HEALTHEVET WEB SERVICES CLIENT (HWSC)

DEVELOPER GUIDE

Version 1.0
February 2011

Revised May, 2013

Department of Veterans Affairs
Office of Information and Technology
Product Development

Revision History

Table i. Revision History

Date	Description	Author(s)
05/2013	Reference to XOBT_1_0_Txx.zip. See page 2-1.	HP VistA Maintenance Team. St Petersburg, Florida • David Elwell—Developer • Bob Sutton—Technical Writer
02/2011	HWSC Version 1.0 Initial release	Product Development Services Security Program HWSC development team. Albany, NY OIFO: • Developer—Mike Kilmade • Developer—Liz Defibaugh Bay Pines, FL OIFO: • Development Manager—Charles Swartz Oakland, CA OIFO: • Developer—Kyle Clarke • SQA—Gurbir Singh • Tester—Padma Subbaraman • Technical Writer—Susan Strack

Contents

Tables

Figures

HealtheVet Web Services Client (HWSC)   May 2013
                                              Developer Guide
                                                Version 1.0

Orientation

How to Use this Manual

Throughout this manual, advice and instructions are offered regarding the installation and use of Health*e*Vet Web Services Client (HWSC) and the functionality it provides for Health*e*Vet-Veterans Health Information Systems and Technology Architecture (VistA) software products.

The installation instructions for HWSC are organized and described in this guide as follows:

I.   Introduction

II.   Sample SOAP and REST Client Applications

III.   VistA/M-Side Development Guide

IV.   Java-Side Considerations

V.   VistA/M-Side API Reference

There are no special legal requirements involved in the use of HWSC.

This manual uses several methods to highlight different aspects of the material:

- Various symbols/terms are used throughout the documentation to alert the reader to special information. The following table gives a description of each of these symbols/terms:

Table ii. Documentation symbol/term descriptions

Symbol	Description
ℹ️	**NOTE/REF:** Used to inform the reader of general information including references to additional reading material.
⚠️	**CAUTION or DISCLAIMER:** Used to inform the reader to take special notice of critical information.

- Descriptive text is presented in a proportional font (as represented by this font).

- "Snapshots" of computer online displays (i.e., roll-and-scroll screen captures/dialogues) and computer source code, if any, are shown in a *non*-proportional font and enclosed within a box.

    – User's responses to online prompts and some software code reserved/key words will be boldface.

    – Author's comments, if any, are displayed in italics or as "callout" boxes.

NOTE: Callout boxes refer to labels or descriptions usually enclosed within a box, which point to specific areas of a displayed image.

- Java software code, variables, and file/folder names can be written in lower or mixed case.

- All uppercase is reserved for the representation of M code, variable names, or the formal name of options, field and file names, and security keys (e.g., the XUPROGMODE key).

Assumptions About the Reader

This manual is written with the assumption that the reader is familiar with the following:

- Health*e*Vet-VistA computing environment:

    - Kernel—VistA M Server software

    - Remote Procedure Call (RPC) Broker—VistA M Server software

    - VA FileMan data structures and terminology—VistA M Server software

    - VistALink—VistA M Server and Application Server software

- Linux or Microsoft Windows environment

- Java Programming language:

    - Java Integrated Development Environment (IDE)

    - J2SE^{TM} Development Kit (JDK)

    - Java Authentication and Authorization Services (JAAS) programming

- M programming language

- WebLogic 9.2 or 10.x Application Server

Reference Materials

The complete HWSC 1.0 end-user documentation package consists of:

- *HWSC 1.0 Installation Guide*
- *HWSC 1.0 System Management Guide*
- *HWSC 1.0 Developers Guide*

They are available from the Product Support anonymous directories and the VHA Software Documentation Library (VDL) Web site (http://www4.va.gov/vdl/application.asp?appid=180).

Health*e*Vet-VistA end-user documentation and software can be downloaded from the Product Support anonymous directories:

- Preferred Method   download.vista.med.va.gov

   This method transmits the files from the first available FTP server.
- Albany OIFO   ftp://ftp.fo-albany.med.va.gov/
- Hines OIFO   ftp://ftp.fo-hines.med.va.gov/
- Salt Lake City OIFO ftp://ftp.fo-slc.med.va.gov/

Health*e*Vet-VistA end-user documentation is made available online in Microsoft Word format and Adobe Acrobat Portable Document Format (PDF). The PDF documents *must* be read using the Adobe Acrobat Reader (i.e., ACROREAD.EXE), which is freely distributed by Adobe Systems Incorporated at the following Web address:

   http://www.adobe.com/

REF: For more information on the use of the Adobe Acrobat Reader, please refer to the *Adobe Acrobat Quick Guide* at the following Web address:

   http://vista.med.va.gov/iss/acrobat/index.asp

DISCLAIMER: The appearance of any external hyperlink references in this manual does not constitute endorsement by the Department of Veterans Affairs (VA) of this Web site or the information, products, or services contained therein. The VA does not exercise any editorial control over the information you may find at these locations. Such links are provided and are consistent with the stated purpose of this VA Intranet Service.

1 Introduction

1.1 Document Overview

This document provides information for M application developers writing code to call web services in Health*e*Vet applications. It presents information on the following topics:

- HWSC architecture
- Implementing Caché code with HWSC to consume to external SOAP-based and REST-based web services
- HWSC Application Program Interface (API) reference

This document assumes the reader is familiar with the following areas:

- M development
- HyperText Transport Protocol (HTTP)
- eXtensible Markup Language (XML)
- REST (for REST-style web service consumption)
- SOAP (for SOAP-style web service consumption)
- Caché Objects

Additional development resources include:

- Caché documentation
- SOAP: *http://en.wikipedia.org/wiki/SOAP*
- REST: *http://en.wikipedia.org/wiki/Representational_State_Transfer*

1.2 Using SOAP and REST to Access Health*e*Vet from M/VistA

M/VistA-based applications have a need to synchronously access Health*e*Vet application services and data (e.g., in-process during an end-user's session). HWSC uses Caché's Web Services Client to invoke web service methods on external servers and retrieve results. It provides helper methods and classes to improve the use of Caché's web service client in a Health*e*Vet-VistA environment.

HWSC supports two modes of synchronous web service access:

- SOAP (Service Oriented Architecture Protocol) – a formal XML-based protocol for accessing services

- REST (REpresentational State Transfer) – an architectural *style* of accessing services via programmatic access to web resources.

The SOAP and Rest approaches to calling Health*e*Vet services are shown in the following diagrams.

Figure 1-1. HWSC Logical View (SOAP-style)
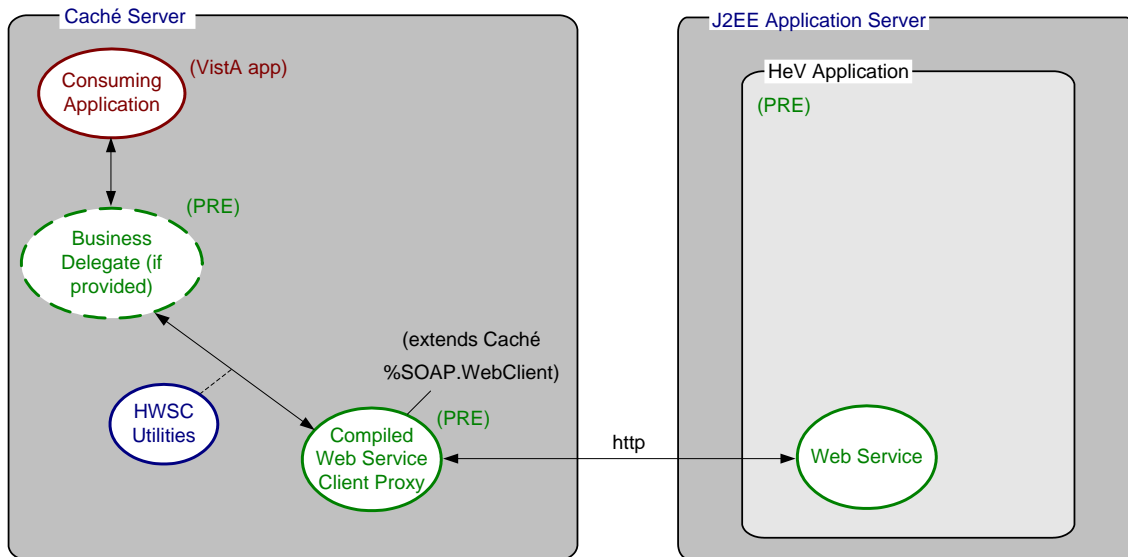
To illustrate web service access using a SOAP example: imagine that a VistA/M-based application needs drug interaction information formerly available in VistA but now maintained in the J2EE-based Pharmacy Re-Engineering (PRE) application. To retrieve the data from the Health*e*Vet side, the following steps would be required:

1. The VistA/M application makes a request to the PRE business delegate to obtain the information.

2. The PRE business delegate uses its compiled web service client proxy class to make the drug interaction request to the PRE web service.

3. PRE's web service processes the request and returns a response.

4. The PRE business delegate receives the response, decomposes the needed data elements from the return value, and returns the requested drug interaction data to the calling VistA application.

Figure 1-2. HWSC Logical View (REST-style)
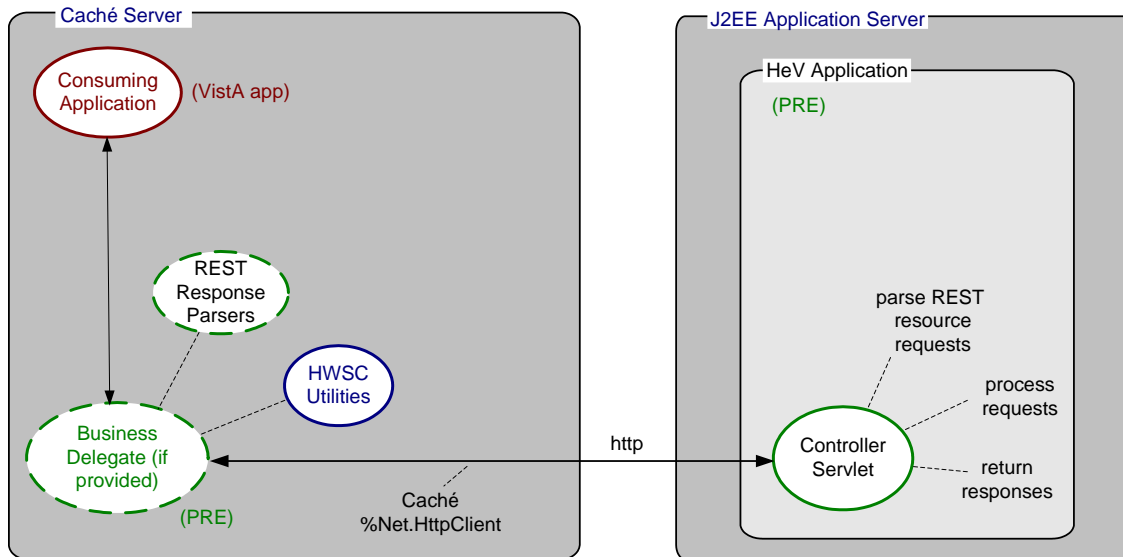
1.3 Caché's Web Services Client Features

The InterSystems Caché product provides a number of features (leveraged by the HWSC project) to enable consumption of external web services:

- **SOAP**: Caché provides APIs to compile proxy objects (in the form of Caché Objects) corresponding to external web services, from the Web Services Description Language (WSDL) document describing the SOAP web services, and invoke calls to the SOAP web service methods. Input and return values for the web service are mapped to Caché object types, which are used when invoking the service.

- **REST**: Caché provides APIs allowing invocation of an external URL via http and retrieval of the response, supporting both POST and GET methods. This supports access to REST-style web services.

- **XML Parsing**: Caché provides an API to invoke a high-performance XML parser (useful for processing very large XML results). Alternatively, Kernel's XML parser can be used.

i   **NOTE:** In order to use these features, use of non-standard M syntax (i.e., Caché Objects) is required, and a waiver has been granted (see the final section in this chapter, below.)

1.4 Role of HWSC

HWSC acts as an adjunct to the web services client functionality provided in Caché, by:

- Leveraging Caché's platform-provided Web services client capabilities.

- Adding a file and user interface (UI) to manage the set of external web server endpoints (IP, port, etc.)

- Adding a file and UI to register and manage the set of external web services.

- Providing runtime API to invoke a specific web service on a specific web server.

- Providing a runtime API to facilitate error processing in a VistA environment.

- Providing a deployment API to install/register a web service proxy from a WSDL file.

- Providing a management UI including the ability to 'ping' (test) a given web service/server combination from VistA/M.

- Supporting both SOAP- and REST-style web services

- Fostering consistent implementation of VistA/M web service consumers.

ⓘ **NOTE:** HWSC does *not* act as an all-inclusive wrapper shielding web service consumers from Caché Objects syntax. Rather, the recommended approach is for the application providing the web service, to also provide a web service delegate for the M/VistA system(s) consuming the web service. The web service delegate should use Caché Objects syntax as needed to consume the web service, but should not expose non-standard M syntax to users of the business delegate.

1.5 HPMO Waiver for Use of Caché-Specific Features

In December 2006, the HPMO Change Control Board voted to move forward with the HWSC project, and grant a waiver allowing the use Caché-specific features during the consumption of web services. At the time of writing, this decision is documented at the following location: http://vaww.va.gov/vhaoitimb/tdr_view.asp?id=33554520 and is reproduced below (Figure 1-3):

Figure 1-3. OITIMB33554520 Technical Decisions Repository Record

OITIMB33554520 - Migration from M2J to VistA Web Services Client (VWSC)	
Keywords	M2J,VWSC,J2EE
Decision Date	12/1/2006
Decision Type	Architecture
Decision Making Body	HPMO CCB
Description	On December 1, 2006, the HPMO Change Control Board voted to accept the migration of VistA from the current M2J solution to the VistA Web Services Client (VWSC). This decision was made for a number of reasons, in particular the fact that the existing 12-year-old M standard has been surpassed by evolving technologies and can no longer address today's requirements. Additionally, we are no longer required to support DSM, previously the primary VistA/M hosting environment. Today, all sites are standardized on Caché 5.0 systems. As such, approvals were granted as follows: Waiver of the requirement to adhere to the existing 1995 M standard (that does not address the implementation of web services); Implementation of an industry standard such as web services for VistA/M to J2EE calls using Caché's built in HTTP and web service client feature; Use of VWSC as an interim solution that ensures continuity of integration between VistA/M applications and migrated J2EE applications as HealtheVet evolves by enabling the consumption of external web services by

	legacy VistA applications; and Deprecation of the original M2J approach.
Rationale	This architectural change allows for a number of improvements, including better scalability, resilience, and performance. Deployment and configuration is far less complicated for administrators, and the APIs can be used by a variety of clients rather than solely M-based. It also places responsibility for support, maintenance, etc. with the vendor rather than OI&T.
Record Type	TDR
State	Approved
Date Submitted	2/14/2007 8:37:24 AM
Supporting Documentation	

Link	Document Title	Description	Date
Download	Migration from M2J to VistA Web Services Client (VWSC) Email Notification	Email notification alerting of the decision	2/13/2007
Download	VWSC Architecture	Proposed architecture view of VWSC	12/1/2006
Download	VWSC Proposed View	Proposed logical view of VistA Web Services Client (VWSC)	12/1/2006

The waiver granted in the technical decision above is *not* intended to be carte blanche to bypass adherence to 1995 M standard. Rather, it permits the use of non-standard M syntax insofar as to allow use of the underlying features of the Caché 5.0 platform to consume external web services.

To better understand the context of the waiver, it is also useful to examine the listed supporting documentation for the technical decision (Figure 1-4 and Figure 1-5 below).

Figure 1-4. OITIMB33554520 Supporting Documentation: VWSC Architecture
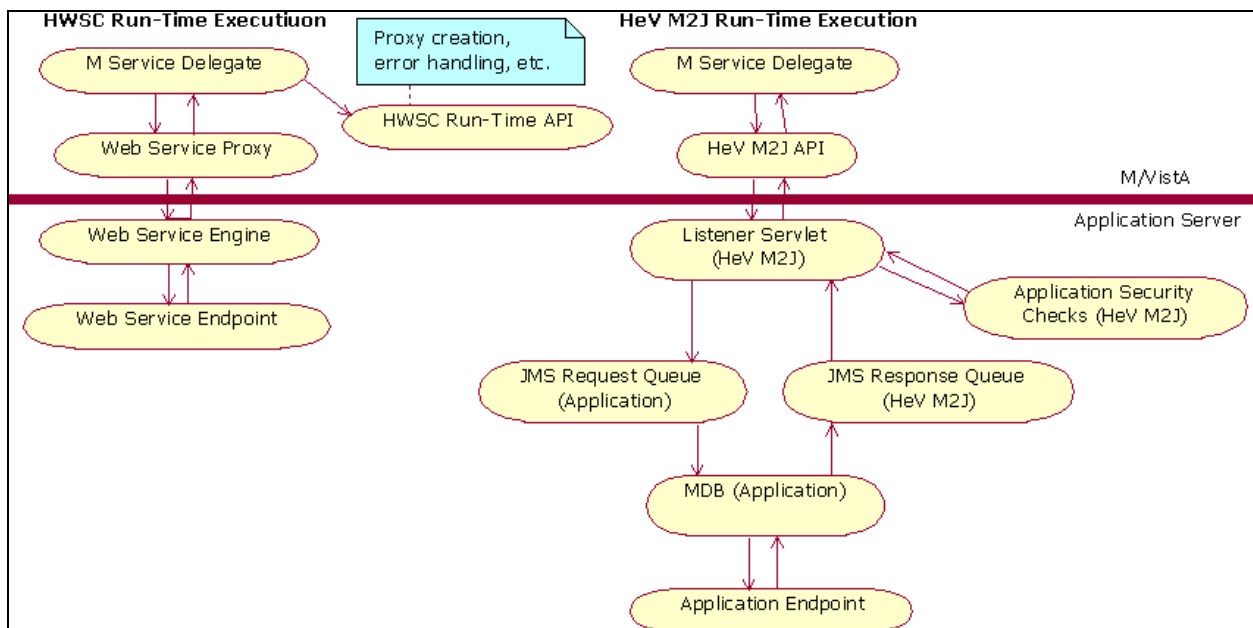
In Figure 1-4, HWSC (the HWSC run-time API) is positioned as an adjunct to the use of Caché's web services client functionality, but is not positioned as a wrapper around that functionality. The M-side business (service) delegate interacts directly with the web service proxy.

Figure 1-5. OITIMB33554520 Supporting Documentation: VWSC Proposed View
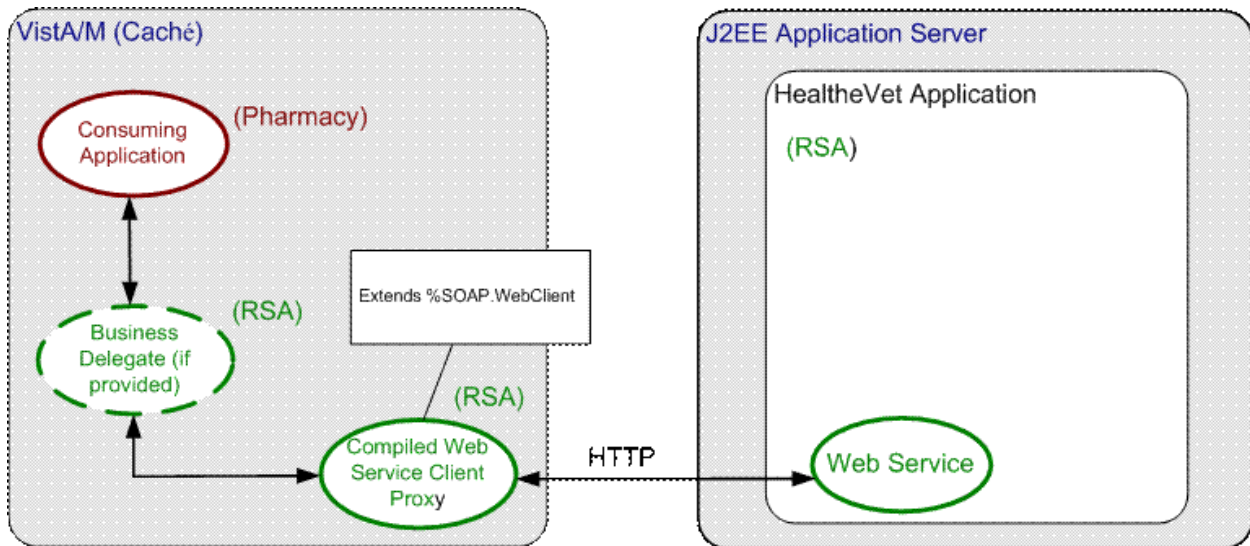
Similarly in the example shown in Figure 1-5:

- The external (J2EE) application providing the web services is RSA

- The web service provider (RSA) also provides an (optional) M-side web service delegate. The delegate directly invokes the compiled Caché Object web service proxy (which extends %SOAP.WebClient)

- The actual application interested in the web service (in this example Pharmacy) interacts with the RSA-provided business delegate to access the RSA web service.

As such, Figure 1-5 captures the recommended model for Health*e*Vet web service providers: that is, to provide an M-side business delegate:

- The M-side business delegate has permission via the waiver discussed above to use non-standard M syntax (specifically, Caché Objects and Caché's web services client functionality) in its consumption of the web service.

- The web service delegate should process the input values and return values, including a variety of errors, as required using Caché Objects syntax. Some of these values may also be very large (and therefore accessed via Caché stream-type interfaces).

- While the delegate needs to interact directly with Caché Objects to process the variety of input and return values, users of the business delegate should be shielded by the delegate from the non-standard M syntax.

HealtheVet Web Services Client (HWSC)
Developer Guide
Version 1.0

1.6 Unsupported Web Service Features on VMS

At the time of HWSC v1.0's release, the following features of HWSC and Caché Web services are **not supported** for production use on the VMS platform:

Feature	Supported	Not Supported	Reason
Encryption	none	SSL	VMS library memory leak
Authentication	None, Basic	Certificate	VMS library memory leak

Therefore, as of the release date of HWSC 1.0, SSL encryption and certificate-based authentication should **not** be used in production **until** either:

1. VA migrates all production Caché platforms to a non-VMS operating system (e.g., Linux), or

2. HP fixes the memory leak identified by InterSystems in the VMS library used by Caché for SSL.

2 Sample SOAP and REST Client Applications

HWSC provides a sample application that demonstrates how to consume both SOAP- and REST-style web services.

The HWSC sample application code functions as example/sample code, and demonstrates many aspects of calling external web services, including parameter passing, return value processing, return values based on custom object types (SOAP), XML response parsing (REST), timeout processing, and error handling.

> **i** **NOTE:** Sample code is provided as an educational tool, and is not intended to be a template for production code.

2.1 Installing the J2EE Sample Web Services EAR

Corresponding sample J2EE-based SOAP and REST web services are provided in an Enterprise ARchive file (EAR) in the HWSC distribution. They provide services for the Caché-based sample client applications to access and retrieve results from.

The HWSC sample application's SOAP web services employ the XFire web service framework; the REST web services are servlet-based.

The WSDL for the sample SOAP application is imported as part of the M-side XOBT installation process for the sample Caché-based *client* application, and is used as the basis to generate Caché proxy objects for the J2EE SOAP-based web service sample.

> **i** **REF:** For installation instructions, see the appendices in the *HWSC Installation Guide*.

> **i** **REF:** For more information on how to obtain the XOBT_1_0_Txx.zip distribution file, contact the OIT PD VistA Maintenance HWSC Developers (Outlook Mail Group: OITPDVistAMaintenanceHWSCDevelopers@va.gov)..

2.2 Using the XOBT M Client Application

The Caché-based SOAP-style web services client application is in the following namespaces:

- XOBTWSA*: sample client application/business delegate caller
- XOBTWSB*: sample 'business delegates' for each SOAP web service call

HWSC also provides a sample Caché-based REST-style web services client application, in the following namespaces:

- XOBTWRA*: sample client application/business delegate caller
- XOBTWRB*: sample 'business delegates' for each REST-style web service call

All non-M-standard Caché Object code in the application samples has been isolated into the sample business delegate routines. The sample "client application" routines (that call the business delegates) employ standard M syntax.

2.2.1 XOBT M-Side Installation

For installation instructions, see the appendices in the *HWSC Installation Guide*.

2.2.2 Create Web Server Entry

In order to access the sample application's J2EE web services, create an entry in the WEB SERVER (#18.12) for the web server they're installed on, using the XOBW WEB SERVER MANAGER option and the 'Add Server' action. The entry should correspond to the server on which you have installed **hwscSampleWs-1.0.0.xxx.ear**.

Use the XOBW WEB SERVER MANAGER option and the 'Add Server' action in the option to enter a new entry in this file, for the WebLogic server on which the sample web service is installed.

Fields / Values

- NAME: *any value to name the target WebLogic server*
- SERVER: *enter the domain name or IP address of WebLogic server*
- PORT: *enter the WebLogic listener port*
- STATUS: **ENABLED**

Security Credentials
====================

- LOGIN REQUIRED: YES//
- USERNAME: *enter WebLogic username that is a member of WebLogic server's XOBW_Server_Proxies group*
- Want to edit PASSWORD (Y/N): **Y**
- PASSWORD: *enter password associated with the WebLogic user*

Authorize Web Services
======================

- Select WEB SERVICE: *(enter/authorize the following two web services)*
- WEB SERVICE: **XOBT TESTER WEB SERVICE**
- STATUS: **ENABLED**

- Select WEB SERVICE:
- WEB SERVICE: **XOBT TESTER REST SERVICE**
- STATUS: **ENABLED**

2.2.3 Sample Application User Interface

A subset of the sample entry points can be executed from the XOBW WEB SERVER MANAGER option's user interface:

- PT Ping Test (PING^XOBTWSA)
- AT Array Test (GA^XOBTWSA)
- ET Echo Test (ECHO^XOBTWSA)
- SP Retrieve System Properties (SP^XOBTWSA)

A description of all actions available in the sample application user interface is provided in the table below.

Table 2-1. Sample Application UI Actions

Action	Description
PT (Ping Test)	Ping the selected web server.
AT (Array Test)	Return an array of text from the selected web server.
ET (Echo Test)	Type in a phrase to echo back from the selected web server.
SP (Retrieve System Properties)	Retrieve the set of java system properties from the selected web server.
DE (Show Demographics)	Display information about the selected web server (name, address, port and status)

To access the "samples" user interface:

1. Option: XOBW WEB SERVER MANAGER
2. Action: Test Server (select your server)
3. Select any of the options to run.

Figure 2-1. Sample Application Screen and Options

```
Web Server Tester          Jun 07, 2007@16:04:56        Page:   1 of   1
                             WEB SERVER: VHAISFZZZC
                                 Demographics

                 NAME: VHAISFZZZC
               SERVER: vhaisfzzzc
                 PORT: 7111
```
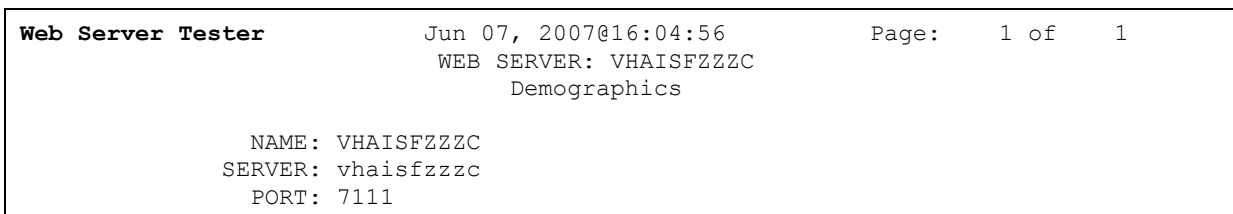
```
          STATUS: ENABLED


        Select Action/Test to Perform on Server
PT  Ping Test                         SP  Retrieve System Properties
AT  Array Test                        DE  Show Demographics
ET  Echo Test
Select Item(s): Quit//
```

2.2.4 Ping the SOAP Tester Web Service from Caché

Perform the following steps to ping the remote "test" web server:

- Option: XOBW WEB SERVER MANAGER
- Action: Test Server (selecting your server entry)
- Sub-Action: Ping Test
- Successful result: "Response: Ping Successful!"

2.2.5 Demonstrating Server Lookup Keys: The XOBT SAMPLE SERVER Lookup Key

The server lookup key XOBT SAMPLE SERVER is installed by the XOBT KIDS build, but is not associated with a particular web server by default.

To determine the web server against which to access the sample web services, the XOBT sample applications first see if a web server has been associated with the XOBT SAMPLE SERVER lookup key. The XOBT sample applications only prompt the end-user to specify a web server if the lookup key-server association is not defined. Similarly, applications using HWSC can define their own server lookup key(s), and avoid hard-coding a particular web server file entry name.

To associate a server with the XOBT SAMPLE SERVER key, use HWSC's Lookup Key Manager, accessed through the Web Server Manager:

1. Use the XOBW WEB SERVER MANAGER option to call up the HWSC Web Server Manager
2. Select the LK action to access the Lookup Key Manager
3. Select the EK (Edit Key) action to edit the XOBT SAMPLE SERVER key
4. Associate it with the server on which you have installed the sample web service.

Then, run the XOBT sample applications again. Web service calls are automatically made against the associated server.

2.3 Sample Client Application Entry Points

The tags in the XOBTWSA*/XOBTWRA* routines invoke a single, corresponding web service operation in the J2EE-based tester web service and display the results. Each call takes a one parameter, which can be either empty ("") or the name of a server in the WEB SERVER file. If empty, you are prompted to select a web server to run the option against.

You can test connectivity by executing any of the following tag^routines and invoking SOAP-style calls (^XOBTWSA):

- PING
- ECHO
- ARRIN
- GA
- SP
- EI
- EIVO
- ECHOEIVO
- EILIST
- SENDXML

Additional SOAP-style calls (^XOBTWSA1):

- DOI
- BOOLEAN
- FLOAT
- DOUBLE
- INT
- SHORT
- LONG
- DATE
- CAL
- TIMEOUT
- RETRY

REST-style calls (^XOBTWRA):

- PING
- SAMPLE
- GA
- EI
- EILIST

Additional REST-style calls (^XOBTWRA1):

HealtheVet Web Services Client (HWSC)
Developer Guide
Version 1.0

- DICTGET
- DICTPOST
- DICTLST

To invoke ALL SOAP-style calls: **ALL^XOBTWSA1**
To invoke ALL REST-style calls (except DICTPOST): **ALL^XOBTWRA**

2.4 Rebuilding the J2EE Sample Web Service Project

The distribution zip file's "**sample-prj**" folder contains an ANT-buildable version of the sample Tester web service. You can optionally modify and rebuild the service as follows:

1. Check that ANT is set up on your system.

2. Ensure your JAVA_HOME environment variable is set to a JDK 1.4 location.

3. In the unzipped "**sample-prj**" folder, copy or rename "**uncommon-build.properties.template**" to "**uncommon-build.properties**".

4. Edit the two file location properties to point to "scratch" locations on your system for the files generated by the build. Also update the WEBLOGIC.DIRPATH property to point to a WebLogic home on your system.

5. In the unzipped "**sample-prj**" folder, from the command line, run ANT (no arguments necessary).

If the build is successful, the files are built in the location you specified in the HWSC.DIST.PATH property.

3 VistA/M-Side Development Guide

3.1 Platform Considerations

3.1.1 Caché-Specific Considerations

You should be aware of the following limitations in Caché, which you may encounter working with web services:

- **Caché Object property length.** Any one Caché Object property is restricted to a length of 32K (e.g., SOAP client input parameter and response classes). This is important if you want to return a long string as the return value of a SOAP call (e.g., an XML document), or pass a long (>32K) string as an input parameter. However, a workaround to overcome this limitation is discussed in the section "Manually Modify SOAP Client Proxies to Overcome Length Limits."

- **Length of REST-style returned data.** Because HWSC returns the response to a REST-style web service in a Caché stream, there is no immediate limit on the length of the data that can be returned by a single call. Of course, the larger the data that is returned, the longer it takes to process the return value.

- **System memory maximum.** The sum of the memory used by all objects and partition variables cannot exceed the system maximum. The maximum varies from system to system, but is currently set to 2 megabytes at most VA production sites. This is mostly an issue for SOAP calls, since results are automatically returned as a Caché object in memory. (For REST calls, the results are returned in an instance of the %Net.HttpResponse class, but large input values could also cause a problem.) The workaround discussed in the section "Manually Modify SOAP Client Proxies to Overcome Length Limits." can also help avoid exceeding memory partition size in some cases.

- **Package name length.** Package names for Caché classes (including the generated Port classes and SAX parser handlers) are limited to 31 characters in length.

- **Unique class names.** Within a Caché package, each class name must be unique within its first 25 characters

- **No punctuation in Caché identifiers.** Punctuation characters, including underscores and hyphens, are not allowed in Caché identifiers, like class names. This becomes an issue when proxy classes are generated based on Java class names: classes are still generated, but punctuation is stripped. This may cause interoperability issues when interacting with the classes on the Java side that still have punctuation in their names. We recommend using UpperCamelCase for web service names.

- **Lack of Full HTTP 1.1 Support**. Caché fully supports HTTP 1.0, but does not fully support HTTP 1.1. This should be adequate for almost all web services interactions.

However, a few small issues may crop up, e.g., the HTTP "Host" header is not required to contain the port number in HTTP 1.0.

3.1.2 WebLogic 8.1-Specific Considerations

You should be aware of that when returning very large result sets (e.g., 10 megabytes or more) with web services, the J2EE server may drop the http connection. This happens when the client (Caché) cannot read in the data stream fast enough to complete the transmission within WebLogic 8.1's http 'Duration' value. This value has a maximum of 120 seconds, and can be set in the WebLogic console under **server | Protocols | http | Duration**. WebLogic interprets the lack of completion as an inactive http connection, and drops the connection at the specified duration limit.

3.2 How to Consume a SOAP-Style Web Service

3.2.1 Compile WSDL into Caché Proxy Classes (SOAP)

To build a web service client to access a web service other than the HWSC sample, in your development account, call GENPORT^XOBWLIB to:

- Import your web service's WSDL by running the Caché SOAP client wizard. This creates proxy classes for communicating with your web service

- Create an entry for your web service in file #18.02, WEB SERVICE

When running GENPORT^XOBWLIB, you need to supply the following as input parameters:

- A name for the WEB SERVICE file entry to be created
- A file system location of your WSDL file (accessible from the Caché install account)
- A Caché package name to use for the compiled proxy classes
- A resource to HWSC to use for availability checks on the web service (optional)

Once completed successfully, you can write M code that calls HWSC APIs to access your web service. For more details on GENPORT^XOBWLIB see the "VistA/M-Side API Reference" section.

3.2.2 Mainline (SOAP)

Once the WSDL is imported/client proxy classes created, your code needs to perform the following main steps to invoke and consume a web service:

1. Get the name of the web server (file #18.12 entry) to call web service on. For simplicity's sake, in the example below the web server name ("VHASERVER1") is hard-coded; for

production code, you can instead use the $$SNAME4KY API and have your install sites associate a "server lookup key" with a specific web server entry at install-time.

2. Set an error trap

3. Get web service proxy object for a specific web server and web service

4. Invoke web service proxy method(s) on web service proxy object to invoke methods on the remote web service.

For example, the following code invokes a doPing method on the MYAPP WEB SVC web service, and writes out the result:

```
NEW MYPROXY,$ETRAP
; -- set error trap
SET $ETRAP="DO ERROR^MYRTN" ; to catch/process errors
; -- obtain client proxy object (replace hard-coded server name ref
;    w/ dynamic retrieval, or site param, or etc.)
SET MYPROXY=$$GETPROXY^XOBWLIB("MYAPP WEB SVC","VHASERVER1")
; -- call web method
WRITE !,"Ping result: ",MYPROXY.doPing()
```

For complete details of the $$GETPROXY^XOBWLIB API, see the "VistA/M-Side API Reference" section.

3.2.3 Passing Input Parameters to Web Services (SOAP)

The HWSC sample applications demonstrate the use of a variety of "IN"-type input parameter types, including:

- Standard web service data types (string, short, int, float, double, boolean, dateTime, etc.)
- Arrays of standard data types (using Caché's %ListOfDataTypes object)
- Service-defined complex types compiled as classes from WSDL

3.2.4 Processing Web Service Return Types (SOAP)

Web service methods return a single object as the method return value. The HWSC sample applications demonstrate retrieving and processing a variety of return types, including:

- Standard web service return types (string, short, int, float, double, boolean, dateTime, etc.)
- Collections of standard return types
- Service-defined complex types compiled as classes from WSDL
- Collections of complex types compiled as classes from WSDL

3.2.5 Large Result Sets (SOAP)

The role in the enterprise architecture currently envisioned for web services, particularly for M/VistA systems accessing Health*e*Vet systems, is to handle requests in "end-user-is-waiting-for-a-response" scenarios. Web service invocations are typically made in real time, synchronously. To match the end-user-is-waiting scenario, their results sets should be of a size that can be assembled and returned in a reasonable amount of time.

For all other VistA-to-J2EE calls (e.g., bulk transfers of data, asynchronous messaging, guaranteed delivery), the responsibility for handling requests falls instead to the messaging infrastructure and use of the interface engine.

There are gray areas, e.g., when a large result set is created on the J2EE server, but the end-user is only expected to look at small portion of the query results before making a decision and moving on. In this case, a J2EE pattern called Value List Handler can be used with web services[1] and HWSC. Optimizing the implementation of a Value List Handler is query-specific. Essentially the implementer of the pattern Caché's the query results on the server and chunks the return value of any single call into a subset of the query results.

In any case, with Caché and SOAP, there are unavoidable consequences to creating large requests and returning huge query result sets, without chunking the requests or results into smaller sets first before sending/returning them. Large requests and responses may cause timeouts and impact server resources on both the sender and receivers sides, and impact overall reliability because:

- The size of data send or returned may exceed the memory size of the Cache partition (client) or the Java virtual machine (JVM) (server). For Caché, at most VA production sites at the time of writing, two megabytes of RAM is dedicated to each M process partition.

- While receiving a large request, the J2EE server may drop the http connection if the duration of the request, or request size exceeds configurable limits.

- While returning a large result set, the J2EE server may drop the http connection if the client (Caché) can't read in the data stream fast enough.

3.2.6 Timeouts (SOAP)

The client timeout is the amount of time the Caché web services client waits for a web service invocation to return over http before aborting. By default, the client timeout is set to 30 seconds in the current Caché implementation used by the HWSC development team (although M administrators can also set a different, per-web-server default timeout).

---

[1] Lechiki, Alois and Kruse, Thomas, *Handling Large Database Result Sets*, WebLogic Journal, volume 3 issue 6, http://wldj.sys-con.com/read/45563.htm.

To explicitly set the client timeout (overriding the system and per-server default settings), use the Timeout property on the Caché web service proxy object in one of the following ways:

- `SET XOBPROXY.Timeout=60`
- `SET XOBPROXY.Timeout=XOBPROXY.Timeout*2`

(The second technique is probably the preferred one, because the site may have already adjusted the default timeout to factor in any local environment situations.)

If a timeout occurs, an error is returned to the calling code. With the current Caché implementation used by the HWSC development team, a Caché object error is returned with its error code set to code #5922 ("Timed out waiting for response").

You can use the TIMEOUT^XOBTWSA sample application call to experiment with timeout behavior.

3.2.7 How to Export an M Business Delegate (SOAP)

Assuming that your application is planning to export an M business delegate, it should export the following items:

- The WSDL file for the web service (separate from the KIDS transport file)
- M business delegate routines to consume the web service (in the KIDS transport file)
- A post-init routine (in the KIDS transport file) that calls $$GENPORT^XOBWLIB to:
  - Import the WSDL file
  - Create an entry in the WEB SERVICE (#18.02) file
  - Create the web service proxy class

You can examine HWSC's KIDS post-init routines – XOBTPST and XOBWPST (and the "Install Questions" section of the XOBT and XOBW KIDS build definitions) to see how they handle prompting and reading files from the host file system.

3.2.8 Manually Modify SOAP Client Proxies to Overcome Memory Limitations

The InterSystems web service development team has noted that to overcome the 32K length limitation of String result values in its SOAP clients (and to avoid overflowing partition memory), generated proxy classes can be manually modified to use Cache streams for %String data types.

The Caché WSDL compiler automatically compiles all WSDL string return values in the proxy classes as `%Library.String`, even though they could also be compiled as Caché stream types. Therefore, all string return type objects are subject to the current 32K-length restriction. But a string return type (on the web service side) is exactly what many web services would use to

return an XML document result (that may exceed 32K, or in some cases even exceed the memory partition size).

The developer can modify the return type in the generated proxy class for a given web service method to `%GlobalCharacterStream` in place of %Library.String:

    a) Change the method return type from `%Library.String` to `%GlobalCharacterStream` in the generated port proxy class

    b) Recompile the port proxy class; this causes other generated proxy classes to be re-compiled

    c) Modify the business delegate code to read the result off of a Caché stream

The modified class should work correctly, but the (string) return value is placed by Caché into a stream, overcoming normal memory limits.

This same approach may also work for %String input parameter types, which may help if large input parameter strings are expected (e.g. a large XML document as an input parameter.

> **NOTE:** An alternative workaround in HWSC for long XML documents is to make a REST call rather than a SOAP call.

3.2.8.1 Disadvantages of Manual Modification

Possible disadvantages, or issues to consider, with manually editing the generated classes, include:

- During the development cycle, whenever the WSDL changes and classes are generated, developers will need to remember to manually restore the modifications to the generated classes.

- The application will need to export the generated Caché Object proxy classes as a package component (as opposed to something generated upon install). Export would be in a separate XML file.

- Web Service (#18.02) file entry needs to be created/installed on target systems separately from WSDL import, since WSDL is only imported at development time. The REGSOAP^XOBWLIB API can be used to do this.

> **NOTE:** For normal situations, HWSC recommends that applications exporting a web service client, do so by exporting their WSDL and using the $$GENPORT^XOBWLIB call to generate client classes on the target install systems (as opposed to exporting the compiled proxy delegate classes themselves).

3.3 How to Consume a REST-Style Web Service

3.3.1 Mainline (REST)

To invoke and consume a REST-style web service, an application needs to perform the following main steps:

1. Get the name of the web server (file #18.12 entry) to call web service on. For simplicity's sake, in the example below the web server name ("MY SERVER") is hard-coded; for production code, you can instead use the $$SNAME4KY API and have your install sites associate a "server lookup key" with a specific web server entry at install-time.

2. Set an error trap.

3. Get a REST proxy object for a specific REST-style web server and web service.

4. Invoke either the Get or Post proxy method(s) on web service proxy object to invoke methods on the remote web service.

5. Process the result (e.g., parse the result if it an XML document).

For example, the following M function accesses a Ping resource on the MYAPP REST SVC REST-style service, using an HTTP GET, and writes out the result:

```
PING ; -- access a Ping response using REST-style Ping service
; resource: <http://server/context>/Ping
 NEW MYREST,PINGRES,MYERR,$ETRAP,X,XOBSTAT,XOBREADR,XOBREAK
 ; -- set error trap
 SET $ETRAP="DO PINGEH^MYRTN"
 ; -- get client REST request object
 SET MYREST=$$GETREST^XOBWLIB("MY REST SERVICE","MY SERVER")
 ; -- retrieve the resource; execute HTTP GET method
 IF $$GET^XOBWLIB(MYREST,"/Ping",.MYERR) DO
 . ;invoke Cache parser
 . SET XOBSTAT = ##class(%XML.TextReader).ParseStream(MYREST.HttpRespons
e.Data,.XOBREADR)
 . IF ($$STATCHK^XOBWLIB(XOBSTAT,.MYERR)) DO
 . . SET XOBREAK=0 FOR QUIT:XOBREAK!XOBREADR.EOF!'XOBREADR.Read() DO
 . . . IF (XOBREADR.NodeType = "element"),(XOBREADR.LocalName = "PingRes
ponse") DO
 . . . . IF XOBREADR.MoveToContent() DO
 . . . . . SET PINGRES=XOBREADR.Value
 . . . . . SET XOBREAK=1
 . WRITE !!,"Ping Result: ",PINGRES
 QUIT
PINGEH ; -- error trap handler for PING
 ; … display error, unwind error trap, set $ECODE="", etc.
 QUIT
```

There are some alternate paths that could be taken. For example, the Get method could be called on the REST HTTP request object (%NET.HttpRequest) directly, rather than by using the

$$GETREST^XOBWLIB wrapper. The main advantage to using the $$GETREST^XOBWLIB wrapper is being able to take advantage of built-in error detection and error trap triggering.

In addition, the above code serves as a mixture of both "business delegate" and "business delegate consumer" code, in that it both kinds invoke the web service and display the results to the end-user. The HWSC sample application's business delegate and business delegate consumer code, on the other hand, are cleanly separated.

For complete details of each XOBWLIB API, please see the "VistA/M-Side API Reference" section.

3.3.2 Parsing XML Responses (REST)

Suppose the XML response for the example above looks like:

```
<?xml version="1.0" encoding="UTF-8" ?>
<PingResponse>Ping Successful!</PingResponse>
```

There are several ways to parse documents, including:

- Use Caché's SAX parser:
    - Call %XML.TextReader methods (parsing is accomplished by traversing a Document Object Model (DOM)-like but mostly forward-only representation of the document – *InterSystems' recommended approach*, or
    - Extend %XML.SAX.ContentHandler (parsing is done via a SAX interface)
    - Call %XML.Reader (parsing is accomplished via a mapped correlation between the XML document and a Caché Object), or
- Use the Kernel MXML parser (parsing is done via a SAX interface)

For parsing implementation examples, see the business delegate (XOBTWRB*) code used for the REST-style web service clients in the HWSC sample application.

3.3.3 Timeouts (REST)

To set the client timeout, set the Timeout property on the request object returned by the $$GETREST^XOBWLIB() call.

3.3.4 How to Export an M Business Delegate (REST)

Assuming that your application is planning to export an M business delegate to access a REST web service, it should export the following items via KIDS:

- M business delegate routines to consume the REST web service
- A KIDS post-init M routine that:
  - Calls REGREST^XOBWLIB to create an entry in the WEB SERVICE (#18.02) file
  - (optionally) calls $$SKEYADD^XOBWLIB to add a server lookup key

3.4 How to Handle Errors (SOAP and REST)

Handling errors is an intrinsic part of working with a distributed communication mechanism such as web services. With Caché Objects and web service calls, there are additional error processing duties beyond simply checking $ZERROR and $ECODE. Some errors can happen at the Caché Objects level, others can be returned as SOAP faults or HTTP errors. Still others occur at the standard M level. Because of this, HWSC is providing a set of utilities to encapsulate and organize the following types of errors:

- Standard M error
- Caché Objects error
- SOAP fault (SOAP)
- HTTP error (REST)
- HWSC "fault" / Dialog entry

Handling errors in web service calls involves setting an error trap, writing the error handler code, and determining in that code whether to continue or quit. The specifics of the error handling code may depend in part on whether you're coding an application calling a web service directly, a business delegate, or an application calling a business delegate.

3.4.1 Business Delegate Level

A business delegate may want to do some processing on the error, but still preserve it so it can also be handled by the actual caller. If so, the business delegate should set up an error handler as follows:

1. "New" $ETRAP (but not $ESTACK)

2. Set $ETRAP=error handler routine for the business delegate context

The business delegate's error handler routine, provided as a pair with business delegate, should:

1. Take corrective action if desired:

   - Process the error condition in the partition and create the HWSC error object:

     - For SOAP calls, call $$EOFAC^XOBWLIB to create the HWSC error object

     - For REST calls, if the call is made via $$GET^XOBWLIB and $$POST^XOBWLIB, check first to see if the HWSC error object has already

been created, in the error variable specified in the call to $$GET or $$POST. Check. If not, call $$EOFAC create it.

- Examine the error object directly (using Caché Object syntax) and decide how to handle and perform any other action appropriate for the error handler

- Optionally, call ZTER^XOBWLIB to record the error in Kernel error trap

2. Return control to the caller:

- Call UNWIND^%ZTER to quit back to the caller of the business delegate

3.4.2 Application (Caller to Business Delegate) Level

The application calling a web service business delegate may want to handle the error itself without necessarily aborting program execution and returning control to Kernel. If so, the calling application should set up an error handler as follows:

1. "New" $ETRAP

2. "New" $ESTACK to establish a new error context before invoking the web service or business delegate

3. Set $ETRAP to an error handler routine for the current error context – that of your application

The caller's error handler routine, provided by the caller, should:

1. Take corrective action, if desired:

- Call ERRDISP^XOBWLIB to display the HWSC error object values (if preserved/returned by business delegate) to the end-user (if in roll & scroll mode)

- Call ERR2ARR^XOBWLIB to decompose the HWSC error object (if preserved/returned by business delegate) into an array, examine it without using Caché Object syntax, and decide how to handle it

2. Resume processing and/or return control to the caller:

- Clear the error stack by setting $ECODE="" and continue processing in the error handler (and eventually QUIT back to Kernel)

- Call UNWIND^%ZTER to immediately quit back to Kernel

3.4.3 REST Error Handling Options

Depending on how you invoke your GET and POST calls, you can increase the convenience of error trapping. In particular:

- If your code calls the GET^XOBWLIB or POST^XOBWLIB wrapper methods for your gets and posts, error processing for all error types is built-in automatically, by default. If

an error is encountered, it is processed into an `xobw.error` object, and an error trap is forced.

- If instead your code calls the Get or Post methods directly on the `xobw.RestRequest` object, you are responsible for capturing the `%Library.Status` return value and the HTTP status code, and for processing them to see if an error occurred during the call.

3.4.4 Automatic Retries

A sample call is provided to demonstrate the use of error trapping to automatically retry a SOAP call a specific number of times. This may be useful if you are expecting to encounter service failures of a short-lived, transient nature. See RETRY^XOBTWSB1 (business delegate) and RETRY^XOBTWSA1 (caller application) for sample code that retries a SOAP call based on trapping a specific type of error. Similar code can be used to retry REST calls as well.

Factors to consider when deciding whether to wrap web service method invocations in retry code include:

- Is the web method idempotent? Can it be retried safely without worry of causing duplicate transactions?

- Is the type of failure likely to be resolved on a retry (e.g., connectivity problem), or does it require client attention/intervention (e.g., application-level error such as "Employee with this ID already exists")?

3.5 Troubleshooting Tips

- To display the most recent error information:

    DO $System.OBJ.DisplayError(%objlasterror)

- To see set of Caché objects instantiated in the partition:

    DO $SYSTEM.OBJ.ShowObjects("d")

- To view the contents of web service messages being sent to and from your service, use a packet-level TCP-IP trace/viewer tool. The tool should allow you to see the actual message requests and responses in their entirety as Cache sends and receives them over TCP.

4 Java-Side Considerations

4.1 SOAP vs. REST Usage Scenarios

When building web services to be consumed by Caché applications, you can choose to build either a SOAP- or REST-style web service. In some cases one type of call (SOAP or REST) may be more advantageous when accessed via a Caché -based client.

- **Automatic parsing of responses.** Caché's SOAP client automatically parses an XML SOAP response and returns the results as a Caché object. No XML parsing is necessary by the consuming application.

- **Sending long responses (e.g., a long XML document)**. Caché client object properties (generated by its WSDL compiler) are currently limited to 32K. So the return of an XML document in a String property, for example, cannot exceed 32K. REST-style calls, on the other hand, use a stream on the Caché side, so there is no theoretical limit to the length of the response.

- **Parsing responses manually.** A REST-style service allows a Caché client to manually parse the response as an XML document. With SOAP-style services, on the other hand, Caché automatically parses the response, which it returns to the client as an object.

- **Standards.** SOAP is a formal (and complex) standard. REST is more of an architectural style (although at the http level it is based on the http standard).

- **Self-Documenting**. SOAP services are self-documenting via their WSDL. REST-style services are not currently self-documenting.

4.2 Supporting HWSC Availability Checking

HWSC provides an M-based console for M system managers to check if a particular web service is available through HWSC. It does this by making a HTTP GET call on some resource hosted by the web service. It deems the service available if it gets a HTTP 200 status code in return.

When registering a web service, you can pass in a parameter that tells HWSC what resource to make the availability check on.

4.2.1 SOAP Web Service

For a SOAP web service, you might configure the resource to be something the web service makes available via HTTP GET (e.g., "?wsdl"). This depends in part on your web service framework.

4.2.2 Rest Web Service

For a REST web service, you might configure the resource to be something that is always available if your service is up, but that does not cause any significant processing. You can also add an explicit resource to your web service for availability checking. (The XOBT sample does this for example, adding "/available" as an explicit availability checking resource.)

5 VistA/M-Side API Reference

5.1 HWSC Caché Classes

Table 5-1. HWSC Caché "Public Use" Classes

Class	Brief Description
xobw.RestRequest	decorates %Net.HttpRequest w/ VistA-specific functionality
xobw.RestRequestFactory	factory to create xobw.RestRequest objects
xobw.WebServiceProxyFactory	factory to create web service proxies
xobw.error.AbstractError	base class for other error classes
xobw.error.BasicError	Object used for basic/standard M error conditions
xobw.error.DialogError	Object used for errors derived from DIALOG file (#.84)
xobw.error.HttpError	Object used for errors derived from HTTP status codes
xobw.error.ObjectError	Object used for errors derived from Caché Object errors
xobw.error.SoapError	Object used for errors derived from SOAP faults

REF: See Caché's online "Documatic" documentation to access class-specific documentation for the above "public use" classes, post-HWSC installation.

5.1.1 Accessing Caché "Documatic" for HWSC Caché Classes

For more information on each HWSC error class, see the online "Documatic" documentation to access the class-specific documentation. To do this, either:

1. Right-click on the class in Caché Studio and select "Show Class Documentation."

Or:

1. Right-click on the Caché cube in the system tray.

2. Choose "Documentation" on a local or remote Caché instance where HWSC is installed.

3. Choose "Class Reference Information" from the Caché "Documentation Home Page" list of documentation.

4. At the top of the left navigation pane in the "Caché Documatic" documentation, select the appropriate "Classes in" namespace that contains the HWSC classes.

5. In the left navigation pane, navigate the package structure to the "xobw.error" node. Documentation for each of the five error classes is provided there.

5.2 HWSC API Overview

All the APIs listed in the following table are tags in the XOBWLIB routine.

Table 5-2. HWSC APIs

Category	M API	Brief Description
SOAP	$$GETPROXY	return web service proxy
	$$GENPORT	import/register web service from WSDL
	REGSOAP	register web service w/o WSDL
	$$GETFAC	return web service proxy factory
	ATTACHDR	add VistaInfoHeader to a web service proxy
	UNREG	un-register/delete a web service
REST	$$GETREST	return REST service request object
	REGREST	register a REST service definition
	$$GET	make HTTP GET call and force error if problem encountered
	$$POST	make HTTP POST call and force error if problem encountered
	$$HTTPCHK	check HTTP status; if not OK create HttpError object
	$$HTTPOK	is current HTTP response status 'ok'?
	$$GETRESTF	return REST service request factory
	UNREG	un-register/delete a web service
Error Handling	$$EOFAC	Error object factory – builds error object based on different error conditions present in partition
	$$EOSTAT	create ObjectError from Caché status object
	$$EOHTTP	create HttpError object from %Net.Response object
	ERRDISP	simple display of error to screen
	ERR2ARR	decompose Error object into M array
	$$STATCHK	check Caché %Library.Status object; if not OK create ObjectError
	ZTER	decompose Error object into M array and call Kernel error trap to record error
Server Lookup	$$SKEYADD	add a server lookup key
	$$SNAME4KY	retrieve server name associated with a server lookup key
Dev Testing Only	DISPSRVS	display server list to screen
	GETSRV	prompt user to select server from list
	SELSRV	display server list to screen / prompt for selection

5.3   SOAP-Related APIs

All the APIs listed in this section are tags in the XOBWLIB routine.

5.3.1 $$GETPROXY (web service name, web server name)

Returns a Caché web service client proxy object for the specified web service, ready to invoke web service methods on the specified web server. Use this method to obtain a web service proxy if you are going to invoke web service methods on a single server only.

Input Parameters

Web Service Name	Name of entry in WEB SERVICE (#18.02) file.
Web Server Name	Name of entry in WEB SERVER (#18.12) file.

Return Value
Web service client proxy object ready to invoke web service methods on the specified web server.

5.3.2 $$GENPORT (.infoarray)

Use in installation post-init routines to import a WSDL file and run the Caché WSDL import wizard. This call:

* Runs the Caché SOAP client wizard to create proxy classes for communicating with an external web service, using the web service's WSDL file.
* Creates entry for web service in file #18.02, WEB SERVICE.

Input Parameters

.infoarray (pass by reference)

infoarray("WSDL FILE")	WSDL file location on host operating system
infoarray("CACHE PACKAGE NAME")	Package name to place generated Caché classes in
infoarray("WEB SERVICE NAME"):	Name to store web service information in file #18.02 (WEB SERVICE) – used for lookups, should be namespaced for your application

infoarray(""AVAILABILITY RESOURCE"")   (optional) resource for HWSC to access via an HTTP GET when checking if the web service is available. HWSC appends the resource to the IP address and context root of the web service.

Return Value

Success: positive value
Failure: 0^failure description

Example

```
SET MYARR("WSDL FILE")="c:\temp\mywsdl.wsdl"
SET MYARR("CACHE PACKAGE NAME")="mypackage"
SET MYARR("WEB SERVICE NAME")="ZZMY WEB SERVICE NAME"
SET MYARR("AVAILABILITY RESOURCE")="?wsdl"
SET XOBSTAT=$$GENPORT^XOBWLIB(.MYARR)
```

5.3.3 REGSOAP (wsname, wsroot, class, [path], [resource])

Use in installation post-init routines to register a web service by creating an entry in the WEB SERVICE file (#18.02), without calling the Caché WSDL compiler. Typical use cases would be:

- Compiled classes are exported for install on the target system rather than just a WSDL, because classes were manually modified by development team after initial import.

- A site calls the WSDL import wizard itself to create a client to a web service, and needs to create a Web Service entry to associate with the imported classes.

Input Parameters

wsname: Web Service Name

wsroot: Web Service context root (without trailing '/')

class: Caché package + class name of the main class created for the web service client proxy, as created by the Caché WSDL compiler.

> **i** **NOTE:** The WSDL compiler uses the value of the *name* attribute (of the *port* element, within the *service* element, in the WSDL file) as the name for the main class it creates.

> **i** **NOTE:** The element names above may be prefaced by a namespace abbreviation (e.g., "wsdl:port", "wsdl:service") depending on the WSDL file.

path: [optional] WSDL file location on host operating system (WSDL file is copied into Web Service file entry.)

resource: [optional] resource for HWSC to access via an HTTP GET when checking if the web service is available. HWSC appends the resource to the IP address and context root of the web service.

Example

```
DO REGSOAP^XOBWLIB("ZZMY WEB SERVICE NAME",
   "myContextRoot","myPackage.myServiceProxyClassName")
```

5.3.4 UNREG (service name)

Use in installation post-init routines to un-register/delete a web service entry in the WEB SERVICE file (#18.02). Can be either a SOAP or REST web service. Also removes the service from any web servers it is authorized to.

Input Parameters

   service name: SOAP or REST Web Service Name (File #18.02)

5.3.5 $$GETFAC (web service name)

Returns a `xobw.WebServiceProxyFactory` object for the specified web service. This factory object is useful if you plan to invoke the same web service on multiple web servers. In this case, you can then use the factory object's `getProxy()` method to obtain multiple web service proxies, one for each individual server.

Input Parameters

   Web Service Name   Name of entry in WEB SERVICE (#18.02) file.

Return Value

   Web service request factory (`xobw.WebServiceProxyFactory`) object.

5.3.6 ATTACHDR (proxy object)

Attach a "VistaInfoHeader" header block to outgoing web service request. This header block contains partition and Kernel environment variables as follows:

duz: the user's DUZ value

mio: the partition's $IO value

mjob: the partition's $JOB value

production: "1" if the calling VistA system is a production system, "0" if test.

station: station # (currently the Kernel site parameter default institution value)

vpid: the user's VPID

It can be processed by the receiving web service as a SOAP header by using a handler, but that processing is currently intended for HWSC-provided handlers only. The sample web service contains one such handler for XFire,
`gov.va.med.webservice.samples.VistaInfoHandler.`

Input Parameters

Client Proxy Object   Web service client proxy object.

5.4 REST-Related APIs

All the APIs listed in this section are tags in the XOBWLIB routine.

5.4.1 $$GETREST (service name, server name)

Return REST service request object. Use to make GET, POST and PUT calls to the specified service and server.

Input Parameters

service name: REST Web Service Name (File #18.02)
server name: Web Server Name (File #18.12)

Return Value

REST service request (xobw.RestRequest) object.

5.4.2 REGREST (service name, context root, [resource])

Use in installation post-init routines to register a REST service by creating an entry in the WEB SERVICE file (#18.02)

Input Parameters

 service name: REST Web Service Name (File #18.02)

 context root: Context Root for the REST service (without leading or trailing '/' characters)

 resource: resource for HWSC to access via an HTTP GET when checking if the web service is available. HWSC appends the resource to the IP address and context root of the web service.

5.4.3 UNREG (service name)

Use in installation post-init routines to un-register/delete a web service entry in the WEB SERVICE file (#18.02). Can be either a SOAP or REST web service. Also removes the service from any web servers it is authorized to.

Input Parameters

 service name: SOAP or REST Web Service Name (File #18.02)

5.4.4 $$GET (RestRequest, resource, [.error], [ForceError])

Make HTTP GET call and (by default) force an error trap if problem encountered.

Input Parameters

 RestRequest: xobw.RestRequest object

 Resource: resource string to use with GET method

 error: (optional) where to store any error encountered (pass by ref) – errors returned as an xobw.error object

 ForceError: (optional) force error trap (1) or not (0). Defaults to 1.

Return Value

True if succeeded, false if an error occurred.

 NOTE: If `ForceError` is set to 1, a $ECODE is thrown and the return value QUIT is never reached.

5.4.5 $$POST (RestRequest, Resource, [.error], [ForceError])

Make HTTP POST call and (by default) force an error trap if problem encountered.

Input Parameters

RestRequest: `xobw.RestRequest` object

Resource: resource string to use with POST method

error: (optional) where to store any error encountered (pass by ref) – errors
                 returned as an `xobw.error` object

ForceError: (optional) force error trap (1) or not (0). Defaults to 1.

Return Value

True if succeeded, false if an error occurred.

NOTE: If ForceError is set to 1, a $ECODE is thrown and the return value QUIT is never reached.

5.4.6 $$HTTPCHK (RestRequest, [.error], [ForceError])

Check HTTP status after a GET, POST, or PUT operation has completed; if HTTP status code indicated condition other than success, create an `HttpError` object and return false.

Input Parameters

`RestRequest: xobw.RestRequest object`

`error:`   (optional) where to store any error encountered (pass by ref) – errors
                 returned as an xobw.error object

`ForceError:`   (optional) force error trap (1) or not (0). Defaults to 1.

Return Value

True if HTTP status judged OK, false if a condition other than success occurred.

NOTE: If `ForceError` is set to 1, a $ECODE is thrown and the return value QUIT is never reached.

5.4.7 $$HTTPOK (http status code)

HealtheVet Web Services Client (HWSC)   May 2013
                                     Developer Guide
                                       Version 1.0

Check HTTP status after a GET, POST, or PUT operation has completed; if HTTP status code indicated condition other than success, return false.

Input Parameters

> http status code: String containing HTTP status code (e.g., from
> `xobw.RestRequest.HttpResponse.StatusCode`)

Return Value

True if HTTP status judged OK, false if a condition other than success occurred.

5.4.8 $$GETRESTF (service name)

Return REST service request factory (use to create one or more REST service request objects for a particular web service).

Input Parameters

> service name: REST Web Service Name (File #18.02)

Return Value

REST service request factory (`xobw.RestRequestFactory`) object.

5.5 Error Handling APIs

All the APIs listed in this section are tags in the XOBWLIB routine.

5.5.1 $$EOFAC ([SOAP proxy object])

For use in error trap handlers during SOAP and REST web services calls, to make it easy to process error conditions. Creates an error object based on the error condition in the partition, representing a SOAP, Caché Object, HWSC dialog, or basic M error. Includes special parsing for <ZSOAP> web service errors.

Intended for use in an error trap handler, i.e., a known error condition is already present in the partition.

Input Parameters

> SOAP proxy object: (optional) SOAP proxy object (if making a SOAP call)

Return Value

Error Object   Caché Object representing the trapped and parsed error (assumes EOFAC^XOBWLIB is being called in an error trap handler) is an instance of one of the following classes in the "xobw.error" package:

Table 5-3. Classes in the "xobw.error" package

Class	Error Type
BasicError	basic M/ Caché error
DialogError	HWSC fault with corresponding DIALOG file (#.84) entry
ObjectError	Caché Object-level error
SoapError	SOAP fault returned from web service invocation
AbstractError	base class for all error types

The type of error returned can be determined by the %IsA method which is available in all of the error classes above.

> **NOTE:** HWSC's HttpError objects are not returned by the $$EOFAC call. HTTP errors are returned by HWSC as follows:
>
> - SOAP: Caché returns HTTP errors encountered during SOAP calls in an object error; $$EOFAC returns such errors in ObjectError objects
>
> - REST: The $$GET and $$POST calls return errors, including an HttpError object if an HTTP error is encountered, directly without the need to call $$EOFAC. Alternatively, if using %Net.Request directly (rather than through the $$GET/$$POST wrappers), code can call $$HTTPCHK to check for HTTP errors and create an HttpError object if an error is found.

5.5.2 $$EOSTAT (status object)

Create ObjectError from Caché status (%Library.Status) object.

Input Parameters

status object: Caché %Library.Status object

Return Value

xobw.error.ObjectError object.

5.5.3 $$EOHTTP (response object)

Create HttpError object from Caché %Net.Response object.

Input Parameters

    response object: %Net.HttpResponse object (e.g., from
              xobw.RestRequest.HttpResponse)

Return Value

    xobw.error.HttpError object.

5.5.4 ERRDISP (error object)

Does a simple display of an error's information to the screen. "Error Object" should be of the type xobw.error.AbstractError or one of its descendants.

Input Parameters

    error object: Any HWSC error object in the xobw.error package.

5.5.5 ERR2ARR (error object, .return array)

Decomposes an error object into an M array carrying the various components of the error object. "Error Object" is should be of the type xobw.error.AbstractError or one of its descendants.

Input Parameters

    error object: Any HWSC error object in the xobw.error package.
    return array: (pass by ref) Array in which to return the decomposed components of the error object.

Output Parameters

See the online "Documatic" documentation for xobw.error to see what array nodes are populated for each xobw.error class type.

5.5.6 $$STATCHK (status object, [.error], [ForceError])

Check Caché %Library.Status status object (returned by many Caché Object calls); if not OK create ObjectError object and return false.

Input Parameters

status object:	Caché %Library.Status object
error:	(optional) where to store any error encountered (pass by ref) – errors returned as an xobw.error object
forceerror:	(optional) force error trap (1) or not (0). Defaults to 1.

Return Value

True if succeeded, false if an error occurred.

> ℹ **NOTE:** If ForceError is set to 1, a $ECODE is thrown and the return value QUIT is never reached.

5.5.7 ZTER (error object)

Performs two functions:

- Decomposes error object into an XOB-namespaced M array carrying the various components of the error object.

- Calls Kernel error trap to record error.

It is useful to decompose the error into an M array before calling the Kernel error trap, because otherwise the Caché Object error information is not captured in the error trap.

Input Parameters

error object:	Any HWSC error object in the xobw.error package (should be of the type xobw.error.AbstractError or one of its descendants).

Output Parameters

See the online "Documatic" documentation for xobw.error to see what array nodes are populated into the XOB-namespaced array for each xobw.error class type.

Example

```
SET MYERROBJ=$$EOFAC^XOBWLIB()
DO ZTER^XOBWLIB(MYERROBJ)
```

5.6 Server Lookup APIs

All the APIs listed in this section are tags in the XOBWLIB routine.

5.6.1 $$SKEYADD (key name, [description], [.error])

Add a new server lookup key, or edit an existing one.

<u>Input Parameters</u>

key name: Name of server lookup key.

description: (optional) Brief description of lookup key.

error: (optional) location to return error description (pass by reference) – returned
                as array node(s) starting at error(1)

<u>Return Value</u>

Success: IEN of new or existing entry (always > 0)

Failure: 0
                Also, error description node(s) are returned in optional error parameter.

5.6.2 $$SNAME4KY (key name, .retvalue, [.error])

Retrieve the server name associated with a server lookup key.

<u>Input Parameters</u>

key name: Name of server lookup key.

retvalue: Storage location to return server name if successful (pass by reference)

error: (optional) location to return error information in if failure (pass by
                reference)

<u>Return Value</u>

Success: IEN of new or existing entry (always > 0) (and the matching server name is
                returned in the "server name" parameter)

Failure: 0
                Also, an error is returned in the optional error parameter.
                [error format]:   error `code^error` text. Possible errors:

186008^description (invalid key)
186009^description (server association missing)

Example

```
SET SUCCESS=$$SNAME4KY^XOBWLIB("PSO LOCAL SERVER",.PSOSRVR,.PSOERR)
I SUCCESS W !,"Using Server: ",PSOBSRV
ELSE  W !,"could not retrieve server: "_$P(PSOERR,U,2)
```

5.7 APIs For Developer Test Account Use Only

5.7.1 $$DISPSRVS^XOBWLIB

For Developer Testing only. Displays, on the current device, a list of all entries defined in the WEB SERVER (#18.12) file.

The primary purpose of this helper procedure is to help developers create testing code and diagnose issues.

Input Parameters

None

Return Value

None. However, the procedure displays the following fields for each application server entry:

- Web server name, IP Address:Port

5.7.2 $$GETSRV^XOBWLIB()

For Developer Testing only. This method is interactive and allows the end user to select an entry in the WEB SERVER file (#18.12) file.

The method returns the site name (.01 field) of the entry selected. The primary purpose of this helper method is to help developers create testing code.

Return Value

Name from the entry selected in the WEB SERVER file (#18.12) file OR the empty string if no entry was selected.

5.7.3 $$SELSRV^XOBWLIB()

For Developer Testing only. Provides interactive display of WEB SERVER, returning user selected web server (combination of $$DISPSRVS and $$GETSRV calls).

<u>Return Value</u>

Name from the entry selected in the WEB SERVER (#18.12) file, OR the empty string, if no entry was selected.

Appendix A: HWSC Error Codes

Error code entries are contained in the DIALOG file (#.84). The following dialog entries are used in the HWSC package:

Table A-1. HWSC APIs

Dialog Number	Short Description
186001	(reserved for future use)
186002	Web Server Disabled
186003	Web Service not registered to server
186004	Web Service disabled for web server
186005	Web Server not defined
186006	Web Service not defined
186007	Web Service is wrong type.
186008	Invalid Server Lookup Key
186009	Server Lookup Key Missing Association

Glossary

AA   *Authentication and Authorization*

Business Delegate   A business delegate acts as a representative of the client components and is responsible for hiding the underlying implementation details of the business service. It knows how to look up and access the business services.

Certificate Authority   "A certificate authority (CA) is an entity that creates and then 'signs'
(CA)   a document or file containing the name of a user and his public key. Anyone can verify that the file was signed by no one other than the CA by using the public key of the CA. By trusting the CA, one can develop trust in a user's public key.

                                The trust in the certification authority's public key can be obtained recursively. One can have a certificate containing the certification authority's public key signed by a superior certification authority *(Root CA)* that he already trusts. Ultimately, one need only trust the public keys of a small number of top-level certification authorities. Through a chain of certificates *(Sub CAs)*, trust in a large number of users' signatures can be established.

                                A broader application of digital certification includes not only name and public key but also other information. Such a combination, together with a signature, forms an extended certificate. The other information may include, for example, electronic-mail address, authorization to sign documents of a given value, or authorization to sign other certificates."[2]

                                Currently, the Department of Veterans Affairs (VA) uses VeriSign, Inc. as the Certificate Authority (CA).

Cryptography   The system or method used to write or decipher messages in code (see "Encryption" and "Decryption").

CSR   *Certificate Signing Request.*

Decryption   Using a secret key to unscramble data or messages previously encrypted with a cipher or code so that they are readable. In some cases, encryption algorithms are one directional (i.e., they only encode and the resulting data *cannot* be unscrambled).

Encryption   Scrambling data or messages with a cipher or code so that they are

---

[2] DEA Web site (http://www.deadiversion.usdoj.gov/ecomm/e_rx/con_ops/index.html): "Public Key Infrastructure Analysis Concept of Operations," Section 3.4.3 "Public Key - The I in PKI"

	unreadable without a secret key. In some cases, encryption algorithms are one directional (i.e., they only encode and the resulting data *cannot* be unscrambled).
HTTP Protocol	Hyper Text Transfer Protocol is the underlying protocol used by the World Wide Web. HTTP defines how messages are formatted and transmitted, and what actions web servers and browsers should take in response to various commands.
HWSC	Health*e*Vet Web Services Client is a support framework that offers VistA/M applications real-time, synchronous client access to n-tier (J2EE) web services through the supplied M-based and Caché APIs.
Intermediate CA	Intermediate Certificate Authority. Currently, the Department of Veterans Affairs (VA) uses VeriSign, Inc. as the Certificate Authority (CA). VeriSign requires the use of an CA Intermediate Certificate. The CA Intermediate Certificate is used to sign the peer's (server) certificate. This provides another level of validation-managed PKI for SSL.
J2EE	The Java 2 Platform, Enterprise Edition (J2EE) defines the standard for developing multi-tier enterprise applications. J2EE defines components that function independently, that can be deployed on servers, and that can be invoked by remote clients. The J2EE platform is a set of standard technologies and is not itself a language. The current J2EE platform is version 1.4.
PKI	Public Key Infrastructure technology adds the following security services to an electronic ordering system:

- Confidentiality — only authorized persons have access to data.

- Authentication — establishes who is sending/receiving data.

- Integrity — the data has not been altered in transmission.

- Non-repudiation — parties to a transaction cannot convincingly deny having participated in the transaction."[3]

Private Certificate	This is the certificate that contains both the user's public and private keys. This certificate will reside on a smart card.

---

[3] DEA Web site (http://www.deadiversion.usdoj.gov/ecomm/e_rx/con_ops/index.html): "Public Key Infrastructure Analysis Concept of Operations," Section 3.3 "Security"

Public Certificate | This is the certificate that contains the user's public key. This certificate will reside in a file or database.

REST | *Representational State Transfer* (REST) is an architectural style for simplified web services, based on accessing resources via HTTP.

Root CA | *Root Certificate Authority*. In cryptography and computer security, a root certificate is an unsigned public key certificate, or a self-signed certificate, and is part of a public key infrastructure scheme. The most common commercial variety is based on the ITU-T X.509 standard. Normally an X.509 certificate includes a digital signature from a Certificate Authority (CA), which vouches for correctness of the data contained in a certificate. Root certificates are implicitly trusted.

Currently, the Department of Veterans Affairs (VA) uses VeriSign, Inc. as the Certificate Authority (CA).

Service Facade | The Service Façade acts as the server-side bridge between the Business Delegate and the capability. The Service Façade is responsible for taking a request from the delegate and doing any translation necessary to invoke the capability and provide the response to the delegate.

Servlet Container | A servlet is managed by a servlet container (formerly referred to as *servlet engine*.) The servlet container is responsible for loading and instantiating the servlets and then calling `init()`. When a request is received by the servlet container, it decides what servlet to call in accordance with a configuration file. A famous example of a servlet container is Tomcat.

The servlet Container calls the servlet's service() method and passes an instance of ServletRequest and ServletResponse. Depending on the request's method (mostly GET and POST), service calls `doGet()` or `doPost()`. These passed instances can be used by the servlet to find out who the remote user is, if and what HTTP POST parameters have been set and other characteristics.

Together with the web server (or *application server*) the servlet container provides the HTTP interface to the world.
It is also possible for a servlet container to run standalone (without web server), or to even run on a host other than the web server. [4]

Signed Certificate | The Signed Certificate (a.k.a. self-signed certificate) is the peer's

---

[4] From the *ADP –Analyse, Design & Programing GmbH* website:
http://www.adp-gmbh.ch/java/servlets/container.html

(server) digital certificate. Currently, the Department of Veterans Affairs (VA) uses VeriSign, Inc. as the Certificate Authority (CA) to sign (validate) digital certificates. VeriSign, Inc. requires the use of CA Root and Intermediate Certificates. The Subject and Issuer will have the same content when signed by VeriSign; the issuer will have VeriSign's content.

SOAP

Simple Object Access Protocol (SOAP) is a protocol for exchanging structured information over a network, often via HTTP.

SSL

Secure Socket Layer. A low-level protocol that enables secure communications between a server and a browser. It provides communication privacy.

TLS

Transport Layer Security. Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL), are cryptographic protocols which provide secure communications on the Internet for such things as web browsing, e-mail, Internet faxing, instant messaging and other data transfers. There are slight differences between SSL 3.0 and TLS 1.0, but the protocol remains substantially the same.

Web Service

A web resource meant to be consumed over a network via HTTP, by an autonomous program.

WebLogic

A BEA product, WebLogic Server 8.1 is a J2EE- v1.3-certified application server for developing and deploying J2EE enterprise applications.

WSDL

Web Services Definition Language. "WSDL is an XML-based service description on how to communicate using web services. The WSDL defines services as collections of network endpoints, or ports. WSDL specification provides an XML format for documents for this purpose.

The abstract definition of ports and messages is separated from their concrete use or instance, allowing the reuse of these definitions. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. Messages are abstract descriptions of the data being exchanged, and port types are abstract collections of supported operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding, where the messages and operations are then bound to a concrete network protocol and message format. In this way, WSDL describes the public interface to the web service.

WSDL is often used in combination with SOAP and XML Schema to provide web services over the Internet. A client program

HealtheVet Web Services Client (HWSC) Developer Guide Version 1.0

connecting to a web service can read the WSDL to determine what functions are available on the server. Any special datatypes used are embedded in the WSDL file in the form of XML Schema. The client can then use SOAP to actually call one of the functions listed in the WSDL."[5]

For a comprehensive list of commonly used infrastructure- and security-related terms and definitions, please visit the Security and Other Common Services Glossary Web page at the following Web address:

http://vista.med.va.gov/iss/glossary.asp

For a comprehensive list of acronyms, please visit the Security and Other Common Services Acronyms Web site at the following Web address:

http://vista/med/va/gov/iss/acronyms/index.asp

---

[5] http://en.wikipedia.org/wiki/Web_Services_Description_Language

HealtheVet Web Services Client (HWSC)
Developer Guide
Version 1.0