

# An Introduction to Writing Web Service APIs

Symsource

Published by the Symbian Developer Network

Version: 1.0 – March 2009

<b>1</b>	<b>INTRODUCTION</b> .....	<b>2</b>
<b>2</b>	<b>WEB SERVICE PROTOCOL STACK</b> .....	<b>2</b>
	2.1 MESSAGE PROTOCOLS .....	2
<b>3</b>	<b>GOLDEN RULES FOR MOBILE-FRIENDLY WEB SERVICES</b> .....	<b>5</b>
	3.1 TOP TIPS SUMMARY .....	5
	3.2 DATA STRUCTURES .....	6
	3.3 CONNECTION ISSUES.....	6
	3.4 TRANSFERRING DATA .....	7
	3.5 COMPRESSION AND DELTAS.....	8
	3.6 ATOMIC COMPOUND APIS.....	9
	3.7 ERROR CODES .....	9
<b>4</b>	<b>SUMMARY</b> .....	<b>9</b>
<b>5</b>	<b>REFERENCES</b> .....	<b>10</b>
<b>6</b>	<b>AUTHOR BIOGRAPHY</b> .....	<b>11</b>

# 1 Introduction

A web service is a web-based software system or application which offers its services to remote client applications via a defined API. Web services can range from simple applications providing information on the weather to more complex services for managing online photo collections.

Offering a web service to mobile clients is a simple way to capture a much larger audience, and designing a good web service API can save time and money when developing a mobile client. There are also additional benefits to end users, who will have access to interesting and timely content.

This paper is intended for developers who are about to design a web services architecture or are planning to update an existing API to integrate mobile clients. Only recently have the hardware capabilities of mobile clients started to become comparable to desktop clients in terms of processing power and memory available for handling complex data types. Some mobile clients are still limited in this regard; many can still suffer unreliable data connections with limited bandwidth and have a limited power supply.

We will examine the available message protocols for their suitability for mobile clients, and then walk through some golden rules for making your web services API more mobile friendly. We will also look at how to avoid some of the common pitfalls that you might not come across when the client for your API is a desktop web browser or application, rather than a mobile phone.

## 2 Web Service Protocol Stack

A web service is defined by the World Wide Web Consortium (W3C) as 'a software system designed to support interoperable machine-to-machine interaction over a network' [1]. In practice, this means that any service offering an interface for remote client applications that uses one or more technologies from the web domain, typically building on HTTP and XML, needs to offer something other than a web browser-only service.

Enabling web services to find each other and to interact requires a set of protocols known as a web service protocol stack. This is a set of four protocols:

- Message protocol: this protocol specifies the format of the message. Typically the format is XML based. Examples include XML-RPC and SOAP.
- Transport protocol: this protocol controls how messages are sent between machines on the network. Examples include FTP, HTTP and SMTP.
- Description protocol: this protocol describes the public interface of a web service. The most common example is the WSDL format [2].
- Discovery protocol: this protocol provides a mechanism to publish the location of a web service with a description of its capabilities. The most notable is the UDDI specification [3].

This paper focuses on web service messaging protocols in the Message protocol layer. The additional layers will be discussed in the context of each specific message protocol.

### 2.1 Message protocols

There are a wide variety of message and transport protocols used in web services today, ranging from simple HTTP GET and POST based requests to complex message protocols such as the industry standard SOAP. Others are not necessarily based on strict standards, but are still de facto standards, for example, the Representational State Transfer (REST) protocol.

The following sections describe the most common web service message protocols.

### 2.1.1 XML-RPC

XML-RPC is a simple protocol created by Dave Winer of UserLand Software in 1998. As the name suggests, it uses XML to perform remote procedure calls. XML-RPC uses HTTP as the transport protocol and has a lightweight XML structure to package a method call with its associated parameters, or a response from the server. Here is an example method call from [4]:

```
<?xml version="1.0"?>
<methodCall>
  <methodName>.getStateName</methodName>
  <params>
    <param>
      <value><i 4>40</i 4</value>
    </param>
  </params>
</methodCall>
```

Due to its lightweight structure, XML-RPC is simple for a developer to understand. It's also easy for a developer to write client software to parse and compose, and is suitable for mobile client web services.

A typical use of the XML-RPC protocol is in blogging web service APIs. Along with creating the XML-RPC specification, Dave Winer also created the XML-RPC MetaWeblog API in 2002 [4], building on the existing Blogger API. Most blogging web sites provide support for both the existing Blogger API and the MetaWeblog API.

### 2.1.2 SOAP

The SOAP v1.2 specification became a recommendation by the World Wide Web Consortium (W3C) in 2003, and is an evolution of the XML-RPC protocol. SOAP uses XML as its message format and can be used over multiple communications protocols such as HTTP and SMTP, but most commonly uses HTTP.

A SOAP message packages a client request in the form of a function method name and parameters, along with a set of headers. The message is wrapped in a special SOAP XML structure called a SOAP envelope. It is up to the server to parse the message to perform the necessary call. The server also packages a response to the client in the same SOAP XML message format.

Here is an example of a SOAP message provided by the W3C [5]:

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
      <n:priority>1</n:priority>
      <n:expires>2001-06-22T14:00:00-05:00</n:expires>
    </n:alertcontrol>
  </env:Header>
  <env:Body>
    <m:alert xmlns:m="http://example.org/alert">
      <m:msg>Pick up Mark at school at 2pm</m:msg>
    </m:alert>
  </env:Body>
</env:Envelope>
```

The WSDL (Web Services Description Language) provides a mechanism to describe a SOAP message or messages and how these messages are exchanged between client and server. A WSDL document is simply XML, so it is human readable. Because it is an XML document, it is also

machine readable and there are a multitude of add-on tools available which can automate the creation of skeleton software in a variety of programming languages (for example, C++, Java, C# and Symbian C++ [6]) to create and parse SOAP messages in the format specified in the WSDL file.

SOAP has a heavyweight XML format which can be overcomplicated for Web Service APIs, which are simply message passing services. It may therefore not be ideal for mobile devices where bandwidth is restricted and processing power available for complex data types is limited. Note that non-SOAP-based XML-based message formats can also quickly become overloaded too, for instance, by using XML namespaces and excessively long node names.

Where the SOAP protocol does provide benefits is with more extensive Web Service APIs, with multiple data parameters or return types. XML is very flexible at describing data but SOAP attempts to apply more structure to describing and serializing these data items.

### 2.1.3 REST

The REST message protocol is not a protocol in the strictest sense as it is not based upon a strict industry standard. Instead, REST is a set of guidelines for a software architectural style coined by Roy Fielding in his PhD dissertation [7], and has since become a de facto standard. The principles of the REST approach describe how to develop a distributed software system, a good example being the World Wide Web itself.

In a REST-style architecture, objects of interest are defined as a 'Resource' and each resource is given a globally unique identifier. Any client can request this object, or at least a representation of the object, using this global identifier.

A REST web service does not package a function call and parameters within a message sent from client to server. Instead, each client request is either a request for a representation of a 'Resource' or a request to create, update or delete a 'Resource'.

The interface to a REST resource is a CRUD (Create, Read, Update, Delete) interface.

A traditional web server is a good example of a REST-style software architecture. Every resource on the web can be accessed by a globally unique ID (URI); HTTP provides a CRUD interface (POST, GET, PUT and DELETE) and a defined set of content types specified by a MIME type.

HTTP is also stateless, an advantage we will come back to later in this paper. Each HTTP request contains all the information required to understand the purpose of the request.

The REST approach offers some advantages:

- The message protocol can be very lightweight (for instance, comma-separated values or very basic XML).
- The transport protocol is stateless, so different servers can be used to service multiple client requests, among other advantages.
- Response times are faster due to the way representations are cached. This also reduces the load on the server. This can be useful if the representation of your data changes infrequently, for example, Amazon product information, and the hit rate on this data is high.

The REST approach also has disadvantages:

- REST is a pattern rather than a specification and implementations are not uniform, so there is no standard definition of the client server interface (for example, a WSDL file). The interface needs to be communicated in a different manner such as plain text documentation.

- There are no tools available to automate client side development, although, in practice, implementing a REST client still requires less effort than manually populating a WSDL document. However, there are tools available to automatically generate WSDL.

Due to its lightweight and stateless nature, the REST approach is well suited for use by mobile clients. According to usage statistics from popular web service API web sites [8], the REST approach is also the most commonly used by a significant margin.

### 3 Golden Rules for Mobile-Friendly Web Services

Web service APIs come in many shapes and sizes, and some offer more support for mobile clients than others. Since the technology is still maturing, examples can still be found of APIs that, for example, return chunks of HTML or self modifying Javascript code, which implies that the client contains a web browser or requires far more effort from a developer using C++ or Java ME rather than C#, Perl or Python.

Therefore, the following sections contain guidelines for those intending to develop a web service that is planned to be used by mobile clients (not necessarily exclusively) or for those with existing web services interested in optimizing the service for mobile clients.

After following these rules you may find an existing API does not conform. If it is not practical to change an existing service API, it may still be possible to provide a service layer to the web service just for mobile clients, to handle session authentication, caching of incomplete transfers and wrapping of APIs. This may still be less effort than writing client code to work around issues with the web service API and will provide benefits to the user such as reduced bandwidth usage.

#### 3.1 Top tips summary

Let's start with a summary of best practice which will be expanded upon later.

The tips are intended to reduce the effort required to write mobile clients and also to enhance the experience for the end user, so as to minimize the impact on bandwidth and power cost on the mobile device, and improve speed and responsiveness. A well designed API will gain advantage from being suitable for all manner of clients, not only mobile. Rich desktop clients, web apps and mashups will benefit from a well designed, client-neutral API.

- **Use simple data structures**  
Use very simple data structures such as comma-separated values, field value pairs or JSON, or very simple XML. Avoid heavyweight XML, SOAP and WBXML.
- **Use stateless APIs**  
Where practicable, use HTTP and basic access authentication with every request, removing the need for a session.
- **Minimize data transfers**  
Mobile clients are prone to connection issues, so the service should have provision for clients to resume downloads of large data files.  
Data should be compressed and delta encoded.
- **Keep transactions atomic**  
Multiple single purpose API calls can be compounded to reduce data transmission and to simplify state and clean-up.

- **Return useful error codes**

Rather than make the client test for state before submitting requests, the client should make the request and the service API should return an appropriate error code.

### 3.2 Data structures

Web service APIs intended for mobile clients should use basic data structures such as field value pairs, comma-separated values, or very simple XML.

Mobile platforms including Symbian, iPhone, Windows Mobile and Java ME offer extensive APIs for the parsing of XML and string-based message formats. Heavyweight XML message formats such as SOAP should be avoided if possible, as the SOAP message format adds additional complexity on already resource constrained devices.

An alternative to field value pairs is the more advanced JSON (JavaScript Object Notation). JSON is a very lightweight language-independent format for data interchange. However, out-of-the-box support for JSON is somewhat limited across mobile platforms apart from Android. Third-party implementations are available [10] [11].

Here is a small example taken from the JSON specification[9]:

```
{
  "Image": {
    "Width": 800,
    "Height": 600,
    "Title": "View from 15th Floor",
    "Thumbnail": {
      "Url": "http://www.example.com/image/481989943",
      "Height": 125,
      "Width": "100"
    },
    "IDs": [116, 943, 234, 38793]
  }
}
```

Despite its frequent application in the mobile space, the use of WBXML should be avoided. WBXML is the worst of both worlds, neither as readable as XML nor as optimal as pure binary.

If using HTTP as the transport protocol, custom HTTP headers are best avoided, as mobile networks can use transcoders that may edit or filter out unrecognized HTTP headers.

### 3.3 Connection issues

Mobile clients of web services will suffer from connection and resource constraints. Data connections can easily be dropped with a loss of signal strength, a user could change or roam to a different network or could switch to a WiFi connection at any time. Maintaining a data connection is also power hungry and can be very costly as well.

There are several ways to help mitigate these problems, the first of which is to choose a stateless web service protocol when designing a web service. This reduces the need for a continuous active connection. Another method is to make your API sessionless.

#### 3.3.1 Sessionless APIs

The first step to a sessionless API is to use HTTP as the underlying transport protocol as each HTTP request typically contains all the information necessary for the server or client to understand

the purpose of the request or response. In the web browser world, it is possible to break this behavior and to use cookies to store information and therefore maintain a user session.

The next step to a sessionless API is to provide user authentication with every request. When a session is created, any authentication is done at the beginning of the session. HTTP provides a mechanism to provide authentication with every request.

Basic access authentication is an HTTP mechanism to pass a username and password as part of a HTTP request. For a user with username 'John' and password 'Smith', the format of an HTTP request containing basic access authentication would look something like the following:

```
GET /public/index.html HTTP/1.1
Host: www.myhost.com
Authorization: Basic Sm9obj pTbWl 0aA==
```

where the string Sm9obj pTbWl 0aA== is a Base64 encoded form of the string ' John: Smi th' .

### 3.3.2 Security

The sessionless API approach has its disadvantage as user authentication information is passed over the internet in plain text format and so may pose a security risk. Interception can be prevented by network security protocols such as TLS (Transport Layer Security), also known as SSL (Secure Sockets Layer).

There is a catch: some mobile devices don't recognize as many certificate authorities as desktop browsers; this is especially true for mobile clients with proprietary operating systems, so it is worth verifying which certificate authorities are supported and whether this can be extended.

### 3.4 Transferring data

It is important to consider if the data being transmitted via a web service is appropriate for the available data connection. Mobile means moving, and even a laptop using a WiFi connection is unlikely to stay in the same location for long.

One solution is to break large data files into smaller chunks. This has the advantages that if a connection failure occurs, only that chunk of data is lost, and breaking the data into smaller chunks reduces the amount of server time required to service each request. Note that there may be a performance and resource issue with this approach on the client-side, for instance, if the client chooses to download multiple chunks in separate threads.

Alternatively, the client may request data, passing a byte offset as an argument to the server, which will then start to send the remaining data.

A simple solution for sending a large file from server to client would be to provide a URL to the resource within a response, allowing the data to be downloaded by whatever means the client finds most appropriate, such as FTP, WebDAV, etc.

When sending data over HTTP and both the server and client are using HTTP/1.1, it is also worth considering switching on 'chunked' transfer-coding. This breaks the data into smaller chunks within the response body. Each smaller chunk within the body has its own set of headers describing the size of the chunk, amongst other things. This is especially useful if total length of the data being sent is unknown by the server, for instance, if we have some dynamic data being generated on the server-side.

### 3.5 Compression and deltas

To minimize data transfers, all data sent should be compressed, if this is supported by the chosen transport protocol. For HTTP, gzip compression should be enabled on the server.

However, compression is also a double edged sword. Compressing small amounts of data actually makes them bigger and uses lots of process cycles. So, a balance needs to be found.

If the client has the latest version of data it is also unnecessary to transfer the data again. The client of a GET request can store the time stamp provided in the 'Last-Modified' server response header.

A client can then perform a conditional GET request on the same resource populating the 'If-Modified-Since' request header with this time stamp. If the resource has not changed, the server will not send the body back to the client in the response, but instead will return an HTTP response code 304 (Not Modified).

Even if the data has changed, delta encoding should be considered. Delta encoding is simply a method of differencing two versions of a resource such as HTML source or an RSS XML feed.

The two supported difference formats are vcdiff [12] and diffe which is simply the output from the UNIX diff-e command.

The following example shows how a client may make a conditional GET request which will only return a body if modified after the supplied time and date:

```
GET /index.html HTTP/1.1
Host: www.foobar.com
If-Modified-Since: Fri, 16 Jan 2009 16:26:20 GMT
A-IM: vcdiff, diffe, gzip
```

The A-IM header is important as it indicates that the client is happy to accept delta-encoded diffs as a response in either vcdiff or diffe formats. We can also see that the client is happy to accept a gzipped response.

#### 3.5.1 Binary transfers

XML protocols are not an obvious choice for sending binary data. The available methods can be inefficient at best and error prone at worst. If binary data must be sent via XML the most common approach is to encode the binary data as text. This is usually done using the UUencode or Base64 encoding algorithms.

The following XML snippet has encoded some binary data within the <test> tag using the Base64 method:

```
<m:data xmlns:m='http://www.myhost.com/test' >
  <test>dGVzdGRhdGE=</test>
</m:data>
```

The disadvantage of this method is that the size of the data transmitted can be up to 15% larger than its binary equivalent. This is not so important when the amount of data to send is small but becomes significant when the size increases. There is also a performance issue as the server or client receiving the encoded data will have to decode it before it can be used.

A common pitfall which can lead to random failures is to insert binary data directly within the XML CDATA tag, as in the example below:

```
<TestData>
  <MyData>
    <![CDATA[! 5B@Z1u1" [LsuP÷Ææ> áûö\ƒXÔÂ<î qÃÎ JË¹É]]>
  </MyData>
</TestData>
```

Here, everything between the start tag `<![CDATA[` and the end tag `>` is interpreted by XML parsers as preformatted text without XML tags or syntax. The main issue here is that, while unlikely, it is possible that the binary data within may coincidentally contain a `']>` end tag. This would terminate the CDATA section prematurely and likely cause the XML parser to fail, resulting in corrupt XML and binary data. So, to use the CDATA section you will need to ensure that you escape out the `']>` end tag using an encoding scheme.

### 3.6 Atomic compound APIs

Consider the following sequence of pseudo API calls to upload a file from client to server:

```

sessi onId = AuthenticateSession(credentials)
return code = CheckFileExists(filename)
UploadFile(file)
return code = CheckFileUploadedSuccessfully(filename)
SignOutSession(sessi onId)

```

This sequence of calls can leave the developer with many different states to track and clean up when things don't work out as expected. This sequence could quite easily be re-factored into a single compound API:

```

return code = UploadFile(filename, file, credentials)

```

As a general rule, if the purpose of an API is to authenticate a session or check a session is still valid, or to check for the existence of a resource, these APIs should be eliminated and this functionality moved into a compound API which returns an appropriate error code.

### 3.7 Error codes

Consider returning success/failure codes from a compound API rather than making the client test for multiple conditions before submitting another request. If you are using a web services protocol over a HTTP, it should be possible to use HTTP response codes to signify success/failure.

For example:

- Invalid authorization: **401 Unauthorized**
- Object already exists (when you don't want it to): **404 Forbidden**
- Object does not exist (when you do want it to): **404 Not Found**
- Out of memory: **507 Insufficient Storage**

## 4 Summary

In this paper, we have looked at the available web service message protocols and discussed which are more appropriate for mobile clients.

We have also gone through some golden rules which offer best practice and, if adhered to, should result in mobile-friendly web services.

Specifications for the web services message protocols we have discussed can be found here:

- The specification for XML-RPC can be found at [www.xmlrpc.com/spec](http://www.xmlrpc.com/spec).
- The SOAP v1.2 specification can be found on the W3C website, at [www.w3.org](http://www.w3.org).
- There is no industry standard for the REST protocol but a good starting point can be found at [en.wikipedia.org/wiki/Representational\\_State\\_Transfer](http://en.wikipedia.org/wiki/Representational_State_Transfer).

Some real world examples of mobile-friendly web service APIs can be found here:

- New Facebook Status, Notes, Links, and Video APIs, which extend the existing Facebook photo uploading and viewing API.  
[developers.facebook.com/news.php?blog=1&story=193](http://developers.facebook.com/news.php?blog=1&story=193)
- Flickr and eBay provide good examples of well documented APIs. The Flickr API offers all the message protocols discussed in this paper and is capable of formatting responses in a variety of formats.  
[flickr.com/services/api/](http://flickr.com/services/api/)
- eBay also offers both REST and SOAP message protocols and can respond in a variety of formats.  
[developer.ebay.com/](http://developer.ebay.com/)

## 5 References

- [1] W3C Web Glossary: [www.w3.org/TR/ws-gloss/](http://www.w3.org/TR/ws-gloss/)
- [2] Web Services Description Working Group: [www.w3.org/2002/ws/desc/](http://www.w3.org/2002/ws/desc/)
- [3] UDDI Specification: [/www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm](http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm)
- [4] MetaWeblog API: <http://www.xmlrpc.com/metaWeblogApi>
- [5] Sample SOAP 1.2 message from W3C: [www.w3.org/TR/2003/REC-soap12-part1-20030624/#firstexample](http://www.w3.org/TR/2003/REC-soap12-part1-20030624/#firstexample)
- [6] Nokia S60 WSDL-to-C++ wizard: [www.forum.nokia.com/info/sw.nokia.com/id/5ddeb939-c4e4-4e64-8f25-282e1e86afed/Nokia\\_WSDL\\_to\\_Cpp\\_Wizard\\_for\\_S60.html](http://www.forum.nokia.com/info/sw.nokia.com/id/5ddeb939-c4e4-4e64-8f25-282e1e86afed/Nokia_WSDL_to_Cpp_Wizard_for_S60.html)
- [7] Architectural Styles and the Design of Network-based Software Architectures dissertation: [www.ics.uci.edu/~fielding/pubs/dissertation/top.htm](http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm)
- [8] Protocol Usage by APIs: [www.programmableweb.com/apis](http://www.programmableweb.com/apis)
- [9] JSON specification: [tools.ietf.org/html/rfc4627](http://tools.ietf.org/html/rfc4627)
- [10] Google Objective-C JSON parser and generator for iPhone: [code.google.com/p/json-framework/](http://code.google.com/p/json-framework/)
- [11] SUN J2ME JSON API: [java.sun.com/developer/technicalArticles/javame/json-me/](http://java.sun.com/developer/technicalArticles/javame/json-me/)
- [12] The VCDIFF Generic Differencing and Compression Data format: [www.faqs.org/rfcs/rfc3284.html](http://www.faqs.org/rfcs/rfc3284.html)

## 6 Author Biography

Symsource's expert engineering team can help you make the most of mobile phone software on Symbian OS and Java platforms, and other key mobile technologies including SMS, WAP and mobile data.

We offer our services for development of applications for Symbian OS, iPhone, Windows Mobile, Java J2ME and BlackBerry, as well as mobile web applications, widgets and SMS services, and soon Android devices as well. We code in C++, C, Objective-C, Java, JavaScript, PHP and Python. And sometimes Lua. Or Ruby.

We can also help with your business case and route to market strategies for taking your existing or startup business into the mobile domain. Symsource's success relies on having the best, most experienced, most capable mobile software development team there is. Our customers know that they can rely on us to deliver on time, on budget and most importantly, solve problems that no-one else can solve