

---

# Web Services Core Programming Guide

Networking, Internet, & Web: Web Client



2009-01-06



Apple Inc.  
© 2009 Apple Inc.  
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.  
1 Infinite Loop  
Cupertino, CA 95014  
408-996-1010

.Mac is a registered service mark of Apple Inc.

Apple, the Apple logo, AppleScript, Carbon, Cocoa, eMac, Mac, Mac OS, and Objective-C are trademarks of Apple Inc., registered in the United States and other countries.

Java and all Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Simultaneously published in the United States and Canada.

**Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY,**

**MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

**IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.**

**THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.**

**Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.**

# Contents

## **Introduction**      **Introduction 7**

---

Organization of This Document 7  
See Also 7

## **Chapter 1**      **About Web Services 9**

---

Web Services Overview 9  
Types of Web Services 9  
Mac OS X Web Services Architecture 10  
About the Web Services API 11  
Types, Method Invocations, and Protocol Handlers 11  
    Types 12  
    Method Invocations 12  
    Protocol Handlers 12  
Using Specific Web Service Types 12  
    XML-RPC 12  
    SOAP 13  
    WSDL 14

## **Chapter 2**      **Using the Web Services Core Framework 17**

---

Creating and Invoking Operations 17  
    The Invocation Reference 17  
    The Operation Parameters 17  
    The Invocation Properties 18  
    Invoking the Operation 18  
    Invoking the Operation Asynchronously 18  
    Dealing With The Response 19  
    Adding Custom Serializers and Deserializers 19  
Example: Calling a SOAP Operation with HTTP Authentication 19

## **Document Revision History 23**

---



# Figures and Listings

## Chapter 1      **About Web Services 9**

---

- Figure 1-1      XML-RPC and SOAP encodings on top of XML on top of HTTP 10
- Figure 1-2      Web Services Core framework 10
- Listing 1-1      RPC-style SOAP envelope 13
- Listing 1-2      Document-style SOAP envelope 13



# Introduction

---

Web services are the interchange of XML-based queries and responses between clients and servers over the Internet or an intranet via standard protocols such as HTTP, HTTPS, or SMTP.

Mac OS X provides support for the client side of these queries and responses, allowing your application to exchange information with remote web servers. Some support for the server side of these operations is also available, primarily the translation between CTypes and XML for SOAP and XML-RPC protocols.

You can communicate with remote servers using Apple events or the Web Services Core framework. Support is provided for using web services from procedural C, Cocoa, or AppleScript. This document describes using the `WebServicesCore` framework from procedural C or Cocoa. For guidance on using Apple events from AppleScript, see *XML-RPC and SOAP Programming Guide*.

The web services API has built-in support for SOAP 1.1, SOAP 1.2, and XML-RPC protocols. The API also supports custom serialization schemes, allowing you to work with other standards or proprietary schemes.

If you are writing an application that needs to exchange information with remote servers using XML over HTTP or HTTPS, you should read this document.

## Organization of This Document

This document consists of two chapters:

- [“About Web Services”](#) (page 9)—a brief introduction to standard methods of exchanging information with remote servers and a general discussion of the web services API.
- [“Tasks”](#) (page 17)—step-by-step examples for accessing web services from a procedural C or Cocoa application using the Web Services Core framework

## See Also

- *Web Services Core Framework Reference*—the reference to the Web Services Core framework functions, callbacks, data types, constants, and error codes.
- *XML-RPC and SOAP Programming Guide*—a guide to using AppleScript and Apple events to obtain access to web services.
- *Event-Driven XML Programming Guide for Cocoa*—how to use `NSXMLParser` to read in and parse XML.
- [Using SOAP with PHP](#)—instructions for writing SOAP clients and servers using PHP.
- [Web Services with AppleScript and PERL](#)—A guide to using AppleScript to access web services and to creating a server using PERL.



# About Web Services

---

## Web Services Overview

Web services provide web-based APIs to support machine-to-machine communication over networks. Because these APIs are web-based, they inherently support interaction between devices running on different architectures and speaking different native languages.

Common examples of web services include weather forecasts, stock market quotes, and book inventories. A server with a database responds to remote queries for data, where the client specifies a particular city, stock symbol, or book title, for example. The client application sends queries to the server, parses the response, and processes the returned data.

All web service schemes utilize a web-based transport mode, such as HTTP, HTTPS, or SMTP, and a method for packaging the queries and responses, typically some sort of XML schema.

Some terminology is unique to web services. Creating an XML file containing outbound messages is sometimes called **serialization**. Extracting information from an incoming XML file is sometimes called **deserialization**. Calling a function or method on a remote server is often called an **invocation**. A web service may expose procedural functions or object-oriented methods. The general term **operation** is used to describe either a function or a method.

## Types of Web Services

The earliest implementations of web services implemented APIs that closely resembled function calls in existing computer languages such as C or Java. These are called remote procedural calls (RPC). A W3C standard has been established for providing web services using XML-based RPC, called XML-RPC. Clients access web services through XML-RPC by calling a series of remote functions, which execute on the server. Parameters are passed and returned in a predefined order. To use RPC-XML services, you must know the URL of the service, the functions that are exposed, and the data type and order of parameters to be sent and received.

A higher-level, more object-oriented approach was later developed called service-oriented architecture. The most popular implementation of this approach is SOAP (formerly Simple Object Access Protocol). In this approach to web services, the client and server exchange messages, rather than making calls and expecting returns. This provides a more loosely coupled interface and is less tied to particular languages. SOAP implementations are extremely common. SOAP parameters are named, rather than sent and received in predefined order, making SOAP calls easier to read and debug. To use SOAP services, you need to know the URL of the service, the methods exposed, and the names and data types of the message parameters.

At a still higher level, there is a definition for a Web Services Description Language (WSDL). This defines an XML document type that describes available web services. WSDL is often used in combination with SOAP to access services over the Internet. A client program connects to a remote server and reads a WSDL file to determine what remote services are available, including a list of operations, parameters, and data types. The client can then use SOAP to call operations listed in the WSDL file.

If a SOAP service is described in a WSDL file, you do not need any preexisting information to access the service except the URL of the WSDL file.

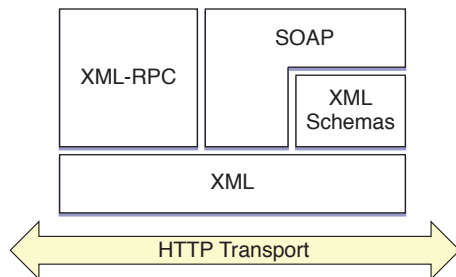
Currently, Mac OS X provides no high-level support for extracting SOAP functions from WSDL files. You can use `NSXMLParser` to read a WSDL file into memory from a URL, however. It is then relatively simple to search the file for a particular method name or to find a URL associated with a service. With a bit more effort, you can extract method names, data types, and parameter names to build a SOAP call. You can then use the Web Services Core framework to make SOAP calls and parse the responses.

**Important:** SOAP data types can be defined within the SOAP message envelope (commonly known as section 5 encoding, from the W3C specification) or they can be defined in external schema contained in WSDL files. The current implementation of the Web Services Core framework supports section 5 encoding *only*. If a web service uses SOAP data types that are not encoded within the SOAP message envelope, there is currently no high-level support for using that web service from Mac OS X. You must use lower-level XML and HTTP messaging functions instead.

## Mac OS X Web Services Architecture

Mac OS X has high-level support for SOAP and RPC, as well as low-level support for XML and HTTP that allows you to access other implementations and schemes. The architecture is illustrated in Figure 1-1.

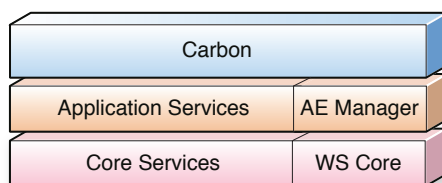
**Figure 1-1** XML-RPC and SOAP encodings on top of XML on top of HTTP



You can make SOAP or RPC calls directly from AppleScript or from within procedural C or Cocoa applications. You can make these calls using either Apple events or the Web Services Core framework.

Web Services Core is a low-level framework that sits alongside CFNetwork, Core Foundation, and Carbon Core, a subframework of Core Services, as shown in Figure 1-2. It is available to all applications, plugins, tools, and daemons.

**Figure 1-2** Web Services Core framework



The framework has no dependency on the window server or login window. It is fully integrated with the Mac OS X system, sitting inside Core Services, and leverages CFXMLParser and CFNetwork. The framework is thread-safe and based on the run loop.

## About the Web Services API

Whether using XML-RPC or SOAP, you use the Web Services Core API to create a call to a server in essentially the same manner:

1. Create a dictionary containing the URL of the server, the name of the operation, and a constant specifying the protocol (XML-RPC, SOAP 1.1, or SOAP 1.2).
2. Create a method invocation ref using `WSMethodInvocationCreate`, passing in the dictionary.
3. Create a dictionary containing the method parameters and their names, and another dictionary specifying their order.
4. Pass these two dictionaries into the invocation ref using `WSMethodInvocationSetParameters`.
5. Pass any additional settings, such as SOAP action headers and debug flags, into the invocation ref using calls to `WSMethodInvocationSetProperty`.
6. Create a callback to handle the response and pass it into the invocation ref using `WSMethodInvocationSetCallback`. This callback parses a dictionary containing the CTypes deserialized from the response.
7. Invoke the procedure using `WSMethodInvocationInvoke` or `WSMethodInvocationScheduleWithRunLoop`.
8. Check the HTML status for authentication challenge or network errors, authenticating if necessary.

The web services API encourages you to asynchronously issue invocation requests on the run loop and receive a reply on your bundle. Because it is CType-based, you have to create CType objects for your strings, records, dictionaries and arrays. If you're an Objective-C programmer, you get "toll-free" bridging with Objective-C types.

Simple data types can be serialized and deserialized using the built-in capabilities of the method invocation. If you need to, you can invoke a custom serializer or deserializer to convert more complex outbound data into XML or to extract data from returned XML. Use `WSMethodInvocationAddSerializationOverride` or `WSMethodInvocationAddDeserializationOverride` to add a custom serializer or deserializer.

You can specify that the response dictionary should contain the raw XML that was sent and/or returned to assist in deserialization or debugging using `WSMethodInvocationSetProperty`.

## Types, Method Invocations, and Protocol Handlers

The web services core framework consists of three header files: `WSTypes.h`, `WSMethodInvocation.h`, and `WSProtocolHandler.h`.

## Types

---

`WSTypes.h` contains the error codes and types unique to the web services framework, and also includes a number of web services types that correspond to core foundation types, such as `eWSNullType`, `eWSBooleanType`, `eWSIntegerType`, and so on.

Because CTypes are determined at runtime, it isn't always possible to produce a static mapping between Core Foundation types and the corresponding serialized XML types used to interact with remote servers. What this means is that when converting between serialized XML data and deserialized CTypes, you need to do a conversion from WSTypes to CTypes, and vice versa. An enum of WSTypes, combined with an API to convert between CTypes and WSTypes, can also be found in `WSTypes.h`.

## Method Invocations

---

`WSMethodInvocation.h` provides the main client-side API described throughout this document: creating an invocation reference, setting parameters, invoking remote operations, and parsing responses.

## Protocol Handlers

---

`WSProtocolHandler.h` contains the API for converting between dictionaries and XML messages without invoking web services. You can use this to support either client-side or server-side operations.

The fundamental object of the `WSProtocolHandler` API is the `WSProtocolHandlerRef`. This references an object that translates dictionaries into a web services request, or an XML message into a dictionary. Typically, it is used to implement the server side of a web service by converting received XML into core foundation types, but you can use the protocol handler API to serialize and deserialize data prior to, after, or instead of invoking an operation for client-side operations as well.

# Using Specific Web Service Types

You can use the Web Services Core framework to access XML-RPC or SOAP-based web services. To access SOAP or XML-RPC services starting with a WSDL document, you must first parse the WSDL document using the Mac OS X XML library (`NSXML...` functions). In most cases, you can then call the described service using the Web Services Core framework. If the described SOAP service uses data whose types are encoded in other WSDL files, however, you need to use XML and network messaging (CFNetwork) to access the service as well.

## XML-RPC

---

A properly formatted XML-RPC message is an HTTP POST request whose body is in XML. The specified remote server executes the requested call and returns any requested data in XML format.

XML-RPC recognizes function parameters by position. Parameters and return values can be simple types such as numbers, strings, and dates, or more complex types such as structures and arrays.

XML-RPC has a significant limitation, in that it defines string parameters as being ASCII text. Some XML-RPC servers enforce this, forcing the user to pass internationalized text as Base-64 encoded data. XML-RPC does support passing binary data in an XML document using Base-64 encoding.

To learn more about XML-RPC messages, see the XML-RPC specification at <http://www.xmlrpc.com/spec>.

## SOAP

---

SOAP is an RPC protocol designed to access servers containing objects whose methods can be called over the Internet. A SOAP request contains a header and an envelope; the envelope in turn contains the body of the request.

SOAP supports two styles for representing web service operation invocations: RPC (remote procedure call) messaging and document-style messaging. RPC messaging is fairly rigid but can generally make use of the Web Services Core framework's built-in serializers and deserializers. Document-style messaging, on the other hand, provides greater flexibility; it allows messages to contain arbitrary data elements. Parsing such messages is more complicated and commonly requires you to provide custom serializers or deserializers. Examples of both representation styles are given in the following two listings.

### Listing 1-1 RPC-style SOAP envelope

```
<soapenv:Envelope
  xmlns:soapenv="soap_ns"
  xmlns:xsd="xml_schema_ns"
  xmlns:xsi="type_ns">
  <soapenv:Body>
    <ns1:getStockPrice
      xmlns:ns1="app_ns"
      soapenv:encodingStyle="encoding_ns">
      <stockSymbol xsi:type="xsd:string">AAPL</stockSymbol>
    </ns1:getStockPrice>
  </soapenv:Body>
</soapenv:Envelope>
```

### Listing 1-2 Document-style SOAP envelope

```
<soapenv:Envelope
  xmlns:soapenv="soap_ns"
  xmlns:xsd="xml_schema_ns"
  xmlns:xsi="type_ns">
  <soapenv:Body>
    <ns1:customerOrder
      soapenv:encodingStyle="encoding_ns"
      xmlns:ns1="app_ns">
      <order>
        <customer>
          <name>Plastic Pens, Inc.</name>
          <address>
            <street>123 Yukon Drive</street>
            <city>Phoenix</city>
            <state>AZ</state>
            <zip>85021</zip>
          </address>
        </customer>
      </orderInfo>
    </ns1:customerOrder>
  </soapenv:Body>
</soapenv:Envelope>
```

```

        <item>
            <partNumber>88</partNumber>
            <description>Blue pen</description>
            <quantity>250</quantity>
        </item>
        <item>
            <partNumber>563</partNumber>
            <description>Red stapler</description>
            <quantity>30</quantity>
        </item>
    </order>
</ns1:customerOrder
</soapenv:Body>
</soapenv:Envelope

```

**Important:** SOAP data types can be defined within the SOAP message envelope (commonly known as section 5 encoding, from the W3C specification) or they can be defined in external schema contained in WSDL files. The current implementation of the Web Services Core framework supports section 5 encoding *only*. If a web service uses SOAP data types that are not encoded within the SOAP message envelope, there is currently no high-level support for using that web service from Mac OS X. You must use lower-level XML and HTTP messaging functions instead.

To learn more about SOAP messages, see the SOAP specification at <http://www.w3.org/TR/soap/>.

## WSDL

---

WSDL files contain namespace definitions that define web services, their URLs, operations, data types, and binding. Many SOAP services can be accessed without prior knowledge of the method names and data types by accessing a WSDL file on the server to obtain this information.

Mac OS X does not currently provide high-level support for WSDL directly. However, it is relatively straightforward to parse the XML of a WSDL file using `NSXMLParser`, then use SOAP or RPC to call the functions described in the WSDL file.

Some web services integrate WSDL and SOAP in more complex ways, actually encoding the SOAP data in external WSDL files. The current implementation of the Web Services Core framework does not support this type of data encoding.

To access services that use external data encoding, you must use lower-level techniques. For example, you can use `NSXMLParser` to read and parse the WSDL file from the URL, then use `NSXMLDocument` to construct an appropriate XML message, then use HTTP or HTTPS messaging to post the message, again using `NSXMLParser` to read and parse the response.

**Note:** The command-line tool `WSMakeStubs` is included with Mac OS X developer tools. It generates the basic source code to access described services from a WSDL file using procedural C, Cocoa, or Applescript. The source code generated by `WSMakeStubs` is not actual working code, however, and cannot be cut and pasted into your application without substantial modification. In most cases, for example, the generated `resultValue` functions should be truncated to simply return a dictionary, which your own code should deal with.

`WSMakeStubs` provides example code that serves as a general guide for writing a program to work with services specified by a given WSDL file, nothing more. While you may find it helpful as a learning aid, you should not rely on it to generate working code “right out of the box.” There are commercial and open-source tools, such as gSOAP for C and C++, or Axis for Java, that are more fully developed and can be used to generate SOAP stubs from WSDL that can be cut and pasted directly into working code.

To learn more about WSDL, see the specification at <http://www.w3.org/TR/wsdl>.



# Using the Web Services Core Framework

---

The fundamental task in using web services is invoking an operation. This is a moderately complex process. This chapter contains two sections. The first section, [“Creating and Invoking Operations”](#) (page 17), explains the steps and provides illustrative code snippets. The second section, [“Example: Calling a SOAP Operation With HTTP Authentication”](#) (page 19), is an annotated example with downloadable sample code you can build and run.

## Creating and Invoking Operations

To call an XML-RPC or SOAP service on a remote server, you need to create an invocation reference, define the parameters and parameter order, pass in the settings, then invoke the service. The invocation reference includes a URL, an operation name, and a protocol (SOAP or XML-RPC). The parameters and settings vary depending on the selected protocol—SOAP parameters are named, for example, whereas RPC parameters are numbered. SOAP requires a namespace and usually a SOAP action header in the settings as well. In addition, there are debug properties you can set.

The following sections describe the steps in more detail.

### The Invocation Reference

---

You create the `WSMethodInvocationRef` object by using the function `WSMethodInvocationCreate`, which takes a URL (`CFURLRef` or `NSURL`), an operation name (`CFStringRef`), and a protocol. The protocol is one of these `CFStringRef` constants:

- `kWSXMLRPCProtocol`
- `kWSSOAP2001Protocol`

Pass in the URL of the service, the name of the operation, and the protocol:

```
NSURL *url = [NSURL URLWithString:@"http://localhost:8888/"];
NSString *methodName = @"echo";
WSMethodInvocationRef mySoapRef = WSMMethodInvocationCreate((CFURLRef)url,
                                                            (CFStringRef)methodName,
                                                            kWSSOAP2001Protocol);
```

### The Operation Parameters

---

Create a dictionary of the operation parameters and an array establishing the parameter order, then pass references to the dictionary and array to `WSMethodInvocationSetParameters`:

```
NSDictionary *params = [NSDictionary dictionaryWithObject:@"firstParamName"
```

```

                                                                    forKey:@"firstParam"];
NSArray *paramOrder = [NSArray arrayWithObject:@"firstParam"];
WSMethodInvocationSetParameters(mySoapRef, (CFDictionaryRef)params,
                                (CFArrayRef)paramOrder);

```

## The Invocation Properties

---

Call `WSMethodInvocationSetProperty` with the appropriate settings for the operation. For example, a SOAP call requires a namespace and a SOAP action header (some SOAP implementations require a header even if it is empty). You might also want to set the request to allow redirects and to turn on the debug properties so that the raw XML is included in the returned dictionary, along with the parsed data.

```

NSString *namespace = @"http://localhost:8888/";
NSDictionary *reqHeaders = [NSDictionary dictionaryWithObject:@" "
forKey:@"SOAPAction"];
WSMethodInvocationSetProperty(mySoapRef, kWSSOAPMethodNamespaceURI,
                              (CFStringRef)namespace);
WSMethodInvocationSetProperty(mySoapRef, kWSHTTPExtraHeaders,
                              (CFDictionaryRef)reqHeaders);
WSMethodInvocationSetProperty(mySoapRef, kWSHTTPFollowsRedirects,
                              kCFBooleanTrue);
// set debug props
WSMethodInvocationSetProperty(mySoapRef, kWSDebugIncomingBody,
                              kCFBooleanTrue);
WSMethodInvocationSetProperty(mySoapRef, kWSDebugIncomingHeaders,
                              kCFBooleanTrue);
WSMethodInvocationSetProperty(mySoapRef, kWSDebugOutgoingBody,
                              kCFBooleanTrue);
WSMethodInvocationSetProperty(mySoapRef, kWSDebugOutgoingHeaders,
                              kCFBooleanTrue);

```

## Invoking the Operation

---

Once you've created the invocation reference and added the settings and the properties, you can invoke the operation. Typically, you then use the CFNetwork framework to check the status of your post—which saves you the trouble of trying to parse a nonresponse in the event of a challenge or network error—using the constant `kWSHTTPResponseMessage` as an identifier for the web services request.

```

NSDictionary *result = (NSDictionary *)WSMethodInvocationInvoke(mySoapRef);
// get HTTP response from SOAP request so we can see the status code
CFHTTPMessageRef res = (CFHTTPMessageRef)
[result objectForKey:(id)kWSHTTPResponseMessage];

```

## Invoking the Operation Asynchronously

---

Because network delays tend to be unpredictably long, you normally want to invoke operations asynchronously, using `WSMethodInvocationScheduleWithRunLoop`, and providing a callback on your run loop to handle the response. You can re-schedule the invocation after it completes. Set your callback using `WSMethodInvocationSetCallback`.

## Dealing With The Response

---

If there are no network, transport, or web services faults, the invocation returns a method response dictionary, either synchronously or on your callback. (Set your callback using `WSMethodInvocationSetCallback`). If you know what the keys are for the values you want, you can retrieve the returned data from the dictionary using the keys.

Alternatively, you can request the `kWSMethodInvocationResult` and parse the XML yourself. If you prefer, you can pass in an expected result parameter name before invoking the operation, in which case `kWSMethodInvocationResult` will be an alias to the parameter that you want.

Because the response dictionary may contain the raw XML as well as the deserialized data, there may be more than one instance of the data in the dictionary.

## Adding Custom Serializers and Deserializers

---

If the CTypes for the data you are working with have corresponding WSTypes, invoking the operation automatically serializes the outbound data from the dictionary into XML and deserializes the returned data from XML into a dictionary. If you need to work with more complex data types, you can write your own serializer and deserializer callbacks. These callbacks are called when the method invocation encounters the specified data type in an outbound dictionary or an inbound XML response.

Set the callbacks using `WSMethodInvocationAddSerializationOverride` and `WSMethodInvocationAddDeserializationOverride`.

## Example: Calling a SOAP Operation with HTTP Authentication

This example calls a SOAP method over HTTP. Web services commonly require authentication, so this example sends the SOAP request and, if challenged, authenticates. You can download the sample code from [http://developer.apple.com/internet/webservices/SOAP\\_AuthExample.dmg](http://developer.apple.com/internet/webservices/SOAP_AuthExample.dmg).

**Downloading and Running a Server:** You can download a small Java program that acts as a SOAP server on your local host. This gives the example code something to talk to, and gives you the ability to test and debug both sides of the transaction. Download the SOAP server from [http://developer.apple.com/internet/webservices/SOAP\\_AuthExampleServer.dmg](http://developer.apple.com/internet/webservices/SOAP_AuthExampleServer.dmg).

Once you have downloaded and unzipped the server, run it on port 8888 by typing `java -jar SOAP_AuthExampleServer 8888` from the console.

The SOAP server exposes one method, `Echo(string)`, which takes a single string parameter and echoes it back to the client. The server challenges the request, but accepts any user name and password in response.

As shown in the example code, begin by declaring or fetching the data that makes up the parameters and other details of the SOAP request, including the SOAP method name, method namespace, request parameters, request parameter order, and SOAP action HTTP header. The SOAP method in this case is named `echo` and takes a single string parameter named `param`. If the call is successful, the method echoes the string parameter ("I hear an echo.") in the SOAP response. Create a dictionary of parameters and their names for SOAP calls (for XML-RPC calls, use ordinal numbers for the parameters instead of names) and an array containing the parameters in order.

```

// SOAP request settings
NSURL *url = [NSURL URLWithString:@"http://localhost:8888/"];
NSString *method = @"echo";
NSString *namespace = @"http://localhost:8888/";

// SOAP request params
NSDictionary *params = [NSDictionary dictionaryWithObject:@"I hear an echo."
                                                    forKey:@"param"];
NSArray *paramOrder = [NSArray arrayWithObject:@"param"];

// SOAP request http headers -- some server implementations require even empty
// SOAPAction headers
NSDictionary *reqHeaders = [NSDictionary dictionaryWithObject:@" "
                    forKey:@"SOAPAction"];

```

It's generally easier to work with Cocoa objects, as shown, and cast the objects to Core Foundation types when necessary. This approach takes advantage of the toll-free bridging between Cocoa and Core Foundation, and makes many common tasks—such as memory management—easier. That said, you can make the same calls from C or C++.

The URL for this example points to the SOAP server running on port 8888 on the local host. Values for the method namespace and SOAPAction HTTP headers are also declared. Many SOAP server implementations require the presence of an empty SOAP action header even if the method itself does not specifically require a header value. Creating a dictionary with an empty value for the SOAPAction key and attaching it to the SOAP request forces an empty SOAP action header to be sent along with the request.

Next, a SOAP request is created from the settings above. A SOAP request is represented as a `WSMethodInvocationRef` type.

```

// create SOAP request
WSMethodInvocationRef soapReq = createSOAPRequest(url, method, namespace, params,
        paramOrder, reqHeaders);

```

Next, creation of the SOAP request is delegated to the `createSOAPRequest` function, which returns a `WSMethodInvocationRef` object.

```

// Custom function to create SOAP request
WSMethodInvocationRef createSOAPRequest(NSURL *url,
                                        NSString *method,
                                        NSString *namespace,
                                        NSDictionary *params,
                                        NSArray *paramOrder,
                                        NSDictionary *reqHeaders)
{
    WSMethodInvocationRef soapReq = WSMethodInvocationCreate((CFURLRef)url,
                                                            (CFStringRef)method,
                                                            kWSSOAP2001Protocol);

    // set SOAP params
    WSMethodInvocationSetParameters(soapReq, (CFDictionaryRef)params,
    (CFArrayRef)paramOrder);
    // set method namespace
    WSMethodInvocationSetProperty(soapReq, kWSSOAPMethodNamespaceURI,
    (CFStringRef)namespace);
    // Add HTTP headers (with SOAPAction header)
    WSMethodInvocationSetProperty(soapReq, kWSHHTTPExtraHeaders,
    (CFDictionaryRef)reqHeaders);
}

```

```

        // for good measure, make the request follow redirects.
        WSMMethodInvocationSetProperty(soapReq,    kWSHTTPFollowsRedirects,
kCFBooleanTrue);
        // set debug props
        WSMMethodInvocationSetProperty(soapReq, kWSDebugIncomingBody,
kCFBooleanTrue);
        WSMMethodInvocationSetProperty(soapReq, kWSDebugIncomingHeaders,
kCFBooleanTrue);
        WSMMethodInvocationSetProperty(soapReq, kWSDebugOutgoingBody,
kCFBooleanTrue);
        WSMMethodInvocationSetProperty(soapReq, kWSDebugOutgoingHeaders,
kCFBooleanTrue);
        return soapReq;
    }

```

Note that in addition to creating the SOAP request, several debug properties of the `WSMethodInvocationRef` object are also set to true. This causes the raw XML contents of the SOAP request and response messages to be included in the result dictionary returned from executing the SOAP request. Viewing this raw XML can be extremely helpful in debugging SOAP client code.

The next step is to invoke the initial SOAP request. Do this using the `WSMethodInvocationInvoke` function, which returns a dictionary containing the SOAP response and other debug information.

Since the service accessed requires HTTP basic authentication, expect the HTTP headers of the SOAP response to contain authentication challenge information. To respond to the authentication challenge, retrieve the HTTP response from the result dictionary using the `kWSHTTPResponseMessage` key. The HTTP response is represented by a `CFHTTPMessageRef` object.

```

// invoke SOAP request
NSDictionary *result = (NSDictionary *)WSMethodInvocationInvoke(soapReq);
// get HTTP response from SOAP request so we can see response HTTP status code
CFHTTPMessageRef res = (CFHTTPMessageRef)[result
objectForKey:(id)kWSHTTPResponseMessage];

```

Now check for an HTTP response code of 401 or 407, which would signal an HTTP authentication challenge. If an authentication challenge is returned, you must:

1. Extract the HTTP response headers from the first SOAP response. These headers contain the necessary authentication challenge information.
2. Create a new `CFHTTPMessageRef` object to represent a new HTTP request.
3. Gather user name and password information by prompting the user or by querying some external data source.
4. Combine the user name and password with the authentication challenge information from the initial SOAP response to create credentials.
5. Attach the credentials to the newly-created HTTP request.
6. Create a new SOAP request and combine it with the new HTTP request to produce a SOAP request with the necessary authentication credentials attached.
7. Invoke the new SOAP request.

```

// get status

```

```

int resStatusCode = CFHTTPMessageGetResponseStatusCode(res);
// if response status code indicates auth challenge, attempt to add authorization
while (401 == resStatusCode || 407 == resStatusCode) {
CFHTTPAuthenticationRef auth =
CFHTTPAuthenticationCreateFromResponse(kCFAllocatorDefault, res);
// extract details of the auth challenge to display
// when prompting the user for username and password information
NSString *scheme = [(NSString *)CFHTTPAuthenticationCopyMethod(auth) autorelease];
NSString *realm = [(NSString *)CFHTTPAuthenticationCopyRealm(auth) autorelease];
NSArray *domains = [(NSArray *)CFHTTPAuthenticationCopyDomains(auth) autorelease];
NSLog(@"Providing auth info for \nscheme: %@\n, realm: %@\n, domains: %@",
scheme, realm, domains);
// Replace with a user prompt or fetch data from remote source
NSString *username = @"uName";
NSString *password = @"pWord";
// create custom http request with authorization
NSString *reqMethod = @"POST";
CFHTTPMessageRef req = CFHTTPMessageCreateRequest(kCFAllocatorDefault,
(CFStringRef)reqMethod,
(CFURLRef)url,
kCFHTTPVersion1_1);

// add auth creds to request.
Boolean success = CFHTTPMessageAddAuthentication(req,
res,
(CFStringRef)username,
(CFStringRef)password,
NULL,
false);

if (!success) {
NSLog(@"failed to add auth to request");
return EXIT_FAILURE;
}
// create a new SOAP request
soapReq = createSOAPRequest(url, method, namespace, params, paramOrder,
reqHeaders);
// add HTTP request auth creds to SOAP request
WSMethodInvocationSetProperty(soapReq, kWSHHTTPMessage, req);
// send SOAP request again
result = (NSDictionary *)WSMethodInvocationInvoke(soapReq);
NSLog(@"result: %@", result);

```

At this point, the console should reflect the string you sent to the SOAP server. You have successfully invoked a service, responded to HTTP authentication challenge, serialized your outbound query, and deserialized the response.

# Document Revision History

---

This table describes the changes to *Web Services Core Programming Guide*.

Date	Notes
2009-01-06	Updated text and sample code. Moved API reference information to "Web Services Core Reference".
2005-08-11	Changed the title from "Using Web Services in Your Application."
2004-02-26	Updated for Mac OS X v10.3.
2003-09-10	Added sample code to illustrate use of <code>WSProtocolHandler</code> functions.
2003-09-05	Added information on <code>WSProtocolHandler</code> functions.
2002-10-10	Updated description of XML-RPC vs. SOAP.
2002-09-19	New document that describes how to use web services in applications.

## REVISION HISTORY

### Document Revision History