Contents

WSDL

WSDL is a specification defining how to describe web services in a common XML grammar. WSDL describes four critical pieces of data:

- Interface information describing all publicly available functions
- Data type information for all message requests and message responses
- Binding information about the transport protocol to be used
- Address information for locating the specified service

In a nutshell, WSDL represents a contract between the service requestor and the service provider, in much the same way that a Java interface represents a contract between client code and the actual Java object. The crucial difference is that WSDL is platform- and language-independent and is used primarily (although not exclusively) to describe SOAP services.

Using WSDL, a client can locate a web service and invoke any of its publicly available functions. With WSDL-aware tools, you can also automate this process, enabling applications to easily integrate new services with little or no manual code. WSDL therefore represents a cornerstone of the web service architecture, because it provides a common language for describing services and a platform for automatically integrating those services.

This chapter covers all aspects of WSDL, including the following topics:

- An overview of the WSDL specification, complete with detailed explanations of the major WSDL elements
- Two basic WSDL examples to get you started
- A brief survey of WSDL invocation tools, including the IBM Web Services Invocation Framework (WSIF), SOAP::Lite, and The Mind Electric's GLUE platform
- A discussion of how to automatically generate WSDL files from existing SOAP services
- An overview of using XML Schema types within WSDL, including the use of arrays and complex types

The WSDL Specification

WSDL is an XML grammar for describing web services. The specification itself is divided into six major elements:

`definitions`
: The `definitions` element must be the root element of all WSDL documents. It defines the name of the web service, declares multiple namespaces used throughout the remainder of the document, and contains all the service elements described here.

`types`
: The `types` element describes all the data types used between the client and server. WSDL is not tied exclusively to a specific typing system, but it uses

the W3C XML Schema specification as its default choice. If the service uses only XML Schema built-in simple types, such as strings and integers, the `types` element is not required. A full discussion of the `types` element and XML Schema is deferred to the end of the chapter.

`message`
: The `message` element describes a one-way message, whether it is a single message request or a single message response. It defines the name of the message and contains zero or more message `part` elements, which can refer to message parameters or message return values.

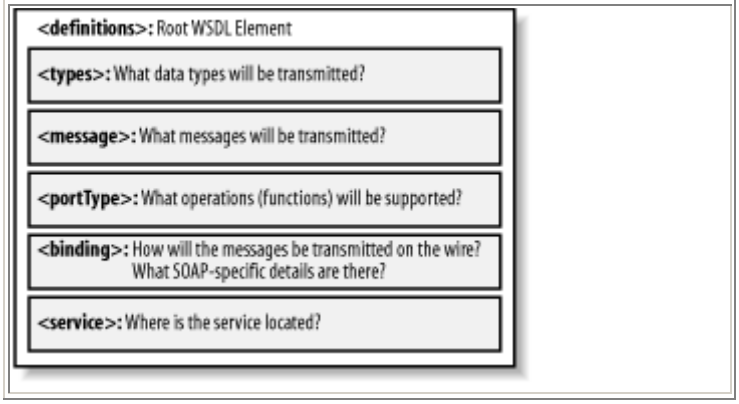
`portType`
: The `portType` element combines multiple `message` elements to form a complete one-way or round-trip operation. For example, a `portType` can combine one request and one response message into a single request/response operation, most commonly used in SOAP services. Note that a `portType` can (and frequently does) define multiple operations.

`binding`
: The `binding` element describes the concrete specifics of how the service will be implemented on the wire. WSDL includes built-in extensions for defining SOAP services, and SOAP-specific information therefore goes here.

`service`
: The `service` element defines the address for invoking the specified service. Most commonly, this includes a URL for invoking the SOAP service.

To help you keep the meaning of each element clear, Figure 6-1 offers a concise representation of the WSDL specification. As you continue reading the remainder of the chapter, you may wish to refer back to this diagram.

Figure 6-1. The WSDL specification in a nutshell



In addition to the six major elements, the WSDL specification also defines the following utility elements:

`documentation`
: The `documentation` element is used to provide human-readable documentation and can be included inside any other WSDL element.

`import`
: The `import` element is used to import other WSDL documents or XML Schemas. This enables more modular WSDL documents. For example, two WSDL documents can import the same basic elements and yet include their

own `service` elements to make the same service available at two physical addresses. Note, however, that not all WSDL tools support the import functionality as of yet.

TIP: WSDL is not an official recommendation of the W3C and, as such, has no official status within the W3C. WSDL Version 1.1 was submitted to the W3C in March 2001. Original submitters included IBM, Microsoft, Ariba, and a half dozen other companies. Most probably, WSDL will be placed under the consideration of the new W3C Web Services Activity's Web Services Description Working Group, which will decide if the specification advances to an official recommendation status. The WSDL Version 1.1 specification is available online at http://www.w3.org/TR/wsdl.

Basic WSDL Example: HelloService.wsdl

To make the previously described WSDL concepts as concrete as possible, let's examine our first sample WSDL file.

Example 6-1 provides a sample *HelloService.wsdl* document. The document describes the HelloService from Chapter 4.

As you may recall, the service provides a single publicly available function, called *sayHello.* The function expects a single string parameter, and returns a single string greeting. For example, if you pass the parameter `world`, the service returns the greeting, "Hello, world!"

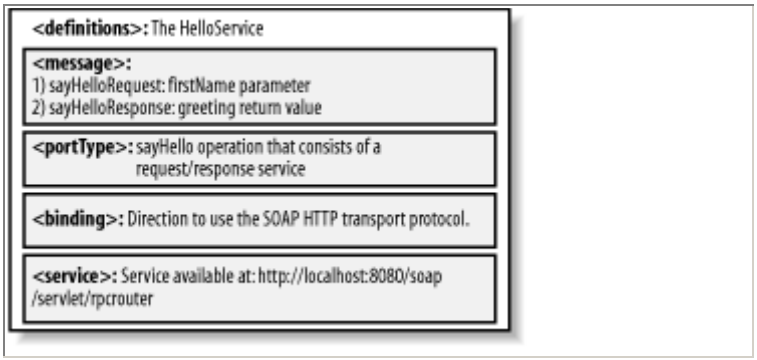
Example 6-1: HelloService.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
   targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <message name="SayHelloRequest">
      <part name="firstName" type="xsd:string"/>
   </message>
   <message name="SayHelloResponse">
      <part name="greeting" type="xsd:string"/>
   </message>

   <portType name="Hello_PortType">
      <operation name="sayHello">
         <input message="tns:SayHelloRequest"/>
         <output message="tns:SayHelloResponse"/>
      </operation>
   </portType>

   <binding name="Hello_Binding" type="tns:Hello_PortType">
      <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
```

```
      <operation name="sayHello">
         <soap:operation soapAction="sayHello"/>
         <input>
            <soap:body

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="urn:examples:helloservice"
                use="encoded"/>
         </input>
         <output>
            <soap:body

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="urn:examples:helloservice"
                use="encoded"/>
         </output>
      </operation>
   </binding>

   <service name="Hello_Service">
      <documentation>WSDL File for HelloService</documentation>
      <port binding="tns:Hello_Binding" name="Hello_Port">
         <soap:address
            location="http://localhost:8080/soap/servlet/rpcrouter"/>
      </port>
   </service>
</definitions>
```

The WSDL elements are discussed in the next section of this chapter. As you examine each element in detail, you may want to refer to Figure 6-2, which summarizes the most important aspects of Example 6-1.

Figure 6-2. A bird's-eye view of HelloService.wsdl



definitions

The `definitions` element specifies that this document is the *HelloService*. It also specifies numerous namespaces that will be used throughout the remainder of the document:

```
<definitions name="HelloService"
  targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
```

```
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

The use of namespaces is important for differentiating elements, and it enables the document to reference multiple external specifications, including the WSDL specification, the SOAP specification, and the XML Schema specification.

The `definitions` element also specifies a `targetNamespace` attribute. The `targetNamespace` is a convention of XML Schema that enables the WSDL document to refer to itself. In Example 6-1, we specified a `targetNamespace` of *http://www.ecerami.com/wsdl/HelloService.wsdl*. Note, however, that the namespace specification does not require that the document actually exist at this location; the important point is that you specify a value that is unique, different from all other namespaces that are defined.

Finally, the `definitions` element specifies a default namespace: *xmlns=http://schemas.xmlsoap.org/wsdl/*. All elements without a namespace prefix, such as `message` or `portType`, are therefore assumed to be part of the default WSDL namespace.

message

Two `message` elements are defined. The first represents a request message, *SayHelloRequest,* and the second represents a response message, *SayHelloResponse*:

```
<message name="SayHelloRequest">
  <part name="firstName" type="xsd:string"/>
</message>
<message name="SayHelloResponse">
  <part name="greeting" type="xsd:string"/>
</message>
```

Each of these messages contains a single `part` element. For the request, the part specifies the function parameters; in this case, we specify a single `firstName` parameter. For the response, the part specifies the function return values; in this case, we specify a single `greeting` return value.

The `part` element's `type` attribute specifies an XML Schema data type. The value of the `type` attribute must be specified as an XML Schema QName--this means that the *value* of the attribute must be namespace-qualified. For example, the `firstName` `type` attribute is set to `xsd:string`; the `xsd` prefix references the namespace for XML Schema, defined earlier within the `definitions` element.

If the function expects multiple arguments or returns multiple values, you can specify multiple `part` elements.

portType

The `portType` element defines a single operation, called *sayHello*. The operation itself consists of a single `input` message (*SayHelloRequest*) and a single `output` message (*SayHelloResponse*):

```
<portType name="Hello_PortType">
  <operation name="sayHello">
    <input message="tns:SayHelloRequest"/>
    <output message="tns:SayHelloResponse"/>
  </operation>
</portType>
```

Much like the `type` attribute defined earlier, the `message` attribute must be specified as an XML Schema QName. This means that the value of the attribute must be namespace-qualified. For example, the `input` element specifies a `message` attribute of t`ns:SayHelloRequest`; the `tns` prefix references the `targetNamespace` defined earlier within the `definitions` element.

WSDL supports four basic patterns of operation:

One-way
: The service receives a message. The operation therefore has a single `input` element.

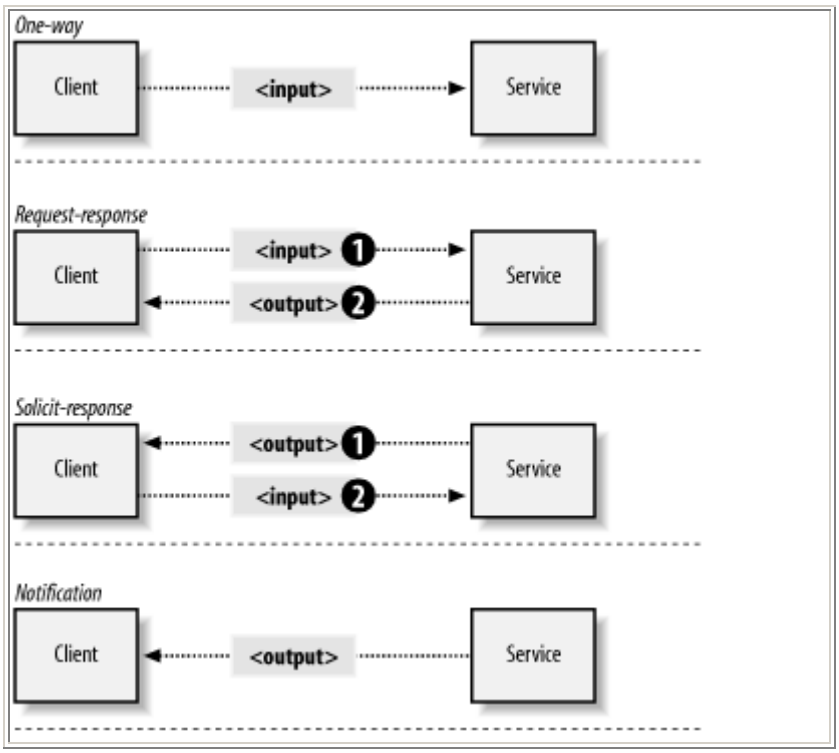
Request-response
: The service receives a message and sends a response. The operation therefore has one `input` element, followed by one `output` element (illustrated previously in Example 6-1). To encapsulate errors, an optional `fault` element can also be specified.

Solicit-response
: The service sends a message and receives a response. The operation therefore has one `output` element, followed by one `input` element. To encapsulate errors, an optional `fault` element can also be specified.

Notification
: The service sends a message. The operation therefore has a single `output` element.

These patterns of operation are also shown in Figure 6-3. The request-response pattern is most commonly used in SOAP services.

Figure 6-3. Operation patterns supported by WSDL 1.1



binding

The `binding` element provides specific details on how a `portType` operation will actually be transmitted over the wire. Bindings can be made available via multiple transports, including HTTP GET, HTTP POST, or SOAP. In fact, you can specify multiple bindings for a single `portType`.

The `binding` element itself specifies `name` and `type` attributes:

```
<binding name="Hello_Binding" type="tns:Hello_PortType">
```

The `type` attribute references the `portType` defined earlier in the document. In our case, the `binding` element therefore references `tns:Hello_PortType`, defined earlier in the document. The binding element is therefore saying, "I will provide specific details on how the *sayHello* operation will be transported over the Internet."

SOAP binding

WSDL 1.1 includes built-in extensions for SOAP 1.1. This enables you to specify SOAP-specific details, including SOAP headers, SOAP encoding styles, and the `SOAPAction` HTTP header. The SOAP extension elements include:

`soap:binding`
: This element indicates that the binding will be made available via SOAP. The `style` attribute indicates the overall style of the SOAP message format. A `style` value of `rpc` specifies an RPC format. This means that the body of the SOAP request will include a wrapper XML element indicating the function

name. Function parameters are then embedded inside the wrapper element. Likewise, the body of the SOAP response will include a wrapper XML element that mirrors the function request. Return values are then embedded inside the response wrapper element.

A `style` value of `document` specifies an XML document call format. This means that the request and response messages will consist simply of XML documents. The document style is flatter than the `rpc` style and does not require the use of wrapper elements. (See the upcoming note for additional details.)

The `transport` attribute indicates the transport of the SOAP messages. The value `http://schemas.xmlsoap.org/soap/http` indicates the SOAP HTTP transport, whereas `http://schemas.xmlsoap.org/soap/smtp` indicates the SOAP SMTP transport.

`soap:operation`

This element indicates the binding of a specific operation to a specific SOAP implementation. The `soapAction` attribute specifies that the `SOAPAction` HTTP header be used for identifying the service. (See Chapter 3 for details on the `SOAPAction` header.)

`soap:body`

This element enables you to specify the details of the input and output messages. In the case of HelloWorld, the `body` element specifies the SOAP encoding style and the namespace URN associated with the specified service.

TIP: The choice between the `rpc` style and the `document` style is controversial. The topic has been hotly debated on the WSDL newsgroup (http://groups.yahoo.com/group/wsdl). The debate is further complicated because not all WSDL-aware tools even differentiate between the two styles. Because the `rpc` style is more in line with the SOAP examples from previous chapters, I have chosen to stick with the `rpc` style for all the examples within this chapter. Note, however, that most Microsoft .NET WSDL files use the `document` style.

service

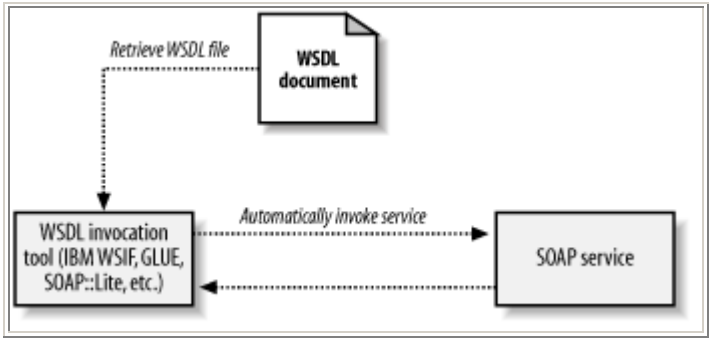
The `service` element specifies the location of the service. Because this is a SOAP service, we use the `soap:address` element, and specify the local host address for the Apache SOAP `rpcrouter` servlet:
`http://localhost:8080/soap/servlet/rpcrouter`.

Note that the service element includes a `documentation` element to provide human-readable documentation.

WSDL Invocation Tools, Part I

Given the WSDL file in Example 6-1, you could manually create a SOAP client to invoke the service. A better alternative is to *automatically* invoke the service via a WSDL invocation tool. (See Figure 6-4.)

Figure 6-4. WSDL invocation tools



Many WSDL invocation tools already exist. This section provides a brief overview of three invocation tools.

GLUE

The Mind Electric provides a complete web service platform called GLUE (available at http://www.themindelectric.com/). The platform itself provides extensive support for SOAP, WSDL, and UDDI. Some of its advanced functionality, including support for complex data types, will be explored later in this chapter.

For now, you can try out the GLUE `invoke` command-line tool. Here is the command-line usage:

```
usage: invoke URL method arg1 arg2 arg3...
```

For example, to invoke the HelloService, make sure that your Apache Tomcat server is running, and place the *HelloService.wsdl* file within a publicly available directory. Then, issue the following command:

```
invoke http://localhost:8080/wsdl/HelloService.wsdl sayHello World
```

Once invoked, GLUE will immediately download the specified WSDL file, invoke the *sayHello* method, and pass `World` as a parameter. GLUE will then automatically display the server response:

```
Output:  result = Hello, World!
```

That's all there is to it!

GLUE also supports an excellent logging facility that enables you to easily view all SOAP messages. To activate the logging facility, set the `electric.logging` system property. The easiest option is to modify the *invoke.bat* file. The original file looks like this:

```
call java electric.glue.tools.Invoke %1 %2 %3 %4 %5 %6 %7 %8 %9
```

Modify the file to include the logging property via the `-D` option to the Java interpreter:

```
call java -Delectric.logging="SOAP" electric.glue.tools.Invoke %1 %2 %3 %4
   %5 %6 %7 %8 %9
```

When you invoke the HelloService, GLUE now generates the following output:

```
LOG.SOAP: request to http://207.237.201.187:8080/soap/servlet/rpcrouter
<?xml version='1.0' encoding='UTF-8'?>
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/
  envelope/' xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
  <soap:Body>
    <n:sayHello xmlns:n='urn:examples:helloservice'>
      <firstName xsi:type='xsd:string'>World</firstName>
    </n:sayHello>
  </soap:Body>
</soap:Envelope>

LOG.SOAP: response from http://207.237.201.187:8080/soap/servlet/rpcrouter
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:xsi='http://www.w3.org/1999/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/1999/XMLSchema'>
  <SOAP-ENV:Body>
    <ns1:sayHelloResponse
      xmlns:ns1='urn:examples:helloservice'
      SOAP-ENV:encodingStyle=
        'http://schemas.xmlsoap.org/soap/encoding/'>
      <return xsi:type='xsd:string'>Hello, World!</return>
    </ns1:sayHelloResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

result = Hello, World!
```

To view additional HTTP information, just set `electric.logging` to `SOAP,HTTP`.

SOAP::Lite for Perl

SOAP::Lite for Perl, written by Paul Kulchenko, also provides limited support for WSDL. The package is available at http://www.soaplite.com.

Example 6-2 provides a complete Perl program for invoking the HelloService.

Example 6-2: Hello_Service.pl

```
use SOAP::Lite;

print "Connecting to Hello Service...\n";
print SOAP::Lite
   -> service('http://localhost:8080/wsdl/HelloService.wsdl')
   -> sayHello ('World');
```

The program generates the following output:

```
Connecting to Hello Service...
Hello, World!
```

IBM Web Services Invocation Framework (WSIF)

Finally, IBM has recently released WSIF. The package is available at http://www.alphaworks.ibm.com/tech/wsif.

Much like GLUE, WSIF provides a simple command-line option for automatically invoking WSDL services. For example, the following command:

```
java clients.DynamicInvoker http://localhost:8080/wsdl/HelloService.wsdl
   sayHello World
```

generates the following output:

```
Reading WSDL document from 'http://localhost:8080/wsdl/HelloService.wsdl'
Preparing WSIF dynamic invocation
Executing operation sayHello
Result:
greeting=Hello, World!

Done!
```

Basic WSDL Example: XMethods eBay Price Watcher Service

Before moving on to more complicated WSDL examples, let's examine another relatively simple one. Example 6-3 provides a WSDL file for the XMethods eBay Price Watcher Service. The service takes an existing eBay auction ID, and returns the value of the current bid.

Example 6-3: eBayWatcherService.wsdl (reprinted with permission of XMethods, Inc.)

```
<?xml version="1.0"?>
<definitions name="eBayWatcherService"
   targetNamespace=
      "http://www.xmethods.net/sd/eBayWatcherService.wsdl"
   xmlns:tns="http://www.xmethods.net/sd/eBayWatcherService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns="http://schemas.xmlsoap.org/wsdl/">

   <message name="getCurrentPriceRequest">
      <part name="auction_id" type = "xsd:string"/>
   </message>
   <message name="getCurrentPriceResponse">
      <part name="return" type = "xsd:float"/>
```

```
   </message>

   <portType name="eBayWatcherPortType">
      <operation name="getCurrentPrice">
         <input
            message="tns:getCurrentPriceRequest"
            name="getCurrentPrice"/>
         <output
            message="tns:getCurrentPriceResponse"
            name="getCurrentPriceResponse"/>
      </operation>
   </portType>

   <binding name="eBayWatcherBinding" type="tns:eBayWatcherPortType">
      <soap:binding
         style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="getCurrentPrice">
         <soap:operation soapAction=""/>
         <input name="getCurrentPrice">
            <soap:body
               use="encoded"
               namespace="urn:xmethods-EbayWatcher"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
         </input>
         <output name="getCurrentPriceResponse">
            <soap:body
               use="encoded"
               namespace="urn:xmethods-EbayWatcher"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
         </output>
      </operation>
   </binding>

   <service name="eBayWatcherService">
      <documentation>
         Checks current high bid for an eBay auction
      </documentation>
      <port name="eBayWatcherPort" binding="tns:eBayWatcherBinding">
         <soap:address

location="http://services.xmethods.net:80/soap/servlet/rpcrouter"/>
      </port>
   </service>
</definitions>
```

Here is an overview of the main WSDL elements:

`messages`
: Two messages are defined: `getCurrentPriceRequest` and `getCurrentPriceResponse`. The request message contains a single string parameter; the response message contains a single float parameter.

`portType`

A single operation, `getCurrentPrice`, is defined. Again, we see the request/response operation pattern.

`binding`
: The `binding` element specifies HTTP SOAP as the transport. The `soapAction` attribute is left as an empty string ("").

`service`
: This element specifies that the service is available at http://services.xmethods.net/soap/servlet/rpcrouter.

To access the eBay watcher service, you can use any of the WSDL invocation tools defined earlier. For example, the following call to GLUE:

```
invoke http://www.xmethods.net/sd/2001/EBayWatcherService.wsdl
   getCurrentPrice 1271062297
```

retrieves the current bid price for a Handspring Visor Deluxe:

```
result = 103.5
```

WSDL Invocation Tools, Part II

Our initial discussion of WSDL invocation tools focused on programming and command-line invocation tools. We now move on to even simpler tools that are entirely driven by a web-based interface.

The GLUE Console

In addition to supporting a number of command-line tools, the GLUE platform also supports a very intuitive web interface for deploying new services and connecting to existing services.

To start the GLUE console, just type:

```
console
```

This will automatically start the GLUE console on the default port 8100. Open a web browser and you will see the GLUE console home page. (See Figure 6-5.)

Figure 6-5. The GLUE console: index page

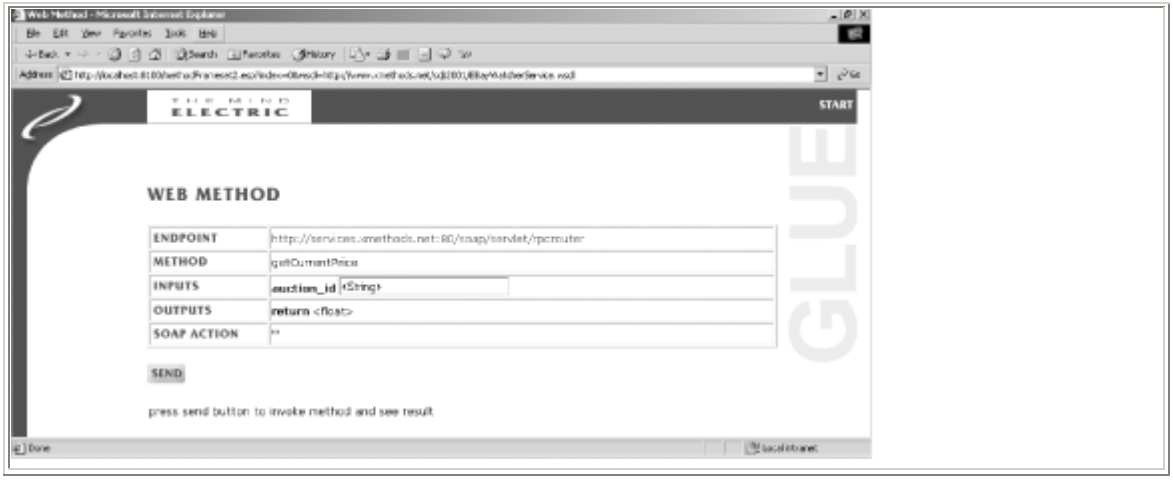
In the text box entitled WSDL, you can enter the URL for any WSDL file. For example, try entering the URL for the eBay Price Watcher Service, http://www.xmethods.net/sd/2001/EBayWatcherService.wsdl.

Click the WSDL button, and you will see the Web Service overview page. (See Figure 6-6.) This page includes a description of the specified service (extracted from the WSDL `document` element) and a list of public operations. In the case of the eBay service, you should see a single `getCurrentPrice` method.

Figure 6-6. The GLUE console: Web Service overview page for the eBay Price Watcher Service

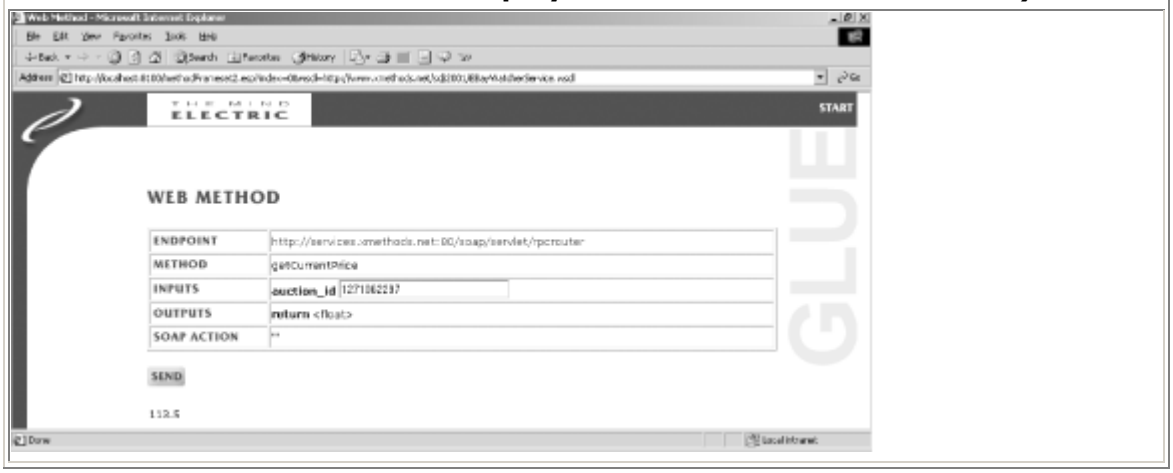
Click the `getCurrentPrice` method, and you will see the Web Method overview page. (See Figure 6-7.) This page includes a text box where you can specify the input auction ID.

Figure 6-7. The GLUE console: Web Method overview page for the getCurrentPrice method



Enter an auction ID, click the Send button, and GLUE will automatically invoke the remote method and display the results at the bottom of the page. For example, Figure 6-8 shows the current bid price for the Handspring Visor Deluxe. Note that the price has already gone up $10 since invoking the service via the GLUE command-line tool!

Figure 6-8. The GLUE console: invoking the getCurrentPrice method (results of the invocation are displayed at the bottom of the screen)



SOAPClient.com

If you would like to try out a web-based interface similar to GLUE, but don't want to bother downloading the GLUE package, consider the Generic SOAP Client available at SOAPClient.com.

Figure 6-9 shows the opening screen to the Generic SOAP Client. Much like the GLUE console, you can specify the address for a WSDL file in this screen.

Figure 6-9. The Generic SOAP Client, available from SOAPClient.com

Specify the same eBay Price Watcher Service WSDL file, and the SOAP Client will display a text box for entering the auction ID. (See Figure 6-10.)

Figure 6-10. The Generic SOAP Client: Displaying information on the XMethods eBay Price Watcher Service

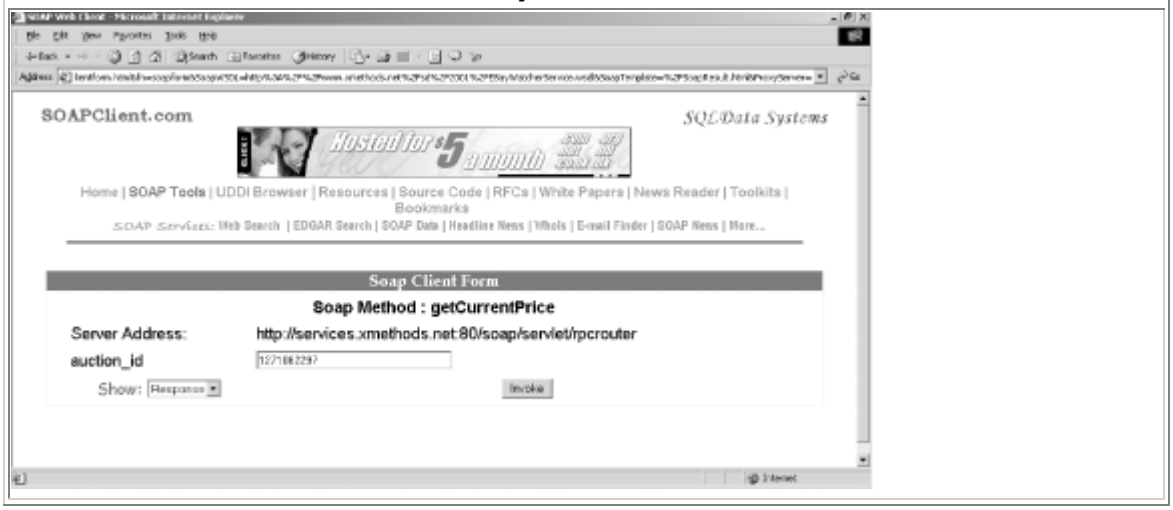
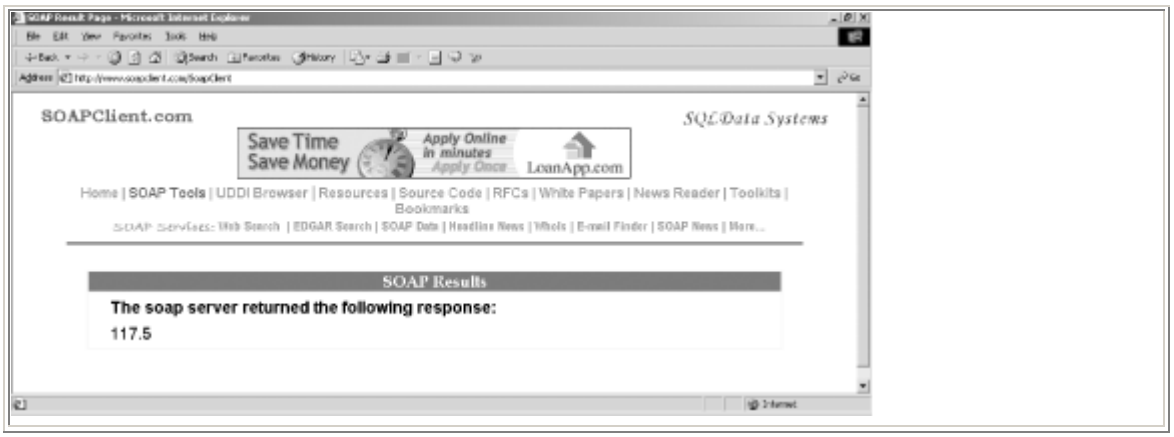


Figure 6-11 displays the result of the eBay service invocation. The Handspring Visor is up another $4!

Figure 6-11. The Generic SOAP Client: Response from the XMethods eBay Price Watcher Service



Automatically Generating WSDL Files

One of the best aspects of WSDL is that you rarely have to create WSDL files from scratch. A whole host of tools currently exists for transforming existing services into WSDL descriptions. You can then choose to use these WSDL files as is or manually tweak them with your favorite text editor. In the discussion that follows, we explore the WSDL generation tool provided by GLUE.

TIP: If you create WSDL files from scratch or tweak WSDL files generated by a tool, it is a good idea to validate your final WSDL documents. You can download a WSDL validator from http://pocketsoap.com/wsdl/. This package requires that you have an XSLT engine and the zvonSchematron (http://www.zvon.org/), but installation only takes a few minutes. Once installed, the validator is well worth the effort and creates nicely formatted HTML reports indicating WSDL errors and warnings.

GLUE java2wsdl Tool

The GLUE platform includes a `java2wsdl` command-line tool for transforming Java services into WSDL descriptions. The command-line usage is as follows:

```
usage: java2wsdl <arguments>

where valid arguments are:
  classname                name of java class
  -d directory             output directory
  -e url                   endpoint of service
  -g                       include GET/POST binding
  -m map-file              read mapping instructions
  -n namespace             namespace for service
  -r description           description of service
  -s                       include SOAP binding
  -x command-file          command file to execute
```

Complete information on each argument is available online within the GLUE User Guide at http://www.themindelectric.com/products/glue/releases/GLUE-1.1/docs/guide/index.html. For now, we will focus on the most basic arguments.

For example, consider the `PriceService` class in Example 6-4. The service provides a single `getPrice( )` method.

Example 6-4: PriceService.java

```
package com.ecerami.soap.examples;

import java.util.Hashtable;
/**
 * A Sample SOAP Service
 * Provides Current Price for requested Stockkeeping Unit (SKU)
 */
public class PriceService {
  protected Hashtable products;

  /**
   * Zero Argument Constructor
   * Load product database with two sample products
   */
  public PriceService (  ) {
    products = new Hashtable(  );
    //  Red Hat Linux
    products.put("A358185", new Double (54.99));
    //  McAfee PGP Personal Privacy
    products.put("A358565", new Double (19.99));
  }

  /**
  *  Provides Current Price for requested SKU
  *  In a real-setup, this method would connect to
  *  a price database.  If SKU is not found, method
  *  will throw a PriceException.
  */
  public double getPrice (String sku)
    throws ProductNotFoundException {
    Double price = (Double) products.get(sku);
    if (price == null) {
      throw new ProductNotFoundException ("SKU: "+sku+" not found");
    }
    return price.doubleValue(  );
  }
}
```

To generate a WSDL file for this class, run the following command:

```
java2wsdl com.ecerami.soap.examples.PriceService -s -e http://localhost:
 8080/soap/servlet/rpcrouter -n urn:examples:priceservice
```

The `-s` option directs GLUE to create a SOAP binding; the `-e` option specifies the address of our service; and the `-n` option specifies the namespace URN for the service. GLUE will generate a *PriceService.wsdl* file. (See Example 6-5.)

TIP: If your service is defined via a Java interface and you include your source files within your CLASSPATH, GLUE will extract your Javadoc comments, and turn these into WSDL `documentation` elements.

Example 6-5: PriceService.wsdl (automatically generated by GLUE)

```
<?xml version='1.0' encoding='UTF-8'?>
<!--generated by GLUE-->
<definitions name='com.ecerami.soap.examples.PriceService'

targetNamespace='http://www.themindelectric.com/wsdl/com.ecerami.soap.
    examples.PriceService/'

   xmlns:tns='http://www.themindelectric.com/wsdl/com.ecerami.soap.
       examples.PriceService/'
   xmlns:electric='http://www.themindelectric.com/'
   xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
   xmlns:http='http://schemas.xmlsoap.org/wsdl/http/'
   xmlns:mime='http://schemas.xmlsoap.org/wsdl/mime/'
   xmlns:xsd='http://www.w3.org/2001/XMLSchema'
   xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
   xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
   xmlns='http://schemas.xmlsoap.org/wsdl/'>
  <message name='getPrice0SoapIn'>
    <part name='sku' type='xsd:string'/>
  </message>
  <message name='getPrice0SoapOut'>
    <part name='Result' type='xsd:double'/>
  </message>
  <portType name='com.ecerami.soap.examples.PriceServiceSoap'>
    <operation name='getPrice' parameterOrder='sku'>
      <input name='getPrice0SoapIn' message='tns:getPrice0SoapIn'/>
      <output name='getPrice0SoapOut' message='tns:getPrice0SoapOut'/>
    </operation>
  </portType>
  <binding name='com.ecerami.soap.examples.PriceServiceSoap'
      type='tns:com.ecerami.soap.examples.PriceServiceSoap'>
    <soap:binding style='rpc'
      transport='http://schemas.xmlsoap.org/soap/http'/>
    <operation name='getPrice'>
      <soap:operation soapAction='getPrice' style='rpc'/>
      <input name='getPrice0SoapIn'>
        <soap:body use='encoded'
         namespace='urn:examples:priceservice'
         encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
      </input>
      <output name='getPrice0SoapOut'>
        <soap:body use='encoded'
        namespace='urn:examples:priceservice'
        encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
      </output>
    </operation>
  </binding>
  <service name='com.ecerami.soap.examples.PriceService'>
    <port name='com.ecerami.soap.examples.PriceServiceSoap'
      binding='tns:com.ecerami.soap.examples.PriceServiceSoap'>
```

```
      <soap:address location='http://207.237.201.187:8080
          /soap/servlet/ rpcrouter'/>
    </port>
  </service>
</definitions>
```

You can then invoke the service via SOAP::Lite:

```
use SOAP::Lite;

print "Connecting to Price Service...\n";
print SOAP::Lite
  -> service('http://localhost:8080/wsdl/PriceService.wsdl')
  -> getPrice ('A358185');
```

Hopefully, this example illustrates the great promise of web service interoperability. We have a WSDL file generated by GLUE, a server running Java, and a client running Perl, and they all work seamlessly together.

```
Connecting to Price Service...
54.99
```

WARNING: The IBM Web Services Toolkit (available at http://www.alphaworks.ibm.com/tech/webservicestoolkit) provides a WSDL generation tool called `wsdlgen`. This tool can take existing Java classes, Enterprise JavaBeans, and Microsoft COM objects and automatically generate corresponding WSDL files. However, as this book goes to press, the `wsdlgen` tool creates files based on the 1999 version of the W3C XML Schema. The WSDL files are therefore incompatible with other WSDL invocation tools, such as SOAP::Lite and GLUE. If you choose to use the IBM tool, make sure to manually update your WSDL files to reflect the latest version of XML Schema (http://www.w3.org/2001/XMLSchema).

XML Schema Data Typing

In order for a SOAP client to communicate effectively with a SOAP server, the client and server must agree on a data type system. By default, XML 1.0 does not provide a data type system. In contrast, every programming language provides some basic facility for declaring data types, such as integers, floats, doubles, and strings. One of the greatest challenges in building web services is therefore creating a common data type system that can be used by a diverse set of programming languages running on a diverse set of operating systems.

WSDL does not aim to create a standard for XML data typing. In fact, WSDL is specifically designed for maximum flexibility and is therefore not tied exclusively to any one data type system. Nonetheless, WSDL does default to the W3C XML Schema specification. The XML Schema specification is also currently the most widely used specification for data typing.

The more you know about XML Schemas, the better you can understand complex WSDL files. A full discussion of XML Schemas is beyond the scope of this chapter. However, two facts are crucially important.

First, the XML Schema specification includes a basic type system for encoding most data types. This type system includes a long list of built-in simple types, including strings, floats, doubles, integers, time, and date. This list, shown in Table 6-1, is excerpted from the XML Schema Part 0: Primer (http://www.w3org/TR/2000/WD=xmlschema=0=20000407/). If your application sticks to these simple data types, there is no need to include the WSDL `types` element, and the resulting WSDL file is extremely simple. For example, our first two WSDL files use only strings and floats.

Table 6-1: A list of the main XML Schema built-in simple types

Simple type	Example(s)
string	Web Services
Boolean	true, false, 1, 0
float	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN
double	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN
decimal	-1.23, 0, 123.4, 1000.00
binary	100010
integer	-126789, -1, 0, 1, 126789
nonPositiveInteger	-126789, -1, 0
negativeInteger	-126789, -1
long	-1, 12678967543233
int	-1, 126789675
short	-1, 12678
byte	-1, 126
nonNegativeInteger	0, 1, 126789
unsignedLong	0, 12678967543233
unsignedInt	0, 1267896754
unsignedShort	0, 12678
unsignedByte	0, 126
positiveInteger	1, 126789
date	1999-05-31
time	13:20:00.000, 13:20:00.000-05:00

Second, the XML Schema specification provides a facility for creating *new* data types. This is important if you want to create data types that go beyond what is already defined within the Schema. For example, a service might return an array of floats or a more complex stock quote object containing the high, low, and volume figures for a specific stock. Whenever your service goes beyond the simple XML Schema data types, you must declare these new data types within the WSDL `types` element.

In the next two sections of this chapter, we present two specific examples of using XML Schemas to create new data types. The first focuses on arrays; the second focuses on a more complex data type for encapsulating product information.

Arrays

Example 6-6, shown later in this section, is a sample WSDL file that illustrates the use of arrays. This is the Price List Service we created in Chapter 5. The service has one public method, called `getPriceList`, which expects an array of string SKU values and returns an array of double price values.

The WSDL file now includes a `types` element. Inside this element, we have defined two new complex types. Very broadly, the XML Schema defines simple types and complex types. Simple types cannot have element children or attributes, whereas complex types can have element children and attributes. We have declared complex types in our WSDL file, because an array may have multiple elements, one for each value in the array.

The XML Schema requires that any new type you create be based on some existing data type. This existing base type is specified via the `base` attribute. You can then choose to modify this base type using one of two main methods: `extension` or `restriction`. Extension simply means that your new data type will have all the properties of the base type plus some extra functionality. Restriction means that your new data type will have all the properties of the base data type, but may have additional restrictions placed on the data.

In Example 6-6, we'll create two new complex types via restriction. For example:

```
<complexType name="ArrayOfString">
  <complexContent>
    <restriction base="soapenc:Array">
      <attribute ref="soapenc:arrayType"
        wsdl:arrayType="string[]"/>
    </restriction>
  </complexContent>
</complexType>
```

Example 6-6: PriceListService.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="PriceListService"
   targetNamespace="http://www.ecerami.com/wsdl/PriceListService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.ecerami.com/wsdl/PriceListService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsd1="http://www.ecerami.com/schema">

   <types>
      <schema xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.ecerami.com/schema"
         xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
         xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">

         <complexType name="ArrayOfString">
            <complexContent>
               <restriction base="soapenc:Array">
```

```
                <attribute ref="soapenc:arrayType"
                wsdl:arrayType="string[]"/>
             </restriction>
          </complexContent>
       </complexType>
       <complexType name="ArrayOfDouble">
          <complexContent>
             <restriction base="soapenc:Array">
                <attribute ref="soapenc:arrayType"
                wsdl:arrayType="double[]"/>
             </restriction>
          </complexContent>
       </complexType>
    </schema>
 </types>

 <message name="PriceListRequest">
    <part name="sku_list" type="xsd1:ArrayOfString"/>
 </message>

 <message name="PriceListResponse">
    <part name="price_list" type="xsd1:ArrayOfDouble"/>
 </message>

 <portType name="PriceList_PortType">
    <operation name="getPriceList">
       <input message="tns:PriceListRequest"/>
       <output message="tns:PriceListResponse"/>
    </operation>
 </portType>

 <binding name="PriceList_Binding" type="tns:PriceList_PortType">
 <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getPriceList">
    <soap:operation soapAction="urn:examples:pricelistservice"/>
       <input>
          <soap:body

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
             namespace="urn:examples:pricelistservice"
             use="encoded"/>
       </input>
       <output>
          <soap:body

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
             namespace="urn:examples:pricelistservice"
use="encoded"/>
       </output>
    </operation>
 </binding>

 <service name="PriceList_Service">
    <port name="PriceList_Port" binding="tns:PriceList_Binding">
       <soap:address
location="http://localhost:8080/soap/servlet/rpcrouter"/>
```

```
      </port>
   </service>
</definitions>
```

The WSDL specification requires that arrays be based on the SOAP 1.1 encoding schema. It also requires that arrays use the name `ArrayOfXXX`, where `XXX` is the type of item in the array. The previous example therefore creates a new type called `ArrayOfString`. This new type is based on the SOAP array data type, but it is restricted to holding only string values. Likewise, the `ArrayOfDouble` data type creates a new array type containing only double values.

When using the WSDL `types` element, you need to be particularly aware of XML namespace issues. First, note that the root `schema` element must include a namespace declaration for the SOAP encoding specification (http://schemas.xmlsoap.org/soap/encoding/). This is required because our new data types extend the array definition specified by SOAP.

Second, the root `schema` element must specify a `targetNamespace` attribute. Any newly defined elements, such as our new array data types, will belong to the specified `targetNamespace`. To reference these data types later in the document, you must refer back to the same `targetNamespace`. Hence, our `definitions` element includes a new namespace declaration:

```
xmlns:xsd1="http://www.ecerami.com/schema">
```

`xsd1` matches the `targetNamespace` and therefore enables us to reference the new data types later in the document. For example, the `message` element references the `xsd1:ArrayOfString` data type:

```
<message name="PriceListRequest">
   <part name="sku_list" type="xsd1:ArrayOfString"/>
</message>
```

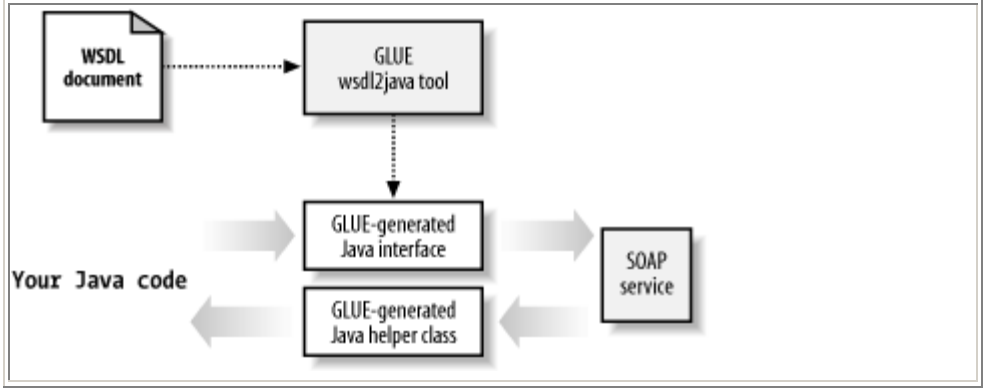
TIP: For an excellent and concise overview of W3C Schema complex types and their derivation via extension and restriction, see Donald Smith's article on "Understanding W3C Schema Complex Types." The article is available online at http://www.xml.com/pub/a/2001/08/22/easyschema.html.

Automatically invoking array services

Once you move beyond basic data types, the simple WSDL invocation methods described previously in this chapter no longer work quite as easily. For example, you cannot simply open the GLUE console, pass an array of strings, and hope to receive back an array of doubles. Additional work is necessary, and some manual code is required. Nonetheless, the additional work is minimal, and the discussion that follows focuses on the GLUE platform. We have chosen to focus on the GLUE platform because it represents the most elegant platform for working with complex data types; other tools, such as the IBM Web Services Toolkit, do, however, provide similar functionality.

To get started, you should become familiar with the GLUE `wsdl2java` command-line tool. The tool takes in a WSDL file and generates a suite of Java class files to automatically interface with the specified service. You can then write your own Java class to invoke the specified service. Best of all, the code you write is minimally simple, and all SOAP-specific details are completely hidden from your view. (See Figure 6-12.)

Figure 6-12. The GLUE wsdl2java tool and the GLUE architecture



Here is the `wsdl2java` command-line usage:

```
usage: wsdl2java <arguments>

where valid arguments are:
  http://host:port/filename     URL of WSDL
  -c                        checked exceptions
  -d directory                output directory for files
  -l user password realm        login credentials
  -m map-file                 read mapping instructions
  -p package                  set default package
  -v                        verbose
  -x command-file             command file to execute
```

Complete information on each argument is available online within the GLUE User Guide at http://www.themindelectric.com/products/glue/releases/GLUE-1.1/docs/guide/index.html. For now, we will focus on the most basic arguments. For example, to generate Java class files for the *PriceListService.wsdl* file, first make sure that the WSDL file is available publicly on a web site or locally via a web server such as Tomcat. Then, issue the following command:

```
wsdl2java.bat http://localhost:8080/wsdl/PriceListService.wsdl -p com.
   ecerami.wsdl.glue
```

The first argument specifies the location of the WSDL file; the second argument specifies that the generated files should be placed in the package `com.ecerami.wsdl.glue`.

GLUE will automatically download the specified WSDL file and generate two Java class files:

```
write file IPriceList_Service.java
```

write file PriceList_ServiceHelper.java

The first file, *IPriceList_Service.java*, is shown in Example 6-7. This file represents a Java interface that mirrors the public methods exposed by the WSDL file. Specifically, the interface shows a `getPriceList( )` method that receives an array of `String` values, and returns an array of `double` values.

Example 6-7: IPriceList_Service.java

```
// generated by GLUE

package com.ecerami.wsdl.glue;

public interface IPriceList_Service
  {
  double[] getPriceList( String[] sku_list );
  }
```

The second file, *PriceList_ServiceHelper.java*, is shown in Example 6-8. This is known as a GLUE helper file, and it can dynamically bind to the service specified by the WSDL file. To access the service, simply call the static `bind( )` method.

Example 6-8: PriceList_ServiceHelper.java

```
// generated by GLUE

package com.ecerami.wsdl.glue;

import electric.registry.Registry;
import electric.registry.RegistryException;

public class PriceList_ServiceHelper
  {
  public static IPriceList_Service bind(  ) throws RegistryException
    {
    return bind( "http://localhost:8080/wsdl/PriceListService.wsdl" );
    }

  public static IPriceList_Service bind( String url )
    throws RegistryException
    {
    return (IPriceList_Service)
       Registry.bind( url, IPriceList_Service.class );
    }
  }
```

Once GLUE has generated the interface and helper files, you just need to write your own class that actually invokes the service. Example 6-9 shows a sample application that invokes the Price List Service. The code first calls `PriceList_ServiceHelper.bind( )`, which then returns an `IPriceList_Service` object. All subsequent code behaves as if the Price List Service is a local object, and all SOAP-specific details are completely hidden from the developer.

Here is a sample output of the `Invoke_PriceList` application:

```
Product Catalog
SKU:  A358185 --> Price:  54.99
SKU:  A358565 --> Price:  19.99
```

Example 6-9: Invoke_PriceList.java

```
package com.ecerami.wsdl;

import com.ecerami.wsdl.glue.*;

/**
 * SOAP Invoker.  Uses the PriceListServiceHelper to invoke
 * SOAP service.  PriceListServiceHelper and IPriceListService
 * are automatically generated by GLUE.
*/
public class Invoke_PriceList {

  /**
   *  Get Product List via SOAP
   */
  public double[] getPrices (String skus[]) throws Exception {
    IPriceList_Service priceListService = PriceList_ServiceHelper.bind(
);
    double[] prices = priceListService.getPriceList(skus);
    return prices;
  }

  /**
   *  Main Method
   */
  public static void main (String[] args) throws Exception {
    Invoke_PriceList invoker = new Invoke_PriceList(  );
    System.out.println ("Product Catalog");
    String skus[] = {"A358185", "A358565" };
    double[] prices = invoker.getPrices (skus);
    for (int i=0; i<prices.length; i++) {
      System.out.print ("SKU:  "+skus[i]);
      System.out.println (" --> Price:  "+prices[i]);
    }
  }
}
```

Complex Types

Our final topic is the use of complex data types. For example, consider a home monitoring service that provides a concise update on your home. The data returned could include multiple data elements, such as the current temperature, security status, and whether the garage door is open or closed. Encoding this data into WSDL requires additional knowledge of XML Schemas, which reinforces the main precept that the more you know about XML Schemas, the better you will understand complex WSDL files.

To explore complex types, consider the WSDL file in Example 6-10. This WSDL file describes our Product Service from Chapter 5. The complex types are indicated in bold.

Example 6-10: ProductService.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="ProductService"
  targetNamespace="http://www.ecerami.com/wsdl/ProductService.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.ecerami.com/wsdl/ProductService.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://www.ecerami.com/schema">

   <types>
     <xsd:schema
         targetNamespace="http://www.ecerami.com/schema"
         xmlns="http://www.w3.org/2001/XMLSchema">
         <xsd:complexType name="product">
            <xsd:sequence>
               <xsd:element name="name" type="xsd:string"/>
               <xsd:element name="description" type="xsd:string"/>
               <xsd:element name="price" type="xsd:double"/>
               <xsd:element name="SKU" type="xsd:string"/>
            </xsd:sequence>
         </xsd:complexType>
      </xsd:schema>
   </types>

   <message name="getProductRequest">
     <part name="sku" type="xsd:string"/>
   </message>

   <message name="getProductResponse">
     <part name="product" type="xsd1:product"/>
   </message>

   <portType name="Product_PortType">
     <operation name="getProduct">
       <input message="tns:getProductRequest"/>
       <output message="tns:getProductResponse"/>
     </operation>
   </portType>

   <binding name="Product_Binding" type="tns:Product_PortType">
     <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
     <operation name="getProduct">
       <soap:operation soapAction="urn:examples:productservice"/>
       <input>
         <soap:body
             encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
             namespace="urn:examples:productservice"
             use="encoded"/>
       </input>
```

```
        <output>
          <soap:body
              encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              namespace="urn:examples:productservice" use="encoded"/>
        </output>
      </operation>
    </binding>

    <service name="Product_Service">
      <port name="Product_Port" binding="tns:Product_Binding">
        <soap:address
location="http://localhost:8080/soap/servlet/rpcrouter"/>
      </port>
    </service>
</definitions>
```

The service in Example 6-10 describes a `getProduct` operation that returns a complex *product* type for encapsulating product information, including product name, description, price, and SKU number.

The new product type is defined in much the same manner as the array definition from the previous example. The main difference is that we are now using the `sequence` element. The `sequence` element specifies a list of subelements and requires that these elements appear in the order specified. XML Schemas also enable you to specify cardinality via the `minOccurs` and `maxOccurs` attributes. If these attributes are absent (as in our example), they default to `1`, requiring that each subelement must occur exactly one time.

Each subelement can also have its own data type, and you can see that we have mixed and matched string data types with double data types in our example.

Automatically invoking complex type services

To automatically invoke the Product Service, we return to the GLUE `wsdl2java` tool. This time around, GLUE will generate a Java interface class and a Java helper class, along with two additional files for handling the new complex type.

For example, the following command:

```
wsdl2java.bat http://localhost:8080/wsdl/ProductService.wsdl -p com.ecerami.
  wsdl.glue
```

generates the following output:

```
write file IProduct_Service.java
write file Product_ServiceHelper.java
write file product.java
write file Product_Service.map
```

The first two files in the output listing are familiar. The first file is a Java interface mirroring the service; the second file is a helper class for dynamically binding to the specified service. (See Example 6-11 and Example 6-12.)

Example 6-11: IProduct_Service.java

```
// generated by GLUE

package com.ecerami.wsdl.glue;

public interface IProduct_Service
  {
  product getProduct( String sku );
  }
```

Example 6-12: Product_ServiceHelper.java

```
// generated by GLUE

package com.ecerami.wsdl.glue;

import electric.registry.Registry;
import electric.registry.RegistryException;

public class Product_ServiceHelper
  {
  public static IProduct_Service bind(  ) throws RegistryException
    {
    return bind( "http://localhost:8080/wsdl/ProductService.wsdl" );
    }

  public static IProduct_Service bind( String url )
   throws RegistryException
    {
    return (IProduct_Service)
      Registry.bind( url, IProduct_Service.class );
    }
  }
```

The third file in the output listing, *product.java*, represents a simple container class for encapsulating product data. (See Example 6-13.) GLUE essentially takes all the complex types defined within the WSDL file and creates a container class for each type. Each subelement is then transformed into a public variable for easy access. For example, the `product` class has four public variables, `name`, `description`, `price`, and `SKU`, corresponding to our new complex data type. Note also that the public variables match the XML Schema types specified within the WSDL file; for example, `name` is declared as a `String`, whereas `price` is declared as a `double`.

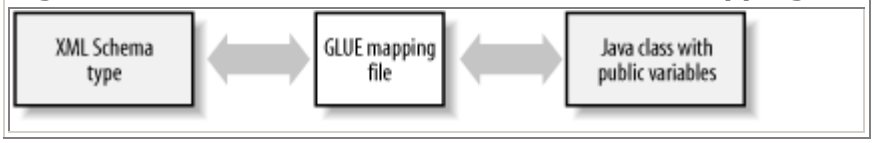
Example 6-13: product.java

```
// generated by GLUE

package com.ecerami.wsdl.glue;

public class product
  {
  public java.lang.String name;
```

```
  public java.lang.String description;

  public double price;

  public java.lang.String SKU;
  }
```

Finally, GLUE generates a Java-to-XML Schema mapping file. (See Example 6-14.) The file itself is extremely concise and is responsible for converting Java to XML Schema types and vice versa. (See Figure 6-13.) The root `complexType` element indicates that elements of type `product` should be transformed into the `product` class located in `com.ecerami.wsdl.glue`. Inside the root complex type, there is a one-to-one mapping between the XML Schema type and the public Java variable. For example, the element `name` is mapped to the `product.name` variable, and the type is specified as `string`. Likewise, the element `price` is mapped to the `product.price` variable, and the type is specified as `double`.

Figure 6-13. The GLUE Java-to-XML Schema mapping file



Example 6-14: Product_Service.map

```
<?xml version='1.0' encoding='UTF-8'?>
<!--generated by GLUE-->
<mappings xmlns='http://www.themindelectric.com/schema/'>
 <schema
  xmlns='http://www.w3.org/2001/XMLSchema'
  targetNamespace='http://www.ecerami.com/schema'
  xmlns:electric='http://www.themindelectric.com/schema/'>
  <complexType name='product'
electric:class='com.ecerami.wsdl.glue.product'>
   <sequence>
    <element name='name' electric:field='name' type='string'/>
    <element name='description'
       electric:field='description' type='string'/>
    <element name='price' electric:field='price' type='double'/>
    <element name='SKU' electric:field='SKU' type='string'/>
    </sequence>
  </complexType>
 </schema>
</mappings>
```

To invoke the Product Service, you must first explicitly load the mapping file via the GLUE `Mappings` class:

```
Mappings.readMappings("Product_Service.map");
```

You can then access the service just like in the previous example. See Example 6-15 for the complete invocation program. Here is some sample output:

```
Product Service
Name:  Red Hat Linux
Description:  Red Hat Linux Operating System
Price:  54.99
```

Example 6-15: Invoke_Product.java

```
package com.ecerami.wsdl;

import java.io.IOException;
import electric.xml.io.Mappings;
import electric.xml.ParseException;
import electric.registry.RegistryException;
import com.ecerami.wsdl.glue.*;

/**
 * SOAP Invoker.  Uses the Product_ServiceHelper to invoke the Product
 * SOAP service.  All other .java files are automatically generated
 * by GLUE.
*/
public class Invoke_Product {

  /**
   * Get Product via SOAP Service
   */
  public product getProduct (String sku) throws Exception {
    //  Load Java <--> XML Mapping
    Mappings.readMappings("Product_Service.map");
    //  Invoke Service
    IProduct_Service service = Product_ServiceHelper.bind(  );
    product prod = service.getProduct(sku);
    return prod;
  }

  /**
   * Main Method
   */
  public static void main (String[] args) throws Exception {
    Invoke_Product invoker = new Invoke_Product(  );
    System.out.println ("Product Service");
    product prod = invoker.getProduct("A358185");
    System.out.println ("Name:  "+prod.name);
    System.out.println ("Description:  "+prod.description);
    System.out.println ("Price:  "+prod.price);
 }
}
```

This is a very small amount of code, but it is capable of doing very real work. Be sure to check The Mind Electric web site (http://themindelectric.com/) for new releases of the GLUE product.