditCard;

  /** @jws:context */
  JwsContext context;
```

This is the operation exposed by the web service. WebLogic Workshop generated the shell of this method when we added the method in the design view:

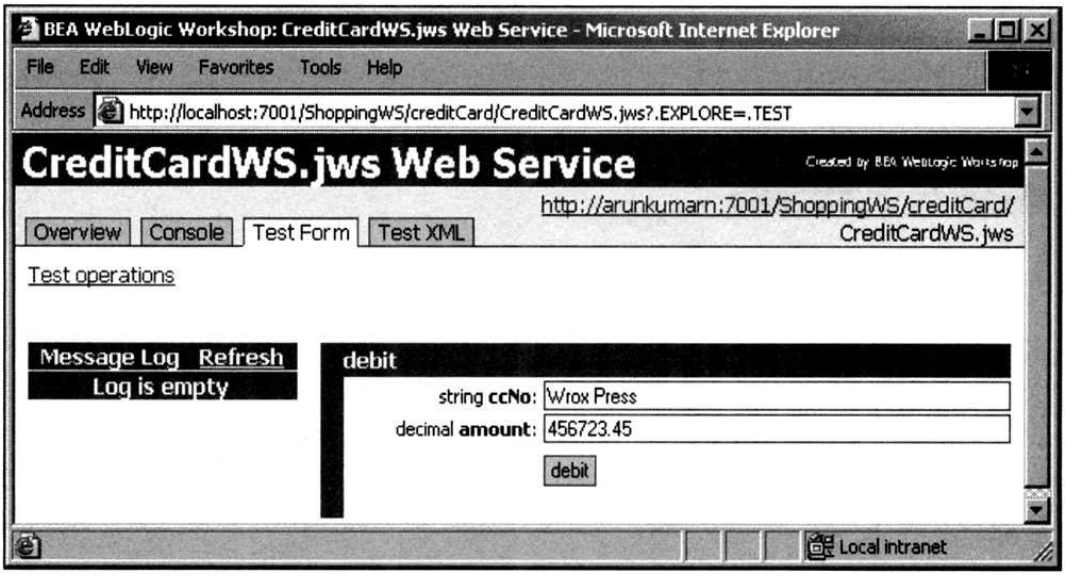
```
/**
* @jws:operation
*/
public boolean debit(String ccNo, double amount) throws RemoteException {
```

Use the EJB control to call the method on the credit card EJB to process the payment:

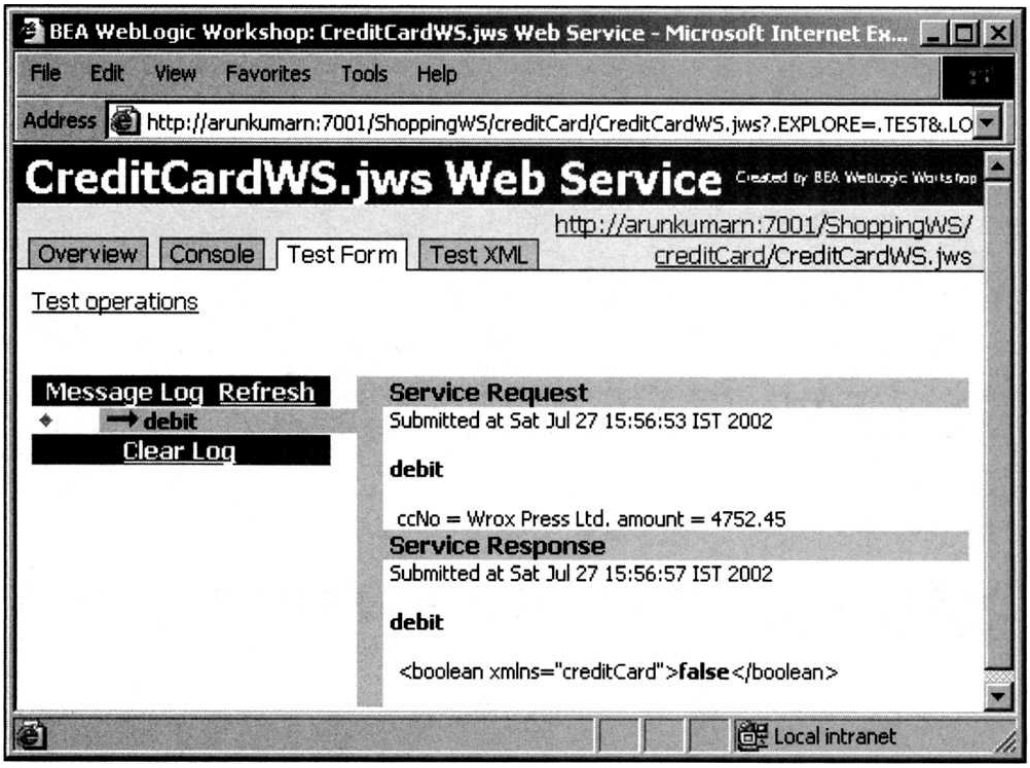
```
    return creditCard.debit(ccNo, amount);
  }
}
```

Test the Web Service

In this section we will check whether our web service is working properly. For this, click on the `Debug` menu and select the `Start option`. This will open the web service home page in a browser window. We will test our web service by using the `Test Form` tab that displays a form for entering the credit card number and amount as shown below:



Submit the form by clicking on the `debit` button. This will execute the web service and display the result in the browser as shown below:



Generate Client Proxy

Now we will generate the client proxy JAR file for accessing the web service from the clients. We will be using this JAR file later in the shopping application for accessing the web service. For this, click on the `Overview` tab and select the `Java Proxy` link. Save the generated JAR file under the name `CreditCardClient.jar`. There are two classes in this file that we will be using for accessing the web service:

- `WebLogic.proxies.jws.CreditCardWS`

  The service class for accessing the web service

- `WebLogic.proxies.jws.CreditCArdWSSoap`

  The interface used for invoking the operation on the web service

The code snippet below shows how the catalog web service can be accessed from the client:

```
CreditCardWS_Impl creditCardWS = new CreditCardWS_Impl();
CreditCardWSSoap creditCardWSSoap = creditCardWS.getCreditCardWSSoap();
creditCardWSSoap.debit("0000 0000 0000 0000", 456.76);
```

Shipping System

As we have already mentioned the shipping system provides a proprietary API for its client systems. We have also mentioned that the performance of the shipping system is time intensive and it has an unacceptable level of performance to be used in a web application scenario. In this section we will wrap the shipping system in a web service and demonstrate how it can be invoked asynchronously from clients by using a queuing mechanism.

Imortant Please note that the implementation of the system is not shown in this chapter. We simulate the delay in accessing the shipping system by letting the thread sleep for thirty seconds within the web service.

Develop the Shipping Web Service

In this section we will develop the credit card web service, which will use the EJB to process credit card payments issued by the web service clients. For this we will perform the following tasks:

- Create a new web service that will process shipping order
- Deploy and test the web service
- Generate the proxies that can be used by clients for accessing this web service

Create the Shipping Web Service

In this section we will create the shipping web service. For this, right-click on the `shipping` folder in the tree on the left-hand side of the IDE and click on the `New File` item in the pop-up menu. This will display a dialog for entering details about the file we are going to create. The `Web Service` radio button will be selected by default. Enter the name of the file `ShippingWS` and click `OK`. Now the IDE will display the design view of the web service in the middle pane. In the property pane on the right-hand side enter the target namespace of the web service as `shipping`.

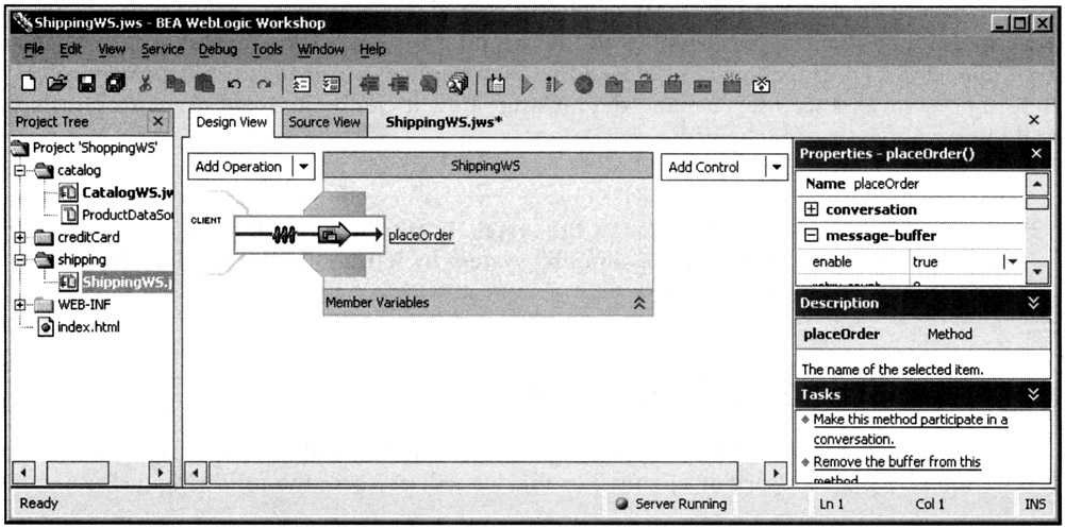
Adding the placeOrder() Method

Now we will add the `placeOrder()` method that will be exposed by the web service to its clients. For this, click on the `Add Operation` drop-down box in the middle pane and select the Add Method item. Enter the name of the method as `placeOrder`.

Now we will see how to add asynchronicity and message buffering facilities to the web service. In the Workshop, IDE methods can be declared as asynchronous by setting the message buffering property to `true` in the method's property sheet. These methods return immediately to the client before executing the body of the method.

The method arguments are stored in an internal buffer for the method to process them later. Asynchronous methods are required to have `void` return type. WebLogic uses internal JMS queues for storing the buffered messages. When we use message buffering, all the argument types should be serializable.

The figure below shows the design view of our web service with the operation on the left side and the control on the right side:



Please note that the graphic representing the method call for `placeOrder()` is different from that for `getCatalog()` and `debit()` in the previous web services. This indicates that the `placeOrder()` method implements message buffering and is invoked asynchronously.

Adding Code to our Web Service

In this section we will add code to our web service for processing the order. For this click on the **Source View** tab of our credit card web service and enter the code shown below in the editor:

```
package shipping;

import weblogic.jws.control.JwsContext;

import java.io.Serializable;

/**
 * @jws:target-namespace namespace="shipping"
 */
public class ShippingWS {
```

This inner class represents the address to which the order is to be sent:

```
  public static class Address implements Serializable {
```

The name of the person:

```
    private String name;
    public String getName() { return name; }
    public void setName(String val) {name = val; }
```

The street:

```
        private String street;
        public String getStreet() { return street; }
        public void setStreet (String val) { street = val; }
```

The city:

```
        private String city;
        public String getCity() { return city; }
        public void setCity (String val) { city = val; }
```

The county:

```
        private String county;
        public String getCounty() { return county; }
        public void setCounty (String val) { county = val; }
```

The post-code:

```
        private String postcode;
        public String getPostcode() { return postcode; }
        public void setPostcode (String val) { postcode = val; }
      }
```

This inner class represents each item in the order:

```
      public static class Item implements Serializable {
```

Unique item code:

```
        private String code;
        public String getCode() { return code; }
        public void setCode (String val) { code = val; }
```

Item quantity:

```
        private int quantity;
        public int getQuantity() { return quantity; }
        public void setQuantity (int val) { quantity = val; }
      }

      /** @jws:context */
      JwsContext context;
```

This is the web service method invoked by clients for placing orders, by passing the address and array of items. WebLogic will accept the incoming SOAP request and deserialize the XML representing the data to the relevant objects. Please also note that the method supports message buffering:

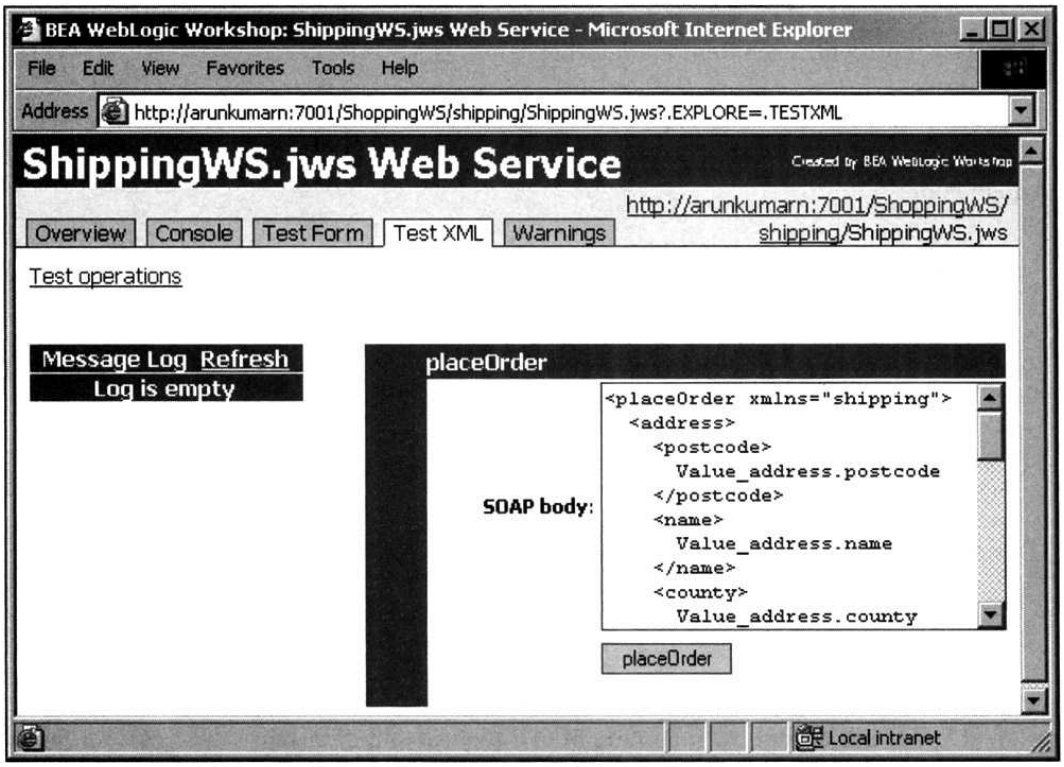
```
      /**
      * @jws:operation
      * @jws:message-buffer enable="true"
      */
      public void placeOrder (Address address, Item[] items) {
```

This is just to simulate the delay in processing the order. Even though the thread sleeps for thirty seconds, the client gets the control back immediately after invocation due to message buffering:

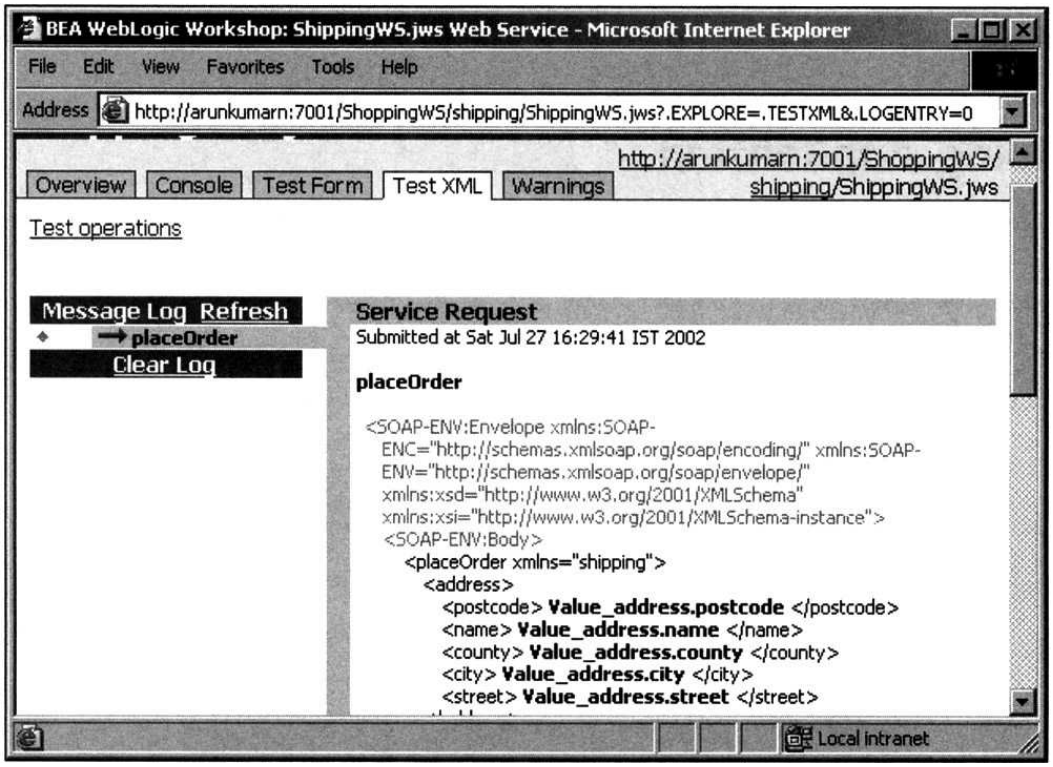
```
        try {
          Thread.sleep (30 * 1000);
        } catch (InterruptedException ex) { }

        System.out.println ("Order processed.");
      }
    }
```

Test the Web Service

In this section we will check whether our web service is working properly. For this, click on the `Debug` menu and select the `Start option`. This will open the web service home page in a browser window. We will test our web service using the `Test XML` tab as WebLogic is unable to display the form because the web service expects complex data types. In this page we can write the XML code that represents the data and submit the data to the web service. WebLogic provides a template XML document that represents the web service's input as shown below:



Submit the form by clicking on the `placeOrder` button. This will execute the web service and display the result in the browser as shown below:



Generate Client Proxy

Now we will generate the client proxy JAR file for accessing the web service from the clients. We will use this JAR file later in the shopping application for accessing the web service. For this, click, on the `Overview` tab and select the `Java Proxy` link. Save the generated JAR file under the name `ShippingClient.jar`. There are four classes in this file that we will be using for accessing the web service:

- `WebLogic.proxies.jws.CreditCardWS`

The service class for accessing the web service.

- `WebLogic.proxies.jws.CreditCArdWSSoap`

  The interface used for invoking the operation on the web service.

- `shipping.Address`

  This class is the client-side representation of the address information to the web service. The client-side stubs and other helper classes will serialize the class to XML before sending to the web service.

- `shipping.Item`

  This class is the client-side representation of the item information to the web service and works in the same way as the `Address` class.

The code snippet below shows how the catalog web service can be accessed from the client:

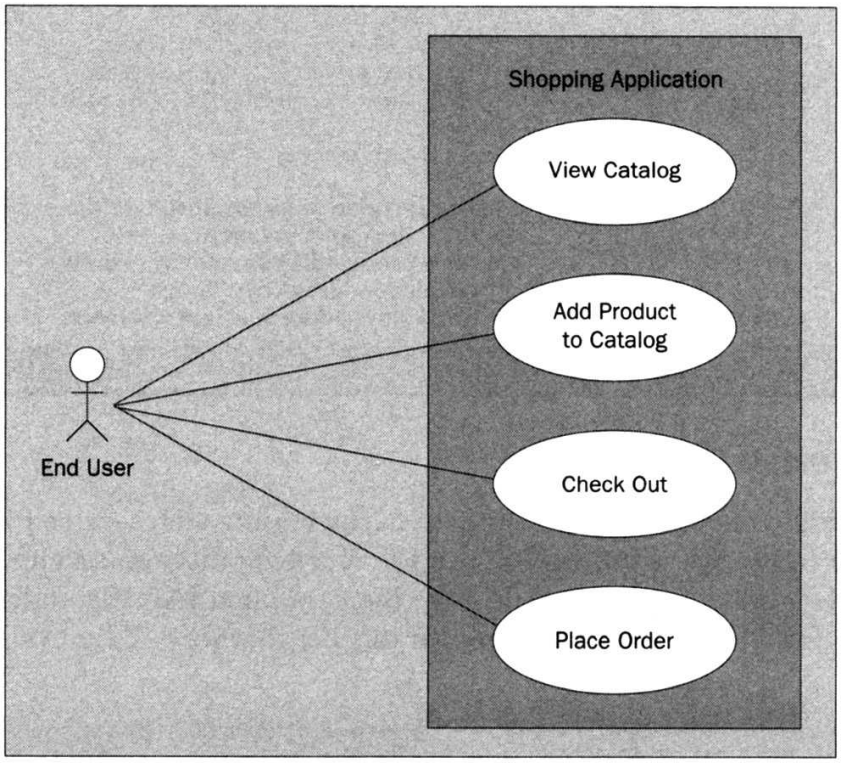
```
ShippingWS_Impl shippingWS = new ShippingWS_Impl();
ShippingWSSoap shippingWSSoap = shippingWS.getShippingWSSoap();
shippingWSSoap.placeOrder (address, items);
```

Shopping Application

In this section we will develop the shopping application, which will make use of the web services we have developed so far in the chapter to provide an online shopping system.

Use Cases

The diagram below depicts the high-level use cases associated with the shopping application:



The end users can visit the web site and browse through the catalog. From the catalog they may choose to add items to the shopping cart and then check out. The system will then display the current cart, and prompt for the payment and dispatch address details. On submitting the order form, the system will authenticate the payment details and instruct the shipping system to deliver the order.

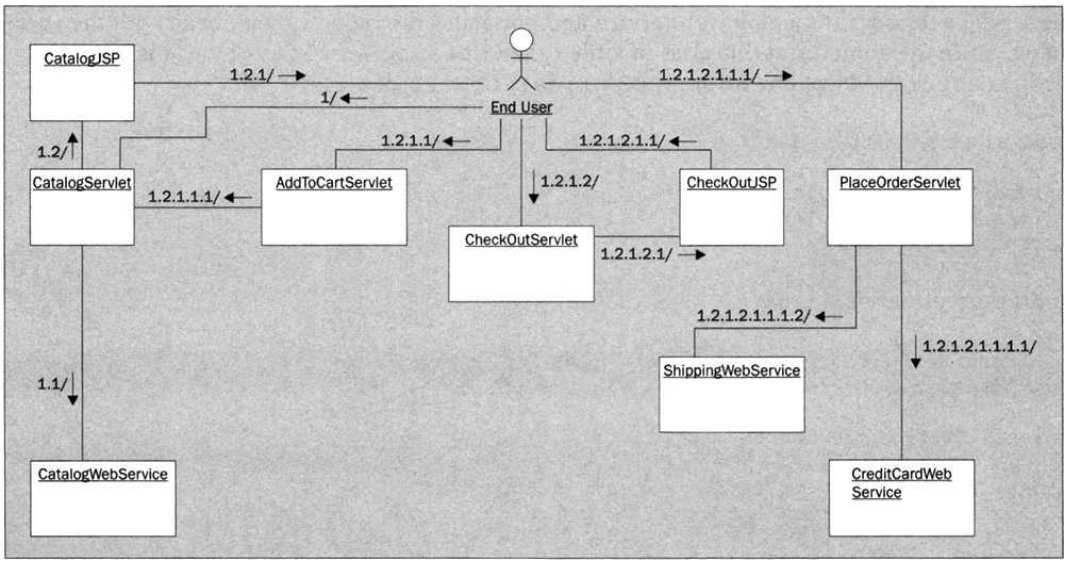
Architecture

The user actions on the client browser are transformed to requests to specific servlets. Depending on the requests, these servlets access the underlying web services to send or retrieve data. The retrieved data is stored as request-or session-scope beans and the request is forwarded to the JSP that renders the next view. The JSP retrieves the data from the request scope bean and renders them as HTML.

A typical set of interactions is listed below:

- The end user accesses the application by requesting the catalog
- The catalog servlet accepts the request and asks the catalog web service for the latest catalog
- This data is stored as a request-scope bean and forwarded to the catalog JSP
- The user adds an item to the shopping cart
- The request is sent to the add-to-cart servlet
- This servlet adds the item to the cart and forwards it to the catalog servlet
- The end user decides to check out
- The request is send to the checkout servlet that stores the order details as a request-scope bean and forwards the request to the checkoutJSP
- The user decides to place the order
- The request is sent to the place-order servlet
- This servlet accesses the credit card web service to authenticate the payment and then accesses the shipping web service to dispatch the order

This is depicted in the collaboration diagram shown below:



Application Classes

Now we will have a look at the various classes and interfaces that constitute the shopping application.

Attributes

This interface enumerates the various bean names used to store session-scope data. Store the contents of this class in a file called `Attributes.java` (it is in the `%jwsDirectory%\Chp11\ShoppingApplication\src`):

```
package com.acme.shopping;

public interface Attributes {
```

This is the name under which the catalog details are stored as a session-scope bean:

```
  public static final String CATALOG = "CATALOG";
```

This is the name under which the shopping cart details are stored as a session-scope bean:

```
  public static final String CART = "CART";
```

This is the name under which the order details are stored as a session-scope bean:

```
  public static final String ORDER = "ORDER";
}
```

CatalogServlet

This servlet accesses the catalog web service and populates the request scope bean with the current catalog. Store the contents of this class in a file called `CatalogServlet.java` (it is in the `%jwsDirectory%\Chp11\ShoppingApplication\src` directory):

```
package com.acme.shopping;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

Import the web services proxy:

```
import weblogic.jws.proxies.CatalogWS_Impl;
import weblogic.jws.proxies.CatalogWSSoap;

import catalog.Product;

public class CatalogServlet extends HttpServlet {
  private ServletContext ctx;

  public void init(ServletConfig config) throws ServletException {
    super.init(config);
    ctx = this.getServletContext() ;
}

protected void processRequest(HttpServletRequest req,
  HttpServletResponse res)throws ServletException, IOException {
```

Get the current session:

```
      HttpSession sess = req.getSession();
```

Use the client proxy for the catalog web service to get the list of products by invoking the web services. Store the list of products as a session scope bean:

```
      CatalogWS_Impl catalogWS = new CatalogWS_Impl();
      CatalogWSSoap catalogWSSoap = catalogWS.getCatalogWSSoap();
      sess.setAttribute(Attributes.CATALOG, catalogWSSoap.getCatalog());
```

Forward the request to the catalogJSP:

```
      ctx.getRequestDispatcher("/Catalog.jsp").forward(req, res);
    }

    protected void doGet (HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
      processRequest(req, res);
    }

    protected void doPost (HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
      processRequest(req, res);
    }
}
```

Catalog.jsp

This JSP code displays the current catalog. Store the contents of this class in a file called `Catalog.jsp` (it is in the `%jwsDirectory%\Chp11\ShoppingApplication\src`):

```
<%@page import="com.acme.shopping.Attributes"%>
<%@page import="catalog.Product"%>

<html>
  <head><title>Product Catalog</title></head>
  <body>
    <table border="1">
```

Retrieve the catalog:

```
      <%
        Product[] products =
          (Product[])session.getAttribute(Attributes.CATALOG);
        for(int i = O;products != null && i < products.length;i++)
        {
```

```
          %>
```

Loop through the catalog and render the details of products:

```
              <tr>
                <td><%= products[i].getName() %></td>
                <td><%= products[i].getDescription() %></td>

                <td><%= products[i].getPrice() %></td>
                <td>
```

Render the link to add the item to the shopping cart:

```
                <a href="addToCart?code=<%= products[i].getCode() %>">
                Add
                </a>
                </td>
              </tr>

            <% } %>
          </table>
```

Render the link to check out:

```
        <a href="checkOut">Check Out</a>

      </body>
    </html>
```

AddToCartServlet

This servlet adds the selected item to the shopping cart. Store the contents of this class in a file called `AddToCartServlet.java`(it is in `%jwsDirectory%\Chp11\ShoppingApplication\src`):

```
    package com.acme.shopping;

    import javax.servlet.*;
    import javax.servlet.http.*;
    import java.io.*;
    import java.util.*;

    public class AddToCartServlet extends HttpServlet {

      private ServletContext ctx;

      public void init(ServletConfig config) throws ServletException {
        super.init(config);
        ctx = this.getServletContext();
      }

      protected void processRequest(HttpServletRequest req,
          HttpServletResponse res) throws ServletException, IOException {
        HttpSession sess = req.getSession();
```

Get the selected item code:

```
        String code = req.getParameter("code");
```

Get the cart from the user's session:

```
        HashSet cart = (HashSet)req.getSession().getAttribute (Attributes.CART);
```

If the cart is not present, create a new cart:

```
        if(cart == null) cart = new HashSet();
```

Add the selected item to the cart:

```
        cart.add(code);
```

Add the cart back to the session:

```
        sess.setAttribute (Attributes.CART, cart);
```

Forward the request to the catalogJSP:

```
        ctx.getRequestDispatcher("/Catalog.jsp").forward(req, rest);
      }
```

```
    protected void doGet (HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
      processRequest(req, res);
    }

    protected void doPost (HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
      processRequest(req, res);
    }
  }
```

CheckOutServlet

This servlet processes the request when the user decides to proceed to checkout. Store the contents of this class in a file called `CheckOutServlet.java`:

```
  package com.acme.shopping;

  import javax.servlet.*;
  import javax.servlet.http.*;
  import java.io.*;
  import java.util.*;
  import catalog.Product;

  public class CheckOutServlet extends HttpServlet {
  private ServletContext ctx;

  public void init(ServletConfig config) throws ServletException {
    super.init(config);
    ctx = this.getServletContext();
  }

  protected void processRequest(HttpServletRequest req,
      HttpServletResponse res) throws ServletException, IOException {
    HttpSession sess = req.getSession();
```

Get the shopping cart from the user's session:

```
      HashSet cart = (HashSet) sess.getAttribute(Attributes.CART);
      if(cart == null || cart.isEmpty()) {
        PrintWriter writer = res.getWriter();
        writer.println("Shopping cart is empty");
        writer.close();
        return;
      }
```

Get the current catalog:

```
      Product[] products = (Product[])sess.getAttribute(Attributes.CATALOG);
      HashSet order = new HashSet();
      for(int i = 0; products != null && i < products.length; i++) {
        Iterator it = cart.iterator();
        while(it.hasNext()) [
          String code = (String)it.next();
          if(code.equals(products[i].getCode()))
            order.add(products[i]);
        }
      }
```

Populate the session with the details of the currently selected items:

```
      sess.setAttribute(Attributes.ORDER, order);
```

Forward the request to the checkoutJSP:

```
      ctx.getRequestDispatcher("/CheckOut.jsp").forward(req, res);
    }

    protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException [
      processRequest(req, res);
    }
```

```
    protected void doPost(HttpServletRequest req,
        HttpServletResponse res) throws ServletException, IOException [
      processRequest(req, res);
    }
  }
```

CheckOut.jsp

This JSP accepts the payment details, shipping address, and the number of items that are ordered. Store the contents of this class in a file called `CheckOut.jsp`:

```
<%@page import="com.acme.shopping.Attributes"%>
<%@page import="catalog.Product"%>
<%@page import="java.util.*"%>

<html>
  <head><title>Product Catalog</title></head>
  <body>
    <form action="placeOrder">
      <table border="1">
        <tr>
          <td colspan="3" align="right">Quantity</td>
        </tr>
```

Loop through the list of currently selected items and render a form showing the details and entering the quantities of selected items required:

```
        <%
          HashSet order =
            (HashSet)session.getAttribute(Attributes.ORDER);
          Iterator it = order.iterator();
          while(it.hasNext())
          {
            Product product = (Product)it.next();
        %>

        <tr>
          <td><%= product.getName() %></td>
          <td><%= product.getDescription() %></td>
          <td>
            <input type="text" name="<%= product.getCode() %>"/>
          </td>
        </tr>
        <% } %>
```

Render the form elements for capturing the billing and shipping details:

```
        <tr>
          <td colspan="2" align="right">
            <b>Name</b>
          </td>
          <td>
            <input type="text" name="name"/>
          </td>
        </tr>
        <tr><beginpage pagenum="452" id="page452"/>
          <td colspan="2" align="right">
            <b>Street</b>
          </td>
          <td>
            <input type="text" name="street"/>
          </td>
        </tr>
        <tr>
          <td colspan="2" align="right">
            <b>City</b>
          </td>
          <td>
            <input type="text" name="city"/>
          </td>
        </tr>
        <tr>
          <td colspan="2" align="right">
```

```
              <b>County</b>
            </td>
            <td>
              <input type="text" name="county"/>
            </td>
          </tr>
          <tr>
            <td colspan="2" align="right">
              <b>Postcode</b>
            </td>
            <td>
              <input type="text" name="postcode"/>
            </td>
          </tr>
          <tr>
            <td colspan="2" align="right">
              <b>Credit Card No:</b>
            </td>
            <td>
              <input type="text" name="creditCard"/>
            </td>
          </tr>
          <tr>
            <td colspan="2" align="right">
            </td>
            <td>
              <input type="submit" value="Place Order"/>
            </td>
          </tr>

        </table>

      </body>
    </html>
```

PlaceOrderServlet

This servlet processes the request when the user decides to place an order. The servlet accesses the underlying web services for authenticating credit card payment and dispatching the shipment. Store the contents of this class in a file called `PlaceOrderServlet.java`:

```
package com.acme.shopping;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

import java.net.URL;
```

Import the web services proxies:

```
import weblogic.jws.proxies.CreditCardWS_Impl;
import weblogic.jws.proxies.CreditCardWSSoap;
import weblogic.jws.proxies.ShippingWS_Impl;
import weblogic.jws.proxies.ShippingWSSoap;

import shipping.Address;
import shipping.Item;
import catalog.Product;

public class PlaceOrderServlet extends HttpServlet {

  private ServletContext ctx;

  public void init (ServletConfig config) throws ServletException {
    super.init (config);
    ctx = this.getServletContext();
  }

  protected void processRequest (HttpServletRequest req,
```

```
        HttpServletResponse res) throws ServletException, IOException {
      HttpSession sess = req.getSession();
```

Get the shopping cart from the session:

```
      HashSet cart = (HashSet)sess.getAttribute(Attributes.CART);
      if(cart == null $$ cart.isEmpty()) {
        PrintWriter writer = res.getWriter();
        writer.println("Shopping cart is empty");
        writer.close();
        return;
      }
```

Get the order details from the session:

```
      HashSet order = (HashSet)sess.getAttribute(Attributes.ORDER);
      if(order == null $$ order.isEmpty()) {
        throw new ServletException("Unable to find order");
      }
```

Create the `Address` object from the request parameter:

```
      Address address = new Address();
      address.setName(req.getParameter("name"));
      address.setStreet(req.getParameter("street"));
      address.setCity(req.getParameter("city"));
      address.setCounty(req.getParameter("county"));
      address.setPostcode(req.getParameter("postcode"));

      ArrayList itemList = new ArrayList();
```

Iterate through the order and calculate the total amount:

```
      double total = 0;
      Iterator it = order.iterator();
      while(it.hasNext()) {
        Product product = (Product) it.next();
        int quantity =
        Integer.parseInt(req.getParameter(product.getCode()));

        Item item = new Item();
        item.setCode(product.getCode());
        item.setQuantity(quantity);
        itemList.add(item);

        total += quantity*product.getPrice();
      }
```

Create the list of items to be ordered:

```
      Item[] items = new Item[itemList.size()];
      for(int i = 0;i < items.length;i++) items[i] = (Item)itemList.get(i);
```

Process the credit card payment and if successful, place the order:

```
      try {
        if(!debitAccount(req.getParameter("ccNo"), total)) {
          PrintWriter writer = res.getWriter();
          writer.println("Credit card authorisation failed");
          writer.close();
          return;
        } placeOrder(address, items);
      } catch(Exception ex) {
        throw new ServletException(ex);
      }

      PrintWriter writer = res.getWriter();
      writer.println("Your order has been successfully processed");
      writer.close();
      return;
    }

    protected void doGet (HttpServletRequest req,
        HttpServletResponse res) throws ServletException, IOException {
```

```
        processRequest(req, res);
      }

      protected void doPost (HttpServletRequest req,
          HttpServletResponse res) throws ServletException, IOException {
        processRequest (req, res);
      }
```

This method uses the credit card web service proxy to invoke the web service for processing payment:

```
      private boolean debitAccount(String ccNo, double amt) throws Exception {
        CreditCardWS_Impl creditCardWS = new CreditCardWS_Impl();
        CreditCardWSSoap creditCardWSSoap = creditCardWS.getCreditCardWSSoap();

        return creditCardWSSoap.debit(ccNo, amt);
      }
```

This method uses the shipping web service proxy to invoke the web service for placing the order:

```
      private void placeOrder(Address address, Item[] items) throws Exception {
        ShippingWS_Impl shippingWS = new ShippingWS_Impl();
        ShippingWSSoap shippingWSSoap = shippingWS.getShippingWSSoap();
        shippingWSSoap.placeOrder(address, items);
      }
    }
```

Deployment Descriptor

The deployment descriptor, `web.xml`, contains the following details:

- The servlet definitions
- The servlet mappings
- Servlet context listener declaration
- Context parameters specifying the endpoint URL and WSDL locations for the various web services

Servlet definitions:

```
    <!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>
```

Add-to-cart servlet:

```
      <servlet>
        <servlet-name>addToCart</servlet-name>
        <servlet-class>com.acme.shopping.AddToCartServlet</servlet-class>
      </servlet>
```

Catalog servlet:

```
      <servlet>
        <servlet-name>catalog</servlet-name>
        <servlet-class>com.acme.shopping.CatalogServlet</servlet-class>
      </servlet>
```

Checkout servlet:

```
      <servlet>
        <servlet-name>checkOut</servlet-name>
        <servlet-class>com.acme.shopping.CheckOutServlet</servlet-class>
      </servlet>
```

Place-order servlet:

```
      <servlet>
        <servlet-name>placeOrder</servlet-name>
        <servlet-class>com.acme.shopping.PlaceOrderServlet</servlet-class>
      </servlet>
```

URL mappings:

```
      <servlet-mapping>
        <servlet-name>addToCart</servlet-name>
```

```
      <url-pattern>/addToCart</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
      <servlet-name>catalog</servlet-name>
      <url-pattern>/catalog</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
      <servlet-name>checkOut</servlet-name>
      <url-pattern>/checkOut</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
      <servlet-name>placeOrder</servlet-name>
      <url-pattern>/placeOrder</url-pattern>
    </servlet-mapping>

  </web-app>
```

Building and Deploying the Application

In this section we will build and deploy our web application on WebLogic server. For this first we need to compile the classes. Please make sure that you have the `weblogic.jar` file in your classpath, while compiling these classes as well as the web service proxy JAR files:

```
set classpath=%BEA_HOME%\server\lib\weblogic.jar;.\CatalogClient.jar;
\CreditCardClient.jar;.\ShippingClient.jar
```

After setting the classpath, we will change to the directory

`%jwsDirectory%\Chp11\ShoppingApplication\src` and compile the files:

```
javac -d ..\classes Attributes.java
javac -d ..\classes CatalogServlet.java
javac -d ..\classes AddToCartServlet.java
javac -d ..\classes CheckOutServlet.java
javac -d ..\classes PlaceOrderServlet.java
```

Once we have compiled the classes, create the following directory structure:

```
/Catalog.jsp
/CheckOut.jsp
/WEB-INF/web.xml
/WEB-INF/lib/CatalogClient.jar
/WEB-INF/lib/CreditCardClient.jar
/WEB-INF/lib/ShippingClient.jar
/WEB-INF/classes/com/acme/shopping/Attributes.class
/WEB-INF/classes/com/acme/shopping/StarupListener.class
/WEB-INF/classes/com/acme/shopping/CatalogServlet.class
/WEB-INF/classes/com/acme/shopping/PlaceOrderServlet.class
/WEB-INF/classes/com/acme/shopping/AddToCartServlet.class
/WEB-INF/classes/com/acme/shopping/CheckOutServlet.class
```

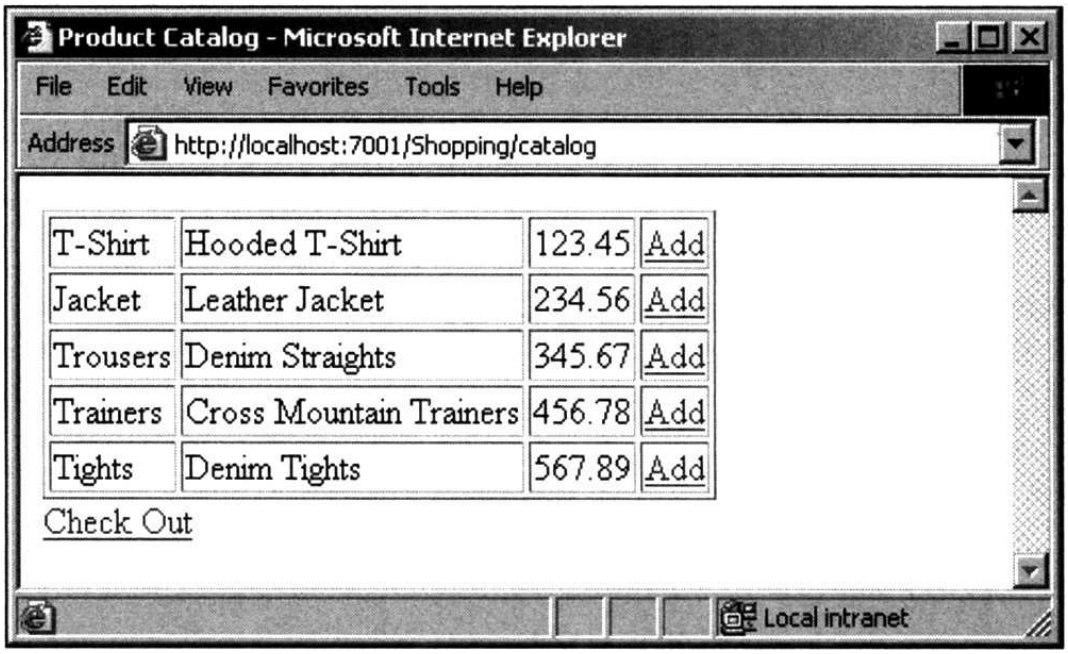
Then run the `jar` command to create a **WAR** file with structure shown above:

```
jar cf Shopping.war *
```

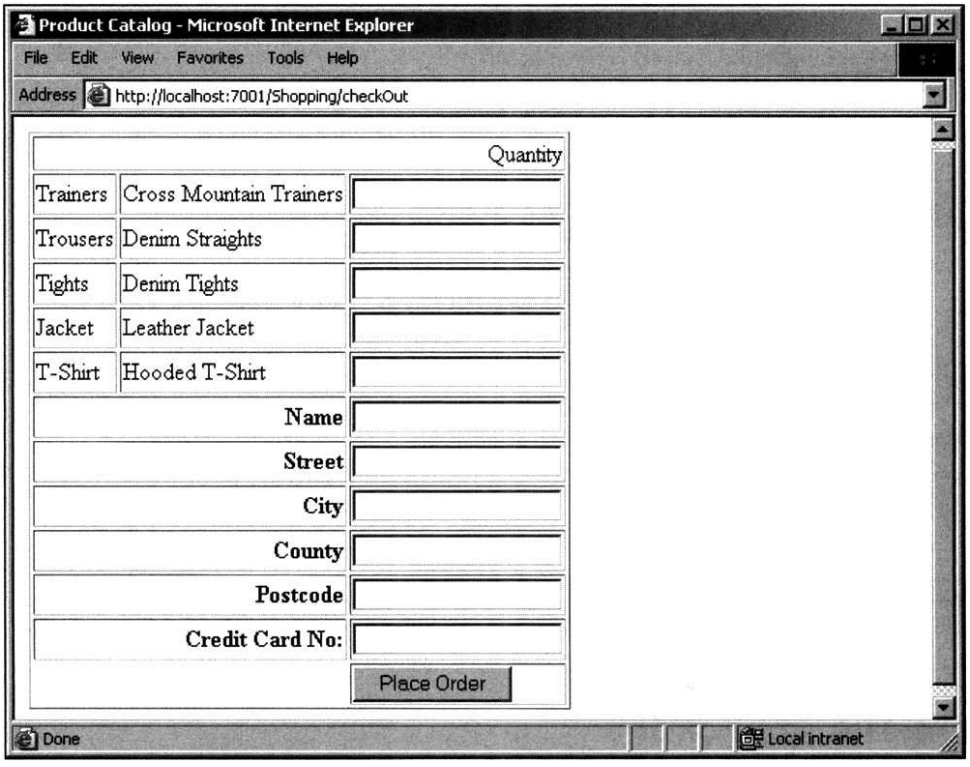
Copy the WAR file to `%WLS_HOME%/Weblogic700/samples/workshop/applications` to deploy the EJB.

Running the Application

The application can be accessed by the URL `http://localhost:7001/Shopping/catalog`:



After adding the items to the cart click on the `Check Out` link:



Enter the required information and submit the form. If the payment is processed successfully we can see the order details in the WebLogic console window.

Summary

In this chapter we have developed a full-fledged case study using web services for integrating a set of disparate applications within an enterprise. We looked at exposing standard J2EE components such as datasources and EJBs as SOAP-based web services. In the case study we developed a web application that leveraged the services provided by a set of existing applications, and covered the use of web services for integrating the new application with a set of existing applications.

We have seen how to use WebLogic Workshop for developing, deploying, and testing web services. We also looked at developing asynchronous web services to improve the transaction throughput and performance in web-based applications.