

# 2

## SOAP Servers and Clients with PHP SOAP Extension

The PHP 5's SOAP extension is implemented as a set of predefined PHP classes that allow the developer to build SOAP servers and clients. In this chapter, you will learn how to use the PHP SOAP extension when building Web services that might then be utilized within SOA applications. In particular you will learn how to:

- Expose application logic as a Web service
- Build Web service providers and requestors
- Encapsulate the underlying logic of a Web service in a PHP class
- Use the XML Schemas specification with WSDL
- Transmit XML documents containing attributes
- Extend predefined classes of the PHP SOAP Extension
- Build Web services supporting parameter-driven operations

### Building Service Providers and Service Requestors

Depending on the interaction scenario in which a Web service is involved, it can either act as a service provider or a service requestor. In the following sections, you will see how to build a Web service provider and a requestor that will consume the service provider.

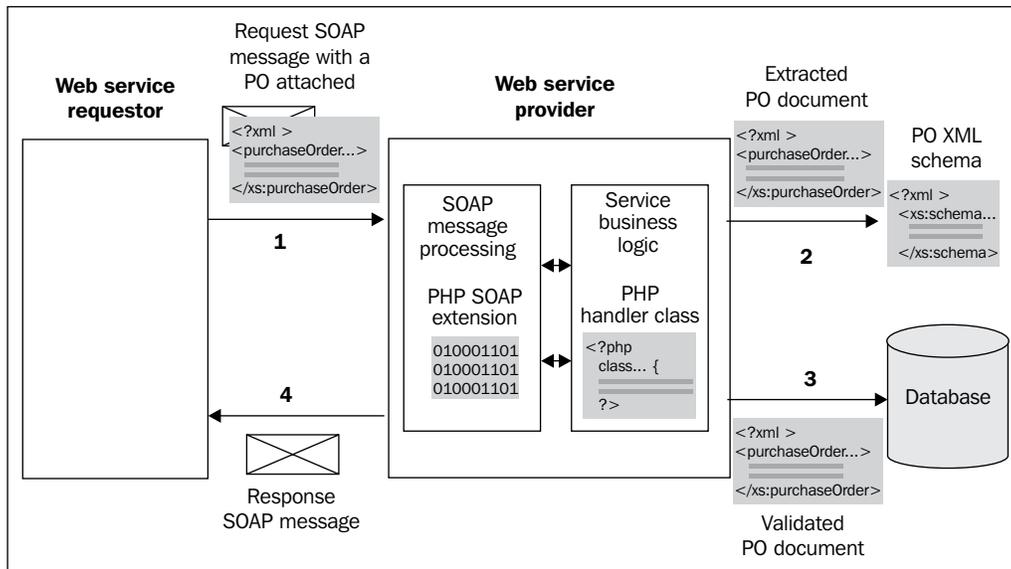
To start with, let's look at a simple example. Suppose that you need to implement an application based on Web services that performs the following sequence of steps:

1. Receives a purchase order (PO) document in XML format
2. Validates a PO against the appropriate XML schema
3. Stores a PO in the database
4. Sends a response message to the requestor

In a real-world situation, to build such an application, you would have to design more than one service and pull these services together into a composite solution. However, for simplicity's sake, the example discussed here uses the only service to handle all of the above tasks.


[
Of course, the above service would be only a part of an entire real-world solution. A service requestor sending a PO document for processing to this service would act as a service provider itself towards another requestor, or would be part of a composite service built, for example, with WS-BPEL.
]

Diagrammatically, the scenario involving the PO Web service that performs the tasks described above might look like the following figure:



Here is the detailed explanation of the steps in the figure:

- Step 1: The service requestor sends a PO XML document wrapped in a SOAP envelope to the service provider.

- Step 2: The service provider extracts the PO document received from the SOAP envelope and validates the extracted PO against the appropriate XML schema.
- Step 3: The service provider stores the validated PO document in the database.
- Step 4: The service provider sends the response message to the service requestor, informing it if the operations being performed have completed successfully or not.

To build the PO Web service depicted in the previous figure, you need to accomplish the following five general steps:

1. Set up a database to store incoming PO documents
2. Develop a PHP handler class implementing the PO Web service logic
3. Design an XML schema to validate incoming PO documents
4. Design a WSDL document describing the PO Web service to its requestors
5. Build a SOAP server to handle incoming messages carrying POs

The following sections take you through each of the above steps. First, you will see how to create a simple PO Web service that actually performs no validation. Then, you will learn how the XML Schema feature can be used with WSDL to define types in messages being transmitted, making sure that transmitted data is valid with respect to a specific XML schema.

## Setting Up the Database

Before we go any further, let's take a moment to set up the database required for this example. This example assumes that you are using Oracle Database Express Edition (XE)—a free edition of Oracle Database, or any other edition of Oracle database.



You can download a copy of Oracle Database from the Download page of the Oracle Technology Network (OTN) Website at <http://www.oracle.com/technology/software/index.html>. In Chapter 3, you will also see an example of using MySQL as the backend database in a Web services application. As for this particular example, Oracle is used because it provides native support for XML, which makes it easier for you to get the job done, allowing you to concentrate on using the PHP SOAP extension while building the application.

To keep things simple, this section actually discusses how to create a minimal set of the database objects required only to store incoming PO documents. When continuing with this example in Chapter 3, you will learn how to leverage the Oracle XML Schema, an Oracle XML feature, to validate incoming POs inside the database.

 The Oracle XML Schema is part of the Oracle XML DB, which is a set of Oracle XML features available in any edition of Oracle Database by default. The Oracle XML DB is discussed in extensive detail in Chapter 3.

With Oracle database, you have several options when it comes to creating, accessing and manipulating the database objects. You can use both the graphical and command-line tools shipped with Oracle Database. As for Oracle Database XE, you might use the Oracle Database XE graphical user interface, a browser-based tool that allows you to administer the database.

However, to create the database objects required for this example, it is assumed that you will make use of Oracle SQL\*Plus, a command-line SQL tool, which is installed by default with every Oracle database installation.

 For information on Oracle database installation, see Appendix A, section *Installing Oracle Database Express Edition (XE)*.

With SQL\*Plus, you interact with the database server by entering appropriate SQL statements at the `SQL>` prompt.

Assuming that you have an Oracle database server installed and running, launch SQL\*Plus and then follow these steps:

Set up a database account that will be used as a container for the database objects by issuing the following SQL statements:

```
//connect to the database as sysdba to be able to create a new
account
CONN /as sysdba

//create a user identified as xmlusr with the same password
CREATE USER xmlusr IDENTIFIED BY xmlusr;

//grant privileges required to connect and create resources
GRANT connect, resource TO xmlusr;
```

Issue the following SQL statements to create a table that will be used to store PO XML documents:

```
//connect to the database using the newly created account
CONN xmlusr/xmlusr;

//create a purchaseOrders table to be used for storing POs
CREATE TABLE purchaseOrders(
    doc VARCHAR2(4000)
);
```

As you can see, the `purchaseOrders` table created by the above statement contains only one column, namely `doc` of `VARCHAR2`. Using the `VARCHAR2` Oracle data type is the simplest option when it comes to storing XML documents in an Oracle database. In fact, Oracle provides much more powerful options for storing XML data in the database. These options will be discussed in detail in Chapter 3.

## Developing the PHP Handler Class

Now that you have set up the database to store the incoming PO documents, it's time to create the PHP code that will perform just that operation: storing incoming POs into the database. Consider the `purchaseOrder` PHP class containing the PO Web service underlying logic. It is assumed that you will save this class in the `purchaseOrder.php` file in the `WebServices\ch2` directory **within the document** directory of your Web server, so that it will be available at `http://localhost/WebServices/ch2/purchaseOrder.php`.

```
<?php
//File purchaseOrder.php
class purchaseOrder {
    function placeOrder($po) {
        if (!$conn = oci_connect('xmlusr', 'xmlusr', '//localhost/xes')){
            throw new SoapFault("Server","Failed to connect to
                                database");
        };
        $sql = "INSERT INTO purchaseOrders VALUES (:po)";
        $query = oci_parse($conn, $sql);
        oci_bind_by_name($query, ':po', $po);
        if (!oci_execute($query)) {
            throw new SoapFault("Server","Failed to insert PO");
        };
        $msg='<rsltMsg>PO inserted!</rsltMsg>';
        return $msg;
    }
}
?>
```

Looking through the code, you may notice that the `purchaseOrder` class actually contains the only method, namely `placeOrder`. As its name implies, the `placeOrder` method is responsible for placing an incoming PO document. What this method actually does is take a PO XML document as the parameter and then store it in the `purchaseOrders` table created in the preceding section. Upon failure to connect to the database or execute the `INSERT` statement, the `placeOrder` method stops execution and throws a SOAP exception.

 For now, you should not necessarily have to understand in detail how the database-related code in the `placeOrder` method works. This will be discussed in greater detail in Chapter 3.

Another important point to note here is that **the `placeOrder` method doesn't** contain any code required to validate an incoming PO document. For simplicity, this example assumes no validation for the moment. However, when continuing with the example in the next sections of this chapter, you will see how XML schema-based validation can be used with WSDL, defining types for parts of the messages described in WSDL definitions. Then, in Chapter 3, you will learn how the incoming PO documents can be automatically validated against a PO XML schema within the database, upon inserting them into the `purchaseOrders` table. As the *Using XML Schemas with Oracle XML DB* section in Chapter 3 will explain, to reach this goal, you need to create and register a PO XML schema against the database and then create an `INSERT` trigger on the `purchaseOrders` table.

## Designing the WSDL Document

To expose the functionality of the `purchaseOrder` PHP class discussed in the preceding section as a Web service, you first need to create a WSDL document that will describe that Web service. Here is the WSDL that might serve this purpose. It is assumed that you will save this document as `po.wsdl` in the `WebServices/wsd1` directory within the document directory of your Web server.

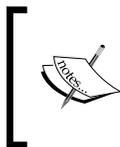
```
<?xml version="1.0" encoding="utf-8"?>
<definitions name="poService"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace=
    "http://localhost/WebServices/wsd1/po.wsdl">
  <message name="getPlaceOrderInput">
    <part name="body" element="xsd:string"/>
  </message>
```

```

<message name="getPlaceOrderOutput">
  <part name="body" element="xsd:string"/>
</message>
<portType name="poServicePortType">
  <operation name="placeOrder">
    <input message="tns:getPlaceOrderInput"/>
    <output message="tns:getPlaceOrderOutput"/>
  </operation>
</portType>
<binding name="poServiceBinding" type="tns:poServicePortType">
  <soap:binding style="document" transport=
    "http://schemas.xmlsoap.org/soap/http"/>
  <operation name="placeOrder">
    <soap:operation
      soapAction="http://localhost/Webservices/ch2/placeOrder"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
<service name="poService">
  <port name="poServicePort" binding="tns:poServiceBinding">
    <soap:address
      location="http://localhost/Webservices/ch2/SOAPserver.php"/>
  </port>
</service>
</definitions>

```

As you may notice, the `getPlaceOrderInput` message described in this document consists of a single part called `body`, which represents an element of `xsd:string`. Actually, the `body` part used here represents the parameter being passed to the `placeOrder` method of the `purchaseOrder` class discussed in the preceding section. So, this WSDL document implies that an incoming PO XML document will be passed from a service consumer to the PO Web service as a string.



As you no doubt have realized, the `string` XSD type is used in this example for simplicity's sake. In the *Using XML Schemas with WSDL* section later in this chapter, you will see an example of using the user-defined XSD types when it comes to describing XML documents being transmitted between a service provider and service requestor.

## Building the SOAP Server

Now that you have created the WSDL definition document describing the PO Web service, the next step is to create a SOAP server that will be responsible for handling and transmitting SOAP messages via HTTP. Save the following PHP script as `SoapServer.php` in the `WebServices/ch2` directory **within the document directory** of your Web server.

```
<?php
//File: SoapServer.php
require_once "purchaseOrder.php";
$wsdl= "http://localhost/WebServices/wsdl/po.wsdl";
$srvc= new SoapServer($wsdl);
$srvc->setClass("purchaseOrder");
$srvc->handle();
?>
```

At the beginning of this script you add the contents of the `purchaseOrder.php` script discussed in the *Developing the PHP Handler Class* section **earlier in this chapter**. Then, you create an instance of the `SoapServer` class.

 The `SoapServer` class, as well as `SoapClient` and `SoapFault` classes discussed in the next section, belongs to the PHP's SOAP extension library, which is not enabled by default. To enable the SOAP extension on a Unix-like platform, you need to recompile your PHP installation with the configure option `--enable-soap`. If you are a Windows user, you need to append the extension `=php_soap.dll` to the list of extensions in the `php.ini` configuration file.

Once you have created an instance of `SoapServer`, you can export the methods of the PHP handler class stored in the `purchaseOrder.php` script. **This is done with the help of the `setClass` method of the `SoapServer` instance.**

Finally, you call the `handle` method of `SoapServer`, which is responsible for handling and processing SOAP requests, calling methods of the handler class, and sending responses back to service consumers.

## Building the Service Requestor

Before you can test the PO Web service built as discussed in the preceding sections, you need to build a service requestor that will interact with the service. Here is a simple client to test the PO Web service. You may save this script in any directory. However, for simplicity's sake you might save it in the same directory as all the other scripts discussed previously.

```
<?php
//File: SoapClient.php
$wsdl = "http://localhost/Webservices/wsdl/po.wsdl";
$handle = fopen("purchaseOrder.xml", "r");
$po= fread($handle, filesize("purchaseOrder.xml"));
fclose($handle);
$client = new SoapClient($wsdl);
try {
    print $result=$client->placeOrder($po);
}
catch (SoapFault $exp) {
    print $exp->getMessage();
}
?>
```

As you can see, this script loads a PO document from the `purchaseOrder.xml` file, which is supposed to be in the same directory as the script. Then, it creates an instance of the `SoapClient` class, passing a URI of the WSDL document to be used, as the parameter. Note that you use the same WSDL document you used for the server discussed in the preceding section. Finally, the script calls the `placeOrder` remote function as a method of the newly created `SoapClient` instance, surrounding that call in a `try` block. If something goes wrong during the `placeOrder` execution and a `SoapFault` exception is thrown, the `catch` block catches it.

A simplified version of a PO document stored in the `purchaseOrder.xml` file being used in this example might look as follows:

```
<purchaseOrder >
  <pono>108128476</pono>
  <billTo>
    <name>Tony Jamison</name>
    <street>24 Johnson Road</street>
    <city>Big Valley</city>
    <state>VA</state>
    <zip>23032</zip>
    <country>US</country>
  </billTo>
  <shipTo>
    <name>Janet Thomson</name>
    <street>11 Maple Street</street>
    <city>Small Valley</city>
    <state>VA</state>
    <zip>23037</zip>
    <country>US</country>
  </shipTo>
```

```
<items>
  <item>
    <partId>743</partId>
    <quantity>4</quantity>
    <price>15.50</price>
  </item>
  <item>
    <partId>235</partId>
    <quantity>7</quantity>
    <price>15.75</price>
  </item>
</items>
</purchaseOrder>
```

In a real-world situation, a PO XML document might be derived from different sources, not necessarily from a file. For example, it might be created on the fly (dynamically) by a PHP script, with the help of the DOM API that is part of the PHP core.



You will see an example of building an XML document with the help of the PHP DOM extension in the *Converting SOAP Messages' Payloads to XML* section later in this chapter.

## Testing the Service

Now you are ready to test the PO Web service. To do this, you simply need to point your browser at the service requestor discussed in the preceding section. If you saved the `SoapClient.php` script in the `WebServices/ch2` directory **within the** document directory of your Web server, enter the following URL in the address box of your browser: `http://localhost/WebServices/ch2/SoapClient.php`.

If everything goes as planned, you will see a **PO inserted!** message in your browser. Otherwise, a SOAP fault message appears. For example, if the `placeOrder` method of the `purchaseOrder` class **fails to connect to the database**, you will see an error message that will look as follows:

### Failed to connect to database

Turning back to the `placeOrder` method of the `purchaseOrder` class discussed in the *Developing the PHP Handler Class* section earlier, you may notice that it also throws a SOAP exception upon failure to insert the received PO into the database.

If the request was successful, the `purchaseOrders` table was created as discussed in the *Setting Up the Database* section earlier should contain one more row. To make sure it does so, you can issue the following query from Oracle SQL\*Plus or any other command-line tool you use to communicate with the database:

```
CONN xmlusr/xmlusr

SELECT * FROM purchaseOrders;
```

When executed, the above `SELECT` statement should output the string representing the same PO XML document as the one shown in the *Building the Service Requestor* section earlier. If so, this means the PO Web service has worked successfully.

## Using XML Schemas with WSDL

The PO Web service discussed above represents a simplified example of a Web service provider. As mentioned, it receives a PO XML document as a simple string and saves it as it is in the database. In practice, of course, it is rarely as simple as this. When receiving an XML document of a specific structure from a consumer, a Web service wants to make sure that the received document has an appropriate structure, that is, the document conforms to a specific schema.

To solve this problem, WSDL allows you to include XML Schema definitions describing the data structures being transmitted between the service provider and its consumers. In WSDL, you can either enclose XML Schema data type definitions within the `types` element or import an XML schema stored in a separate file using the `import` statement. In the following sections, you will look at both these approaches.

## Including XML Schema Data Type Definitions in WSDL

In the PO Web service, you might want to modify its WSDL document so that it includes an XSD data type definition for the PO XML document received with the input message. Assuming that the PO Web service expects to receive a PO XML document having the same structure as the one shown in the *Building the Service Requestor* section earlier, the WSDL document describing the PO Web service might look now as follows. It is assumed that you save this document as `po_typed.wsdl` in the `WebServices/wsdl` directory in which you saved `po.wsdl` document discussed in the *Designing the WSDL Document* section previously.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions name="poService"
```

```
        targetNamespace="http://localhost/Webservices/wsd1/po/"
        xmlns:tns="http://localhost/Webservices/wsd1/po/"
        xmlns:http="http://schemas.xmlsoap.org/wsd1/http/"
        xmlns:soap="http://schemas.xmlsoap.org/wsd1/soap/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd1="http://localhost/Webservices/schema/"
        xmlns="http://schemas.xmlsoap.org/wsd1/">
<types>
  <schema targetNamespace="http://localhost/Webservices/schema/"
    xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="purchaseOrder">
      <complexType>
        <sequence>
          <element name="pono" type="xsd:string" />
          <element name="shipTo" type="xsd1:AddressType" />
          <element name="billTo" type="xsd1:AddressType"/>
          <element name="items" type="xsd1:LineItemsType"/>
        </sequence>
      </complexType>
    </element>
    <complexType name="AddressType">
      <sequence>
        <element name="name" type="xsd:string"/>
        <element name="street" type="xsd:string"/>
        <element name="city" type="xsd:string"/>
        <element name="state" type="xsd:string"/>
        <element name="zip" type="xsd:int"/>
        <element name="country" type="xsd:NMTOKEN" />
      </sequence>
    </complexType>
    <complexType name="LineItemsType">
      <sequence>
        <element minOccurs="1" maxOccurs="unbounded" name="item"
          type="xsd1:LineItemType" />
      </sequence>
    </complexType>
    <complexType name="LineItemType">
      <sequence>
        <element name="partId" type="xsd:int"/>
        <element name="quantity" type="xsd:decimal"/>
        <element name="price" type="xsd:decimal"/>
      </sequence>
    </complexType>
  </schema>
</types>
```

```
        </sequence>
    </complexType>
</schema >
</types>
<message name="getPlaceOrderInput">
    <part name="body" element="xsd1:purchaseOrder"/>
</message>
<message name="getPlaceOrderOutput">
    <part name="body" element="xsd:string"/>
</message>
<portType name="poServicePortType">
    <operation name="placeOrder">
        <input message="tns:getPlaceOrderInput"/>
        <output message="tns:getPlaceOrderOutput"/>
    </operation>
</portType>
<binding name="poServiceBinding" type="tns:poServicePortType">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeOrder">
        <soap:operation
            soapAction="http://localhost/Webservices/ch2/placeOrder"/>
        <input>
            <soap:body use="literal" />
        </input>
        <output>
            <soap:body use="literal" />
        </output>
    </operation>
</binding>
<service name="poService">
    <port name="poServicePort" binding="tns:poServiceBinding">
        <soap:address
            location="http://localhost/Webservices/ch2/SOAPServer_typed.php"/>
    </port>
</service>
</definitions>
```

As you can see, the `getPlaceOrderInput` message described in this document includes a `body` part representing an element of a complex XSD type, namely `xsd1:purchaseOrder`. This type is described in the XML schema defined within the `types` construct of the WSDL document. What this means is that a PO XML document passed to the `placeOrder` method as the input argument must now conform to the `xsd1:purchaseOrder` type definition.

## Importing XML Schemas into WSDL Documents

In the preceding section you saw how an XML schema containing data type definitions used for typing messages' contents can be added to a WSDL document. However, to achieve better reusability you might save that XML schema in a single file and then import it into the WSDL document. In this case, you won't have to modify your WSDL document when you modify a type definition in the imported XML schema. Instead, you will modify the document containing the schema, while leaving the WSDL document representing the contract between the service provider and its consumers untouched.

Returning to the WSDL document discussed in the preceding section, you first need to separate the XML schema enclosed within the `types` element. It is assumed that you save the following schema document as `po.xsd` in the `WebServices/schema` directory within the document directory of your Web server.

```
<?xml version='1.0'?>
<schema targetNamespace="http://localhost/WebServices/schema/po/"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:types1="http://localhost/WebServices/schema/po/">
  <element name="purchaseOrder">
    <complexType>
      <sequence>
        <element name="pono" type="string" />
        <element name="shipTo" type="types1:AddressType" />
        <element name="billTo" type="types1:AddressType" />
        <element name="items" type="types1:LineItemsType" />
      </sequence>
    </complexType>
  </element>
  <complexType name="AddressType">
    <sequence>
      <element name="name" type="string"/>
      <element name="street" type="string"/>
      <element name="city" type="string"/>
      <element name="state" type="string"/>
      <element name="zip" type="int"/>
      <element name="country" type="NMTOKEN" />
    </sequence>
  </complexType>
  <complexType name="LineItemsType">
    <sequence>
      <element minOccurs="0" maxOccurs="unbounded" name="item">
```

```

                                type="types1:LineItemType" />
    </sequence>
  </complexType>
  <complexType name="LineItemType">
    <sequence>
      <element name="partId" type="int"/>
      <element name="quantity" type="decimal"/>
      <element name="price" type="decimal"/>
    </sequence>
  </complexType>
</schema >

```

Now you can import the entire XML schema shown above into the WSDL document describing the PO Web service, rather than enclosing that schema in the `types` element in the WSDL document. To achieve this, you use the `import` WSDL element, modifying the `po_types.wsdl` document discussed in the previous section as shown below. It is assumed that you save this document as `po_imp.wsdl` in the `WebServices/wsdl` directory.

```

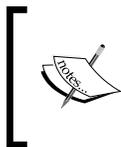
<?xml version="1.0" encoding="utf-8"?>
<definitions name="poService"
  targetNamespace="http://localhost/WebServices/wsdl/po/"
  xmlns:tns="http://localhost/WebServices/wsdl/po/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://localhost/WebServices/schema/po/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <import namespace="http://localhost/WebServices/schema/po/"
    location="http://localhost/WebServices/schema/po.xsd" />

  ...
</definitions>

```

In this example, you associate the `xsd1` namespace defined in the WSDL document with the PO XML schema stored in a separate file, using the `namespace` and `location` attributes of the `import` statement.



Looking through the XML schema and WSDL documents discussed here, you may notice that each of these documents uses a different prefix for the namespace whose URI is `http://localhost/WebServices/schema/po/`. In fact, you might use the same prefix here. However, it doesn't matter as long as the URI is the same.

## Getting Data Types Defined in the XML Schema

With a great number of XSD data types defined in the XML schema document or documents used by the WSDL definition, there is often a need to be able to look into those types from within the client code at run time.



In the design stage, the above is not a problem. Regardless of whether the WSDL document describing the service incorporates the XML schema in the way you saw in the *Including XML Schema Data Type Definitions in WSDL* section or imports one or more XML schemas as discussed in the *Importing XML Schemas into WSDL Documents* section, you can examine the XSD types used by looking either through the `types` section of the WSDL document or the imported XML schema documents. This is possible because service providers share their WSDL definitions with their consumers.

To meet this challenge, the PHP SOAP extension introduces the `__getTypes` method of the `SoapClient` class. To simply output the information about actual XSD data types, you might use `__getTypes` as follows:

```
<?php
...
$wsdl= "http://localhost/WebServices/wsdl/po_imp.wsdl";
$client = new SoapClient($wsdl);
print_r($client->__getTypes());
...
?>
```

The highlighted line in the above code will output the following array of structures that are representations of the XSD types defined in the `po.xsd` XML schema document imported into the `po_imp.wsdl` WSDL definition document:

```
Array
(
    [0] => struct purchaseOrder {
        string pono;
        AddressType shipTo;
        AddressType billTo;
        LineItemsType items;
    }
    [1] => struct AddressType {
        string name;
        string street;
        string city;
```

```
string state;
int zip;
NMTOKEN country;
}
[2] => struct LineItemsType {
    LineItemType item;
}
[3] => struct LineItemType {
    int partId;
    decimal quantity;
    decimal price;
}
)
```

While the above example simply outputs the array of structures returned by the `__getTypes` method, in a real-world application you might make practical use of this information. For example, you might dynamically build an input form based on these structures, so that a user could manually input data to be sent to the service. While constructing such a form, you might use information about the types of the fields of the returned structures when defining validation rules.

 To quickly build such a form, you might take advantage of the `PEAR::HTML_QuickForm` package. The discussion of this package, though, is outside the scope of this book. To find out more about `PEAR::HTML_QuickForm`, you can visit [http://pear.php.net/HTML\\_QuickForm](http://pear.php.net/HTML_QuickForm).

## Transmitting Complex Type Data

In the preceding example, you simply send a string representing a PO XML document as the input argument of the `placeOrder` method exposed by the PO Web service. Now that you have modified the WSDL document describing the PO Web service so that the PO Web service receives a PO XML document conforming to a specific structure, you cannot send that document as a simple string any more.

The following sections explain in detail how the complex type data structures are transmitted between SOAP nodes built with the PHP SOAP extension.

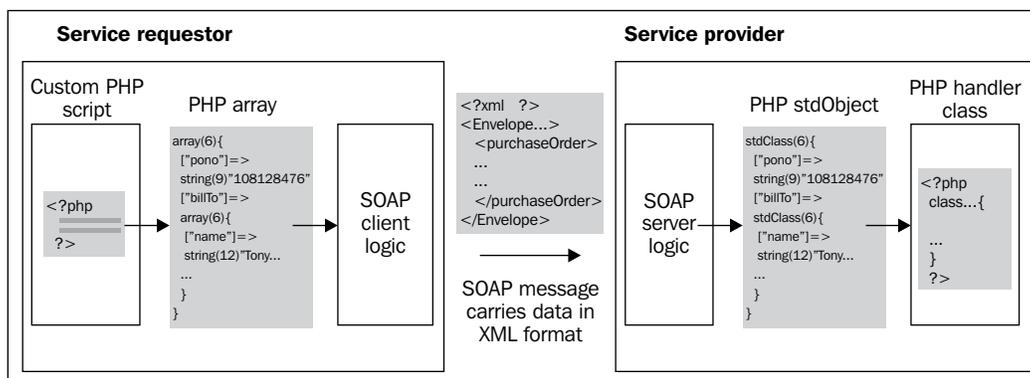
## Exchanging Complex Data Structures with PHP SOAP Extension

Like any other SOAP-based interfaces, service providers and service consumers built with the PHP SOAP extension use XML format when it comes to exchanging structured and typed data. However, in the case of the PHP SOAP extension you are not supposed to provide an XML document ready for transmitting. Instead, you provide an array having an appropriate structure and containing all the data to be sent. The SOAP software transforms this array to an XML document conforming to the specified type definition defined in the XML schema employed, and then sends that XML document wrapped in a SOAP envelope to the receiver. The receiver extracts the document from the SOAP envelope assuming that the receiver uses the PHP SOAP extension and converts it to an instance of the `stdClass` built-in PHP class.



In practice, a service requestor built with the PHP SOAP extension may send requests to a service provider built with other software, and also a PHP SOAP extension-based service provider may be consumed by a requestor built with a non-PHP tool. In either case, the requestor and provider will exchange the data organized as specified in the corresponding WSDL and XML schema documents. The details of manipulating the data structures being exchanged, though, will vary depending on the SOAP software specifics. In Chapter 5, for example, you will learn how a WS-BPEL process service handles complex type data arriving with request messages sent by consumers of that service. You also will learn how a WS-BPEL process handles complex data being sent to its partners.

Diagrammatically, the process of transmitting a complex data structure between two SOAP nodes built with the PHP SOAP extension might look like the following figure:



Generally, when you call a function exposed by the service, in the way you did in the `SoapClient.php` script discussed in the *Building the Service Requestor* section earlier, the instance of the `SoapClient` class assumes that you pass arrays as the arguments of that function.

 However, in the case of the `SoapClient.php` mentioned here you don't have to worry about this, since you send a simple string as the parameter of the exposed function.

For example, the following PHP array might be used as the argument of the `placeOrder` function exposed by the PO Web server described by the `po_imp.wsdl` document shown in the *Importing XML Schemas into WSDL Documents* section earlier:

```
array(4) {
  ["pono"]=>      string(9) "108128476"
  ["billTo"]=>   array(6) {
    ["name"]=>    string(12) "Tony Jamison"
    ["street"]=>  string(15) "24 Johnson Road"
    ["city"]=>    string(10) "Big Valley"
    ["state"]=>   string(2) "VA"
    ["zip"]=>     string(5) "23032"
    ["country"]=> string(2) "US"
  }
  ["shipTo"]=>   array(6) {
    ["name"]=>    string(13) "Janet Thomson"
    ["street"]=>  string(15) "11 Maple Street"
    ["city"]=>    string(12) "Small Valley"
    ["state"]=>   string(2) "VA"
    ["zip"]=>     string(5) "23037"
    ["country"]=> string(2) "US"
  }
  ["items"]=>    array(1) {
    ["item"]=>    array(2) {
      [0]=>       array(3) {
        ["partId"]=> string(3) "743"
        ["quantity"]=> string(1) "4"
        ["price"]=>  string(7) "10.5"
      }
      [1]=>       array(3) {
        ["partId"]=> string(3) "235"
        ["quantity"]=> string(1) "7"
        ["price"]=>  string(2) "15.75"
      }
    }
  }
}
```

We could pass the variable containing this array to the `placeOrder` function as the parameter like the following:

```
<?php
...
$wsdl= "http://localhost/Webservices/wsdl/po_imp.wsdl";
$client = new SoapClient($wsdl);
...
$result=$client->placeOrder($poarray);
...
?>
```

The SOAP software operating on the client side will transform this array into an XML document conforming to the `purchaseOrder` data type definition described in the `po.xsd` XML schema document shown in the *Importing XML Schemas into WSDL Documents* section, thus generating a PO XML document like that you saw in the *Building the Service Requestor* section earlier. This XML document is then wrapped in an SOAP envelope and sent to the server.

On the server side, the posted document is extracted from the SOAP envelope and by default is transformed to an instance of the `stdClass` built-in PHP class. You may look into that instance with the help of the `var_dump` standard PHP function and output the instance structure and data to a file as shown:

```
<?php
class purchaseOrder {
    function placeOrder($po) {
        ...
        ob_start();
        var_dump($po);
        $buffer = ob_get_flush();
        file_put_contents('buffer.txt', $buffer);
        ob_end_clean();
        ...
    }
}
?>
```

On inspecting the `buffer.txt` file you see that the instance of `stdClass` containing the data received by the server is similar in structure to the array processed and posted by the client, and contains the same actual data as that array. In this particular example, the instance of `stdClass` would look as follows:

```
object(stdClass)#2 (4) {
    ["pono"]=> string(9) "108128476"
    ["shipTo"]=> object(stdClass)#3 (6) {
        ["name"]=> string(13) "Janet Thomson"
```

```
["street"]=> string(15) "11 Maple Street"
["city"]=> string(12) "Small Valley"
["state"]=> string(2) "VA"
["zip"]=> int(23037)
["country"]=> string(2) "US"
}
["billTo"]=> object(stdClass)#4 (6) {
["name"]=> string(12) "Tony Jamison"
["street"]=> string(15) "24 Johnson Road"
["city"]=> string(10) "Big Valley"
["state"]=> string(2) "VA"
["zip"]=> int(23032)
["country"]=> string(2) "US"
}
["items"]=> object(stdClass)#5 (1) {
["item"]=> array(2) {
[0]=> object(stdClass)#6 (3) {
["partId"]=> int(743)
["quantity"]=> string(1) "4"
["price"]=> string(7) "10.5"
}
[1]=> object(stdClass)#7 (3) {
["partId"]=> int(235)
["quantity"]=> string(1) "7"
["price"]=> string(2) "15.75"
}
}
}
}
```

Examining the difference between the array and `stdClass` object discussed here, you may notice that the latter contains fields in an order that is different from that used in the former. Specifically, the first upper element called `pono` is followed by the `shipTo` element in the `stdClass` object but by the `billTo` element in the array. To understand why the order of the elements has changed, you need to come back to the `po.xsd` XML schema discussed in the *Importing XML Schemas into WSDL Documents* section. Looking through the schema, you may notice that the `purchaseOrder` XSD type assumes that the order of the upper-level elements in its type representations must be as follows:

1. `pono`
2. `shipTo`
3. `billTo`
4. `items`

As you no doubt have guessed, the SOAP client, while processing the input array containing the data being sent, applied the required changes to the input structure, changing the order of the elements so that the XML document being transmitted conforms to the `purchaseOrder` XSD type definition described in the `po.xsd` XML schema document.



It's interesting to note that if the input array discussed here contained some extra fields that did not have corresponding elements defined within the `purchaseOrder` XSD type, the `stdClass` object on the server side actually would not change. The fact is that the SOAP client not only makes sure that the elements in the XML document being sent are in the correct order, but also prevent unnecessary elements presented in the input array from being included in that document.

## Structuring Complex Data for Sending

Now that you know the basics of how the service requestors and services providers based on the PHP SOAP extension handle the data being exchanged, it's a good time to see how all this works in practice.

In this section, you will see an example of how you can prepare a complex type data structure being sent as the argument of the function exposed by a Web service. In the following section, you will see how to handle the received data on the service provider side.

Suppose you are building a service requestor that will take the information to be sent from a file holding the data in XML format. In this case, you need to create the code that will first read an XML document from the file, and then convert the uploaded XML document to a PHP array being specified as the argument of the function exposed by the service provider. To read a well-formed XML document from a file into a PHP structure that might be easily converted to an array, you might take advantage of the `simplexml_load_file` PHP function that reads the XML document from the file specified as the argument to an object of the `SimpleXMLElement` class. Once you have the XML document as an instance of `SimpleXMLElement`, you can convert it to an array with the help of the function as follows:

```
<?php
//File: obj2Arr.php
function obj2Arr($obj)
{
    $result = NULL;
    if(!is_array($obj))
    {
        if($var = get_object_vars($obj))
```

```

    {
        foreach($var as $key => $value)
            $result[$key] = obj2Arr($value);
    }
    else
        return $obj;
    }
    else
    {
        foreach($obj as $key => $value)
            $result[$key] = obj2Arr($value);
    }
    return $result;
}
?>

```

As you can see, the `obj2Arr` custom function takes the object to be converted as the argument, and may perform a number of recursive calls (calling itself), depending on the complexity of the structure being converted.

 Please note that the `obj2Arr` function shown above assumes that the `SimpleXMLElement` object passed in as the argument represents an XML document containing no attributes. Processing XML documents containing attributes will be discussed in the *Dealing with Attributes* section later.

With the `simplexml_load_file` and `obj2Arr` functions, the client script calling the `placeOrder` function might now look as follows. It is assumed that you save this script as `SoapClient_typed.php` in the `WebServices/ch2` directory.

```

<?php
//File: SoapClient_typed.php
require_once "obj2Arr.php";
$wsdl = "http://localhost/WebServices/wsdl/po_imp.wsdl";
$xmlDoc = simplexml_load_file('purchaseOrder.xml');
$xmlarr = obj2Arr($xmlDoc);
$client = new SoapClient($wsdl);
try {
    print $result=$client->placeOrder($xmlarr);
}
catch (SoapFault $exp) {
    print $exp->getMessage();
}
?>

```

However, before you can test this client code you need to create the SOAP server and the PHP handler class to handle requests coming from the client. Both are discussed in the next section.

## Converting SOAP Messages' Payloads to XML

As discussed previously, on the server side, assuming that the server is built with the PHP SOAP extension, the exposed methods of the PHP handler class receive their arguments carrying complex type data as instances of the `stdClass` class. So, in this particular example, the `placeOrder` method of the `purchaseOrder` PHP handler class will receive its argument as a `stdClass` object.

Suppose you want to convert the `stdClass` object received by the `placeOrder` method back to XML. To handle this task, you might want to create a custom class. Here is the code for the `obj2Dom` class that takes care of converting a `stdClass` to XML:

```
<?php
class obj2Dom {
    private $dom;
    private $rootNode;
    private $arrayName;

    public function __construct($rootElmName='root')
    {
        $this->dom = new DomDocument('1.0');
        $root = $this->dom->createElement($rootElmName);
        $this->rootNode = $this->dom->appendChild($root);
    }

    private function buildDom($result, $node) {
        $attrFlag=0;
        foreach($result as $key => $value) {
            if (!is_int($key)){
                $nodeName=$key;
            }
            else {
                $nodeName=$this->arrayName;
            }
            if (!is_object($value)){
                if (is_array($value)) {
                    $this->arrayName=$key;
                    $this->buildDom($value,$node);
                }
            }
        }
    }
}
```

```

        else {
            $elm = $this->dom->createElement($nodeName);
            $elm = $node->appendChild($elm);
            $txt = $this->dom->createTextNode($value);
            $txt = $elm->appendChild($txt);
        }
    }
    else {
        $elm = $this->dom->createElement($nodeName);
        $elm = $node->appendChild($elm);
        $this->buildDom($value, $elm);
    }
}
}
public function trans2Dom($result)
{
    $this->buildDom($result, $this->rootNode);
}
public function printDomTree()
{
    return $this->dom->saveXML();
}
}
?>

```



Like the `obj2Arr` function discussed in the preceding section, the `obj2Dom` class shown here assumes that the `stdClass` objects being converted represent XML documents that do not contain attributes. In the *Dealing with Attributes* section, though, you will see a modified version of `obj2Dom` that can handle XML documents containing attributes.

Once you have created the `obj2Dom` class, you can include the file containing it in the PHP handler script, and then use this class as follows. It is assumed that you save the following PHP handler class in the `purchaseOrder_typed.php` file.

```

<?php
//File purchaseOrder_typed.php
require_once 'obj2Dom.php';
class purchaseOrder {
    function placeOrder($po) {
        $obj = new obj2Dom('purchaseOrder');
        $obj->trans2Dom($po);
        $po=$obj->printDomTree();
        if(!$conn = oci_connect('xmlusr', 'xmlusr', '//localhost/XE')){
            throw new SoapFault("Server","Failed to connect to database");
        }
    }
}

```

```
};
$sql = "INSERT INTO purchaseOrders VALUES (:po)";
$query = oci_parse($conn, $sql);
oci_bind_by_name($query, ':po', $po);
if (!oci_execute($query)) {
    throw new SoapFault("Server","Failed to insert PO");
};
$msg='<rsltMsg>PO inserted!</rsltMsg>';
return $msg;
}
}
?>
```

In the `placeOrder` method shown above, you first create an instance of the `obj2Dom` custom class, passing `'purchaseOrder'` as the argument in order to explicitly set up the name of the root element in the XML document being built. Then, you call the `trans2Dom` method of the newly created instance, passing in the value of the argument received by the `placeOrder` method. As discussed previously, `placeOrder` is supposed to receive the `stdClass` object representing the PO document posted by a service consumer. The `trans2Dom` method will translate the `stdClass` object received as the argument into an instance of the `DomDocument` class. By calling the `printDomTree` method of the `obj2Dom` class in the next step, you obtain the generated XML document as a string, which then is inserted into `purchaseOrders` table in the database.

Now that you have created the PHP handler class that will translate an incoming structure representing a PO XML document back into XML format, you have to build a SOAP server that will receive and process requests coming from the client. It is assumed that you save the following server as `SoapServer_typed.php` in the `WebServices/ch2` directory.

```
<?php
//File: SoapServer_typed.php
require_once "purchaseOrder_typed.php";
$wsdl= "http://localhost/WebServices/wsdl/po_imp.wsdl";
$srvc= new SoapServer($wsdl);
$srvc->setClass("purchaseOrder");
$srvc->handle();
?>
```

Now you can test the client shown in the preceding section. To do this, you need to point your browser at `http://localhost/WebServices/ch2/SoapClient_typed.php`. If everything goes as planned, you will see a **PO inserted!** message in your browser. Otherwise, a SOAP fault message appears.

## Using PHP SOAP Extension Tracing Capabilities

In the development and testing stage, there's often a need to look at the incoming and outgoing SOAP messages. To look through the headers of the last SOAP request and response, you can use the `__getLastRequestHeaders` and `__getLastResponseHeaders` methods of a `SoapClient` instance respectively. To look through the entire messages representing the last SOAP request and response, you can use the `__getLastRequest` and `__getLastResponse` methods respectively, as shown in the following example:

```
<?php
//File: SoapClient_trace.php
require_once 'obj2Arr.php';
$wsdl = "http://localhost/Webservices/wsdl/po_imp.wsdl";
$xml = simplexml_load_file('purchaseOrder.xml');
$arr = obj2Arr($xml);
$client = new SoapClient($wsdl, array('trace' => 1));
try {
    print "RESULT:\n".$result=$client->placeOrder($arr)."\n";
}
catch (SoapFault $exp) {
    print $exp->getMessage();
}
print "REQUEST:\n".htmlspecialchars
        ($client->__getLastRequest())."\n";
print "RESPONSE:\n".htmlspecialchars
        ($client->__getLastResponse())."\n";
?>
```

To test the above script, you don't have to write another SOAP server or PHP handler class—those discussed in the preceding sections will do. So, you specify the same WSDL document as you did in the `SoapClient_typed.php` script discussed in the *Structuring Complex Data for Sending* section earlier.



If you recall, the physical part of the WSDL document describes the concrete characteristics of the Web service, including information about the concrete network address of the service provider.

If you execute the `SoapClient_trace.php` script as shown previously, it should return the following output (the output has been formatted for clarity and the PO XML document in the request has been cut down to save space):

**RESULT:**

PO inserted!

**REQUEST:**

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:
  SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ns1=
    "http://localhost/Webservices/schema/po/">
<SOAP-ENV:Body>
  <ns1:purchaseOrder>
    <pono>108128476</pono>
    ...
  </ns1:purchaseOrder>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

**RESPONSE:**

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:
  SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
  <body><rsltMsg>PO inserted!</rsltMsg></body>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

## Dealing with Attributes

In the preceding sections, you learned how an XML document can be transmitted via SOAP as a complex data structure, and then converted back to an XML format on the receiver side. It was assumed, though, that the document being transmitted contains no attributes. This section discusses how to deal with documents containing attributes.

On the client side, perhaps the safest way to go when it comes to dealing with an XML document containing attributes is to first convert the attributes to elements and then transform the document into an array, as discussed in the *Structuring Complex Data for Sending* section earlier. The SOAP software converting this array to XML to be transmitted as the payload of a SOAP message will generate an XML document conforming to a certain XSD type, as defined in the WSDL definition for this particular part of the message.

Turning back to the `po.xsd` XML schema document discussed in the *Importing XML Schemas into WSDL Documents* section, you might now use it as the basis for another XML schema document, changing it a bit by adding the `id` attribute to the `item` element. The highlighted line in the `po_attr.xsd` XML schema document shown below is the only difference between this document and the `po.xsd` document discussed previously (the `po_attr.xsd` document shown here has been cut down to save space):

```
<?xml version='1.0'?>
<schema targetNamespace="http://localhost/Webservices/schema/po/"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:types1="http://localhost/Webservices/schema/po/">
  <element name="purchaseOrder">
    ...

  <complexType name="LineItemType">
    <sequence>
      <element name="partId" type="int"/>
      <element name="quantity" type="decimal"/>
      <element name="price" type="decimal"/>
    </sequence>
    <attribute name="id" type="int"/>
  </complexType>
</schema >
```

 The full versions of the PHP scripts, WSDL definitions, XML schemas, and other documents discussed here can be found in the downloadable archive on the book's Web page. 

Now you can call the `placeOrder` SOAP function, passing as the argument the following array, which is the same as the one shown in the *Exchanging Complex Data Structures with PHP SOAP Extension* section, except for the `id` fields added:

```
array(4) {
  ["pono"]=>      string(9) "108128476"
  ...

  ["items"]=>     array(1) {
    ["item"]=>    array(2) {
      [0]=>      array(3) {
        ["id"]=>  string(2) "24"
        ["partId"]=> string(3) "743"
```

```

    ["quantity"]=> string(1) "4"
    ["price"]=>    string(7) "10.5"
  }
  [1]=>          array(3) {
    ["id"]=>      string(2) "25"
    ["partId"]=>  string(3) "235"
    ["quantity"]=> string(1) "7"
    ["price"]=>  string(2) "15.75"
  }
}
}
}
}

```



The following section explains how to convert attributes to elements in the XML documents to be transmitted, so that you can generate an array, like the one shown above, with the help of the `obj2Arr` function discussed in the *Structuring Complex Data* for sending section previously.

When generating the payload of the message to be sent, the SOAP software will automatically recognize the attributes in the input array and produce the appropriate XML document. In this particular example, you will have the following document as the payload (it has been cut down to save space):

```

<ns1:purchaseOrder>
...
<items>
  <item id="24">
    <partId>743</partId>
    <quantity>4</quantity>
    <price>15.5</price>
  </item>
  <item id="25">
    <partId>235</partId>
    <quantity>7</quantity>
    <price>15.75</price>
  </item>
</items>
</ns1:purchaseOrder>

```

As you can see, the SOAP software correctly generated `item` elements, according to the `LineItemType` complex type definition described in the XML schema document.

However, the most interesting thing about documents containing attributes is how these documents are handled on the receiver side, assuming the receiver of the message is built with the PHP SOAP extension.

When the message carrying this payload reaches the receiver (in this example, it's the PO Web service provider), the SOAP software operating on the receiver side will convert the payload to the following `stdClass` object before sending it to the `placeOrder` method of the `purchaseOrder` PHP handler class (the object has been cut down to save space):

```
object(stdClass)#2 (4) {
    ["pono"]=> string(9) "108128476"
    ...

    ["items"]=>
    object(stdClass)#5 (1) {
        ["item"]=>
        array(2) {
            [0]=>
            object(stdClass)#6 (4) {
                ["partId"]=> int(743)
                ["quantity"]=> string(1) "4"
                ["price"]=> string(7) "15.5"
                ["id"]=> int(24)
            }
            [1]=>
            object(stdClass)#7 (4) {
                ["partId"]=> int(235)
                ["quantity"]=> string(1) "7"
                ["price"]=> string(2) "15.75"
                ["id"]=> int(25)
            }
        }
    }
}
```

As you can see, the SOAP software operating on the SOAP server side treats attributes like elements when processing a payload representing an XML document. Obviously, if you now try to transform the above `stdClass` object back to XML, using methods of the `obj2Dom` class as discussed in the *Converting SOAP Messages' Payloads to XML* section earlier, you will have an XML document in which all attributes have been converted to elements.



In the following section, you will learn how to handle this problem by applying XSLT transformations to the XML documents derived from `stdClass` objects.

It's important to note that the above example shows only the case when the element containing the attributes also contains nested elements. But, what if the element containing the attributes represents a text node? For example, you might use the currency name as the attribute of the `price` element in the `purchaseOrder` document discussed here, as shown below:

```
<purchaseOrder>
...

<items>
  <item id="24">
    <partId>743</partId>
    <quantity>4</quantity>
    <price currency = "USD">15.5</price>
  </item>
  <item id="25">
    <partId>235</partId>
    <quantity>7</quantity>
    <price currency = "USD">15.75</price>
  </item>
</items>
</purchaseOrder>
```

The `price` element in the above snippet might be described by the highlighted type definition in the `po_attr_price.xsd` XML schema document shown below. It is assumed that you save this document in the `WebServices/schema` directory.

```
<?xml version='1.0'?>
<schema targetNamespace="http://localhost/WebServices/schema/po/"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:types1="http://localhost/WebServices/schema/po/">
  <element name="purchaseOrder">
  <element name="purchaseOrder">
  <complexType>
  <sequence>
    <element name="pono" type="string" />
    <element name="shipTo" type="types1:AddressType" />
    <element name="billTo" type="types1:AddressType"/>
    <element name="items" type="types1:LineItemsType"/>
```

```
    </sequence>
  </complexType>
</element>
<complexType name="AddressType">
  <sequence>
    <element name="name" type="string"/>
    <element name="street" type="string"/>
    <element name="city" type="string"/>
    <element name="state" type="string"/>
    <element name="zip" type="int"/>
    <element name="country" type="NMTOKEN" />
  </sequence>
</complexType>
<complexType name="LineItemsType">
  <sequence>
    <element minOccurs="0" maxOccurs="unbounded" name="item"
      type="types1:LineItemType" />
  </sequence>
</complexType>
<complexType name="LineItemType">
  <sequence>
    <element name="partId" type="int"/>
    <element name="quantity" type="decimal"/>
    <element name="price">
      <complexType>
        <simpleContent>
          <extension base="decimal">
            <attribute name="currency" type="string"/>
          </extension>
        </simpleContent>
      </complexType>
    </element>
  </sequence>
  <attribute name="id" type="int"/>
</complexType>
</schema >
```

As you can see, the above document is the same as the `po_attr.xsd` discussed previously, except for the highlighted definition describing the `price` element.

Now, if you call the `placeOrder` function to transmit the `purchaseOrder` document shown prior to the above XML schema document, you should pass the following array as the argument (it has been cut down to save space):

```
object(stdClass)#2 (4) {
  ["pono"]=> string(9) "108128476"

  ...

  ["items"]=>
  array(1) {
    ["item"]=>
    array(2) {
      [0]=>
      array(4) {
        ["id"]=> string(2) "24"
        ["partId"]=> string(3) "743"
        ["quantity"]=> string(1) "4"
        ["price"]=>
        array(2) {
          ["_"]=> string(4) "15.5"
          ["currency"]=> string(3) "USD"
        }
      }
      [1]=>
      array(4) {
        ["id"]=> string(2) "25"
        ["partId"]=> string(3) "235"
        ["quantity"]=> string(1) "7"
        ["price"]=>
        array(2) {
          ["_"]=> string(5) "15.75"
          ["currency"]=> string(3) "USD"
        }
      }
    }
  }
}
```

Note that each `price` element is represented as a two-field array in which the value of the `price` element is mapped to an `_` (underscore) field, and the `currency` attribute is mapped to the `currency` field.

In this example, the `stdClass` object generated by the SOAP server and then sent to the `placeOrder` method of the `purchaseOrder` PHP handler class as the argument is as follows (again, fields of the object that are unimportant to this discussion have been omitted to save space):



```
if (!is_int($key)) {
    $nodeName=$key;
}
else {
    $nodeName=$this->arrayName;
}
if ($attrFlag==1) {
    $node->setAttribute($nodeName,$value);
    continue;
}
if ($nodeName=='_') {
    $txt = $this->dom->createTextNode($value);
    $node->appendChild($txt);
    $attrFlag = 1;
    continue;
}
if (!is_object($value)) {
    if (is_array($value)) {
        $this->arrayName=$key;
        $this->buildDom($value,$node);
    }
    else {
        $elm = $this->dom->createElement($nodeName);
        $node->appendChild($elm);
        $txt = $this->dom->createTextNode($value);
        $elm->appendChild($txt);
    }
}
else {
    $elm = $this->dom->createElement($nodeName);
    $node->appendChild($elm);
    $this->buildDom($value,$elm);
}
}
```

Now, when invoked, the `buildDom` method shown above will correctly handle the `_` fields in the input `stdClass` object, creating attributes in the appropriate text-node elements of the resultant DOM document.

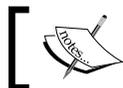
However, note that the updated `obj2Dom` class still doesn't provide you a mechanism to create attributes in the resultant document when it comes to dealing with attributes of elements containing nested elements. To handle this problem, you might add the following method to the `obj2Dom` class:

```
public function elmToAttr($nodeName)
{
    $items = $this->dom->getElementsByTagName($nodeName);
    $count= $items->length;
    for ($i = 0; $i < $count; $i++) {
        $node = $items->item(0);
        $parent = $node->parentNode;
        $parent->setAttribute($node->nodeName, $node->nodeValue);
        $parent->removeChild(
            $parent->getElementsByTagName($nodeName)->item(0));
    }
}
```

The above method takes the name of the element to be processed as the parameter. If the DOM tree contains more than one element with the name specified, this method will process each of these elements, converting such an element to an attribute of its parent element. You will see this method in action in the following section, when converting `id` elements in the `item` constructs to `id` attributes.

## Transforming XML Documents with XSLT

As you learned in the preceding section, if you need to send an XML document containing attributes from a SOAP node built with the PHP SOAP extension, then you first have to convert that document into an array in which both the attributes and elements of the document are represented as fields. To build such an array on an attribute-containing XML document, you might find it useful first to transform that document into the one containing no attributes but only elements. This is where XSLT (eXtensible Stylesheet Language Transformations) may come in very handy.



To learn more about XSLT, you can visit the following resource:  
<http://www.w3.org/TR/xslt>.

Suppose you need to transform the following PO XML document, say, saved as `po.xml`, so that the result document can be easily translated into an array to be passed as the argument to the `placeOrder` function exposed by the PO Web service.

```
<?xml version="1.0" ?>
<purchaseOrder>
  <pono>108128476</pono>
  ...
  <items>
    <item id="24">
```

```

    <partId>743</partId>
    <quantity>4</quantity>
    <price currency="USD">15.5</price>
  </item>
  <item id="25">
    <partId>235</partId>
    <quantity>7</quantity>
    <price currency="USD">15.75</price>
  </item>
</items>
</purchaseOrder>

```

Now, to transform this document into another one that in turn can be easily converted into an array to be passed to the `placeOrder` function as the argument, you can create an XSL stylesheet that might look as follows. It is assumed that you save this XSL stylesheet as `AttrsToElms.xsl` in the `WebServices/ch2` directory.

```

<?xml version='1.0' encoding='utf-8' ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:output method="xml"/>
  <xsl:template match="purchaseOrder">
    <purchaseOrder>
      <xsl:apply-templates/>
    </purchaseOrder>
  </xsl:template>
  <xsl:template match="@*|*|text()">
    <xsl:copy>
      <xsl:apply-templates select="@*|*|text()"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="items">
    <items>
      <xsl:for-each select="item">
        <xsl:element name="{name()}">
          <xsl:for-each select="@*">
            <xsl:element name="{name()}">
              <xsl:value-of select="."/>
            </xsl:element>
          </xsl:for-each>
          <xsl:for-each select="*">
            <xsl:choose>
              <xsl:when test="name()='price'">
                <price>

```

```
<_>
  <xsl:value-of select="."/>
</_>
<xsl:for-each select="@*">
  <xsl:element name="{name()}">
    <xsl:value-of select="."/>
  </xsl:element>
</xsl:for-each>
</price>
</xsl:when>
<xsl:otherwise>
  <xsl:element name="{name()}">
    <xsl:value-of select="."/>
  </xsl:element>
</xsl:otherwise>
</xsl:choose>
</xsl:for-each>
</xsl:element>
</xsl:for-each>
</items>
</xsl:template>
</xsl:stylesheet>
```

In the first highlighted block you transform all the attributes of the `item` element being processed into nested elements of this `item` element.

In the second highlighted block you process the elements nested in the `item` elements, performing conditional processing with the `xsl:choose` construct. Specifically, when the nested element being processed is `price`, all its attributes are transformed into elements nested in `price`, and its value is wrapped in the `_` element. Otherwise, the `item`'s nested element being processed remains the same as before.

To test the XSL stylesheet, create the following script:

```
<?php
//File: XSLTest.php
$xml = new DOMDocument();
$xml->load('po.xml');
$xsl = new DOMDocument();
$xsl->load('AttrsToElms.xsl');
$proc = new XSLTProcessor;
$proc->importStyleSheet($xsl);
print $proc->transformToXML($xml);
?>
```

If you execute this script, it should produce the following document:

```
<?xml version="1.0" ?>
<purchaseOrder>
  <pono>108128476</pono>

  ...

  <items>
    <item>
      <id>24</id>
      <partId>743</partId>
      <quantity>4</quantity>
      <price>
        <_>15.5</_>
        <currency>USD</currency>
      </price>
    </item>
    <item>
      <id>25</id>
      <partId>235</partId>
      <quantity>7</quantity>
      <price>
        <_>15.75</_>
        <currency>USD</currency>
      </price>
    </item>
  </items>
</purchaseOrder>
```

If you see the above document in your browser, it means the XSL transformation performed within the `XSLtest.php` script has been successfully applied, and everything works as expected. If so, you can move on and use this mechanism in a SOAP client script to transform the `po.xml` document shown at the beginning of this section to the above XML document, the one that is then translated into the array to be passed to the `placeOrder` function as the argument.

For example, you might create the following script and save it as `SoapClient_attr_price.php` in the `WebServices/ch2` directory.

```
<?php
//File: SoapClient_attr_price.php
require_once "obj2Arr.php";
$wsdl = "http://localhost/WebServices/wsdl/po_attr_price.wsdl";
```

```
$xml = new DOMDocument();
$xml->load('po.xml');
$xmls1 = new DOMDocument();
$xmls1->load('AttrsToElms.xsl');
$proc = new XSLTProcessor;
$proc->importStyleSheet($xmls1);
$poxml = $proc->transformToXML($xml);
$xmlldoc = simplexml_load_string($poxml);
$xmlarr = obj2Arr($xmlldoc);
$client = new SoapClient($wsdl);
try {
    print $result=$client->placeOrder($xmlarr);
}
catch (SoapFault $exp) {
    print $exp->getMessage();
}
?>
```

As you can see, the above script starts by performing the XSL transformation, the same as the one you saw in the `XSLTest.php` script earlier. Next, it loads the resultant document into a SimpleXML object, which is then transformed into an array being passed to the `placeOrder` function as the argument.

However, before you can execute the above SOAP client script you need to create a SOAP server and PHP handler class to handle responses from the client.

When creating the `SoapServer_attr_price.php` SOAP server script, you can use the `SoapServer.php` script discussed in the *Building the SOAP Server* section as the basis, changing the included PHP handler class to `purchaseOrder_attr_price.php` and the WSDL document location to `http://localhost/Webservices/wsd1/po_attr_price.wsdl`.



If you don't have the `po_attr_price.wsdl` created, you should build it now. As the base, you can use the `po_imp.wsdl` document discussed in the *Importing XML Schemas into WSDL Documents* section, importing the `po_attr_price.xsd` XML schema discussed in the *Dealing with Attributes* section and pointing the `soap:address` location attribute to `http://localhost/Webservices/ch2/SOAPServer_attr_price.php`.

The `purchaseOrder_attr_price.php` script containing the handler class should look as follows:

```
<?php
//File purchaseOrder_attr_price.php
require_once 'obj2Dom.php';
class purchaseOrder {
    function placeOrder($po) {
        $obj = new obj2Dom('purchaseOrder');
        $obj->trans2Dom($po);
        $obj->elmToAttr('id');
        $po=$obj->printDomTree();
        if(!$conn = oci_connect('xmlusr', 'xmlusr', '//localhost/XE')){
            throw new SoapFault("Server","Failed to connect to database");
        };
        $sql = "INSERT INTO purchaseOrders VALUES (:po)";
        $query = oci_parse($conn, $sql);
        oci_bind_by_name($query, ':po', $po);
        if (!oci_execute($query)) {
            throw new SoapFault("Server","Failed to insert PO");
        };
        $msg='<rsltMsg>PO inserted!</rsltMsg>';
        return $msg;
    }
}
?>
```

As you can see, the `placeOrder` function is similar to the one in the `purchaseOrder_typed.php` script discussed in the *Converting SOAP Messages' Payloads to XML* section earlier. The only difference is that you utilize the `elmToAttr` method of the `obj2Dom` instance here, passing `'id'` as the argument.

 As an alternative to the `elmToAttr` method here, you might apply an XSL transformation to the resultant XML document returned by the `printDomTree` method, converting `id` elements into attributes, as they were in the original document.

Now, if you execute the `SoapClient_attr_price.php` script, the `placeOrder` function should insert into the `purchaseOrders` table the `po.xml` document shown at the beginning of this section.

## Extending PHP SOAP Extension Predefined Classes

You can extend predefined classes of the PHP SOAP extension as needed. Here is an example of how you might extend the `SoapServer` class. It is assumed that you save this script as `SoapServer_ext.php` in the `WebServices\ch2`:

```
<?php
//File: SoapServer_ext.php
require_once "purchaseOrder.php";
class MySoapServer extends SoapServer {
    var $client;
    function __construct($wsdl1, $wsdl2) {
        parent::__construct($wsdl1);
        $this->client = new SoapClient($wsdl2);
    }
    function handle() {
        ob_start();
        parent::handle();
        $buf=ob_get_contents();
        ob_get_flush();
        $buf=html_entity_decode($buf);
        $env = simplexml_load_string($buf);
        $rslt= $env->xpath('//rsltMsg');
        if ($rslt==null) {
            $rslt= $env->xpath('//faultstring');
        }
        $this->client->regOrder(htmlentities((string) $rslt[0]));
    }
}
$wsdl1= "http://localhost/WebServices/wsdl/po_ext.wsdl";
$wsdl2= "http://localhost/WebServices/wsdl/reg.wsdl";
$srvc= new MySoapServer($wsdl1, $wsdl2);
$srvc->setClass("purchaseOrder");
$srvc->handle();
?>
```

The `MySoapServer` class extending the `SoapServer` predefined class overrides the constructor and the `handle` method of the parent class. The overridden constructor takes links to the two WSDL documents as the parameters, and creates a `SoapClient` instance that is then used in the overridden `handle` method to invoke the `regOrder` SOAP function.

Before you put this SOAP server script into action, you have to create a few other scripts and documents. First of all, make sure to create the `po_ext.wsdl` and `reg.wsdl` documents used here.

To create the `po_ext.wsdl` document, you can use the `po.wsdl` file discussed in the *Designing the WSDL Document* section as the base. The only thing you have to change is the value of the `location` attribute in the `soap:address` element within the service definition of the document. In particular, you should specify the following URL: `http://localhost/WebServices/ch2/SOAPServer_ext.php`. In the case of `reg.wsdl`, you should specify the following value for the `soap:address` location attribute: `http://localhost/WebServices/ch2/SOAPServer_reg.php`.

The next step is to create the `SOAPServer_reg.php` SOAP server script that will be automatically invoked each time the overridden `handle` method of `MySoapServer` is called. The `SOAPServer_reg.php` should look like the following:

```
<?php
//File: SoapServer_reg.php
$wsdl= "http://localhost/WebServices/wsdl/reg.wsdl";
require_once "reg.php";
$srv = new SoapServer($wsdl);
$srv->setClass("reg");
$srv->handle();
?>
```

As you can see, the above SOAP server exposes methods of the `reg` custom class. So, make sure to create the `reg` class. It might look like the following:

```
<?php
//File reg.php
class reg {
    function regOrder($reginfo) {
        if(!$conn = oci_connect('xmlusr', 'xmlusr', '://localhost/XE')){
            throw new SoapFault("Server","Failed to connect to
                                database");
        };
        $sql="INSERT INTO regDocs VALUES(SYSDATE, :reginfo)";
        $query = oci_parse($conn, $sql);
        oci_bind_by_name($query, ':reginfo', $reginfo);
        if (!oci_execute($query)) {
            throw new SoapFault("Server","Failed to execute query");
        };
        $msg='Ok!';
        return $msg;
    }
}
?>
```

Looking through this code, you may notice that the `regOrder` method takes one parameter and inserts it into the `regDocs` database table. So you need to create the `regDocs` table before you can use `regOrder`. This can be done from SQL\*Plus by issuing the following statements:

```
CONN xmlusr/xmlusr;

CREATE TABLE regDocs (
    dateTime DATE,
    msg VARCHAR2(100)
);
```

Finally, you have to create the `SoapClient_ext.php` script that will call the `placeOrder` method of the `purchaseOrder` class exposed by the `SoapServer_ext.php` SOAP server script. The `SoapClient_ext.php` is almost the same as the `SoapClient.php` script discussed in the *Building the Service Requestor* section, except for the WSDL document specified. In the case of `SoapClient_ext.php`, you should specify `http://localhost/WebServices/wsd1/reg.wsd1` as the WSDL document.

Now, if you execute the `SoapClient_ext.php` script, you should see a **PO inserted!** message. Then, you can check out the `regDocs` table by issuing the following statements from SQL\*Plus:

```
CONN xmlusr/xmlusr;

SELECT * FROM regDocs;
```

The above should return output that might look as follows:

```
DATE TIME      MSG
-----
02-APR-07     PO inserted!
```

## Defining Parameter-Driven Operations

As mentioned in Chapter 1, using parameter-driven service operations allows you to invoke the required piece of underlying logic depending on the arguments passed in. So, you can expose a single operation—passing parameters identifying what actually has to be done. This section shows a simple example of using this technique. You'll build a Web service that exposes a single function, namely `getOrder`. This function takes two parameters: the ID of a `purchaseOrder` and the parameter identifying what you want to receive: the document itself or its status.

To start with, you need to create a database table to store the orders' status information. Here are the statements you should issue from SQL\*Plus:

```
CONN xmlusr/xmlusr;

CREATE TABLE poStatusInfo (
    pono VARCHAR2(9),
    status VARCHAR2(15)
);

INSERT INTO poStatusInfo VALUES (
    '108128476',
    'shipped'
);

COMMIT;
```

The next step is to create the underlying service logic. To achieve this, create the following class and save it in the `orderInfo.php` file:

```
<?php
//File orderInfo.php
class orderInfo {
    function getOrder($pono, $par) {
        if(!$conn = oci_connect('xmlusr', 'xmlusr', '//localhost/XE')){
            throw new SoapFault("Server","Failed to connect to
                database");
        };
        switch ($par) {
            case 'doc':
                $sql="SELECT doc FROM purchaseOrders WHERE
                    extractValue(XMLType(doc), '/purchaseOrder/pono')=:pono
                    AND rownum=1";
                break;
            case 'status':
                $sql="SELECT status FROM poStatusInfo WHERE pono=:pono";
                break;
        }
        $query = oci_parse($conn, $sql);
        oci_bind_by_name($query, ':pono', $pono);
```

```
    if (!oci_execute($query)) {
        throw new SoapFault("Server","Failed to execute query");
    };
    oci_fetch($query);
    $rslt = oci_result($query, strtoupper($par));
    return $rslt;
}
}
?>
```

As you can see, `orderInfo` uses a different SQL statement querying the database, depending on the value passed in with the second parameter.

Next, you need to create the WSDL document describing the Web service discussed here. Here is the WSDL document being used in this example. It is assumed that you save it as `po_params.wsdl` in the `WebServices/wsdl` directory:

```
<?xml version="1.0" encoding="utf-8"?>
<definitions name="poInfoService"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace=
        "http://localhost/WebServices/wsdl/poInfo">
    <message name="getOrderInfoInput">
        <part name="pono" element="xsd:string"/>
        <part name="par" element="xsd:string"/>
    </message>
    <message name="getOrderInfoOutput">
        <part name="body" element="xsd:string"/>
    </message>
    <portType name="poInfoServicePortType">
        <operation name="getOrder">
            <input message="tns:getOrderInfoInput"/>
            <output message="tns:getOrderInfoOutput"/>
        </operation>
    </portType>
    <binding name="poInfoServiceBinding"
        type="tns:poInfoServicePortType">
        <soap:binding style="rpc"
            transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="getOrder">
            <soap:operation
                soapAction="http://localhost/WebServices/ch2/getOrder"/>
```

```
<input>
  <soap:body use="literal"/>
</input>
<output>
  <soap:body use="literal"/>
</output>
</operation>
</binding>
<service name="poInfoService">
  <port name="poInfoServicePort"
    binding="tns:poInfoServiceBinding">
    <soap:address
location="http://localhost/WebServices/ch2/SOAPServer_params.php"/>
    </port>
  </service>
</definitions>
```

Note that the `getOrderInfoInput` message in the above document consists of two parts that represent parameters of the `getOrder` operation described in the document.

To expose the `getOrder` method of the `orderInfo` class, you use the following SOAP server script, saved as `SOAPServer_params.php`:

```
<?php
//File: SoapServer_params.php
require_once "orderInfo.php";
$wsdl= "http://localhost/WebServices/wsdl/po_params.wsdl";
$srvc= new SoapServer($wsdl);
$srvc->setClass("orderInfo");
$srvc->handle();
?>
```

Once you've done all that, you can test the Web service. To do this, you might build and then execute the following SOAP client.

```
<?php
//File: SoapClient_params.php
$wsdl = "http://localhost/WebServices/wsdl/po_params.wsdl";
$client = new SoapClient($wsdl);
$pono='108128476';
$par='doc';
try {
  print $result=$client->getOrder($pono, $par);
}
catch (SoapFault $exp) {
  print $exp->getMessage();
}
?>
```

When executed, this script should output the entire PO XML document whose `pono` is `108128476`. However, if you specify `$par='status'` in the above code, you will get only the message saying `shipped`.

## Summary

As you have learned in this chapter, creating service providers and service requestors with the PHP SOAP extension is quite easy in most cases – you simply manipulate predefined SOAP classes. Things become a bit more complicated when it comes to transmitting complex type data – especially if you are dealing with XML documents whose elements contain attributes. This is where intermediate transformations are required. We looked at how to employ a custom PHP class to perform such transformations and how to use standard XSLT mechanism.

In this chapter, you also learned how to extend predefined classes of the PHP SOAP extension and how standard methods of these classes can be overridden to suit the needs of your application. The chapter wrapped up by explaining how to build Web services supporting parameter-driven operations.

