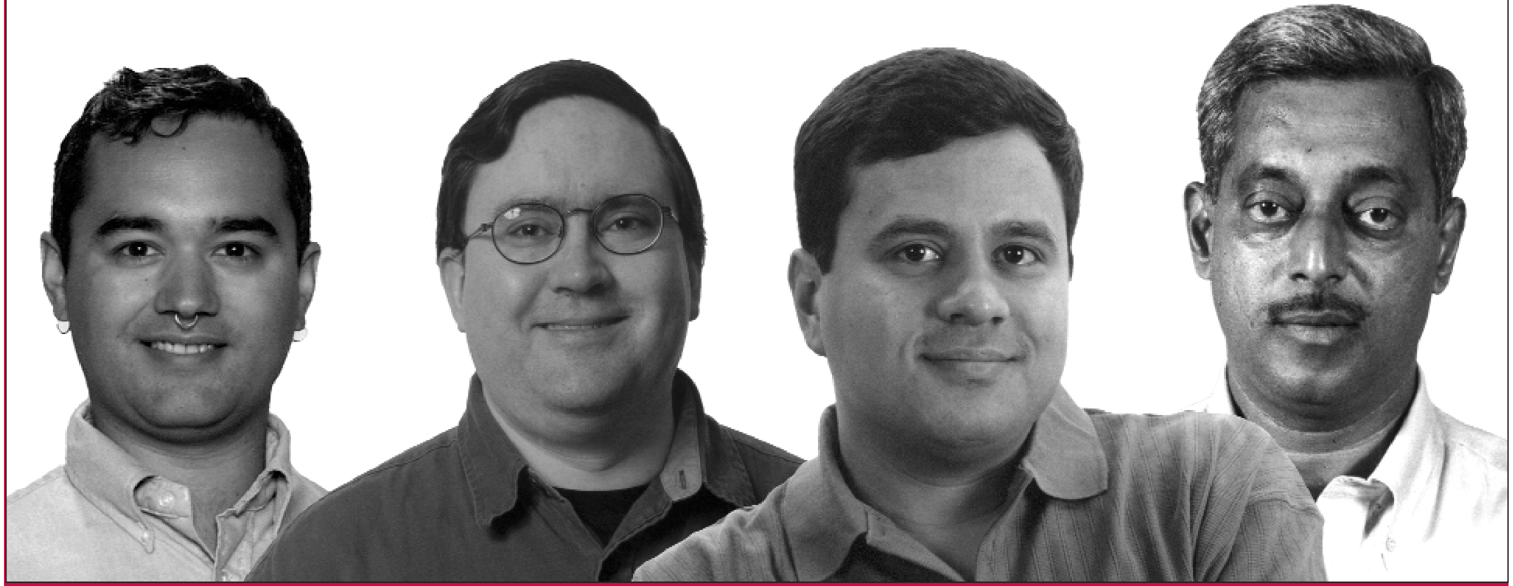
Programmer to Programmer ™



Professional Open Source Web Services

Dietrich Ayala, Christopher Browne, Vivek Chopra,
Dr. Poornachandra Sarang, Kapil Apshankar, Tim McAllister

wrox
Programmer to Programmer

Wrox technical support at: **support@wrox.com**
Updates and source code at: **www.wrox.com**
Peer discussion at: **p2p.wrox.com**

Summary of Contents

8

PHP and Web Services

PHP Hypertext Processor (PHP) is a widely-used general-purpose scripting language that is specially suited for web development as it was designed to work on the Web and can be embedded into HTML. It is a procedural language, with some object-oriented capabilities and has syntax similar to C, Perl, and Java. PHP was first released on June 8, 1995. PHP version 4, which was released in May 2000, was a major milestone, with its explosive performance increase and many features, such as native session support.

In this chapter, we will first present PHP with its advantages for Web Services development. Then we will cover the installation and configuration of PHP on a Unix system.

The second section will concentrate on SOAP usage in PHP. We'll look at the NuSOAP toolkit, and learn how to use it for both consumption and deployment of SOAP Web Services. We'll also explore NuSOAP's functionality for several advanced Web Services scenarios such as using HTTP proxy, SOAP over HTTPS, and document style messaging. We'll discuss some of the issues facing Web Service development in PHP, such as security and language to data type mapping, and how NuSOAP solves them.

The final section covers XML-RPC in PHP. We'll explore the particular features of XML-RPC, as well as discuss scenarios for using XML-RPC versus using SOAP. We will then use the Useful, Inc. implementation to create XML-RPC client and server applications.

PHP Features

Several of PHP's features are advantageous for Web Service development. The first would be its object-oriented programming capabilities. This may be limited in comparison to other natively object-oriented languages, but it provides the support needed to create extendable and reusable tools for PHP. It also allows SOAP and XML-RPC toolkits to be split into a group of classes each supporting parts of the entire Web Service transaction, but which on its own can be reused to accomplish the plethora of different kinds of transactions that are possible and tasks that may be necessary, such as native type serialization, network operations, XML parsing, and XML generation.

Another advantage of PHP is its XML support. The Expat parser is bundled with PHP, providing SAX capability out of the box. There are several PHP extensions available for expanded XML functionality, such as the `domxml` extension that use the libxml library to provide DOM, Xpath, and Xlink support. The `xslt` extension is a wrapper for different third-party XSLT libraries such as Sablotron and libxslt. There are also experimental extensions for XML-RPC and SOAP.

A very helpful PHP extension for Web Service development is the `CURL` extension, described as a Client URL Library. CURL allows you to communicate via various different protocols such as HTTP, HTTPS, FTP, telnet, and LDAP. The HTTPS support is particularly helpful for Web Service usage in PHP as it allows a (Web Service) client to make a secure connection with the server.

PHP and Web Services

PHP Web Service newbies can experiment with Web Services using PHP by writing only a few lines of code, without having to set up a special environment, and without needing to know anything about how Web Services actually work. But this ease-of-use is a double-edged sword and general advice is to know as much as possible about the technologies used, to stay abreast of security issues especially, considering that by nature Web Services are exposed to the network.

PHP is already broadly deployed for data-centric web applications. If you are planning to develop Web Services using PHP, your PHP-driven web sites may have components that may be reused by exposing their methods using XML-RPC or SOAP. The conversion from wrapping your data in HTML to serving it as SOAP messages is trivial using the tools available today.

An excellent reason to use PHP for Web Services is that sometimes it's the only choice you've got. Web sites that are hosted by PHP hosting services and want to use Web Services to share data with their partners to access services cannot do so, since they have no control over their PHP installation. They cannot install Java, don't have compiler access, and have no permissions to restart Apache if assuming they could install new software on the server. Now users in such a scenario can easily consume and deploy Web Services in short order if the server has PHP.

Currently PHP does not have standard SOAP or XML-RPC support. There are several different implementations each of SOAP and XML-RPC and the stable ones are written in pure PHP. There has been a recent push by developers to create standard support for SOAP in PHP. Several of the PHP core developers have expressed interest in seeing this happen, and a PHP-SOAP mailing list was started to pursue the effort. Hopefully, this activity will culminate in a SOAP extension that will be compiled into PHP by default, thereby being available out of the box to developers everywhere, regardless of system-related limitations such as permissions or compiler access.

The XML-RPC support in PHP has increased with Epinions opening the source to their XML-RPC extension. The extension has been in the main PHP CVS repository for quite some time, yet is still marked experimental and is not recommended for production use.

There are several advantages and disadvantages in using SOAP against XML-RPC for Web Services:

- Strong and extensible typing – SOAP allows user- and schema-defined types to be passed in SOAP message bodies. XML-RPC is very limited in the types of data it supports.
- Character set – SOAP allows the user to define the character set used in the message (like US-ASCII, UTF-8, UTF-16) while XML-RPC does not.
- Specifies recipient – SOAP can specify the recipient of a message, as well as can be routed through intermediaries.
- Failure if not understood – SOAP allows the sender to force a recipient to fail if the recipient cannot understand its message.
- Ease of use – Both PHP and XML-RPC have a short learning curve. It is easy for users new to PHP and Web Services to consume and deploy Web Services quickly and easily.
- Simplicity by design – XML-RPC was designed to be simple; to accomplish remote procedure calls with a minimum of infrastructure. SOAP is overkill for many Web Services that pass only simple or complex values.

Configuring PHP

Installing and configuring PHP for the examples in this chapter requires the Apache web server. To compile and configure PHP for Apache, we first download the PHP source from http://www.php.net/downloads.php and unpack it. Then change your directory to the root directory of the PHP source distribution. To run the configure script type `./configure` on the command line. You can add options to this command to compile PHP with non-standard functionality. Some useful options are listed below:

- **Apache**
  To enable PHP to be run as an Apache module, use the `--with-apxs` option, adding the path to your APXS binary as is appropriate for your server. For example:
  `--with-apxs=/www/bin/apxs`
- **DOMXML**
  This extension provides XML DOM, Xpath, and Xlink functionality. Currently, this extension is not required in Web Services, but is useful for reading and manipulation XML documents. Domxml support requires a version of the libxml libraries to be present on the system (with a version number greater than or equal to 2.4.2). For example, `--with-dom=DIR`. Here `DIR` is the `libxml` install directory, which defaults to `/usr`.
- **XSLT**
  This extension is also not necessary for using Web Services with PHP, but is useful for transforming XML data returned in Web Service messages into other types of documents. The configure option for this extension is `--enable-xslt --with-xslt-sablot`. The Sablotron XSLT library from Gingerall (http://www.gingerall.com/) must be installed in a location where your compiler can find it, usually `/usr/lib` or `/usr/local/lib`.

- **CURL**
  The CURL extension provides the ability to initiate secure socket connections using SSL. This is a necessary extension if you want to conduct SOAP client operations using SSL. The configure option is `--with-curl=DIR` where `DIR` is the location of the CURL libraries on your system. The CURL extension requires version 7.0.2-beta or higher of the CURL libraries.

Finally, run `make` and then `make install` on the command line, and the installation is complete. The final installation process on the command line using the previous options would look like this:

```
> cd PHP-xxx
> ./configure --with-apxs=/www/bin/apxs --with-dom=/usr/local/lib --enable-xslt
--with-xslt-sablot --with-curl=/usr/local/lib
> make
> make install
```

The PHP web site at http://www.php.net/manual/en/installation.php provides instructions for installing PHP with many other options on a variety of platforms.

PHP Web Services Using NuSOAP

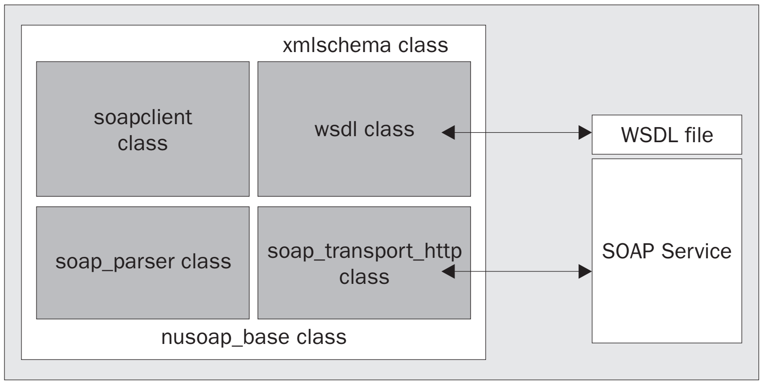
NuSOAP is a collection of PHP classes that allow users to send and receive SOAP messages over HTTP. NuSOAP, formerly known as SOAPx4, is distributed by the NuSphere Corporation (http://www.nusphere.com/). It is open source, licensed under the GNU LGPL. SOAPx4 has been used as the core of several Web Services toolkits for PHP, including PEAR-SOAP and Active State software's simple Web Services API project.

One of the benefits of NuSOAP is that it's not a PHP extension, but is written in pure PHP. This means that nearly all PHP developers, regardless of the web server or permissions restrictions, can use NuSOAP.

NuSOAP is a component-based Web Services toolkit. It employs a base class that provides utility methods such as variable and envelope serialization, as well as namespace information and mappings of different types to different namespaces. Web Service interaction is achieved through a high-level client class called `soapclient`. This high-level class allows users to specify options such as HTTP authorization credentials, HTTP proxy information, as well as managing the actual sending and receiving of the SOAP message itself. It uses several helper classes to accomplish the sending and receiving of SOAP messages.

SOAP operations may be executed by passing the name of the operation you'd like to execute to the `call()` method. If the service to be consumed provides a WSDL file, the `soapclient` class takes the URL of the WSDL file as an argument to its constructor, and uses the `wsdl` class to parse the WSDL file and extract all the data. The WSDL class has methods that extract data on a per-operation or per-binding basis.

The `soapclient` uses this data from the WSDL file to encode parameters and create the SOAP envelope when the user executes a call to a service. When the call is executed, the `soapclient` class uses the `soap_transport_http` class to send the outgoing message and receive the incoming message. The incoming message is parsed using `soap_parser` class. The diagram opposite describes the process of consuming a SOAP Web Service using NuSOAP.



If the Web Service to be consumed does not provide a WSDL file, then the process is different. The URL of the service is passed to the `soapclient` class constructor. Operations are still executed using the call method of the `soapclient` object, but details that are otherwise provided by the WSDL file must be passed as arguments. Parameters that are custom types can be represented using the `soapval` class, which allow users to customize a parameter's serialization.

Installation and Configuration

The installation of NuSOAP is a pretty straightforward process. It can be achieved in a few steps as follows:

- Download the files from http://dietrich.ganx4.com/nusoap/.
- Extract the file `nusoap.php` from its zip.
- For easy access, copy the classes to a location in your include path, or into the directory in which you'll be using the classes.
- Include the class in your script. The path to `nusoap.php` can be relative or absolute:

```
include('nusoap.php');
```

In this example, we have used the `include()` function to include the NuSOAP classes in our script. This function will generate a warning if the path to `nusoap.php` is incorrect, but will continue to process the rest of the script. There are several other alternatives:

- `require()`
  This function is identical to `include()`, but handles failure by emitting a fatal error, which will halt processing of the script
- `require_once()`
  Identical to `require()`, except that if the file to be included is already included in the script, it will not repeat the inclusion
- `include_once()`
  Identical to `include()`, except that if the file to be included has been previously included in the script, it will not include it again

Language to Data Mapping

SOAP and WSDL both make heavy use of the data types described in the XML Schema specification. This can be problematic since PHP does not natively support most of the data types defined in the specification. Also, the XML Schema data types are fine-grained and explicitly defined, whereas PHP is a loosely-typed language and will convert data types automatically when deemed appropriate. NuSOAP solves this problem at three different levels:

- In WSDL, NuSOAP's `soapclient` class will encode a value's type according to the type specified in the WSDL document.
- The NuSOAP `soapval` class allows users to explicitly define a value's type.
- If no type is explicitly declared when instantiating a `soapval` object, NuSOAP will analyze the value passed to it using PHP's built-in variable introspection functions as well as regular expressions and other means where necessary, and classify it as a valid XML Schema data type, or a valid type as described in Section 5 of the SOAP 1.1 specification.

A Simple PHP SOAP Client Example

In this example we will pass a string to a SOAP server that echoes the same string back. This example demonstrates the basic process of creating a SOAP client, calling a SOAP service and passing it parameters, and receiving the response. We will name this file `echoStringClient.php`.

In this chapter to start a script in a standard way would be to include the NuSOAP classes first. We'll use the `require()` function in our examples because we'd like the script to halt execution if it cannot find the NuSOAP file:

```
<?php
require('nusoap.php');
```

Let's create a variable for the string we'd like to send.

```
$myString = 'Dietrich Ayala';
```

Our parameters must be passed as an array to the SOAP client, so let's create one:

```
$parameters = array($myString);
```

Now we can instantiate the `soapclient` object. It takes the URL of the server as an argument to its constructor:

```
$s = new soapclient('http://localhost/wrox/nusoap/echoStringServer.php');
```

This step is where all the magic takes place. Using the `call()` method, we tell the `soapclient` object which service we'd like to access, then pass our array of parameters, and the method then returns the response from the server. This response is a PHP native type, such as a string, integer, or array. It is the result of the decoding that takes place when NuSOAP parses the response message:

```
$result = $s->call('echoString',$parameters);
```

NuSOAP offers error detection through the `getError()` method. If an error has occurred this method returns a string describing the error, and returns `false` otherwise. In our example, we print our result after checking for errors if there are none. If there are errors, we'll print the error message:

```
if(!$err = $s->getError()){
    echo 'Result: '.$result;
} else {
    echo 'Error: '.$err;
}
```

This final bit of code is very helpful for debugging SOAP operations. The `request` and `response` properties of the `soapclient` class contain strings of the respective messages, including the HTTP headers sent with each:

```
echo '<xmp>'.$s->request.'</xmp>';
echo '<xmp>'.$s->response.'</xmp>';
?>
```

SOAP Request and SOAP Response

Here is the request message from the previous example:

```
POST /wrox/nusoap/echoStringServer.php HTTP/1.0
User-Agent: NuSOAP v0.6.1
Host: localhost
Content-Type: text/xml
Content-Length: 569
SOAPAction: ""

<?xml version="1.0" encoding="ISO-8859-1"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:si="http://soapinterop.org/xsd">
<SOAP-ENV:Body>
  <nu:echoString xmlns:nu="http://testuri.org">
    <soapVal xsi:type="xsd:string">Dietrich Ayala</soapVal>
  </nu:echoString>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This is the server's response message from the previous example. Notice that the first element in the body of the message is the name of the operation we called, with `Response` appended to it. Also, the element name of the return value is called `return`. Both of these are standard practice for serializing SOAP responses:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Tue, 04 Jun 2002 18:47:53 GMT
X-Powered-By: PHP/4.1.2
Status: 200 OK
Server: NuSOAP Server v0.6.1
Connection: Close
Content-Type: text/xml; charset=UTF-8
Content-Length: 1525

<?xml version="1.0" encoding="ISO-8859-1"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:si="http://soapinterop.org/xsd">
<SOAP-ENV:Body>
  <echoStringResponse>
    <soapVal xsi:type="xsd:string">Dietrich Ayala</soapVal>
  </echoStringResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A Simple PHP SOAP Server Example

This example is the server that was accessed by our client example. It implements the `echoString` service.

The first step as usual is to include the NuSOAP classes:

```
<?php

require('nusoap.php');
```

Now we can instantiate the server object, provided by the `soap_server` class:

```
$s = new soap_server;
```

To allow our function to be called remotely, we must register it with the server object. If this is not done, the server will generate a fault indicating that the service is not available if a client accesses the service. In the absence of such a registration process, any PHP functions would be remotely available, which would present a serious security risk:

```
$s->register('echoString');
```

Now we can define our function that we are exposing as a service. Notice that we first check to make sure a string was passed. If the parameter is not a string, we use the `soap_fault` class to return an error to the client indicating that they must pass a string value as the parameter to this function:

```
function echoString($inputString){

    if(is_string($inputString)){
        return $inputString;
    } else {
        return new soap_fault('Client','','The parameter to this service must be a 
string.');
    }
}
```

The final step is to pass any incoming posted data to the SOAP server's `service` method. This method processes the incoming request, and calls the appropriate function. It will then formulate the response, and print it:

```
$s->service($HTTP_RAW_POST_DATA);

?>
```

Fault Handling

The `soap_fault` class provides a way to specify errors and return them when developing services with NuSOAP.

The properties of the `soap_fault` class are below. These are also the arguments to the `soap_fault` constructor, in the same order as below:

Fault	**Description**
`faultcode`	This property must have a value. The values available to the user are `Client` and `Server`. Client class errors indicate that the message didn't contain the information required for the operation to succeed. Server class errors indicate processing problems on the server.
`faultactor`	This property is not functional yet in NuSOAP, and can be left empty. Its purpose is to indicate the location of the fault among multiple actors in a message path.
`faultstring`	This is a human-readable error message. This is the best place for you to describe errors.
`faultdetail`	The value of the `faultdetail` property is XML data used to detail the application-specific errors related strictly to the `Body` element of the SOAP message. You can insert your own XML markup here.

The `soap_fault` class has only one method besides the constructor that is `serialize()`. The `serialize()` method takes the fault information and serializes it, returning a complete SOAP message.

An example of using the `soap_fault` class is shown here:

```
$fault = new soap_fault(
     'Client','','The inputString parameter must not be empty');

echo $fault->serialize();
```

The SOAP message below is what is returned by the `serialize()` method of the `soap_fault` object instantiated above:

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:si="http://soapinterop.org/xsd">
<SOAP-ENV:Body>
<SOAP-ENV:Fault>
  <faultcode>Client</faultcode>
  <faultactor></faultactor>
  <faultstring>The inputString parameter must not be empty</faultstring>
  <detail></detail>
</SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Using Arrays

Transmitting and receiving arrays is transparently done in NuSOAP. You can pass PHP arrays in the parameters array argument to the `soapclient` class's call method, and NuSOAP will detect the type of the array contents, and serialize accordingly. This script uses a service called `echoArray` that accepts an array and returns the same array. While running this example, try modifying the parameter array by adding different data types, and see how the serialization changes.

First, include the NuSOAP classes:

```
<?php

require('nusoap.php');
```

Create the array we'd like to send:

```
$arr = array('string1','string2');
```

Create the array of parameters:

```
$parameters = array($arr);
```

Instantiate the `soapclient` object, passing it the endpoint URL:

```
$s = new soapclient('http://localhost/wrox/nusoap/echoArrayServer.php');
```

Now we call the `echoArray` operation, passing it our array of parameters, and receive the result. Then print the result on the screen, and view the request and response messages, as follows:

```
$result = $s->call('echoArray',$parameters);

if(!$err = $s->getError()){
     echo 'Result: '.$result;
} else {
     echo 'Error: '.$err;
}

echo '<xmp>'.$s->request.'</xmp>';
echo '<xmp>'.$s->response.'</xmp>';

?>
```

Creating Complex Types

SOAP allows users to define their own types, called "general compound types" in the specification. NuSOAP provides the `soapval` class for defining custom types, or for situations where you want to override NuSOAP's default serialization behavior. An example of overriding NuSOAP's behaviors would be where let's say NuSOAP would type the value `2.3433` as a float, but the service requires it to be typed as a double. You could create the parameter like:

```
$param = new soapval('','double',2.3433);
```

NuSOAP would then serialize the parameter like this:

```
<soapVal xsi:type="xsd:double">2.3433</soapVal>
```

Here is an example of using the `soapval` class to create a custom type for some contact information:

```
<?php

include('nusoap.php');

$address = array(
     'street' => '123 Freezing Lane',
     'city' => 'Nome',
     'state' => 'Alaska',
     'zip' => 12345,
    'phonenumbers' => array('home'=>'1234567890','mobile'=>'0987654321')
);

$s =new soapval('myAddress','address',$address,'','http://myNamespace.com');

print "<xmp>".$s->serialize()."</xmp>";

?>
```

Here is the result of running the above code:

```
<myAddress xmlns:ns8467="http://myNamespace.com" xsi:type="ns8467:address">
<street xsi:type="xsd:string">123 Freezing Lane</street>
<city xsi:type="xsd:string">Nome</city>
<state xsi:type="xsd:string">Alaska</state>
<zip xsi:type="xsd:int">12345</zip>
<phonenumbers>
<home xsi:type="xsd:string">1234567890</home>
<mobile xsi:type="xsd:string">0987654321</mobile>
</phonenumbers>
</myAddress>
```

Using WSDL and soap_proxy

WSDL is an XML language used to describe a Web Service. It is a machine-readable format that provides Web Service clients with all the information necessary to access the service. NuSOAP provides a class for parsing WSDL files, and extracting data from them. The `soapclient` object uses the `wsdl` class to ease the burden of the developer calling a service. With the help of the WSDL data to create messages, a programmer only needs to know the name of the operation to call, and the parameters required by the operation.

Using WSDL with NuSOAP provides several benefits:

- All service meta data such as namespaces, endpoint URLs, parameter names, and much more are read from the WSDL, allowing the client to dynamically cope with changes from the server. This information no longer needs to be hard-coded into the user's script since it's on the server.
- It allows us to use the `soap_proxy` class. This class is an extended `soapclient` class with new methods for each of the operations detailed in the WSDL file. Now the user can call these methods directly. This process is described below.

The `soapclient` class contains a method called `getProxy()`. This method returns an object of the class `soap_proxy`. The `soap_proxy` class extends the `soapclient` class with methods that correspond to the operations defined in the WSDL document, and allows users to call the remote methods of an endpoint as if they were local to the object. This is only functional when the `soapclient` object has been instantiated using a WSDL file and has the advantage of easy access for a user. The disadvantage though is the performance – object creation is expensive in PHP – and this functionality serves no utilitarian purpose.

Here is an example of using WSDL to call a stock quotes service from Xmethods, using the `soap_proxy` class. We start by including the NuSOAP classes:

```
<?php

include('nusoap.php');
```

When instantiating the client object using WSDL, we pass the URL or path to the WSDL file as an argument to the constructor, as well as an argument that lets the client know we've passed it WSDL and not a SOAP endpoint:

```
$s = new soapclient(
'http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl', 'wsdl');
```

Now we'll generate the proxy class. This is achieved by invoking the `getProxy()` method of the `soapclient` class:

```
$p = $s->getProxy();
```

We can now invoke the remote method as a method of the proxy class, and pass our parameters directly. The proxy object will handle the details such as matching up parameter values to their names, assigning namespaces, and types:

```
$sq = $p->getQuote('ibm');
```

Lastly let's check for errors, and if none are present print out the results:

```
if(!$err = $p->getError()){
     print "IBM current stock price: $sq.";
} else {
     print "ERROR: $err";
}


print '<xmp>'.$p->request.'</xmp>';
print '<xmp>'.str_replace('><',">\n<",$p->response).'</xmp>';

?>
```

Using an HTTP Proxy Server

The `soapclient` class has a method called `setHTTPProxy()`, which allows you to use an HTTP proxy server. It takes two arguments: the proxy hostname and the port address. For example:

```
$s = new soapclient('http://www.remoteserver.com/soap_server.php');
$s->setHTTPProxy('proxy.myCompany.com',8080);
```

You can then continue your SOAP calls as we did above. Calling this method will force all requests to be sent to the specified proxy server, which then forwards the request to its intended recipient.

HTTP Authentication

The `soapclient` object provides a method called `setCredentials()`. This method is used when HTTP authentication is needed to access a SOAP server. The two arguments to the method are a username and password. Its functionality is shown in the example below:

```
$s = new soapclient(
'http://www.remoteserver.com/protected_directory/soap_server.php');

$s->setCredentials('myUsername','myPassword');
```

You can then continue your SOAP calls, as you would normally do. The most common way of implementing this level of security on the server is to use Apache's `.htaccess` files to implement required authorization to protect a directory or file.

SSL

Using SSL with the NuSOAP client requires the CURL extension, which was discussed earlier in the chapter, to be installed. The CURL functions provide a way to make a secure connection to a URL. Standard PHP doesn't provide this functionality. If you have the CURL extension, and you would like to connect to a secure Web Service using SSL, you need do nothing other than enter the URL as you normally would. If the URL is secure, you'll notice it starts with `https` instead of the normal `http`:

```
$s = new soapclient('https://mySecureServer.com');
```

Implementing SSL on the server side is outside the scope of PHP's abilities. Check your web server's documentation for instructions on how to achieve this.

Using Document Style Messaging

All of the examples we have seen so far were remote procedure calls, or SOAP RPC, in which we call remote methods, and pass them, encoded parameters. There is another style of message exchange in SOAP, which is known as document style. Document style messaging involves the exchange of literal XML documents. This means that the message bodies are not modeled after procedure calls, and the elements in the message bodies do not have attributes containing type information.

This section provides two examples of document/literal usage using NuSOAP: one using WSDL and one without WSDL. The more common scenario would be to use a WSDL document, but it is useful being able to send arbitrary XML documents and document fragments via SOAP. The service used in the examples is a simple service that allows users to pass a ZIP code, and it returns a listing of 10 ATM cash machines in the ZIP code. The service is listed on Xmethods (http://www.xmethods.net/), and is provided by ServiceObjects (http://www.serviceobjects.com/).

The initial step, as usual, is to include the NuSOAP classes:

```
<?php

require('nusoap.php');
```

Next let's set the URL or path to the WSDL file:

```
$wsdlfile = 'http://ws.serviceobjects.net/gc/GeoCash.asmx?WSDL';
```

Now we'll create the XML document to be sent to the service. This document can be created manually by examining the schema defined in the WSDL document, or by using an XML writing tool, or a PHP class that supports the generation of document skeletons from an XML Schema:

```
// set parameters
$msg =
'<GetATMLocations xmlns="http://www.serviceobjects.com/">
  <strInput>32804</strInput>
  <strLicenseKey>0</strLicenseKey>
</GetATMLocations>';
```

We can now instantiate our `soapclient` object, and pass it the WSDL location:

```
$s = new soapclient($wsdlfile,'wsdl');
```

Finally, we make the call and get the result (which will be an XML document). When using document style messaging with NuSOAP, the document is available as a string via the document property of the `soapclient` class. Here we are printing out the document, and the request and response messages:

```
$s->call('GetATMLocations',array($msg));

print 'RESULT:<xmp>';
var_dump($s->document);
print '</xmp>';

print 'REQUEST:<xmp>'.$s->request.'</xmp>';
print 'RESPONSE:<xmp>'.$s->response.'</xmp>';

?>
```

Sometimes no WSDL is available for a service, or you just need to send arbitrary XML documents via SOAP. The method for sending arbitrary documents is described below:

```
<?php

require('nusoap.php');
```

First we create the XML messages we would like to send:

```
$body =
'<GetATMLocations xmlns="http://www.serviceobjects.com/">
  <strInput>32804</strInput>
  <strLicenseKey>0</strLicenseKey>
</GetATMLocations>';
```

Now we will instantiate the `soapclient` object by passing the service endpoint to the constructor, like this:

```
$s = new soapclient('http://ws.serviceobjects.net/gc/GeoCash.asmx');
```

The `serializeEnvelope()` method can be used to wrap a SOAP envelope around XML content:

```
$msg = $s->serializeEnvelope($body);
```

Use the `send()` method to actually send the message. The method takes two arguments: the message content, and the `SOAPAction` header value. Once again, we can examine the result via the document property of the `soapclient` class, and we'll print out our request and response messages:

```
$s->send($msg,'http://www.serviceobjects.com/GetATMLocations');

print 'RESULT:<xmp>';
var_dump($s->document);
print '</xmp>';

print 'REQUEST:<xmp>'.$s->request.'</xmp>';
print 'RESPONSE:<xmp>'.$s->response.'</xmp>';

?>
```

Other PHP SOAP Implementations

The following are PHP Web Services toolkits for sending and receiving SOAP messages.

Active State SWSAPI

Active State, a software company in Vancouver, Canada, used SOAPx4 as the core for their SOAP and WSDL toolkit for PHP, which is Web Service API called the Simple Web Services API (SWSAPI). It is a unified API for Web Service programming for Active State's Perl, Python, and PHP distributions.

PEAR

PHP Extension and Add-on Repository or PEAR is a group of classes bundled with the PHP distribution. PEAR is a community effort to create a set of tools that provide functionality common to the many uses of PHP. Shane Caraveo recently converted the SOAPx4 classes to meet the PEAR standards, and the toolkit was added to the repository. He also added many new features, such as HTTPS, SMTP support, and function overloading.

Krysalis

Krysalis is an application development platform for PHP from Interakt. It is similar in design to the Cocoon Application Framework, and was inspired by it. In the latest version of Krysalis, 1.0.4, Interakt has implemented a simple SOAP client and SOAP server. Until now, much of the message creation is done manually. Use of Krysalis requires PHP's XSLT Sablotron extension, and access to the PEAR classes. http://www.interakt.ro/products/Krysalis/.

PHP Web Services and XML-RPC

XML-RPC in PHP provides a simple and accessible alternative to SOAP. It is widely used, and there are several implementations to choose from. For many PHP developers, XML-RPC serves their RPC needs, without the complexity and infrastructure required by other RPC methods.

XML-RPC Data Types

As discussed above, XML-RPC specification defines a limited number of supported data types. The supported types are listed below with their PHP equivalents and examples:

XML-RPC	PHP	Example
`i4`	Integer	`23`
`int`	Integer	`23`
`boolean`	Boolean	`true`, `false`

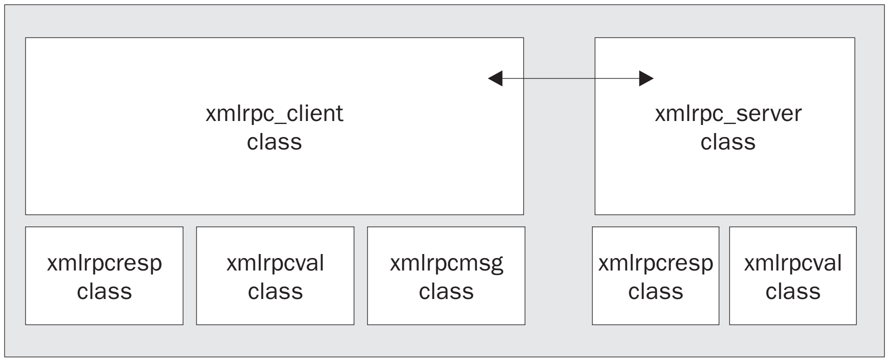
XML-RPC	PHP	Example
`string`	String	`'Dietrich Ayala'`
`double`	Double or float	`23.3`
`dateTime.Iso8601`	no native PHP equivalent	`-`
`base64`	Base64 encoded string	`base64 ('Dietrich Ayala')`
`array`	Array	`array ( 1, 2, 'red', 'blue');`
`struct`	Associative array	`array ('color' => 'red', 'number'=> 2)`

Useful Inc. XML-RPC Implementation

The XML-RPC toolkit that we'll be using for our examples is a set of PHP classes authored by Edd Dumbill of Useful, Inc. The code and documentation can be found at Sourceforge.net (http://phpxmlrpc.sourceforge.net/). The toolkit contains both client and server implementations of XML-RPC.

The toolkit comes with two important classes: the `xmlrpc_client` class for the client and the `xmlrpc_server` class for server-side programming. These classes are mainly used for sending and receiving the XML-RPC messages. An `xmlrpcval` class is used to encode the PHP variables into their XML-RPC equivalents and is used to pass parameters to a remote method. The reverse process of decoding back into the PHP equivalent is done using the `xmlrpc_decode()` function. An XML-RPC message is created using the `xmlrpcmsg` class by passing a parameter list to it.

The `xmlrpc_client` class sends XML-RPC messages that are created by the `xmlrpcmsg` class. On the server side, the `xmlrpc_server` class parses these incoming messages back into an `xmlrpcmsg` object. This message object is then passed as the single argument to the user's deployed function. This function must return a response in the form of an `xmlrpcresp` object that the `xmlrpc_server` class uses to serialize and return back to the client. This basic architecture of the toolkit is shown in the diagram below:



Installation and Configuration

The installation can be achieved in a few steps as follows:

- Download XML-RPC for PHP from http://phpxmlrpc.sourceforge.net/
- Unpack it
- Move `xmlrpc.inc` and `xmlrpcs.inc` to a location in your include path, or save them to the same directory as the script(s) in which you expect to use them

To use the XML-RPC client you must include the client code that is done by adding the following line at the top of your script. Here, the argument to the `include()` function is the path to the `xmlrpc.inc` file:

```
include('xmlrpc.inc');
```

To use the XML-RPC server class – to create and deploy your own XML-RPC Web Services – you must include the client and the server code:

```
include('xmlrpc.inc');
include('xmlrpcs.inc'); //server code
```

A Simple XML-RPC Client Example

This example demonstrates an XML-RPC client that passes a string variable to an XML-RPC server that echoes the string back to the client.

The first step is to include the XML-RPC client code:

```
<?php

// include the xml-rpc classes
require('xmlrpc.inc');
```

Now we can create a client object, and pass it our connection information. The `xmlrpc_client` class is a high-level class that serializes parameters and messages. It is also responsible for sending the messages to the server. Its constructor takes three arguments: the path, hostname, and port address of the server:

```
$s = new xmlrpc_client('/wrox/xmlrpc/xmlrpc_server.php','localhost',80);
```

The next step is to create the string variable that we would like to send to the `echoString` service. We do this using the `xmlrpcval` class. It allows us to create abstractions of PHP variables into XML elements according to the XML-RPC specification:

```
// create xmlrpcval object, which allows the encoding of our variable
$inputString = new xmlrpcval('Dietrich Ayala','string');
```

Let's add our parameter to an array that can hold multiple parameters. We do this because the `xmlrpcmsg` object constructor takes a parameter list as its second argument:

```
// create an array of parameters
$parameters = array($inputString);
```

The final preparatory step is to create the XML-RPC message using the `xmlrpcmsg` object. Its constructor takes the method name we are calling as its first argument, and our array of parameters as its second:

```
    // create the message object
    $msg = new xmlrpcmsg('echoString',$parameters);
```

Now we are ready to send our message. To do this we use the `send()` method of the `xmlrpc_client` object. It takes our `xmlrpcmsg` object as an argument and returns an `xmlrpcresp` object. The `xmlrpcresp` object has three methods of use to the client:

- ❑ `faultCode()`
  This method returns the code associated with any errors returned
- ❑ `faultString()`
  This method returns a string description of the error
- ❑ `value()`
  This method returns the value contained in the response in the form of an `xmlrpcval` object

If no errors occur, the `faultCode()` method will return a zero. This is an easy way to check for errors and this is what we'll do in our example:

```
    // send the message, get the response
    $rsp = $s->send($msg);

    // check for errors
    if($rsp->faultcode() == 0){
```

If no errors occurred, we can go ahead and retrieve the value returned by the server. To do this, we will first call the `value()` method of the response object, which returns an `xmlrpcval` object. To convert this into its equivalent PHP type, we use the `xmlrpc_decode()` function that returns the value converted into a PHP type:

```
         // decode the response to a PHP type
         $response = xmlrpc_decode($rsp->value());
```

Let's print the result, or any errors, and view the messages:

```
         // print results
        print '<pre>';
        var_dump($response);
        print '</pre>';
```

```
    } else {
        // print errors
        print 'Error: '.$rsp->faultcode().', '.$rsp->faultstring().'<br>';
    }

    // show messages
    $msg->createpayload();
    print 'REQUEST:<xmp>'.$msg->payload.'</xmp>';
    print 'RESPONSE:<xmp>'.$rsp->serialize().'</xmp>';

    ?>
```

XML-RPC Request and Response

This is the request message generated in the previous code sample. Notice the readability of the message. Though verbose, it is intuitive and easy to decipher its meaning:

```
<?xml version="1.0"?>
<methodCall>
<methodName>echoString</methodName>
<params>
<param>
<value><string>Dietrich Ayala</string></value>
</param>
</params>
</methodCall>
```

Here is the response message from the server:

```
<methodResponse>
<params>
<param>
<value><string>Dietrich Ayala</string></value>
</param>
</params>
</methodResponse>
```

A Simple XML-RPC Server Example

This example implements the service that we used in our client example that is the `echoString` service. Here we'll see how to use the `xmlrpc_server` class to create and deploy a service and then serve requests using the `xmlrpcresp` class to encapsulate and return responses.

We must first include the XML-RPC files. Note the inclusion of `xmlrpcs.inc`, which provides the code necessary for the server:

```
<?php

// include the client and server classes
require('xmlrpc.inc');
require('xmlrpcs.inc');
```

Let's define our function that will handle the incoming request. The function to be deployed must take only one argument, which is an `xmlrpcmsg` object:

```
// service that echoes a string
function echoString($msg){
```

The first step in our function is to take the parameters from the message object and decode them into PHP types. The decoding is done using the `xmlrpc_decode()` function that we discussed above:

```
    // decode parameters into native types
     $inputString = xmlrpc_decode(array_shift($msg->params));
```

Here we check the parameter's type to see that it is valid. If it is, we return it using the `xmlrpcresp` object. To return a value, the `xmlrpcresp` class constructor takes one argument of type `xmlrpcval`. Thus, any returned value must first be encoded as an `xmlrpcval`:

```
      // check for input parameter validity
       if(is_string($inputString)){
          return new xmlrpcresp( new xmlrpcval($inputString, 'string') );
  }
```

If the parameter is invalid, we may use the `xmlrpcresp` object again to return a fault. This is done by passing zero to the `xmlrpcresp` constructor as its first argument followed by the error code. The XML-RPC specification says that fault codes are to be user-defined. XML-RPC for PHP defines a constant named `$xmlrpcerruser`, which sets an error level. For errors taking place outside of the XML-RPC classes themselves, the standard convention recommended by the author of these classes is to increment the value of this variable by one. The last argument is the fault code, which is a human-readable string describing the error:

```
      else {   // or return a fault

        return new xmlrpcresp(0, $xmlrpcerruser+1,
        "Parameter type ".gettype($inputString)." mismatched expected type.");

      }
  }
```

The final step in the process is to instantiate the server class, and register our function in it. The `xmlrpc_server` constructor takes an associative array as its first argument. This array is what is called the "dispatch map." It is an associative array of associative arrays. The key values of the outer array are the public names of the functions being deployed that are the names by which XML-RPC clients would call them. The inner array contains three members:

- `function`
  This is a mandatory entry and its value is the name of the PHP function you've defined to handle this service. In our example above this is the same as the service name that is `echoString`. For example, you could deploy a service called `reverser` that maps to a PHP function `strrev()` that takes a string as an argument and returns the string reversed. The most important point to note is that the value of `function` must be a valid function in the global scope.
- `signature`
  This is an optional member. A `signature` is defined by creating an array of parameter types for each input and output parameters. It is an array of possible signatures to validate requests against. The value of this member takes an array in which each member is a possible signature. Here, the last one is the output parameter while all the previous ones are input parameters.
- `docstring`
  This is an optional member that may contain the documentation for your service, and may even have HTML contents.

Instantiating the server class initiates the processing of any incoming requests. The second parameter to the `xmlrpc_server` constructor is optional. When this is set to zero, the server will not immediately process requests. The processing can be executed later by using the server's `service()` method:

```
// instantiate the server object and register our functions
$s = new xmlrpc_server(array(
           'echoString' => array(
                 'function' => 'echoString',
                 'signature' => array( array('string','string') ),
                 'docstring' => 'This service echoes the input string back to the
client.'
           )
     ));

?>
```

Other PHP XML-RPC Implementations

The other open source implementations of the XML-RPC protocol for PHP are either experimental or not well documented. Here we list a couple of them.

XMLRPC-EPI

XMLRPC-EPI is an XML-RPC implementation written in C. It was originally developed for internal use at Epinions (hence the -EPI). Epinions (http://epinions.com/) released XMLRPC-EPI as open source in March 2000. Since then the project has been hosted and managed on Sourceforge.net. Epinions also donated a PHP module for the code, written by Dan Libby. This module has since been folded into the PHP core distribution and has been available in PHP since version 4.1.0. It is currently labeled experimental in the PHP manual.

Noyade

This is a pure PHP implementation by Matt Bean. It has no documentation available. The source is available at http://rpc.lemurpants.com/noyrpc.phps. The script for testing the implementation against Userland Software's Validator is provided at http://rpc.lemurpants.com/rpc-validator1.phps.

Future of PHP Web Services

The momentum of Web Services in PHP is growing rapidly. The core PHP developers are discussing the possibility of having standard SOAP support bundled with the main PHP distribution. This would enable Web Service usage in PHP to explode by making it available in every new PHP installation.

Brad LaFountain has recently released the first version of a SOAP extension for PHP. It looks very promising, and will probably end up being PHP's standard SOAP extension. It uses the `libxml` library, which provides implementations of XML DOM, Xpath, and XLink. If it were bundled with PHP it would be a double win for PHP developers who use XML.

Other Web Services activity in PHP is the creation of PEAR-SOAP. PEAR (PHP Extension and Application Repository) is a collection of reusable PHP components. Shane Caraveo, primary developer of the initial port of PHP to Windows, ported SOAPx4 (NuSOAP's predecessor) to PEAR, which became PEAR-SOAP.

Summary

With NuSOAP we've been able to create a client that retrieves content by consuming services offered by remote servers using SOAP, and guided by WSDL. We've also deployed our own service that is accessible by a multiplicity of clients of all different platforms and languages. These basic operations are boilerplates for your imagination as you can use them to create robust interoperable Web Services for yourself.

We have also seen how to use XML-RPC for PHP to create a client that can pass encoded data to a server and detect success or failure from the response. We have seen how to create an XML-RPC server and deploy services using it. These examples demonstrate how PHP and XML-RPC can be used together to access remote applications, as well as to create your own.

Though the question that arises is "which is a better choice: SOAP or XML-RPC?", actually there is no black-and-white answer to this question. The best solution is the one that solves the stated problem best, or achieves the goals of your application. Both XML-RPC and SOAP have different abilities, different strengths and weaknesses. You should weigh these against the needs of your application to make your decision.

XML-RPC is designed for simplicity. It contains limitations that some developers might welcome, such as a small number of types to support and a small XML vocabulary. It may perform faster as it requires much less infrastructure than SOAP.

SOAP is designed to accomplish much more complex messaging problems than XML-RPC. It allows users to define a near limitless amount of types, as well as different styles of messaging. You can specify the character set of the message, as well as specify a recipient, and force the recipient to reply with a fault if it can't understand the message. All in all, SOAP provides a much richer set of features than XML-RPC, but at the cost of more complexity. The final decision of which is better will be decided by the needs of your application or project.

This chapter has demonstrated the use of PHP tools to consume and deploy Web Services with XML-RPC and SOAP, which is not only possible, but robust and interoperable as well. The examples in this chapter have shown that PHP and Web Services can turn your web applications into distributed systems that can provide interoperable services to others.