Develop a web service without an IDE, Part 1: Focus on the Server: Create a web service provider on the command line

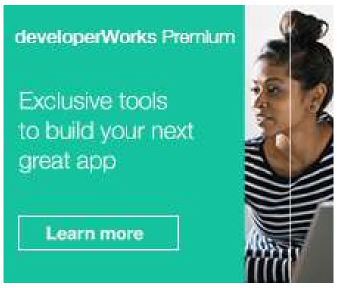
Learn how to create a web service provider, including the deployment descriptors and the Java classes, and see a demonstration of the Java compiler, Java2WSDL, and WSDL2Java command-line tools.

Share:

Alec Go is a Software Engineer for IBM WebSphere Application Server Level 2 Support. He has a Bachelor's degree in Computer Engineering from the Pennsylvania State University and graduated with honors.

09 December 2005
Also available in

Introduction



Although an Integrated Development Environment (IDE) like WebSphere® Studio Application Developer or Rational® Application Developer can help create web services, sometimes customers do not have access to such tools. This article describes how to create a Web service provider using only the command-line tools, and includes several Java classes and deployment descriptors. You will learn about the essential files needed to create a very simple Web service. In the process of developing the Web service provider, you'll use command-line tools that come with WebSphere Application Server such as the Java compiler, WSDL2Java, and Java2WSDL. Part 2 in this series will describe how to create a client that accesses this Web service.

The basic steps to create a Web service provider using only the command-line tools are:

1. Create a service endpoint interface (SEI).
2. Generate a Web Services Description Language (WSDL) file from the SEI.
3. Generate the Web service skeleton from the WSDL file.
4. Package a Web Archive (WAR) file.
5. Deploy the Web service.

This article assumes the following prerequisites:

1. You're using Microsoft® Windows®.
2. You have a licensed or trial version of WebSphere Application Server 5.x or 6.x (please see the Resources section for information on how to obtain WebSphere Application Server 5.x or 6.x).

Some alternatives to starting from a SEI are:

1. A Stateless Session Bean EJB remote interface
2. A JavaBean
3. A WSDL file

Creating a Web service from either an EJB remote interfaces, or from Java beans is similar to the process described in this article. More information about this can be found in the WebSphere InfoCenter in a document titled, Developing Web services based on Web services for J2EE.

If, instead, you are creating a Web service from a WSDL file, you can start at the section Generate the server-side artifacts and service endpoint implementation from the WSDL.

Create the SEI

In this article, the Web service is generated from a SEI.

A SEI is a description of what a Web service function should look like. Programming interfaces are often used to describe remote functions in distributed computing. For example, Enterprise JavaBeans have the concept of component interfaces, Common Object Request Broker Architecture (CORBA®) has the concept of Interface Definition Language (IDL) files, and Java RMI has the concept of Java interfaces that extend java.rmi.Remote. J2EE Web Services follow a similar pattern by requiring a SEI.

For the "Hello World" Web service, the SEI will look like the following:

```
package mypackage;

public interface HelloWorld extends java.rmi.Remote {
        public java.lang.String sayHello() throws java.rmi.RemoteException;
}
```

This is a very simple interface that has a function called sayHello. This function returns a java.lang.String object. For an interface to be a valid SEI, the interface must extend java.rmi.Remote and each method should throw java.rmi.RemoteException.

Perform the following steps:

1. Create a starting directory c:\temp\Server. Inside that, create a directory called mypackage. This directory will hold all the relevant class files for the Web service.
2. Inside c:\temp\Server\mypackage, create a file called HelloWorld.java. Copy and paste the following code into this file:
   ```
   package mypackage;

   public interface HelloWorld extends java.rmi.Remote {
           public java.lang.String sayHello() throws java.rmi.RemoteException;
   }
   ```
3. The WebSphere command line tools work only on class files and not source files. In order to continue, you must compile the Java file to a class file. Open a new command-line window. Be sure to use this command-line window for the rest of the tutorial. In it, issue the following commands:
   ```
   SET WAS_HOME=C:\Program Files\WebSphere\AppServer5.1

   call "%WAS_HOME%\bin\setupcmdline.bat"

   "%JAVA_HOME%\bin\javac" -extdirs "%WAS_CLASSPATH%;%WAS_EXT_DIRS%;." mypackage\*.java
   ```

   The first line sets the WAS_HOME environment variable to where you have WebSphere Application Server installed. The second line calls a script that will set up additional environment variables, like JAVA_HOME. The final line calls the Java compiler to compile the SEI that you created.
4. After running these commands, a new file called HelloWorld.class should have been created in c:\temp\Server\mypackage. Double check that the class file was created.

Generate the WSDL file from the SEI

A WSDL file describes the Web service. It defines the interface between the Web service and the Web service client. Although an SEI also describes the Web service, the description is in a Java-specific manner that is only readable to Java programs. The WSDL file describes the Web service in a language-independent manner. A .NET developer or a PHP developer can take the WSDL file to develop their respective clients. To generate the Web service, you need to first have a WSDL file. A WSDL can be generated from the SEI by using the Java2WSDL command, as follows (this command should be on one line):

```
call "%WAS_HOME%\bin\Java2WSDL" -style document -use literal -verbose -location
http://localhost:9080/HelloWorldWAR/services/HelloWorld mypackage.HelloWorld
```

Here are some details on the options:

Style - this can either be "document" or "rpc." This relates to the formatting of the SOAP message. Generally speaking, document is more interoperable than rpc, so choose document here. Rpc style can often cause trouble with arrays.

Literal - this can be either "literal" or "encoded." This relates to how the server and client interpret the information from the SOAP message. Using encoded will almost always cause interoperability problems, so choose literal.

Verbose - messages will be printed to the console for every step that is taken.

Location - this specifies the endpoint of the Web service. The endpoint is where the Web service client will be sending its information.

Class - the last command-line argument is the class name.

After running the Java2WSDL command, the following should appear in the console:

```
WSWS3429I: Binding-specific properties are {MIMEStyle=WSDL11, use=literal, debug=false,
    style=document,
    bindingName=HelloWorld, encodingStyle=http://schemas.xmlsoap.org/soap/encoding/,
    verbose=true, wrapped=true, portTypeName=HelloWorld, servicePortName=HelloWorld,
    intfNS=http://mypackage, location=
    http://localhost:9080/HelloWorldWAR/services/HelloWorld,
    soapAction=DEFAULT}
WSWS3010I: Info: Generating portType {http://mypackage}HelloWorld
WSWS3010I: Info: Generating message {http://mypackage}sayHelloRequest
WSWS3010I: Info: Generating message {http://mypackage}sayHelloResponse
WSWS3010I: Info: Generating binding {http://mypackage}HelloWorldSoapBinding
WSWS3010I: Info: Generating service {http://mypackage}HelloWorldService
WSWS3010I: Info: Generating port HelloWorld
```

Note that the only file ever generated from Java2WSDL is the WSDL file.

Generate the server-side artifacts and SEI from the WSDL

The WSDL file is crucial to generating the Web service. A J2EE Web service is basically a collection of class files and deployment descriptors. Use the WSDL2Java command (which should be on one line) to create these artifacts:

```
call "%WAS_HOME%\bin\WSDL2Java" -genJava overwrite -genXML overwrite
-role server -container web -verbose -output . HelloWorld.wsdl
```

The following should be displayed in the console:

```
WSWS3185I: Info: Parsing XML file:  HelloWorld.wsdl
WSWS3282I: Info: Generating .\mypackage\HelloWorld.java.
WSWS3282I: Info: Generating .\mypackage\HelloWorldSoapBindingImpl.java.
WSWS3282I: Info: Generating .\WEB-INF\webservices.xml.
WSWS3282I: Info: Generating .\WEB-INF\ibm-webservices-bnd.xmi.
WSWS3282I: Info: Generating .\WEB-INF\ibm-webservices-ext.xmi.
WSWS3282I: Info: Generating .\WEB-INF\HelloWorld_mapping.xml.
```

Note the files generated. The actual implementation class is HelloWorldSoapBindingImpl.java, which contains the following:

```
package mypackage;

public class HelloWorldSoapBindingImpl implements mypackage.HelloWorld{
    public java.lang.String sayHello() throws java.rmi.RemoteException {
        return null;
    }
}
```

The sayHello function can actually be called from a remote client. Compare this again with the SEI (HelloWorld):

```
package mypackage;

public interface HelloWorld extends java.rmi.Remote {
    public java.lang.String sayHello() throws java.rmi.RemoteException;
}
```

HelloWorldSoapBindingImpl is the actual endpoint implementation, and HelloWorld is the interface. HelloWorldSoapBindingImpl implements HelloWorld.

WSDL2Java generated HelloWorldSoapBindingImpl, which is basically a skeleton of the implementation. Currently it does nothing -- sayHello just returns null. The developer should fill in the code for the implementation. For this example, change the line "return null" to instead return a string. Change the HelloWorldSoapBindingImpl.java to look like the following:

```
package mypackage;

public class HelloWorldSoapBindingImpl implements mypackage.HelloWorld{
    public java.lang.String () throws java.rmi.RemoteException {
        return "Hello World";
    }
}
```

Now that the implementation is finished, all the Java classes should be compiled. Issue the following command:

```
"%JAVA_HOME%\bin\javac" -extdirs "%WAS_CLASSPATH%;%WAS_EXT_DIRS%;." mypackage\*.java
```

Package the WAR file

Web services are part of the J2EE Specification. The services try to reuse as much of the current specifications as possible. One way this can be done is to be packaged into a Web Archive (WAR) file, just like servlets. In a WAR file, the WEB-INF directory is used to store files that are not directly accessible to the public. For a Web service WAR file, the general structure looks like the following:

1. WEB-INF\classes\ - holds compiled classes (for example, Web service implementation and interface classes)
2. WEB-INF\wsdl\ - holds the Web services WSDL file
3. WEB-INF\web.xml - WAR file deployment descriptor
4. WEB-INF\webservices.xml - Web services deployment descriptor

Put together these elements to create the Web services WAR manually.

1. First, create a directory called classesin WEB-INF. Copy the mypackage directory into the classes directory. You should have the following directory structure:
   ```
   WEB-INF\classes\mypackage\HelloWorld.class
   WEB-INF\classes\mypackage\HelloWorld.java
   WEB-INF\classes\mypackage\HelloWorldSoapBindingImpl.class
   WEB-INF\classes\mypackage\HelloWorldSoapBindingImpl.java
   ```
2. Next, a wsdl directory should have been created by the Java2WSDL command. Copy the WSDL file into the WEB-INF\wsdl\ directory. This is the required location of the WSDL file.
3. Create the web.xml file inside the WEB-INF folder:
   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <!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 id="WebApp">
           <display-name>HelloWorldWAR</display-name>
           <servlet>
                   <servlet-name>mypackage_HelloWorldSoapBindingImpl</servlet-name>
                   <servlet-class>mypackage.HelloWorldSoapBindingImpl</servlet-class>
                   <load-on-startup>1</load-on-startup>
           </servlet>
   </web-app>
   ```

   Please note: Lines 2 and 3 above should be on the same line. For the next two deployment descriptor examples, please also put lines 2 and 3 on the same line. The entries within the servlet tag are important. This element describes the Web service endpoint implementation. The servlet-name element (mypackage_HelloWorldSoapBindingImpl) acts as an identifier for the Web service implementation. The "servlet-class" element (mypackage.HelloWorldSoapBindingImpl) identifies the class file in the WEB-INF\classes directory that is being used as the implementation. Alter the WEB-INF\webservices.xml. Initially, the file will look like the following:
   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <!DOCTYPE webservices PUBLIC "-//IBM Corporation, Inc.//DTD J2EE Web services 1.0//EN"
       "http://www.ibm.com/webservices/dtd/j2ee_web_services_1_0.dtd">
   <webservices>
     <webservice-description>
       <webservice-description-name>HelloWorldService</webservice-description-name>
       <wsdl-file>WEB-INF/wsdl/HelloWorld.wsdl</wsdl-file>
       <jaxrpc-mapping-file>WEB-INF/HelloWorld_mapping.xml</jaxrpc-mapping-file>
       <port-component>
         <port-component-name>HelloWorld</port-component-name>
         <wsdl-port>
           <namespaceURI>http://mypackage</namespaceURI>
           <localpart>HelloWorld</localpart>
         </wsdl-port>
         <service-endpoint-interface>mypackage.HelloWorld</service-endpoint-interface>
         <service-impl-bean>
           <servlet-link>??SET THIS TO servlet-name ELEMENT OF web.xml??</servlet-link>
         </service-impl-bean>
       </port-component>
   ```

```
  </webservice-description>
</webservices>
```

4. The element of concern is servlet-link. Change this value to mypackage_HelloWorldSoapBindingImpl. This links the Web service to an entry in the web.xml file. The webservices.xml file should now look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE webservices PUBLIC "-//IBM Corporation, Inc.//DTD J2EE Web services 1.0//EN"
    "http://www.ibm.com/webservices/dtd/j2ee_web_services_1_0.dtd">
<webservices>
  <webservice-description>
    <webservice-description-name>HelloWorldService</webservice-description-name>
    <wsdl-file>WEB-INF/wsdl/HelloWorld.wsdl</wsdl-file>
    <jaxrpc-mapping-file>WEB-INF/HelloWorld_mapping.xml</jaxrpc-mapping-file>
    <port-component>
      <port-component-name>HelloWorld</port-component-name>
      <wsdl-port>
        <namespaceURI>http://mypackage</namespaceURI>
        <localpart>HelloWorld</localpart>
      </wsdl-port>
      <service-endpoint-interface>mypackage.HelloWorld</service-endpoint-interface>
      <service-impl-bean>
        <servlet-link>mypackage_HelloWorldSoapBindingImpl</servlet-link>
      </service-impl-bean>
    </port-component>
  </webservice-description>
</webservices>
```

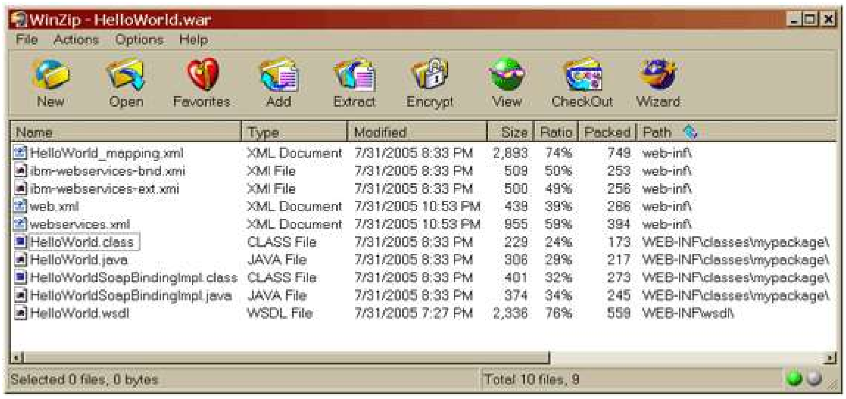
5. Create the WAR file. Use your favorite zip program to create a WAR file from the WEB-INF directory. If you are using WinZip, you can follow these steps:

   Right click on the **WEB-INF** directory.

   Click on **WinZip** --> Add to **zip**

   Call the file HelloWorld.war. The WAR file should look like the following:

Figure 1. View from WinZip



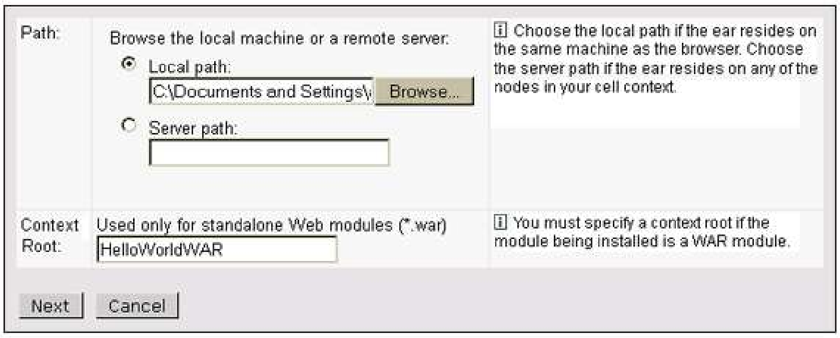
Name	Type	Modified	Size	Ratio	Packed	Path
HelloWorld_mapping.xml	XML Document	7/31/2005 8:33 PM	2,893	74%	749	web-inf\
ibm-webservices-bnd.xmi	XMI File	7/31/2005 8:33 PM	509	50%	253	web-inf\
ibm-webservices-ext.xmi	XMI File	7/31/2005 8:33 PM	500	49%	256	web-inf\
web.xml	XML Document	7/31/2005 10:53 PM	439	39%	266	web-inf\
webservices.xml	XML Document	7/31/2005 10:53 PM	955	59%	394	web-inf\
HelloWorld.class	CLASS File	7/31/2005 8:33 PM	229	24%	173	WEB-INF\classes\mypackage\
HelloWorld.java	JAVA File	7/31/2005 8:33 PM	306	29%	217	WEB-INF\classes\mypackage\
HelloWorldSoapBindingImpl.class	CLASS File	7/31/2005 8:33 PM	401	32%	273	WEB-INF\classes\mypackage\
HelloWorldSoapBindingImpl.java	JAVA File	7/31/2005 8:33 PM	374	34%	245	WEB-INF\classes\mypackage\
HelloWorld.wsdl	WSDL File	7/31/2005 7:27 PM	2,336	76%	559	WEB-INF\wsdl\

Selected 0 files, 0 bytes
Total 10 files, 9

Deploy the Web service

Install the new WAR file in the WebSphere Application Server Administration Console. Go to **Applications --> Enterprise Applications --> Install.** For the Local Path specify the location of HelloWorld.war, and for the Context Root specify HelloWorldWAR."

Figure 2. View from the Administration Console



Path: Browse the local machine or a remote server:
(●) Local path: C:\Documents and Settings\ [Browse...]
( ) Server path:
Choose the local path if the ear resides on the same machine as the browser. Choose the server path if the ear resides on any of the nodes in your cell context.

Context Root: Used only for standalone Web modules (*.war)
HelloWorldWAR
You must specify a context root if the module being installed is a WAR module.

Note that when the WSDL file was first created, the location parameter was set to "http://localhost:9080/HelloWorldWAR/services/HelloWorld." Therefore, the Context Root must be

"HelloWorldWAR" so that the WSDL file can be used to generate clients, without manual alteration.

Click **Next** and continue clicking **Next** through all the defaults. Finish installing the WAR file. Remember to save all the changes, and to manually start the HelloWorld application.

Congratulations! You have successfully created a Web service provider!

Conclusion

Creating a Web service without an Integrated Development Environment like Application Developer or Rational Application Developer can be daunting. However, it is essential to learn what is actually occurring underneath the IDE Wizards. In Part 2 of this series, you will create a Web service client to invoke this Web service.

Download

Description	Name	Size
Hello World WAR file - the finished product	ws-noide1code2006-01-15.war	5KB

Resources

Learn

"Develop, test, and deploy Web services using Rational Application Developer V6.0" (developerWorks, January 2005) offers a tutorial on using Rational Application Developer to create a Web service provider and client.

J2EE Web Services is a comprehensive guide to developing and deploying Web services using J2EE technology.

W3Schools offers Web-building tutorials from basic HTML and XHTML to advanced XML, Multimedia and WAP.

WebSphere Version 6 Web Services Handbook Development and Deployment is an IBM Redbook that discusses Web Services.

Documentation on WSDL2Java can be found on the InfoCenter.

Java2WSDL documentation is also located on the InfoCenter.

The developerWorks Technical Library offers more developerWorks articles on Web services.

Get products and technologies

WebSphere Application Server V6.0 Trial Version -- Download the trial version in order to follow the steps in this article.

Discuss

Participate in the discussion forum.