Web services with WebSphere Studio: Build and test

Presented by WebSphere Developer Domain

www7b.software.ibm.com/wsdd/

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

1. Introduction .............................................................. 2
2. Work order management ............................................. 5
3. Creating the project .................................................... 10
4. Implementing the service ............................................. 13
5. Creating the Web service ............................................ 17
6. Generating the WSDL ................................................. 23
7. Building a proxy and a test client .................................. 25
8. Packaging your application for deployment .................. 29
9. Summary ................................................................. 31
Section 1. Introduction

Should I take this tutorial?

This is the first part in a two-part series on developing and deploying Web services.

You should take this tutorial if you want to develop Web services, if you are new to the Application Developer configuration of WebSphere Studio, and if you want to quickly learn how to use this tool for building Web services. The tutorial assumes basic knowledge of Java. You do not need to have a background in Web services and in technologies such as the Simple Object Access Protocol (SOAP), the Web Services Description Language (WSDL) and Universal Description Discovery and Integration (UDDI). Application Developer takes care of building all of this for you. The tutorial also explains the code that Application Developer generates, so you have a chance to learn how Application Developer uses SOAP and WSDL in the work order scenario, the example application created in this tutorial.

You'll find the tutorial easier to follow if you download and install the Application Developer configuration of WebSphere Studio. You can download the tool from the Tools and resources on page 4 page. However, this is not a prerequisite, and you can follow the tutorial even if you do not have a working copy of Application Developer at your disposal. If you go ahead and install Application Developer, the last dialog in the installation will ask what your primary user role is. Select Web Services Developer as shown below to get started:
What is this tutorial about?

In this tutorial you'll learn the following skills:

- How to develop Web services using Application Developer
- How to use existing functionality that you have already developed in Java and expose it as a Web service
- How to develop the Web service description and package the project as an enterprise archive

You'll do all this using the set of tools packaged within Application Developer, so you are not required to manually build any of these components. You'll also learn how to test your Web service from within Application Developer using a simple client.

The tutorial is based on a real business scenario involving a work management application. The Web service you'll build, called `createWorkOrder`, allows clients to create new work orders for a company's workforce. The work orders are managed by the work manager object.

*Part 2* of this tutorial takes the Web service you develop here and shows you how to deploy it on a WebSphere Application Server, how to publish the service in a registry, and how to test the Web service after you deploy it.
Tools and resources

The following tools are necessary if you plan to run the examples in this tutorial:

- **WebSphere Studio Application Developer** trial edition. (313MB)
- **WebSphere Application Server Advanced Developer Edition** version 4.0. (146MB)

---

About the author

Ron Ben-Natan is chief technology officer at ViryaNet Inc., a software provider of wireless workforce management and field service solutions. Prior to that he worked for companies such as Intel, AT&T Bell Laboratories, Merrill Lynch and as a consultant at J.P. Morgan. He has a Ph.D. in Computer Science in the field of distributed computing and has been architecting and developing distributed applications for over 15 years. His hobby is writing about how technology is used to solve real problems, and he has authored and co-authored numerous books, including *CORBA: A Guide to Common Object Request Broker Architecture*, *CORBA on the Web*, and *The SanFrancisco Developer's Guide* all published by McGraw-Hill; *IBM WebSphere Starter Kit* and *IBM WebSphere: The Complete Reference* both published by Osborne/McGraw-Hill; and *Integrating Service Level Agreements* published by John Wiley & Sons. He is also the author of numerous articles and tutorials. He can be reached at rbennata@hotmail.com.
Section 2. Work order management

Overview

The scenario used in this tutorial is taken from a real business environment. It is a bit unfortunate that most examples of Web services follow a stock ticker scenario or some permutation of the "Hello World" program. Given these examples you may think that Web services are useful only in environments involving access to simple information from a Web site. In fact, Web services shine in business environments that have decoupled applications that need to interact in order to complete an end-to-end business process. These systems may or may not reside on the same network segment and may not even be managed by the same organization. Before Web services this type of environment would require complex and costly integration and private networks. Web services use the connectivity of the Internet, the simplicity of HTTP, and the wide adoption of SOAP, WSDL and UDDI to provide the same level of integration at a fraction of the cost.

Let's move on to review the work order business scenario.

Work order management scenario

Imagine a company that's in the business of doing TV satellite dish installations. In order to protect the innocent, let's call them InstallCo. They own a workforce of some 2,500 installers who specialize in satellite dish installations and install satellite dishes all over the U.S. Every day each of these installers has 3-5 work orders for installing a dish. In the morning, each installer gets a to do list and goes off to complete the day's work.

InstallCo does not get the customer order directly. Dish companies and retailers sell the service to consumers and book the appointment. These companies then forward the information to InstallCo, which creates a work order in its work management system. InstallCo is responsible for performing the installation and updating the originator regarding the status of the work order.

InstallCo is paid based on the number of completed work orders. It is therefore in InstallCo's best interest to work with as many retailers and dish companies as possible. The most difficult thing in getting a new retailer to work with InstallCo is the way in which new orders are transferred from the retailer to InstallCo. Most retailers capture new customers in their own, separate system. It is not feasible to perform complex integration with so many systems, and yet InstallCo does not want these orders to arrive as paper requests because this makes the work handling process expensive and error prone. So how does InstallCo provide a simple, standards-based way to electronically create a new installation order over the Web? Using a Web service, of course.
The createWorkOrder service

InstallCo's IT department inspected the order management system and listed the information required to capture a new work order. This information includes the appointment times, the customer address, and the type of installation. It also includes the name of the company that initiated the installation order. Once the order is created in InstallCo's work management system, the order number is returned to the client who invoked the Web service. This number can later be used to track the progress of the order.

The createWorkOrder Web service is based on this analysis. In making a call to this service, the required work order information is sent and the order number is returned. The service is the front end to InstallCo's work management system and is exposed over the Web for all retailers and dish companies to use. Because InstallCo is using a Web service to expose this interface, the order data is passed as an XML document:

As the illustration shows, when a retailer or dish provider sets an appointment with a customer, the order information is packaged as XML data and used to invoke the Web service, which in turn creates the order within the work management system.

The fact that work order creation is packaged as a Web service makes it available over the public Internet and thus accessible to all at a very low cost of ownership. Packaging
this function as a Web service has one other important effect: It makes InstallCo easy
to find as a service provider and makes the createWorkOrder function easy to
interface with even if you have never used it before. Making the function a Web service
means that it is self-describing and easy to publish.

Publishing self-describing services

InstallCo works with many retailers. Some of these retailers have large IT departments
and some have very small IT departments. Because InstallCo wants to get as many
work orders as possible, it is important that building an interface to InstallCo is simple.
It is also important that the createWorkOrder service be self-describing so that a
retailer does not have to approach InstallCo's IT department and go through a lengthy
integration project. And it is important that the service can be used regardless of the
retailer's platform, such as WebSphere systems, .NET systems, etc. All this is satisfied
through Web services.

In the same way that InstallCo may be listed in the yellow pages, it also wants to use
the Web services framework to self-publish the createWorkOrder service on public
registries. This means that a retailer who does not know InstallCo can search through
such registries for companies who do dish installations, discover InstallCo, and then
quickly build an interface to InstallCo using the self-describing interface point.

The combination of a published, self-describing interface means that "electronic
bonding" is easy. A new retailer who needs to commit to new customers for dish
installations can search public registries looking for companies that can do such
installations. At the end of this discovery process, using Universal Description
Discovery and Integration (UDDI), they find InstallCo. The retailer then uses the Web
Service Description Language (WSDL) file to learn how to invoke the service. A proxy
is generated based on the WSDL; this proxy is used to invoke the createWorkOrder
service. The invocation is done using the Simple Object Access Protocol (SOAP):
All this is possible because of the technologies forming Web services: UDDI, WSDL and SOAP.

60 seconds on SOAP, WSDL and UDDI

This tutorial does not teach you about SOAP, WSDL and UDDI (there are many resources on these subjects), and you do not need to have any previous knowledge of these technologies in order to follow this tutorial. This is one of the great things about Application Developer -- it takes care of all generation involved with both creating and invoking the Web service.

Still, if you are not yet familiar with SOAP, WSDL and UDDI, it is worthwhile spending a few minutes familiarizing yourself with them so you have the right reference framework.

Web services are functional elements deployed on a node on the network and accessed over the Internet. This description is quite generic and doesn't say too much;
what makes such a service endpoint a Web service is the *how*, not the *what*. Web services are based on a set of standards -- specifically SOAP, WSDL, and UDDI. SOAP is the protocol by which a remote client can invoke the functionality implemented by the Web service. Developers of Web services use WSDL to define the metadata describing the Web service, and remote clients use WSDL to learn what arguments are required for invoking the Web service (as well as other things required to make the remote call). Web services providers use UDDI to publish their Web services, and clients of Web services use UDDI to discover where these Web services (and the metadata describing them) reside.

Enough with theory. Let's move on to build the service.
Section 3. Creating the project

Creating a new Web project

The first thing you need to do is create a new Web project. Select File => New => Project from the menu bar. This opens the New Project wizard. Select Web in the left pane and Web Project in the right pane. Then click Next. This will bring up the next dialog in the wizard (shown below) in which you define the Web project. Enter the project name and the enterprise application project name. The default context root will be identical to the project name, but you can override it. Click Finish. Application Developer creates all the required components for the project.
Adding the Xerces and SOAP libraries

Because you are going to be working with XML in this project, you need to add the Xerces library to your project. Many of the XML parsers and libraries come from the Apache organization (see Resources on page 31). Xerces is one of the active projects within apache.org. The Xerces libraries (named after the Xerces Blue butterfly) implement XML parsers as well as mechanisms for generating XML.

Since you will also be using SOAP, you will need to load the Apache SOAP library as well.

To add an external library to your project, right-click on the project folder and select Properties. In the left pane, select Java Build Path and click the Libraries tab as shown below:

Select Add External JARs... and use the file dialog to locate the xerces.jar and the soap.jar files in plugins/com.ibm.etools.websphere.runtime/lib. Click OK.

Your environment is now all set and you are ready to move on to implement the Web service.
Because this tutorial is meant to be self-sufficient and not to involve the use of a large system that you have no access to, you will build a very simple back end in the next section so that you can use Web services. You will build a Java class for work orders as well as a Java class for the work order manager.
Section 4. Implementing the service

Creating the WorkOrder class

The first class that you need is the `WorkOrder` class. This class maintains the data required to perform a work order.

First you need to open the Java perspective. Choose **Perspective**=> **Open**=> **Java** from the menu bar. You are now in the Java developer mode. Open the `WorkOrderManagementIfc` folder in the Packages pane. Right-click on the sources folder and choose **New=> Class**. This brings up the Java Class dialog allowing you to create a new Java class. Enter the package name and the class name as shown below:
Now you need to write the code for the work order class. You can download the sample code. The code for the work order class is very simple and some of the get and set methods are omitted for the sake of brevity. The order number is set in the constructor when you create a new work order:

```java
package com.installco;
import java.util.Date;
import java.lang.String;
public class WorkOrder {
    // input data
    private String customerName;
    private String addressStreet;
    private String addressCity;
    private String addressState;
    private String addressZip;
}```
private String sourceCompany;
private Date appointmentDate;

// output data
private int orderNumber;

// manage a sequence to assign order number
static int nextOrderNumber = 0;

public WorkOrder() {
    // set the order number upon creation.
    this.orderNumber = nextOrderNumber;
    // increment the sequence
    nextOrderNumber++;
}

public String getCustomerName() {
    return this.customerName;
}

public void setCustomerName(String customerName) {
    this.customerName = customerName;
}
...

public Date getAppointmentDate() {
    return this.appointmentDate;
}

public void setAppointmentDate(Date appointmentDate) {
    this.appointmentDate = appointmentDate;
}

// no set method - the order number is created by the constructor and cannot be changed later
public int getOrderNumber() {
    return this.orderNumber;
}


Creating the WorkOrderManager class

The only other class you need for the back end is the WorkOrderManager class. This class manages a vector of work orders and implements the methods used for creating a new work order. This is the method that you will later wrap as a Web service.

To create the class, follow the same steps as shown in the previous panel. The code for this class follows:

package com.installco;
import java.util.Vector;
import java.util.Date;
public class WorkOrderManager {

static Vector orders = new Vector();

// Get input data and create a new work order.
// Return the order number
public int createNewWorkOrder(
    String customerName,
    String addressStreet,
    String addressCity,
    String addressState,
    String addressZip,
    String sourceCompany,
    Date appointmentDate) {

    WorkOrder newOne = new WorkOrder();
    newOne.setCustomerName(customerName);
    newOne.setAddressStreet(addressStreet);
    newOne.setAddressCity(addressCity);
    newOne.setAddressState(addressState);
    newOne.setAddressZip(addressZip);
    newOne.setSourceCompany(sourceCompany);
    newOne.setAppointmentDate(appointmentDate);

    orders.addElement(newOne);

    return newOne.getOrderNumber();
}

Now that you have completed the back end you are ready to build the Web service wrapper.
Section 5. Creating the Web service

Selecting the project and the class

From the menu bar choose **File**=> **New**=> **Other**. This brings up a dialog on which you select **Web Services** from the left pane and then **Web Service** from the right pane. Click **Next** to move to the next page in the wizard.

Select the WorkOrderManagementIfc Web project using the pulldown combo box. If this is not the first time you are generating the Web service, select the **Overwrite files without warning** checkbox. This will ensure that if you change some of the properties of the code but have a previously saved version of the Web service files, you will replace them. Click **Next**.

In the next screen of the wizard you are asked the class which you want to expose as a Web service. Select the WorkOrderManager entry and click **Next**:

This will bring up the Java Bean Selection page in which you click **Next**; the right Java bean is already being displayed. At this stage you have finished specifying which functionality you want to wrap as a Web service and you are ready to go ahead and define the properties of the Web service itself.

Configuring the Web service

In the Web Service Java Bean Identity page, change the Web service URI to `urn:WorkOrderManager`. This resource name is used by the Web services.
infrastructure to uniquely identify the service and map an incoming request with an implementation routine. Choose **Session** from the pulldown so that the Scope value is **Session**. This means that a Web service invocation is always done within the context of a client session. This is not important in this simple example, but it is always safer to use session scope in real applications where you may need to manage sessions and global application data. The wizard page should look as follows:

![Web Service Wizard Page](image)

You can leave the rest of the values as is and click **Next**. When using Application Developer you can usually click **Finish**. We will continue by walking through the step-based process in order to describe some of the more advanced features of the tool.

---

**About the service URN**

The service URN uniquely identifies the service. Because you are building a Web service wrapper to an existing piece of code (the `WorkOrderManager` class), you need to tell the server how to map the unique service identifier to the implementation class.

While you do not need to do the mapping explicitly (this is one of the files that is automatically generated for you by Application Developer), it is useful to know where
this mapping occurs. Open the following folders in the navigator pane:

**WorkOrderManagementIfc**=> **webApplication**=> **WEB-INF**=> **isd**=> **java**=> **com**=> **installco**. Click **WorkOrderManager.isd** to view the deployment descriptor; it should read as follows:

```xml
<isd:service id="urn:WorkOrderManager"
    xmlns:isd="http://xml.apache.org/xml-soap/deployment">
  <isd:provider type="java" scope="Session" methods="createNewWorkOrder">
    <isd:java class="com.installco.WorkOrderManager" static="false"/>
  </isd:provider>
  <isd:mappings/>
</isd:service>
```

The URN entered in the wizard is used as the identifier of the service. The provider defines the Java class implementing the Web service. This is used by the SOAP server to perform the method invocation when a request comes into the server.

---

**Selecting the method and the encoding style**

Once you define the service URN and click **Next**, you need to pick the method(s) that you want wrapped as a Web service. In this case you just have one method that you need to wrap: the **createNewWorkOrder** method. You also need to determine the encoding method used both for the input arguments as well as for the output arguments. Make sure that SOAP encoding is selected for both as shown below:
More on encoding style

Encoding style must match the way that you code your methods. Using SOAP encoding means that Java types are mapped to predefined SOAP types. In the example used in this tutorial, all the arguments are simple types (like `int`) or standard Java classes (like `String` and `Date`), hence SOAP encoding makes perfect sense. Using SOAP encoding saves you quite a bit of work and leaves the type mapping and XML creation up to the SOAP libraries.

Literal XML encoding means that you need to explicitly do the data extraction and injection using XML element objects. When using literal XML encoding you need to explicitly create an XML document object into which you build in all the XML elements that you require. You need to write more code -- usually quite a bit of code. The advantage of this method is that it is completely generic. If you are using literal XML encoding for your output, you can build any XML structure you need and it will be inserted directly into the SOAP message. For example, if you had to write the `createNewWorkOrder` method to use literal encoding for its output, it would look like the following:

```java
// Get input data and create a new work order.
// Return the order number within an XML document
// which needs to be manually built
public Element createNewWorkOrder{
```
String customerName,
String addressStreet,
String addressCity,
String addressState,
String addressZip,
String sourceCompany,
Date appointmentDate) {

    WorkOrder newOne = new WorkOrder();
    newOne.setCustomerName(customerName);
    newOne.setAddressStreet(addressStreet);
    newOne.setAddressCity(addressCity);
    newOne.setAddressState(addressState);
    newOne.setAddressZip(addressZip);
    newOne.setSourceCompany(sourceCompany);
    newOne.setAppointmentDate(appointmentDate);

    orders.addElement(newOne);

    // Generate the output XML - explicitly
    DocumentImpl doc = new DocumentImpl();
    try {
        Element oderNumElement = doc.createElement("OrderNumber");
        doc.appendChild(oderNumElement);
        oderNumElement.appendChild(
            doc.createTextNode(new String(newOne.getOrderNumber())));
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    return doc.getDocumentElement();
}

So long as you're using standard types, SOAP encoding means that the whole Web service request/response process is completely transparent to you. Incidentally, if you are using SOAP encoding and want to see what mapping will be created (or need to change it to create a custom mapping) select the **Show server (Java to XML) type mappings** checkbox. This allows you to see the default mapping and modify it as necessary. As an example, the following wizard step shows that java.util.Date is mapped to the XML dateTime type:
Web Service

Web Service Java to XML Mappings

Review your Web service type mappings and make any necessary changes before proceeding to the next page.

java.lang.String, SOAP encoding
int, SOAP encoding
java.util.Date, SOAP encoding

- Show and use the default Java bean mapping
- Show and use the default DOM Element mapping
- Edit and use a customized mapping

Encoding style: http://schemas.xmlsoap.org/soap/encoding/
XML type namespace: http://www.w3.org/2001/XMLSchema

XML type name: dateTime
Bean class: java.util.Date
Serializer class: 
Deserializer class: 

< Back  Next >  Finish  Cancel
Section 6. Generating the WSDL

Auto-generation of the WSDL

WSDL is used to describe the Web service. This description is used by clients that wish to invoke the service. The service description is the metadata that makes a Web service self-describing.

In Part 2 of this tutorial, you will learn how to publish the WSDL file to a registry. This allows a new retailer to query the registry and learn about the existence of the service. The client can then retrieve the WSDL and learn from it how to invoke the Web service.

WSDL is an application of XML. While it is not very difficult to write WSDL, Application Developer saves you this bit as well. Application Developer analyzes the metadata it extracts from the Java class as well as the bit of properties you set in the wizard, and generates the WSDL for you. This saves you quite a bit of work as well as the hassle involved with debugging errors.

There are two files that are generated by Application Developer:
- WorkOrderManager-service.wsdl contains the service interface and
- WorkOrderManager-bindings.wsdl contains the service implementation. The service interface describes the abstract type interface and its protocol binding while the service implementation describes the service access (location) information.

Let's look at the WSDL files generated by Application Developer for your project.

WorkOrderManager-service.wsdl

To see the two WSDL files generated by Application Developer, use the Navigator pane to select the WorkOrderManagementIfc=> webApplication=> wsdl folder. The service interface definition is the high-level description of the service. The most important element is the soap:address element which defines the URL that should be used to make the Web service invocation. In the WSDL file shown below, the URL is a servlet that is part of the Web services runtime called the RPC router. You'll find more on this in the following panels: About the proxy on page 25 and About the RPC router on page 26.

```xml
</definitions>
```
WorkOrderManager-binding.wsdl

The service implementation for the WorkOrderManager Web service defines the data structures used for the request and response of the Web service, the method exposed as a Web interface, and the SOAP attributes (such as encoding style). The service implementation generated by Application Developer for the createNewWorkOrder service is shown below (some of the verbose XML has been omitted for brevity):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="WorkOrderManagerRemoteInterface" ...>
  <message name="createNewWorkOrderRequest">
    <part name="customerName" type="xsd:string"/>
    <part name="addressStreet" type="xsd:string"/>
    <part name="addressCity" type="xsd:string"/>
    <part name="addressState" type="xsd:string"/>
    <part name="addressZip" type="xsd:string"/>
    <part name="sourceCompany" type="xsd:string"/>
    <part name="appointmentDate" type="xsd:dateTime"/>
  </message>
  <message name="createNewWorkOrderResponse">
    <part name="result" type="xsd:int"/>
  </message>
  <portType name="WorkOrderManagerJavaPortType">
    <operation name="createNewWorkOrder">
      <input name="createNewWorkOrderRequest" ... />
      <output name="createNewWorkOrderResponse" ... />
    </operation>
  </portType>
  <binding name="WorkOrderManagerBinding" ... >
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="createNewWorkOrder">
      <input name="createNewWorkOrderRequest" ... />
      <output name="createNewWorkOrderResponse" ... />
    </operation>
  </binding>
</definitions>
```
Section 7. Building a proxy and a test client

Generating the proxy

Part of the beauty of Application Developer is that it not only generates the server-side code (the Web services wrapper), it also generates a client-side proxy. This proxy is used by client code to make a Web service invocation. In making such an invocation you do not need to write SOAP and XML documents; you merely need to use the proxy Java object.

In this tutorial you use the proxy and the test client to invoke the Web service from within Application Developer.

After clicking Next on the dialog from the previous section (the Web service Java bean methods wizard page) you move to the Web service binding proxy generation page of the wizard. This allows you to define which binding to base proxy generation on and the class name for the proxy. Leave both unchanged and click Next as shown:

About the proxy

WSDL generates a class implementing a proxy to your service. While you don't need to know anything about the proxy, it is enlightening to look at what it does. You can view
the source for the generated proxy by using the Navigator pane and selecting
WorkOrderManagementIfc=> source=> proxy=> soap=> com=> installco and
double-clicking on WorkOrderManagerProxy.java.

For each of the methods that you wrap as a Web service, Application Developer
generated a method within the proxy. In the work order example, the generated method
is createNewWorkOrder as shown below (some code is omitted for brevity):

```java
class NameSpace {
    public synchronized int createNewWorkOrder(java.lang.String
    customerName, java.lang.String
    addressStreet, java.lang.String addressCity, ...) throws Exception
    {
        String targetObjectURI = "urn:WorkOrderManager";
        String SOAPActionURI = "";
        ...
        call.setMethodName("createNewWorkOrder");
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
        call.setTargetObjectURI(targetObjectURI);
        Vector params = new Vector();
        Parameter customerNameParam = new Parameter("customerName",
        java.lang.String.class, customerName, Constants.NS_URI_SOAP_ENC);
        params.addElement(customerNameParam);
        Parameter addressStreetParam = new Parameter("addressStreet",
        java.lang.String.class, addressStreet, Constants.NS_URI_SOAP_ENC);
        params.addElement(addressStreetParam);
        ...
        call.setParams(params);
        // getURL here returns http://localhost:8180/WorkOrderManagementIfc/servlet/rpcroute
        Response resp = call.invoke(getURL(), SOAPActionURI);
        //Check the response.
        if (resp.generatedFault())
        {
            Fault fault = resp.getFault();
            call.setFullTargetObjectURI(targetObjectURI);
            throw new SOAPException(fault.getFaultCode(), fault.getFaultString());
        }
        else
        {
            Parameter refValue = resp.getReturnValue();
            return ((java.lang.Integer)refValue.getValue()).intValue();
        }
    }
}
```

Each invocation is done through a call object. The proxy sets the input arguments in
a vector, the method name to be invoked, and the service URN (which is then used on
the server side to associate an incoming request with an implementation). Quite clever,
isn't it?

---

**About the RPC router**

The rpc router is a servlet that exists both within the WebSphere v4.0 Test
Environment within Application Developer as well as within the WebSphere Application
Server (which you will use in Part 2 of this tutorial). As you saw in the previous panel,
all proxies forward requests to the rpc router. The rpc router uses the service
URN to look up the Java class that implements the Web service and delegates the request to that class. It is also responsible for converting the SOAP types to Java types and vice versa (if using SOAP encoding). The full calling scheme follows:

![Diagram showing the flow of data between a Retailer, Proxy, SOAP, InstallCo, RPC Router, and WorkOrderManager class]

---

**Generating and launching the test client**

The last element that Application Developer generates for you is a simple test client.

Recall that the last wizard step you completed was the Web service binding proxy generation in which you specified the proxy class name (or used the default). When you click Next, you move on to the Web service test client page. Select the **Launch the test client** checkbox and click **Next**. In the Web service sample generation step of the wizard, select the **Generate a sample** checkbox and click **Finish**.

At this point, Application Developer goes ahead and generates all the WSDL files, the proxy class, and other definitions required for runtime. You are now ready to test your code using a sample test client that Application Developer generates for you.

The client is a simple JSP that uses the proxy. To launch the test client, use the Navigator pane and select `WorkOrderManagementIfc=> webApplication=> sample=> WorkOrderManager`. Right-click on `TestClient.jsp` and select **Run on server**. If the WebSphere test environment is not running, Application Developer will start the server for you.

The test application has three panes. In the left pane you select which Web service method you wish to invoke. Click **createNewWorkOrder**. The top right pane will then contain a form matching the input arguments to the service. Notice that because the WSDL service implementation file contains all typing information, the form knows that the last argument is a date and tells you the data format that it is expecting. Type in some input data and click **Invoke**. The lower right pane displays the output of the service -- the order number that was created:
As you are probably guessing, the sample test application (specifically `Result.jsp`) instantiates a proxy object and uses it to make the call.

You have now finished developing the Web service and have tested it to ensure that it works. The last step you need to do within Application Developer is package your application so that you can deploy it onto the WebSphere Application Server and publish your service on a UDDI registry. These latter two topics are covered in Part 2 of this tutorial.
Section 8. Packaging your application for deployment

Modifying the WSDL

The service interface file (WorkOrderManager-service.wsdl) includes address information about where the service resides and where the WSDL files reside. Application Developer points to these locations at the test environment when it generates these files. For example, the SOAP address element will look as follows (assuming the test environment is listening on port 8180):

```xml
<soap:address location="http://localhost:8180/WorkOrderManagementIfc/servlet/rpcrouter",
```

This is good for testing your code, but once you are ready to deploy your application, you should modify the address information. If you are deploying on a node called xxx.yyy.zzz on port ppp then modify every instance of localhost:8180 (or 8080 if you are using the default test environment port number) to xxx.yyy.zzz:ppp or xxx.yyy.zzz if deploying on port 80.

Building the EAR

The last step is to package your application as an EAR. Rebuild your project one last time by choosing Project=> Rebuild All from the menu bar.

In the Navigator pane, right-click on WorkOrderManagementEAR and choose Export EAR File... Select a directory to export the file to and click Finish. Application Developer will create the EAR file with an embedded WAR file, which includes all the Web service specifics as well as the implementation code:
<table>
<thead>
<tr>
<th>Name of File</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkOrderManagement.war</td>
<td>WAR File</td>
</tr>
<tr>
<td>Manifest.mf</td>
<td>MF File</td>
</tr>
<tr>
<td>ibm-application-ext.xml</td>
<td>XML File</td>
</tr>
<tr>
<td>application.xml</td>
<td>XML Document</td>
</tr>
<tr>
<td>serverPreferences</td>
<td>File</td>
</tr>
<tr>
<td>web.xml</td>
<td>XML Document</td>
</tr>
<tr>
<td>loc.html</td>
<td>HTML Document</td>
</tr>
<tr>
<td>TestClient.jsp</td>
<td>JSP File</td>
</tr>
<tr>
<td>stop.jsp</td>
<td>JSP File</td>
</tr>
<tr>
<td>start.jsp</td>
<td>JSP File</td>
</tr>
<tr>
<td>soap poles.jar</td>
<td>Executable Jar File</td>
</tr>
<tr>
<td>soapadmin.gif</td>
<td>GIF Image</td>
</tr>
<tr>
<td>soap.xml</td>
<td>XML Document</td>
</tr>
<tr>
<td>showdetails.jsp</td>
<td>JSP File</td>
</tr>
<tr>
<td>Result.jsp</td>
<td>JSP File</td>
</tr>
<tr>
<td>Method.jsp</td>
<td>JSP File</td>
</tr>
<tr>
<td>Master.css</td>
<td>Cascading Style Sheet</td>
</tr>
<tr>
<td>Manifest.mf</td>
<td>MF File</td>
</tr>
<tr>
<td>list.jsp</td>
<td>JSP File</td>
</tr>
<tr>
<td>input.jsp</td>
<td>JSP File</td>
</tr>
<tr>
<td>index.html</td>
<td>HTML Document</td>
</tr>
<tr>
<td>ibm-web-ext.xml</td>
<td>XML File</td>
</tr>
<tr>
<td>ibm-web-bed.xml</td>
<td>XML File</td>
</tr>
<tr>
<td>header.html</td>
<td>HTML Document</td>
</tr>
<tr>
<td>dials.xml</td>
<td>XML Document</td>
</tr>
<tr>
<td>blankpage.html</td>
<td>HTML Document</td>
</tr>
</tbody>
</table>
Section 9. Summary

Summary

In this tutorial you developed a Web service using the Application Developer configuration of WebSphere Studio. You built a simple work order manager as a Java class and then proceeded to wrap it as a Web service. You used the Web services wizard within Application Developer to build the Web service, setting all the properties and letting Application Developer generate the code for you. While this was not the focus of this tutorial, you should have a clear idea of not only how to use the Application Developer wizard, but also what happens behind the scenes. You saw what WSDL definitions were created by Application Developer for the deployment phase, and also learned about the proxy that helps client applications invoke your service. Finally, you packaged your application and Web service in an enterprise archive so that you can deploy it in Part 2 on WebSphere Application Server.

In "Building Web Services with WebSphere Studio, Part 2: Deploy and Publish" you will continue with the work management scenario. You'll deploy the EAR on the WebSphere Application Server and publish the service on a UDDI registry so that potential users of the Web service can easily find it and learn how to invoke the service.

Resources

- See "Building Web Services with WebSphere Studio, Part 2: Deploy and Publish" to continue with this project.
- Download the WebSphere Studio Application Developer trial edition. (313MB)
- WebSphere Application Server Advanced Developer Edition version 4.0. (146MB)
- Get the source code for the work order management scenario.
- The Web Services Toolkit (WSTK) is a software development kit that includes a run-time environment, a demo, and examples to aid in building and using Web services.
- Check out these developerWorks tutorials:
  - Building Java HTTP servlets (developerWorks, September 2000).
  - Intro to JavaServer Pages technology (developerWorks, August 2001).
  - Registering and publishing your Web service (developerWorks, June 2001). This tutorial describes UDDI and the IBM UDDI4J toolkit allowing you to access UDDI registries from SOAP clients.
  - Implementing Web services with the WSTK 3.0.1 (developerWorks, January 2002). This tutorial shows you how to use the set of technologies packaged within the Web Services Toolkit to implement Web services.
- Take a look at these articles on the WebSphere Developer Domain:
  - "Web Services Development and Deployment with IBM Tools and Technologies - Part 1"
- "Web Services Programming with WebSphere Studio Application Developer -- Part 1: Web Services Discovery and Evaluation"
- Check out the Web services resources on the WebSphere Developer Domain.
- The JavaServer Pages home page includes the latest specifications as well as pointers to custom tag libraries and other advanced JSP technologies.
- The Xerces project at Apache.org provides XML parsers for a variety of languages such as Java, C++ and Perl. Follow the link for the Java 2 libraries.
- The SOAP resource center contains articles, examples, FAQs, mailing lists, specifications, tutorials, and other material pertaining to SOAP programming.
- The OASIS WSDL resource page contains many useful links related to WSDL including the WSDL specification version 1.1.
- The OASIS UDDI resource page contains many useful links related to UDDI including the UDDI technical specification and various UDDI repositories.
- Download the Web Services Toolkit for dynamic e-business, a software development kit that includes a run-time environment, a demo, and examples to aid in designing and executing Web service applications.
- Test-drive Web services with these Web Services demos.

Feedback

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.