Web Service

Implementation using SOAP and REST

Contents

Web Services

"Web Service" has been a popular buzzword these days. With a tendency to move software systems toward network implementation, particularly the internet, there has been a rising interest in utilizing web services technology. It is therefore important to have knowledge regarding this topic because it will most likely be prevalent in majority of software systems in the near future.

Definition

Web Service has become a loose terms which can refer to many things in the software system. As such, definition might vary depending on how we view it.

W3C defines Web Service as follows.

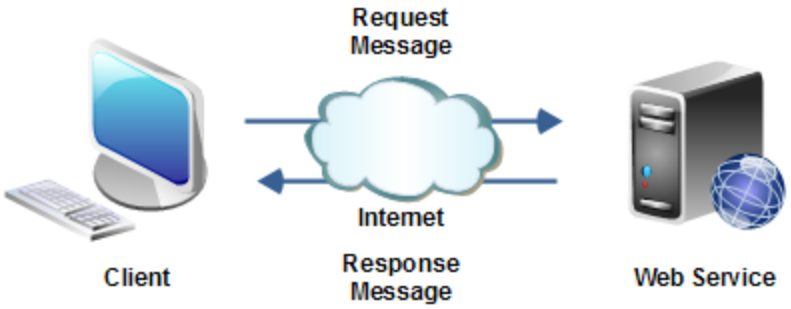
> "A software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

Definition from Microsoft's Developer Network put Web Service as such.

> "Web services extend the World Wide Web infrastructure to provide the means for software to connect to other software applications. Applications access Web services via ubiquitous Web protocols and data formats such as HTTP, XML, and SOAP, with no need to worry about how each Web service is implemented."

In a programmer's view, a Web Service can be described as **functions** or **methods** provided publicly which can be used by others without needing to know the exact implementation. To use a Web Service, we simply need to know the **name** of the service, **location** of said service, and **input/output** parameter necessary for the service.

Here is a graphical representation of how Web Service works.



1. Client would call the required service through the network (request message)
2. Server accept request message and do necessary processing
3. Server then packages the return value as response message and sent it back to client

As we can see from the description above, Web Service is a surprisingly simple concept. Client doesn't need to know how server processed the requested data. All the client needs to know is the response message.

Service on the Web

Web Service is usually provided in World Wide Web URI address, for example:

http://www.webservicex.net/stockquote.asmx

This address provides the location where we can get and use (**consume**) the service. The method to access the web service depends on how the particular web service is built. With SOAP, we need to have a SOAP server and SOAP client to interact with each other, while with REST we only need to call the URL and get the data immediately.

Programming Language for Web Service

At this point I should say that there is no particular language to create a web service. The nature of Service-Oriented Architecture means that a Web Service is language independent. The important part is that our functions or methods are available for others to use. This fact makes Web Services a powerful system because any application can interact with our services without being obstructed by language constraints.

Or course there are favorite languages in developing web services. These languages are geared toward web development, thus making it easy to create a web-deployment ready service. Here are some recommended languages to develop a web service:

1. ASP.NET (http://www.asp.net/)
2. PHP (http://php.net/)
3. Ruby (https://www.ruby-lang.org/en/)
4. Java (https://www.oracle.com/java/index.html)

We can actually use any language to create web service, even C++, as long as we can make our functions usable on the network.

How This Module Works

In this module we will explore the concept of Web Services as well as the implementation of two most popular web service systems as of now: Simple Object Access Protocol (SOAP) and Representational State Transfer (REST).

This module will explore the two implementations and provides know-how to implement your own web services.

While this concept can be applied with any programming languages, the sample codes would mostly be programmed in PHP and ASP.NET language.

Prerequisites

This module will assume that readers have at least basic knowledge of Web protocols, HTML, XML, PHP, JavaScript, and ASP.NET.

Readers are also required to have these software in order to try the examples in this module.

- Browser with JavaScript support. Recommended Google Chrome (http://www.google.com/chrome/)
- JQuery library for JavaSript. Download here: http://jquery.com/download/
- PHP at least version 5.4. Download here: http://php.net/
- ASP.NET with at least Microsoft Visual Studio 2010 or newer. Download here: http://www.asp.net/

Simple Object Access Protocol (SOAP)

Web Services are merely functions exposed in the internet, waiting to be consumed by user clients around the world. The problem is that server and clients might have different operating system running, if not even difference in programming language and possible data types.

This problem calls for a uniform standard by which all system can communicate with each other regardless of how they are built. This standard is known as Simple Object Access Protocol (SOAP).

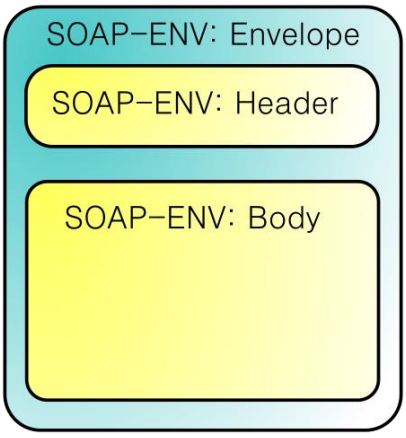
What is SOAP?

SOAP was designed as an object-access protocol in 1998 by Dave Winer, Don Box, Bob Atkinson, and Mohsen Al-Ghosein for Microsoft, where Atkinson and Al-Ghosein were working at the time. The SOAP specification is currently maintained by the XML Protocol Working Group of the World Wide Web Consortium.

SOAP is a specification which every application must adheres to use. It is the standard which allows all applications to interact in the same language. In this case, all services use **SOAP message format** as its input and output parameter. Data might come in many forms before being sent, but in the network they are all in the form of SOAP message format.

SOAP Message

SOAP message is simply an XML document. The general format can be shown as follows.



SOAP message is further divided into two types: **request** and **response**. Request message is sent by client to call functions in the server. Likewise, the server will return the result of process by wrapping it inside the response message.

Here is a sample request message.

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

   <soapenv:Header>
      <ns1:RequestHeader
      soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
      soapenv:mustUnderstand="0"
      xmlns:ns1="https://www.google.com/apis/ads/publisher/v201403">
         <ns1:networkCode>123456</ns1:networkCode>
            <ns1:applicationName>
               DfpApi-Java-2.1.0-dfp_test
            </ns1:applicationName>
         </ns1:RequestHeader>
   </soapenv:Header>

   <soapenv:Body>
      <getAdUnitsByStatement
      xmlns="https://www.google.com/apis/ads/publisher/v201403">
         <filterStatement>
            <query>WHERE parentId IS NULL LIMIT 500</query>
         </filterStatement>
      </getAdUnitsByStatement>
   </soapenv:Body>

</soapenv:Envelope>
```

As we can see, the `soapenv:Header` and `soapenv:Body` tag is enclosed inside the `soapenv:Envelope` tag as seen in the diagram above. Header is used to show application-specific information which the actual functions do not need to know. Body is the real message which states which function to use and what parameter is given. In above example, the function is named `getAdUnitsByStatement` and the parameter is a `filterStatement` using SQL query.

This request gives the following XML as the response.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

  <soap:Header>
    <ResponseHeader
    xmlns="https://www.google.com/apis/ads/publisher/v201403">
      <requestId>xxxxxxxxxxxxxxxxxxxx</requestId>
      <responseTime>1063</responseTime>
    </ResponseHeader>
  </soap:Header>

  <soap:Body>
    <getAdUnitsByStatementResponse
    xmlns="https://www.google.com/apis/ads/publisher/v201403">
      <rval>
        <totalResultSetSize>1</totalResultSetSize>
        <startIndex>0</startIndex>
        <results>
          <id>2372</id>
          <name>RootAdUnit</name>
          <description></description>
          <targetWindow>TOP</targetWindow>
          <status>ACTIVE</status>
          <adUnitCode>1002372</adUnitCode>
          <inheritedAdSenseSettings>
            <value>
              <adSenseEnabled>true</adSenseEnabled>
              <borderColor>FFFFFF</borderColor>
              <titleColor>0000FF</titleColor>
              <backgroundColor>FFFFFF</backgroundColor>
              <textColor>000000</textColor>
              <urlColor>008000</urlColor>
              <adType>TEXT_AND_IMAGE</adType>
              <borderStyle>DEFAULT</borderStyle>
              <fontFamily>DEFAULT</fontFamily>
              <fontSize>DEFAULT</fontSize>
            </value>
          </inheritedAdSenseSettings>
        </results>
      </rval>
    </getAdUnitsByStatementResponse>
  </soap:Body>

</soap:Envelope>
```

We can easily find the result by looking at a tag with the called function's name appended with "Response". In this case, it is stored in `getAdUnitsByStatementResponse` tag. Every children of this tag is a variable in programming sense. In this case, the result is a data structure named `rval` with attributes of `totalResultSetSize`, `startIndex`, and `results`. `results` is another data structure with more attributes. You can read it yourself.

Web Service Definition Language

Your web service functions reside in the server. Clients need to know the exact name of the functions, parameters needed for each function, and then call them through SOAP request message. Although they can do this by asking the service provider, it becomes impossible if the provider is located in remote places. It is better to have a documentation which shows what functions are available for clients and what parameters they accept.

Enter Web Service Definition Language (WSDL).

WSDL is simply an XML document which describes the services. This means the function names, functions parameter, protocol used, and access type.

An XML document has to follow the official format to be recognized as a WSDL. Namely, it must contain these elements.

Element	Definition
<types>	A container for data type definitions used by the web service
<message>	A typed definition of the data being communicated
<portType>	A set of operations supported by one or more endpoints
<binding>	A protocol and data format specification for a particular port type

Here is a sample of WSDL document.

```
<definitions name="HelloService"
targetNamespace="http://www.examples.com/wsdl/HelloService.wsdl"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://www.examples.com/wsdl/HelloService.wsdl"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <types>
      <schema xmlns="http://www.w3.org/2000/10/XMLSchema">
         <element name="TradePriceRequest">
            <complexType>
               <all>
                  <element name="tickerSymbol" type="xsd:string"/>
```

```
            </all>
         </complexType>
      </element>
   </schema>
</types>

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

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

<binding name="Hello_Binding" type="tns:Hello_PortType">
   <soap:binding style="rpc"
   transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="sayHello">
      <soap:operation soapAction="sayHello"/>
      <input>
         <soap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         namespace="urn:examples:helloservice"
         use="encoded"/>
      </input>
      <output>
         <soap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         namespace="urn:examples:helloservice"
         use="encoded"/>
      </output>
   </operation>
</binding>

<service name="Hello_Service">
   <documentation>WSDL File for HelloService</documentation>
   <port binding="tns:Hello_Binding" name="Hello_Port">
      <soap:address
         location="http://www.examples.com/SayHello/">
   </port>
</service>
```

```
</definitions>
```

Let's break it down bit by bit. A WSDL document always starts with `definitions` tag with desired XML schema information you need.

`types` tag is used to define data types outside standard XML Schema Definition (XSD). We define the data type definition in `element` tags. complexType tag means we are making a custom data types containing one or more variables. In this case we have the `tickerSymbol` variable with data type of standard XSD string (`xsd:string`). Think of it like making a custom data type in C language using struct.

`message` tag is used to show the parameter for requesting function and returning results. For example, in this WSDL we define a `message` tag with SayHelloRequest and SayHelloResponse as the name. Inside `message` tag we have children in the form of `part` tag. `part` tag has 2 attributes: the **name** for the variable and the **type** of the variable. The type available is either standard defined in XSD or the custom data types in `types` tag. In this example we use standard XSD string format.

`portType` tag contains the name for server functions defined in `operation` tags. Inside operation tag is the input and output message (request and response) which is referenced by `message` attribute to our previously-defined `message` tag. In this case, we refer back to `SayHelloRequest` as the input sent by client and `SayHelloResponse` as the output sent by server.

`binding` tag is the one which defines the protocol for each `portType` tag. The most important part is the `type` attribute which refers to our previously-defined `portType` tag (in this example, `Hello_PortType`). Inside binding we have `soap:binding` tag which decides our transport style: RPC (Remote Procedure Call) or document. In this example we use RPC style. The rest is similar to contents of `portType` tag. Important to note is the `use` attribute inside `soap:body` tag. We have 2 options: literal or encoded. Literal type means that there is no specific encoding for the data value, while encoded is vice-versa. In this example the encoding format is specified in `encodingStyle` attribute (`"http://schemas.xmlsoap.org/soap/encoding/"`).

Lastly the `service` tag is used to define our service description, the port binding used, and service address in URI.

SOAP Implementation with PHP

SOAP might looked overly complicated due to heavy usage of XML tags. Fortunately, many languages provides library to implement SOAP in our program. We don't need to create request and response message by ourselves; we will leave it to the library.

In this module we will try to implement SOAP using PHP language. PHP has provides a built-in library to support SOAP message generation and passing. It divides the task into two classes: **SoapServer** which

acts as the container for server functions and **SoapClient** which consumes the functions from remote place.

PHP SoapServer

SoapServer is the class which handles request message from clients and point toward the right function that the client wants. We would initialize a SoapServer with this syntax.

```
server_var = new SoapServer(wsdl_file[, options]);
```

We have two things to put in SoapServer. First is the location of our WSDL document if we have made one. The other one is options for the server, like the target namespace and SOAP version. Option is in form of associative array.

If you don't specify any WSDL file, we can put NULL into the first parameter but we need to put `uri` data into the `options` array containing our target namespace.

Here is an example. Note that the content of `uri` is the **target namespace** of your web service and not necessarily a real URI location.

```
$options = array('uri' => 'http://localhost/SOAPDemo');
$server  = new SoapServer(NULL, $options);
```

SoapServer is merely a PHP class, so we will mostly work with its methods.

Methods	Description
addFunction	Adds one or more functions to handle SOAP requests
addSoapHeader	Add a SOAP header to the response
__construct	SoapServer constructor
fault	Issue SoapServer fault indicating an error
getFunctions	Returns list of defined functions
handle	Handles a SOAP request
setClass	Sets the class which handles SOAP requests
setObject	Sets the object which will be used to handle SOAP requests
setPersistence	Sets SoapServer persistence mode
SoapServer	SoapServer constructor

Let's try exposing functions so others can use our web service. Here are two simple functions as sample services.

```
function hello(){
   return 'Hello Binus';
}

function plus($a, $b){
   return $a + $b;
}
```

We will register the functions to our server object with **addFunction** method. What we need to put into the method is the name of functions in string format.

```
//adding function one by one
$server->addFunction('hello'));
$server->addFunction('plus'));
```

We can also put the function names using array.

```
//adding function using array
$server->addFunction(array('hello', 'plus'));
```

Or we can simply use SOAP_FUNCTIONS_ALL constants to put **all** functions in our php file into server.

```
//adding all function in this php file
$server->addFunction(SOAP_FUNCTIONS_ALL);
```

Another method to register the functions is by creating a PHP class to encapsulate all functions that we want to expose through SOAP. So our sample functions will be created and called as such:

```
class TestService{
   public function hello(){
      return 'Hello Binus';
   }

   public function plus($a, $b){
      return $a + $b;
   }
}

$server->setClass('TestService');
```

Finally, we only need to call **handle** method to start our web service.

```
$server->handle();
```

PHP SoapClient

SoapClient is the class PHP provides to consume SOAP services. Note that it does not means the server must be created using SoapServer. Any server would do as long as they follow SOAP message format. This is why SOAP can be a cross-platform format!

The syntax for SoapClient is similar to SoapServer:

```
client_var = new SoapClient(wsdl_file[, options]);
```

If we don't specify any WSDL file location, we need to put `location` and `uri` data in `options`. `location` is the **actual** address where the web service server is located, while `uri` is simply the target namespace of the service. Let's make a client pointing to our server from before. Note that the location in this sample is based on my server configuration. The one in your deployment might be different.

```
$options = array(
   'uri'      => 'http://localhost/SOAPDemo',
   'location' => 'http://10.22.68.59:8099/SOAPDemo/server.php'
);
$client = new SoapClient(NULL, $options);
```

The methods in SoapClient is as follows.

Methods	Description
__construct	SoapClient constructor
__doRequest	Performs a SOAP request
__getFunctions	Returns list of available SOAP functions
__getLastRequest	Returns last SOAP request
__getLastRequestHeaders	Returns the SOAP headers from the last request
__getLastResponse	Returns last SOAP response
__getLastResponseHeaders	Returns the SOAP headers from the last response
__getTypes	Returns a list of SOAP types
__setCookie	The __setCookie purpose
__setLocation	Sets the location of the Web service to use
__setSoapHeaders	Sets SOAP headers for subsequent calls
__soapCall	Calls a SOAP function
SoapClient	SoapClient constructor

Most of the methods in SoapClient is intended for internal use. Usually we have no need to tinker with them.

With SoapClient, we can call the service functions as if they belonged to client object as methods. So to call our sample functions we only need to do it like this.

```
echo $client->hello(); //print Hello Binus
echo $client->plus(5, 8); //print 13
```

Web Service with WSDL

This kind of server and client without WSDL is allowed because we used the same technology for server and client (PHP SoapServer and SoapClient). In reality, we could not know what kind of technology is used by the server or client. If we make our server with ASP.NET, for example, our SoapClient will break because we can't find out what functions are exposed in the server.

In this kind of situation we would need to make WSDL file. WSDL is a universal language for describing web services, and all SOAP clients from any programming language will be guaranteed to understand it.

Here is the WSDL for our sample service above. Save this document with the name TestService.wsdl in your SOAP server folder. Note that I made this document manually by conforming to WSDL standard template. Refer to previous section about WSDL to help you understand it. As usual, change any URI used in this sample with your own location in your server.

```
<definitions name="TestService"
targetNamespace="http://10.22.68.59:8099/SOAPDemo/"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://10.22.68.59:8099/SOAPDemo/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <message name="helloRequest" />
   <message name="helloResponse">
      <part name="return" type="xsd:string"/>
   </message>
   <message name="plusRequest">
      <part name="a" type="xsd:int"/>
      <part name="b" type="xsd:int"/>
   </message>
   <message name="plusResponse">
      <part name="return" type="xsd:int"/>
   </message>

   <portType name="Test_PortType">
      <operation name="hello">
         <input message="tns:helloRequest"/>
         <output message="tns:helloResponse"/>
      </operation>
      <operation name="plus">
         <input message="tns:plusRequest"/>
         <output message="tns:plusResponse"/>
      </operation>
   </portType>

   <binding name="Test_Binding" type="tns:Test_PortType">
      <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="hello">
         <soap:operation soapAction="hello"/>
         <input>
            <soap:body
            namespace="urn:testservice:hello"
            use="literal"/>
         </input>
         <output>
            <soap:body
            namespace="urn:testservice:hello"
            use="literal"/>
         </output>
```

```
      </operation>
      <operation name="plus">
         <soap:operation soapAction="plus"/>
         <input>
            <soap:body
            namespace="urn:testservice:plus"
            use="literal"/>
         </input>
         <output>
            <soap:body
            namespace="urn:testservice:plus"
            use="literal"/>
         </output>
      </operation>
   </binding>

   <service name="Test_Service">
      <documentation>WSDL File for TestService</documentation>
      <port binding="tns:Test_Binding" name="Test_Port">
         <soap:address
            location="http://10.22.68.59:8099/SOAPDemo/server.php" />
      </port>
   </service>

</definitions>
```

Now in `SoapServer` we don't need to put `uri` data in `options`. We can even skip `options` entirely like this. Instead, we exchange `NULL` with the location of our WSDL file.

```
$server  =
   new SoapServer('http://10.22.68.59:8099/SOAPDemo/TestService.wsdl');

class TestService{
   public function hello(){
      return 'Hello Binus';
   }

   public function plus($a, $b){
      return $a + $b;
   }
}

$server->setClass('TestService');
```

```
$server->handle();
```

And in SoapClient we do the same. Again, with WSDL the `options` part is not necessary.

```
$client =
   new SoapClient('http://10.22.68.59:8099/SOAPDemo/TestService.wsdl');

echo $client->hello(); //print Hello Binus

echo $client->plus(5, 8); //print 13
```

WSDL can be viewed with text editor or web browser as it is simply an XML document. Thus to know what service is provided by a server, we only need to read the WSDL file. Even better, with WSDL our client could know what function is available from the start. We can print it with `__getFunction()` method in SoapClient.

```
echo '<pre>';
print_r($client->__getFunctions());
echo '</pre>';
```

WSDL Generator Library

Although WSDL is the norm, PHP provides no native library to generate it automatically. As such, we need to build it ourselves. The process is error-prone as WSDL does not allow any mistake in the parsing. Fortunately, there are many libraries that provides framework for SOAP server in PHP which of course includes WSDL generation capabilities. Some of them are listed here.

1. NuSOAP - is a ready-use SOAP library for PHP which provides classes to easily create SOAP server and client. NuSOAP is actually created to provide SOAP capabilities back when PHP has not yet provided any native SOAP classes. NuSOAP server is bundled with automatic WSDL generator so we do not need to create one manually.
   Link: http://sourceforge.net/projects/nusoap/

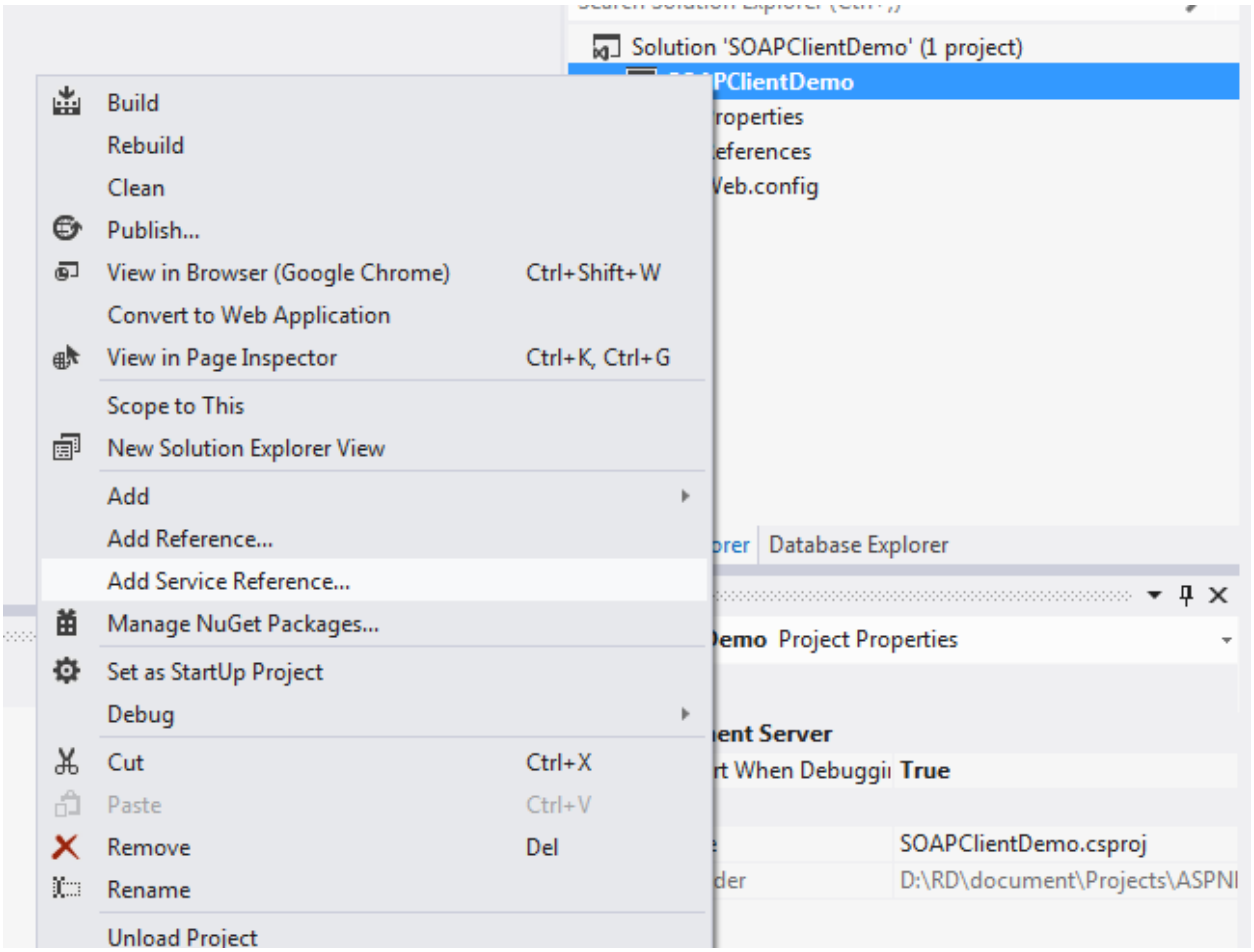
2. PHP2WSDL - is a package by Protung Dragos which provides WSDL generation based on the PHP class used in our web service. It works by parsing the methods in PHP class.
   Link: http://www.phpclasses.org/package/3509-PHP-Generate-WSDL-from-PHP-classes-code.html

3. WSO2 Web Service Framework – is a ready-use web service framework for PHP. This framework is quite complex, more than NuSOAP, and intended for enterprise-level web service usage.
   Link: http://wso2.com/products/web-services-framework/php/

Test Case: Connecting Different Technology

Finally armed with WSDL, our web service is now portable for any technology to connect to as long as they follow SOAP standard. Let's try using ASP.NET website to consume our PHP web service.

Consuming a SOAP web service in ASP.NET using Visual Studio is easy to do with its integrated user interface. What we need to do is to add a Service Reference to our project.

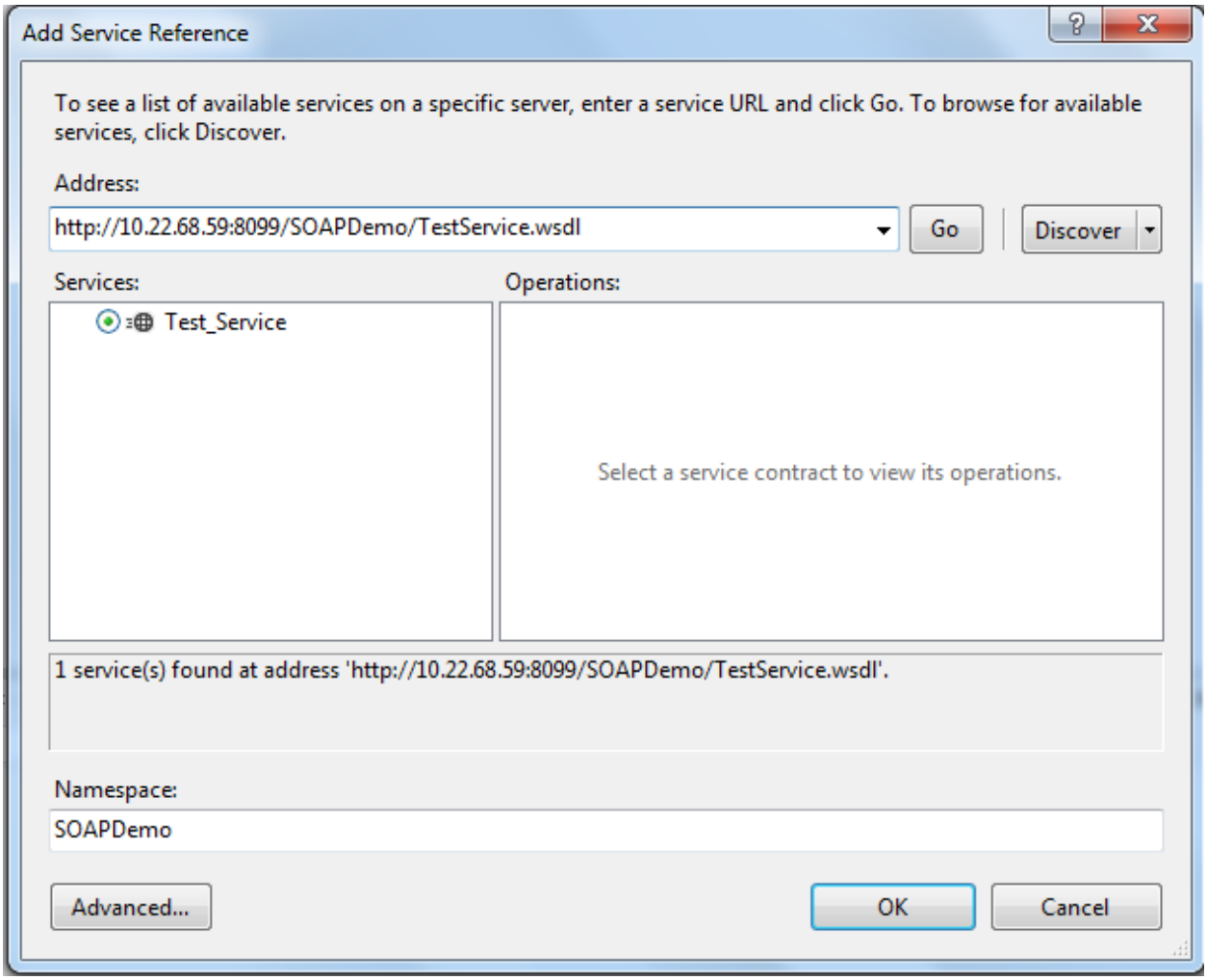


It will open the Add Service Reference dialog. This dialog is used to import services into our programming environment, usually within its own namespace. In this way, we can use services as if they are a class.

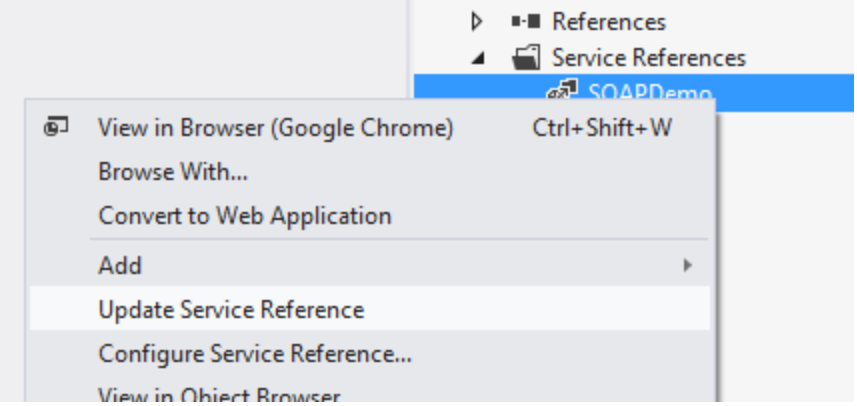
Put the URI for our WSDL file in Address field, and then click Go button. If it succeeded, then the services defined in WSDL will be shown in the left panel. In this sample, we use the previous WSDL and so we

only have 1 service: TestService. If the reference could not be loaded, there might be some problem or mistake in your WSDL file.

The last thing to do would be to specify our desired namespace and then click OK.



Our service will appear under Service Reference folder. It is now ready to be used. Note that if by any chance we made changes to our services WSDL, we need to update it by right-clicking it and select "Update Service Reference".



Now let's try consuming this service. We'll use a simple ASP.NET web form to call and then display the result of functions in our TestService. First create a new ASP.NET web form. I named it Default.aspx so it will be called first by default. The HTML content would be something like this. I am using Visual C# as my development language. Feel free to use Visual C++ or Visual Basic if you want.

```
<%@ Page Language="C#"
    AutoEventWireup="true"
    CodeBehind="Default.aspx.cs"
    Inherits="SOAPClientDemo.Default" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>SOAP Client Demo</title>
</head>
<body>
    <form id="DefaultForm" runat="server">
    <table>
        <tr>
            <td>Return value for hello</td>
            <td><asp:Label runat="server" ID="helloText"></asp:Label></td>
        </tr>
        <tr>
            <td>Return value for plus</td>
            <td><asp:Label runat="server" ID="plusText"></asp:Label></td>
        </tr>
    </table>
    </form>
</body>
</html>
```

Now in Default.aspx.cs when the form loads we put code as follows.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace SOAPClientDemo
{
    public partial class Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            //create service object
            SOAPDemo.Test_PortTypeClient service
                = new SOAPDemo.Test_PortTypeClient();
            //call hello method
            helloText.Text = service.hello();
            //call plus method
            plusText.Text = service.plus(23, 11).ToString();
        }
    }
}
```

`SOAPDemo.Test_PortTypeClient` is the name of our service's class. `SOAPDemo` is the namespace we've specified earlier, while `Test_PortTypeClient` is the name of our `portType` tag in WSDL. By default, Service Reference would add "Client" after the `portType` name. If we have 2 or more `portType` tags in our WSDL file, we would have more classes to make.

The rest of the code is simply calling the operations in `Test_PortType` as if we're calling a method.

Representational State Transfer (REST)

The biggest opposition to SOAP is the complexity to learn it. SOAP uses its own protocol in addition to commonly-used HTTP and such web programmers need to learn more about SOAP rules and standards. Some developers looked back on current web protocols, like HTTP, and thought to themselves “hey, our protocol is good enough! No need to have a new one!”

Thus RESTful web services started to gain popularity.

REST Is Not Protocol

The term Representational State Transfer is first coined by Roy Fielding in his doctoral dissertation at UC Irvine. REST is used as constraints in software architecture focusing on connection constraint between elements in software system. We have actually used REST in the internet architecture without needing to know about it. This is because REST’s constraints are described as follows.

1. Client-Server
   System is separated into Client and Server part. Client focuses on requesting and displaying data from Server while Server gives data responses, most of the time processed, depending on what the Client is requesting.
2. Stateless
   No Client context is being stored in Server between requests. Each request has enough information for the Server to provide adequate response. Server might store Client context in external storage, like database, only for important session state like authentication data. The less Client context is needed, the better.
3. Cacheable
   Response can be cached by Client to increase interaction speed. As cached response would be outdated, responses would need to identify themselves as cacheable or not depending on whether the data needs to be updated frequently or not.
4. Layered System
   Client only know that it is connected to an end point, not necessarily a Server. Likewise, a Server doesn’t need to know whether it is connected to a Client or not. That means connecting an intermediary point like secondary server between Server and Client would be applicable. This allows for ease of scalability.
5. Code-on-Demand (Optional)
   Allow for temporary enhancement of Client capabilities with codes transferred from Server, for example JavaScript or Java Applets. This constraint is optional for REST architecture.
6. Uniform Interface
   Applies a standard interface for resources and their manipulation. This standard is free from any internal implementation. For example, a Server would send data in form of HTML or XML regardless

of its internal implementation (Server might uses Windows or Linux-based system; C++, Java, or PHP as its programming language).

Have you ever seen a technology like this? That's right. The **internet** that we are using right now depends on this architectural implementation. HTTP and its document formats is the best representation of REST architecture.

As **REST is not a protocol**, there is no standard or format like SOAP. HTTP is a protocol. REST is software architecture. While HTTP implements REST's constraints, **it does not mean that REST would always be HTTP**. Any technology can be RESTful if it implements the constraints described above.

REST as Web Service

The idea in using REST is that we don't need a new protocol: HTTP is sufficient for our data passing needs. In accordance to uniform interface constraint of REST, anything we need to do should be following HTTP standard and no more. The advantage of this is that we all are already drenched deep in HTTP anyway with our heavy usage of internet so learning a few new syntax would not take much of our time.

Here's how it works. HTTP defines some methods (or more famously called **verbs**) which is called on each HTTP request-response. The methods determine what to be done and what data to be returned. Here are the methods.

Verb	Description
GET	Retrieve representation of data in specified resource. In browser, this resource is usually in form of URI address and the representation is a file (mostly HTML)
HEAD	Same as GET but we only take the response header data, usually to get metadata
POST	Send data in message body and request server to take it as new information. Usually we use this to send data to be saved in the server
PUT	Send data to be saved in specified resource. If data already exists in resource, then update the data with the new one. Similar to copy and replace in Windows system
DELETE	Delete data in specified resource
TRACE	Return back message to see changes in intermediary server (if any)
OPTIONS	Return supported HTTP verbs. Usually to check whether server support needed verbs
CONNECT	Request connection to transparent TCP/IP tunnel
PATCH	Applies partial modification to resource

And now you see how complicated everyday browsing actually is behind the scene.

From the list of HTTP methods above, four of them caught our attention because they are similar to the four principal actions in database system: CRUD (Create, Read, Update, Delete). The four methods are as follows.

HTTP Verb	CRUD Equivalent	Usage
GET	Read (Select)	Retrieve data
POST	Create (Insert)	Insert new data
PUT	Update	Update existing data
DELETE	Delete	Delete existing data

Another advantage of using REST with HTTP as web service is that all of this is already there in our browser and internet technology. We used GET everyday every time we typed something in URL bar. We used POST every time we register for a new account. Only PUT and DELETE would be rare for everyday user because it involves manipulation of server data, but website administrator would be familiar with it.

URI as Resource

When using REST with HTTP as web service, we would need to view URI address as **resource**. For example, an URI in location "http://www.example.com/users" would denote user resource. Depending on HTTP verb used, data on this resource would be manipulated. Here's an example.

HTTP Verb	Effect on Users Resource or Returned Data
GET	Retrieve list of users with URI address to each single data detail
POST	Insert new user into the server
PUT	Update entire user list OR return error
DELETE	Delete entire user list OR return error

Did you notice that PUT and DELETE has the option to return error? That means sometimes you don't want to update or delete the **entire** data list. That would be catastrophic error especially if the data is important.

In exchange, we might specify the resource better in the URI location. Let's say that each user is defined with their unique username. To point to a specific user we would say something like "http://www.example.com/users/binuslover". Now the HTTP verb would give different result.

HTTP Verb	Effect on Users Resource or Returned Data
GET	Retrieve the detail for user "binuslover"
POST	return error
PUT	Update data for user "binuslover"
DELETE	Delete user "binuslover"

As you can see, the effect got narrowed down to a single user rather than the entire list of users. Note that POST return error now because Create / Insert operation only deals with the entire list (as it adds new item to list. There is no meaning with performing insert on a single row of data).

We can make conclusion that RESTful Web Service, that is web service which conforms to REST constraints using HTTP, is a combination of **meaningful URL** and **HTTP verbs**.

A Practical Example of PHP RESTful Web Service

It would be clearer if we dive straight into an example of how RESTful Web Service could be implemented. The system that we will make would be a simple one which only has one service. However, what the service could do would depend on the HTTP verb used.

Remember that **REST is not a protocol**. This example is just one way to implement RESTful Web Service. As long as we follow REST constraints, any system can be called RESTful Web Service. Use your creativity to build your own system and, better yet, conceive a more efficient one!

System Overview

Our Web Service system would be centered in a PHP file called service.php. Any request would enter this PHP file and be processed depending on the request method. Therefore, service.php would need to know the request method and act accordingly.

We will consume our services using JavaScript AJAX technology. To make it easier for us to use AJAX, we will use JQuery library for its built-in AJAX method.

Redirecting with .htaccess

To make sure that user can use URL address to its fullest, we will redirect any request to service.php using .htaccess file. Here is the code. Save this code as a .htaccess file and put it in your project folder.

```
RewriteEngine on

RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_URI} !service\.php
RewriteRule ^service/(.*)$ service.php [NC,L,PT]

RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_URI} !service\.php
RewriteCond %{REQUEST_URI} !index\.php
RewriteRule ^.*/?$ index.php [L,PT]
```

As you can see with this .htaccess, if we call “service/” in URL address followed with anything, it will be redirected to service.php file. If we simply call anything else, we will be redirected to index.php.

Web Service Class

The easiest way to make collection of functions is to create a class containing all of those functions. Let's make a TestService class to simulate our CRUD operations. In this example we will use an array as data source, which is instantiated when the class is created. In real life situation, we will most likely use database connection as data source.

```
class TestService{
   private $data;

   public function __construct(){
      $this->data = array('joko', 'budi', 'andre', 'rusman');
   }

   public function getAllData(){
      return $this->data;
   }

   public function getSingleData($index = null){
      $count = count($this->data);

      if($count > 0 &&
         is_numeric($index) &&
         $index >= 0 &&
         $index < $count)
      {
         return $this->data[$index];
      }

      return false;
   }

   public function insertData($name){
      $this->data[] = $name;
      return true;
   }

   public function updateData($name, $index = null){
      $count = count($this->data);

      if($count > 0 &&
         is_numeric($index) &&
         $index >= 0 &&
         $index < $count)
      {
         $this->data[$index] = $name;
         return true;
```

```
      }

      return false;
   }

   public function deleteData($index = null){
      $count = count($this->data);

      if($count > 0 &&
         is_numeric($index) &&
         $index >= 0 &&
         $index < $count)
      {
         array_splice($this->data, $index, 1);
         return true;
      }

      return false;
   }
}
```

URL manipulation

Our skeleton framework for REST service would utilize URL parsing. Basically, we want our web service to be invoked when we say "service" in the URL address accompanied with appropriate service name and (optionally) additional parameters. So a call to our service might look like this.

```
http://www.example.com/service/test/2
```

Where **test** is the service name and **2** is the optional parameter. The inclusion of "service" before them is simply our naming choice in the .htaccess file. There is no rule which states that you can't change it to something else or even bypass it completely.

So the idea is we want to get this **test** and **2** part of our URL. We achieve this with `parse_url` function in PHP and a few built-in variables.

```
if($url = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH)){
   $basepath = pathinfo($_SERVER['PHP_SELF'])['dirname'] . '/service/';
   $options = explode('/', trim(str_replace($basepath, '', $url), '/'));
   $service = isset($options[0]) ? $options[0] : false;
   $param = isset($options[1]) ? $options[1] : false;
   $method = $_SERVER['REQUEST_METHOD'];

   //service region
   ... -> we will code this part in a moment
```

```
}
else header('index.php');
```

There is a lot going on in the code above. Using `parse_url`, we will process the URL

```
http://www.example.com/service/test/2
```

into this string

```
/service/test/2
```

Now, following our schema, "service" doesn't really mean anything but a way to tell that we want to use the web service part of our website, not the usual user display pages. So we need to take out this /service/ part from our URL string. We achieve this by using `str_replace` to replace it with empty string ("") and then erase any "/" left in the front and end part using `trim`. The result is this string

```
test/2
```

This string is converted into array using `explode` with "/" as separator. Then we simply say that the first index is the service name and the second index is the additional parameter. It is advised to follow how the code above checks whether the array index has been set or not because we can't tell expect user to always follow the "correct" way when calling our service URL.

Finally, to find out the request method we simply take `$_SERVER['REQUEST_METHOD']` built-in variable.

Implementing HTTP verb

With this information we can start making our service constraint. We will continue the code above in "service region" area.

Here is the bare bone concept for implementing the HTTP verb.

```
//service region
switch($service){
   case 'test':
      $svc = new TestService();
      switch($method){
         case 'GET':
            break;
         case 'POST':
            break;
         case 'PUT':
            break;
         case 'DELETE':
            break;
      }
```

```
        break;
    default:
        echo json_encode(false);
}
```

We opt to use `switch` because it offers a simple-to-read interface. First we check what kind of service the user is calling (stored in `$service` variable). If it doesn't match any, then the `default` is called which return `false` to client.

When the service exists, we now check what kind of request method is used by the client (stored in `$method` variable). In our "test" service, first we instantiate our `TestService` class and then we begin the method selection. Let's see it one by one.

GET method is simple. If `$param` exists (the additional parameter in URL), then we call `getSingleData` in `TestService`. However, if not, then we take all of the data using `getAllData`.

```
            case 'GET':
                if($param)
                    echo json_encode($svc->getSingleData($param));
                else
                    echo json_encode($svc->getAllData());
                break;
```

POST works differently. Remember that POST must not allow any additional parameter because using parameters in the URL usually means focusing on a specific resource detail (for example, a single user out of the list of users). Therefore in POST we would return `false` to client if `$param` exists. Also, we need to check if the POST data is sent by client (in this example we want a POST value with name "name"). If not, we also return `false`.

```
            case 'POST':
                $input = isset($_POST['name']) ? $_POST['name'] : false;
                if($param || $input == false)
                    echo json_encode(false);
                else
                    echo json_encode($svc->insertData($input));
                break;
```

PUT works the other way around. We don't usually want to update the **entire** data list. Therefore, when using PUT we want the `$param` to always exists. Then we need to check if the update data in PUT is sent by client.

```
case 'PUT':
    //read PUT data
   parse_str(file_get_contents("php://input"),$put);
   $input = isset($put['name']) ? $put['name'] : false;
   if($param && $input !== false)
      echo json_encode($svc->updateData($input, $param));
   else
      echo json_encode(false);
   break;
```

DELETE is like PUT except client doesn't need to send any data. It simply takes the `$param` value and then proceed to delete data.

```
case 'DELETE':
   if($param)
      echo json_encode($svc->deleteData($param));
   else
      echo json_encode(false);
   break;
```

And now our web service is done. Of course in real life situation we would have to implement authentication and validation to POST, PUT, and DELETE services because it involves changes to the data. The implementation of them is, however, different to each problem domain and so this module will not cover them.

Consuming Our Web Service

The easiest way to test our web service is by calling it using URL address in browser, as it is a GET operation. For POST we need to use HTML form to send our data. PUT and DELETE, however, is not supported by HTML standard and so not all browsers may support it.

REST Web Service is easier to implement in scripting language like JavaScript because there is no need to create XML message format like the one in SOAP. In REST all we need to use is the URL and HTTP verb, so it fits JavaScript perfectly especially with AJAX. In this example we will try to consume our service using AJAX with JQuery.

We only need JQuery and JavaScript for this, so our HTML will mainly be empty except for the script. For example, here is how I call the GET service. Suit the `url` string with your service's location.

```
<!doctype html>
<html lang="en">
<head>
  <title>REST Demo Service Home</title>
  <script type="text/javascript" src="jquery-1.11.1.min.js"></script>
  <script type="text/javascript">
      $(document).ready(function(){
         $.ajax({
            'url': 'http://10.22.68.59:8099/RESTDemo/service/test/',
            'type': 'GET',
            'data': {},
            'dataType': 'json',
            'success': function(result){
               document.write(result);
            },
            'error': function(error){
               document.write(error.responseText);
            }
         });
      });
   </script>
</head>
<body></body>
</html>
```

This will return, as expected, the list of data in the service. If we call it with additional parameter in the `url`, we will get the specific data.

```
         $.ajax({
            'url': 'http://10.22.68.59:8099/RESTDemo/service/test/1',
            'type': 'GET',
            'data': {},
            'dataType': 'json',
            'success': function(result){
               document.write(result);
            },
            'error': function(error){
               document.write(error.responseText);
            }
         });
```

To insert a new data, we use the POST verb.

```
$.ajax({
   'url': 'http://10.22.68.59:8099/RESTDemo/service/test/',
   'type': 'POST',
   'data': {
      'name': 'ultraman'
   },
   'dataType': 'json',
   'success': function(result){
      document.write(result);
   },
   'error': function(error){
      document.write(error.responseText);
   }
});
```

To test the other HTTP verbs, simply change the `url`, `type`, and `data` properties in AJAX method as necessary.

SOAP or REST?

SOAP Advantages and Disadvantages

As SOAP is a protocol by itself, it has standardized notations and templates which mean developers would have same mindset and knowledge on how to implement it. SOAP defines a formal contract for client and server communication. Another good feature of SOAP is its WSDL documentation format which not only provides information on what services are available to use by client but also relatively readable by human because it uses XML.

However, SOAP is quite complicated to learn as it is a different technology than existing HTTP. SOAP request and response messages also require bigger space because it is wrapped in XML. SOAP also doesn't have built-in error handling so developers need to handle it by themselves.

REST Advantages and Disadvantages

REST uses existing technology like HTTP and HTML and thus developers already know the basics. REST is also simple. There is no need for additional document like WSDL or wrapping requests and responses in XML wrapper. HTTP has built-in error handling and many useful features which are ready to be used by REST services from the start.

The simplicity that is REST is also its disadvantage. First, there is no formal documentation like WSDL and so there is no assurance that a service is available in the server. To know about a service, clients need to search by trial-and-error or ask the server administrator directly (WSDL 1.2 is actually an attempt to clear this problem as it is also geared toward RESTful web services). REST also doesn't have a formal standard as it is merely architecture, so implementation might differs from one developer to another.

Conclusion

There is no definite answer over which style of web service is better. Both SOAP and REST is a viable option when developing web service. With SOAP, developer builds the system with SOAP protocol and standard while REST utilizes HTTP as its protocol.

As a rule of thumb, use SOAP when the web service requires formal contract between client and server and use REST when the restriction is less rigid. REST is also better if there is no time to train developers in SOAP because most developers should be familiar with HTTP already.

Bibliography

W3Schools on Web Services

http://www.w3schools.com/webservices/

World Wide Web Consortium (W3C) WSDL specification

http://www.w3.org/TR/wsdl

World Wide Web Consortium (W3C) HTTP specifications

HTTP 1.1 specifications: http://www.w3.org/Protocols/rfc2616/rfc2616.html

List of specifications superseding HTTP 1.1: http://www.w3.org/Protocols/Specs.html

PHP SOAP Documentation

http://php.net/manual/en/book.soap.php