Java™ 6 Platform Revealed

John Zukowski

Apress®

Java™ 6 Platform Revealed

Copyright © 2006 by John Zukowski

ISBN-13 (pbk): 978-1-59059-660-9

ISBN-10 (pbk): 1-59059-660-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 7

Web Services

Who doesn't use web services these days? Due to the increasing popularity of web services, the Java APIs for taking advantage of the functionality are moving from the latest Java EE release into the Java SE 6 platform. In other words, there are no add-on kits for web services, and both platforms have the same API. Mustang adds a handful of different web services–related APIs to the standard tool chest: Web Services Metadata for the Java Platform with JSR 181, the Java API for XML-Based Web Services (JAX-WS) 2.0 via JSR 224, and the SOAP with Attachments API for Java (SAAJ) 1.3 as part of JSR 67.

Before continuing with the chapter, it is necessary to point out one very important point: this is not a book about web services. I've seen 1,000-plus-page books on web services that still require you to understand some level of XML, SOAP, or some other Java API to take full advantage of the described capabilities. In this chapter, I'll do my best to show examples of using the new APIs in the context of a Java SE program. If you need more information about creating web services, consider getting one of Apress's other titles or some of the many online tutorials on the topic. You will need to "convert" the online tutorial to Mustang, but the web services APIs are pretty much the same, just in a new environment: Java SE, instead of Java EE.

The packages associated with the three web services APIs are new to Java SE, so no need for tables showing the differences between Java 5 and 6. The JAX-WS API is found in the `javax.xml.ws` packages, the SAAJ classes are in `javax.xml.soap`, and the Web Services Metadata classes are found under `javax.jws`.

The javax.jws Package

JSR 181 and its specification of Web Services Metadata for the Java Platform provide a mechanism to utilize annotations in classes to design and develop web services. For those unfamiliar with annotations, they were introduced with J2SE 5.0 and have been expanded somewhat with Java 6. They are described more fully in Chapter 10; but they essentially allow you to add `@tags` to classes, methods, and properties to describe associated metadata. A parser can then locate the tags and act appropriately; though when that action happens is dependent upon the tag itself.

The two packages involved here are `javax.jws` and `javax.jws.soap`. Both packages only define enumerations and annotations. There are neither classes nor interfaces here. By importing the appropriate package for the annotations, you can annotate the classes that represent web services, and their methods, as shown in Listing 7-1. Be sure to include a `package` statement. If you don't, when you run the `wsgen` tool later, you'll get an error message, as follows:

```
modeler error: @javax.jws.Webservice annotated classes that do not belong to a
    package must have the @javax.jws.Webservice.targetNamespace element.
    Class: HelloService
```

Listing 7-1. *An Annotated Hello World Service*

```
package net.zukowski.revealed;

import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class HelloService {

  @WebMethod
  public String helloWorld() {
    return "Hello, World";
  }
}
```

There are two basic annotations specified here: `@WebService` and `@WebMethod`. The `@WebService` annotation identifies the `HelloService` class as a web service. If not specified otherwise, the `@WebService` annotation assumes the name is that of the class. You can also specify a namespace, service name, WSDL location, and endpoint interface.

But what can you do with the source file? Running the `javac` compiler against the source just spits out a `.class` file—nothing else special. You still need to do that. But after compiling the class, you also need to run the `wsgen` command-line tool (`wsgen` is short for web service generator).

The `wsgen` tool generates a handful of source files and then compiles. Since the package name of the example here is `net.zukowski.revealed`, the new classes are generated into the `net/zukowski/revealed/jaxws` directory. Listings 7-2 and 7-3 show the source for the classes generated from running the following command:

```
> wsgen -cp . net.zukowski.revealed.HelloService
```

Listing 7-2. *The First Generated Class for the Web Service*

```
package net.zukowski.revealed.jaxws;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(name = "helloWorld", namespace = "http://revealed.zukowski.net/")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "helloWorld", namespace = "http://revealed.zukowski.net/")
public class HelloWorld {
}
```

Listing 7-3. *The Second Generated Class for the Web Service*

```
package net.zukowski.revealed.jaxws;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(name = "helloWorldResponse", ➥
    namespace = "http://revealed.zukowski.net/")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "helloWorldResponse", namespace = "http://revealed.zukowski.net/")
public class HelloWorldResponse {

    @XmlElement(name = "return", namespace = "")
    private String _return;

    /**
     *
     * @return
     *     returns String
     */
    public String get_return() {
        return this._return;
    }
    /**
     *
     * @param _return
     *     the value for the _return property
     */
    public void set_return(String _return) {
        this._return = _return;
    }
}
```

There is certainly much more you can do with annotations here. In fact, the annotations found in the `javax.jws.soap` package are where you get the SOAP bindings, which takes us to the next section.

The javax.xml.ws and javax.xml.soap Packages

The JAX-WS API is very closely associated with JAXB. Where JAXB is the Java-to-XML mapping, and vice versa, JAX-WS is the mapping of Java objects to and from the Web Services Description Language (WSDL). Together, with the help of the SAAJ, the trio gives you the API stack for web services.

What does all that mean? With the help of last chapter's JAXB, you can use the SAAJ and JAX-WS APIs to generate SOAP messages to connect to web services to get results. You're not going to deploy web services with Mustang. Instead, you're going to connect to preexisting services to get answers. The available APIs are what you get with Java EE 5.

SOAP Messages

In the last chapter, you saw an example creating a SOAP message. The `javax.xml.soap` package provides a `MessageFactory`, from which you get an instance to create the message.

```
SOAPMessage soapMessage = MessageFactory.newInstance().createMessage();
```

This generates a SOAP 1.2 message format. If you need to create a message for the SOAP 1.1 protocol, you can pass the protocol name to the `createMessage()` method.

```
SOAPMessage soapMessage =
  MessageFactory.newInstance().createMessage(SOAPConstants.SOAP_1_1_PROTOCOL);
```

The SOAP message in turn consists of a series of other objects: `SOAPPart`, `SOAPEnvelope`, `SOAPBody`, and `SOAPHeader`. All of these bits are in XML format. To include non-XML data within the message, you would use one or more `AttachmentPart` type objects. These are created from either an Activation Framework `DataHandler`, or directly from an `Object` and mime type. Think of e-mail attachments here. There is built-in support in SAAJ 1.3 for mime types of type `text/plain`, `multipart/*`, and `text/xml` or `application/xml`. For other mime types, that's where the `DataHandler` comes into play.

```
AttachmentPart attachment = soapMessage.createAttachmentPart();
attachment.setContent(textContent, "text/plain");
soapMessage.addAttachmentPart(attachment);
```

What you put in your SOAP message really depends upon the service you are connecting to. Just be sure to identify the destination in the `SOAPEnvelope`.

Table 7-1 includes a list of the key interfaces and classes found in the `javax.xml.soap` package.

Table 7-1. *Key SOAP Classes and Interfaces*

Name	Description
`AttachmentPart`	`SOAPMessage` attachment
`Detail`	`DetailEntry` container
`DetailEntry`	`SOAPFault` details
`MessageFactory`	`SOAPMessage` factory
`MimeHeader`	Mime type details
`MimeHeaders`	`MimeHeader` container
`Name`	XML name
`Node`	Element of XML document
`SAAJMetaFactory`	SAAJ API factory
`SAAJResult`	JAXP transformation or JAXB marshalling results holder
`SOAPBody`	SOAP body part of `SOAPMessage`
`SOAPBodyElement`	`SOAPBody` contents
`SOAPConnection`	Point-to-point connection for client
`SOAPConnectionFactory`	`SOAPConnection` factory
`SOAPConstants`	SOAP 1.1 protocol constants
`SOAPElement`	`SOAPMessage` element
`SOAPElementFactory`	`SOAPElement` factory
`SOAPEnvelope`	`SOAPMessage` header information
`SOAPException`	Standard exception for SOAP-related operations
`SOAPFactory`	`SOAPMessage` elements factory

Continued

Table 7-1. *Continued*

Name	Description
`SOAPFault`	`SOAPMessage` element for error status information
`SOAPFaultElement`	`SOAPFault` contents
`SOAPHeader`	`SOAPMessage` header element
`SOAPHeaderElement`	`SOAPHeader` contents
`SOAPMessage`	Base class for SOAP messages
`SOAPPart`	`SOAPMessage` element for SOAP-specific pieces
`Text`	Textual node contents

The contents of the `SOAPMessage` generated really depend upon the web service you are connecting to. Similar to how the example from Chapter 6 was built up, you just create the different elements and put the pieces together. As the javadoc for the package states, there are lots of things you can do with the `javax.xml.soap` package:

- Create a point-to-point connection to a specified endpoint
- Create a SOAP message
- Create an XML fragment
- Add content to the header of a SOAP message
- Add content to the body of a SOAP message
- Create attachment parts and add content to them
- Access/add/modify parts of a SOAP message
- Create/add/modify SOAP fault information
- Extract content from a SOAP message
- Send a SOAP request-response message

The JAX-WS API

The next identical packages shared with Java EE 5 are the `javax.xml.ws` package and subpackages, which include JAX-WS. Again, there are whole books written just about this API, so I'll just show off some highlights. Since the API is identical to that of Java EE 5, those books go into much more detail of the API than can be shown in a Mustang quick start–type book. The key difference now is that the JAX-WS libraries are standard with the Java SE platform, so no additional libraries are needed.

The key thing to understand when using the JAX-WS API with Mustang—outside the context of Java EE—is that you need to think in the context of a consumer of web services, not as a developer of them. For instance, take Google, which is a rather popular search engine. Google offers a set of web services to utilize its services from your programs. You can now use these programs directly in your program, provided you get a free key from them, which is limited to 1,000 usages a day.

Listing 7-4 provides the source for a simple web services client. Provided that you pass the file name for the SOAP request to the program, it will connect to the Google site to get results.

Listing 7-4. *Connecting to Google Web Services*

```
import java.io.*;
import java.net.*;
import javax.xml.ws.*;
import javax.xml.namespace.*;
import javax.xml.soap.*;
public class GoogleSearch {
  public static void main(String args[]) throws Exception {
   URL url = new URL("http://api.google.com/GoogleSearch.wsdl");
   QName serviceName = new QName("urn:GoogleSearch", "GoogleSearchService");
   QName portName = new QName("urn:GoogleSearch", "GoogleSearchPort");
   Service service = Service.create(url, serviceName);
   Dispatch<SOAPMessage> dispatch = service.createDispatch(portName,
   SOAPMessage.class, Service.Mode.MESSAGE);
   SOAPMessage request = MessageFactory.newInstance().createMessage(
     null, new FileInputStream(args[0]));
   SOAPMessage response = dispatch.invoke(request);
   response.writeTo(System.out);
  }
}
```

That's a typical web services client. You can either build up the SOAP message with the previously described `javax.xml.soap` package, or, as the program does, just place the XML for the SOAP request in a file (with *your* Google key) as part of the SOAP message. Then, you can query Google from your program.

```
> java GoogleSearch search.xml
```

■**Note** For more information on using Google's APIs, see `www.google.com/apis`.

To change this client to use another web service, you'll need to change the URL you connect to, as well as the qualified name, `QName`, for the service. And, of course, adjust the XML of the SOAP request accordingly. Not to belittle the JAX-WS API, but that is really all there is to using the API for a client, as opposed to creating the service itself.

■**Note** Information on web services available from Amazon can be found at `http://developer.amazonwebservices.com`, if you want another source to try out.

Summary

The web services support added to Mustang is meant purely for the client-side aspect of web services. There is no web server to deploy your services to. Through the three-pronged approach of JAXB, JAX-WS, and SAAJ, you can connect to any preexisting services for tasks that used to be done with RMI and CORBA, among many other preexisting remote procedure call (RPC)–like services. The APIs themselves aren't new to the Java platform—they're just new to the standard edition.

Chapter 8 moves beyond what you can think of as standard libraries into the APIs related to directly using the platform toolset. Working with JSR 199, you can now compile your Java programs from your Java programs to create new Java programs, or at least new classes. The API even allows you to compile straight from memory, without files. Turn the page to learn about the `javax.tools` API.