HealtheVet Web Services Client (HWSC) 1.0

Developer's Guide

October 2016

Department of Veterans Affairs (VA)

Office of Information and Technology (OI&T)

Enterprise Program Management Office (EPMO)

Revision History

Date	Version	Description	Author
10/20/2016	2.0	Tech Edits: • Updated the "Orientation" section. • Updated Figure 1, Figure 2, and Figure 5 for HTTPS. • Updated/Renamed Section 1.6. Added new subsections for HTTPS. • Updated and reformatted Section 5, APIS. • Converted Word document to .docx format. • Reformatted document to follow latest documentation standards and formatting rules. Also, formatted document for online presentation vs. print presentation (i.e., for double-sided printing). These changes include: o Revised section page setup. o Removed section headers. o Revised document footers. o Removed blank pages between sections. o Revised all heading style formatting. • Updated organizational references (e.g., "Product Development [PD]" to "Enterprise Program Management Office [EPMO]"). • Redacted document for the following information: o Names (replaced with role and initials). o Production IP addresses and ports. o VA Intranet websites. o Server geographic locations and node names.	HealtheVet Web Services Client (HWSC) Project Team
05/--/2013	1.1	Changed reference to XOBT_1_0_Txx.zip in Section 2.1.	HP VistA Maintenance Team. St Petersburg, Florida • Developer—D. E. • Tech Writer—B. S.

Date	Version	Description	Author
02/--/2011	1.0	HealtheVet Web Services Client (HWSC) Version 1.0 Initial release.	HWSC development team. Albany, NY OIFO: • Developer—M. K • Developer—L. D. Bay Pines, FL OITFO: • Development Manager—C. S. Oakland, CA OITFO: • Developer—K. C. • SQA—G. S. • Tester—P. S. • Tech Writer—S. S.

Table of Contents

List of Figures

List of Tables

Orientation

How to Use this Manual

Throughout this manual, advice and instructions are offered regarding the installation and use of HealtheVet Web Services Client (HWSC) and the functionality it provides for Veterans Information Systems and Technology Architecture (VistA).

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

1. Introduction
2. Sample SOAP and REST Client Applications
3. VistA M-Side Development Guide
4. Java-Side Considerations
5. VistA M-Side API Reference

Intended Audience

The intended audience of this manual is the following stakeholders:

- Enterprise Program Management Office (EPMO)—VistA legacy development teams.
- System Administrators—System administrators at Department of Veterans Affairs (VA) sites who are responsible for computer management and system security on the VistA M Servers.
- Information Security Officers (ISOs)—Personnel at VA sites responsible for system security.
- Product Support (PS)—Personnel who support Kernel-related products.

Disclaimers

Software Disclaimer

This software was developed at the Department of Veterans Affairs (VA) by employees of the Federal Government in the course of their official duties. Pursuant to title 17 Section 105 of the United States Code this software is *not* subject to copyright protection and is in the public domain. VA assumes no responsibility whatsoever for its use by other parties, and makes no guarantees, expressed or implied, about its quality, reliability, or any other characteristic. We would appreciate acknowledgement if the software is used. This software can be redistributed freely provided that any derivative works bear some notice that they are derived from it.

CAUTION: Kernel routines should *never* be modified at the site. If there is an immediate national requirement, the changes should be made by emergency Kernel patch. Kernel software is subject to FDA regulations requiring Blood Bank Review, among other limitations. Line 3 of all Kernel routines states:

Per VHA Directive 2004-038, this routine should not be modified

CAUTION: To protect the security of VistA systems, distribution of this software for use on any other computer system by VistA sites is prohibited. All requests for copies of

Kernel for *non*-VistA use should be referred to the VistA site's local Office of Information Field Office (OIFO).

Documentation Disclaimer

This manual provides an overall explanation of using Kernel; however, no attempt is made to explain how the overall VistA programming system is integrated and maintained. Such methods and procedures are documented elsewhere. We suggest you look at the various VA Internet and Intranet SharePoint sites and websites for a general orientation to VistA. For example, visit the Office of Information and Technology (OI&T) Enterprise Program Management Office (EPMO) Intranet Website.

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

Documentation Conventions

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

- Various symbols are used throughout the documentation to alert the reader to special information. Table 1 gives a description of each of these symbols:

Table 1: Documentation Symbol Descriptions

Symbol	Description
	**NOTE / REF:** Used to inform the reader of general information including references to additional reading material.
	**CAUTION / RECOMMENDATION / DISCLAIMER:** Used to caution 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 commands and online displays (i.e., screen captures/dialogues) and computer source code, if any, are shown in a *non*-proportional font and may be enclosed within a box.
  - User's responses to online prompts are **boldface** and (optionally) highlighted in yellow (e.g., **<Enter>**).
  - Emphasis within a dialogue box is **boldface** and (optionally) highlighted in blue (e.g., STANDARD LISTENER: RUNNING).
  - Some software code reserved/key words are **boldface** with alternate color font.
  - References to "**<Enter>**" within these snapshots indicate that the user should press the **Enter** key on the keyboard. Other special keys are represented within < > angle brackets. For example, pressing the **PF1** key can be represented as pressing **<PF1>**.

- Author's comments 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.

- This manual refers to the M programming language. Under the 1995 American National Standards Institute (ANSI) standard, M is the primary name of the MUMPS programming language, and MUMPS is considered an alternate name. This manual uses the name M.
- Descriptions of direct mode utilities are prefaced with the standard M ">" prompt to emphasize that the call is to be used *only in direct mode*. They also include the M command used to invoke the utility. The following is an example:

  ```
  >D ^XUP
  ```

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

  NOTE: Other software code (e.g., Delphi/Pascal and Java) variable names and file/folder names can be written in lower or mixed case (i.e., CamelCase).

How to Obtain Technical Information Online

Exported VistA M Server-based software file, routine, and global documentation can be generated through the use of Kernel, MailMan, and VA FileMan utilities.

NOTE: Methods of obtaining specific technical information online are indicated where applicable under the appropriate section.

Help at Prompts

VistA M Server-based software provides online help and commonly used system default prompts. Users are encouraged to enter question marks at any response prompt. At the end of the help display, you are immediately returned to the point from which you started. This is an easy way to learn about any aspect of VistA M Server-based software.

Obtaining Data Dictionary Listings

Technical information about VistA M Server-based files and the fields in files is stored in data dictionaries (DD). You can use the List File Attributes option [DILIST] on the Data Dictionary Utilities menu [DI DDU] in VA FileMan to print formatted data dictionaries.

REF: For details about obtaining data dictionaries and about the formats available, see the "List File Attributes" chapter in the "File Management" section in the *VA FileMan Advanced User Manual*.

Assumptions

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

- VistA computing environment:
  - Kernel 8.0—VistA M Server software
  - Remote Procedure Call (RPC) Broker 1.1—VistA M Server software
  - VA FileMan 22.0 data structures and terminology—VistA M Server software
  - VistALink 1.6—VistA M Server and Application Server software
- Linux or Microsoft® Windows environment
- Java Programming language:
  - Java Integrated Development Environment (IDE)
  - J2SE™ Development Kit (JDK)
  - Java Authentication and Authorization Services (JAAS) programming
- M programming language
- WebLogic 9.2 or 10.x Application Server

Reference Materials

Readers who wish to learn more about HWSC should consult the following:

- *HWSC 1.0 Installation Guide*
- *HWSC 1.0 Systems Management Guide*
- *HWSC 1.0 Developer's Guide* (this manual)
- *HWSC 1.0 Patch XOBW*1.0*4 Release Notes*
- *HWSC 1.0 Patch XOBW*1.0*4 Installation, Back-Out, and Rollback Guide*
- *HWSC 1.0 Patch XOBW*1.0*4 Security Configuration Guide*

VistA documentation is made available online in Microsoft® Word format and in Adobe® Acrobat Portable Document Format (PDF). The PDF documents *must* be read using the Adobe® Acrobat Reader, which is freely distributed by Adobe® Systems Incorporated at: http://www.adobe.com/

VistA documentation can be downloaded from the VA Software Document Library (VDL): http://www.va.gov/vdl/

REF: HWSC manuals are located on the VDL at: http://www.va.gov/vdl/application.asp?appid=180

VistA documentation and software can also be downloaded from the Product Support (PS) Anonymous Directories.

1 Introduction

1.1 Document Overview

This document provides information for M application developers writing code to call Web services in HealtheVet applications. It presents information on the following topics:

- HWSC architecture.
- Implementing Caché code with HWSC to consume to external Service Oriented Architecture Protocol (SOAP)-based and Representational State Transfer (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 HealtheVet from VistA M

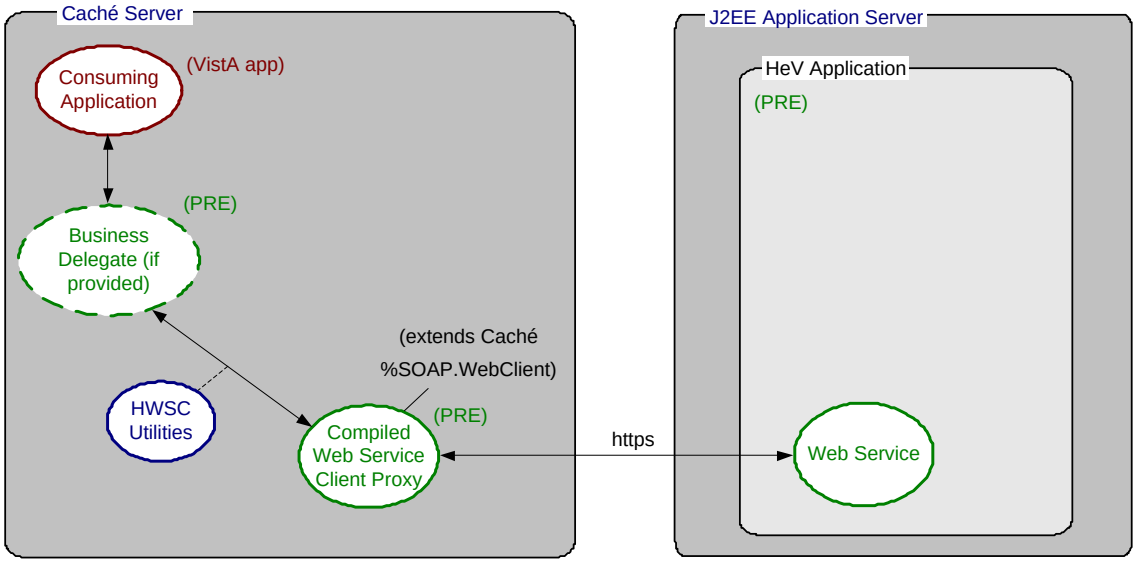
Veterans Health Information Systems and Technology Architecture (VistA) M-based applications have a need to synchronously access VistA application services and data (e.g., in-process during an end-user's session). HWSC uses Caché's Web Services Client to invoke Web service methods on external servers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

HWSC supports two modes of synchronous Web service access:

- SOAP (Service Oriented Architecture Protocol)—Formal XML-based protocol for accessing services.
- REST (REpresentational State Transfer)—Architectural *style* of accessing services via programmatic access to Web resources.

The SOAP and Rest approaches to calling VistA services are shown in Figure 1 and Figure 2.

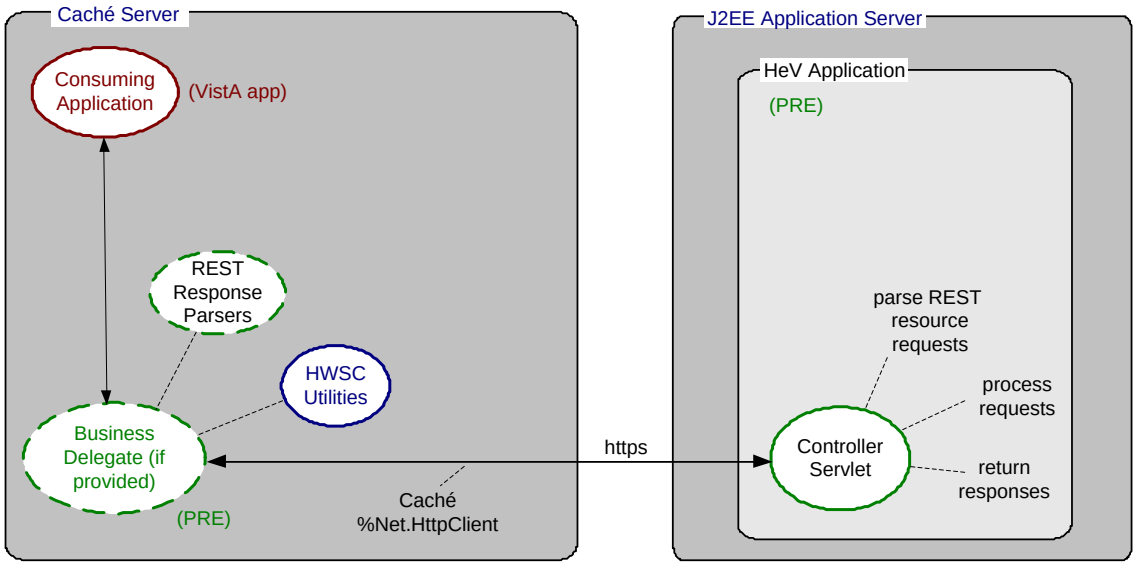
Figure 1: HWSC Logical View (SOAP-style)



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 HealtheVet 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 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.

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 "HPMO Waiver for Use of Caché-Specific Features" section).

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 VistA M systems 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 reproduced below (Figure 3):

Figure 3: 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

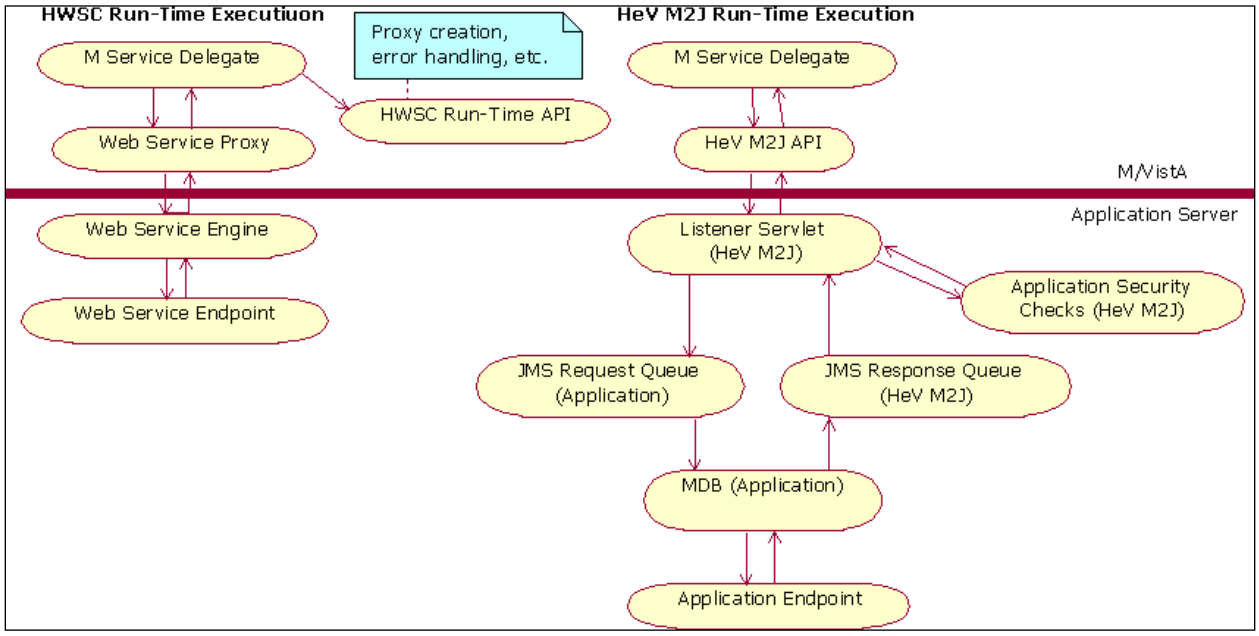
Supporting Documentation

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.

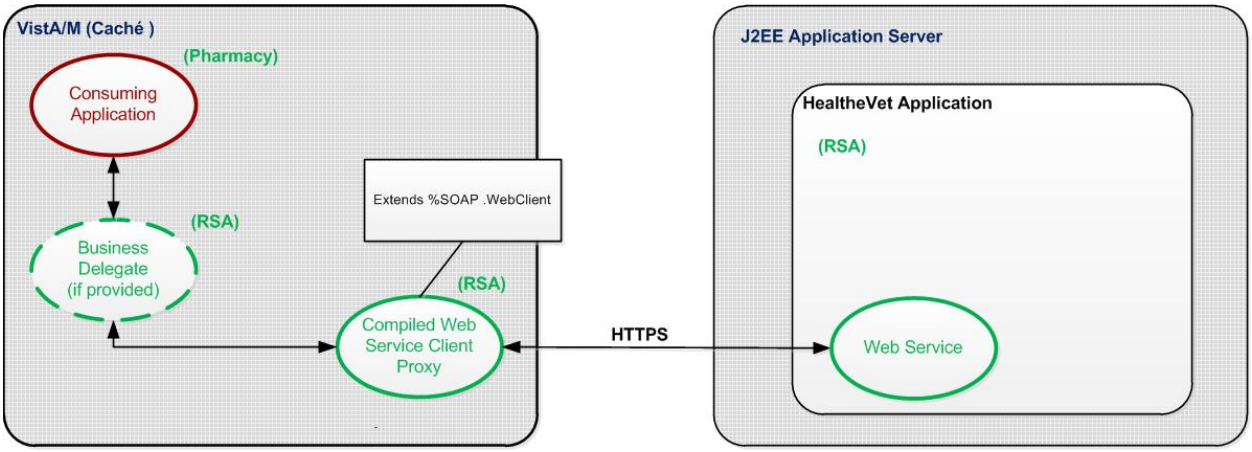
To better understand the context of the waiver, it is also useful to examine the listed supporting documentation for the technical decision (see Figure 4 and Figure 5).

Figure 4: Supporting Documentation: VWSC Architecture



In Figure 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 5: Supporting Documentation: VWSC Proposed View



Similarly in the example shown in Figure 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 5 captures the *recommended* model for HealtheVet 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 can 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.

1.6 Securing HWSC Applications Using SSL/TLS (HTTPS)

As of Patch XOBW*1.0*4, HWSC enabled the use of Secure Socket Layer/Transport Layer Security (SSL/TLS) encryption. This allows VistA applications to make Hypertext Transfer Protocol (Secure) HTTP(S) connections from VistA on all supported operating system (OS) platforms to remote HTTP(S) servers. HWSC uses a Caché library that makes HTTP or HTTPS requests. A secondary benefit of using SSL/TLS security is the use of Certificate-Based Authentication.

REF: For more information and configuration setup instructions, see the *HWSC 1.0 Patch XOBW*1.0*4 Security Configuration Guide*.

1.6.1 Support for Encryption and Certificate-Based Authentication

With the use of SSL/TLS to encrypt communication, VistA applications can make use of SSL/TLS Certificate Authentication.

REF: For instructions on setting up an HWSC application to use Certificate Based Authentication, see the "Client Certificate Authentication Configuration" section in the *HWSC 1.0 Patch XOBW*1.0*4 Security Configuration Guide*.

1.6.2 Support for Encryption and HTTP Basic Authentication

Support for HTTP Basic Authentication existed with the initial release of HWSC; however, username and password credentials were transported in clear text. With the use of SSL/TLS to encrypt the transmission of credentials, VistA applications also have a secured alternative to using Certificate-Based Authentication.

REF: For instructions on setting up an HWSC application to use Encryption and HTTP Basic Authentication, see the "Encryption-Only Setup" and "HTTP Basic Authentication Configuration" sections in the *HWSC 1.0 Patch XOBW*1.0*4 Security Configuration Guide*.

2 Sample SOAP and REST Client Applications

HealtheVet Web Services Client (HWSC) provides a sample application that demonstrates how to consume both Service Oriented Architecture Protocol (SOAP)- and Representational State Transfer (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
- Error handling

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 Web Services Description Language (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.

REF: For installation instructions, see the appendices in the *HWSC 1.0 Installation Guide*.

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 1.0 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 file (#18.12) for the Web server on which they are installed, 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 Table 2.

Table 2: 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. Select option: XOBW WEB SERVER MANAGER.
2. Select Action: **Test Server**.
3. Select your server entry.
4. Select any of the options to run.

Figure 6: 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:

1. Select option: XOBW WEB SERVER MANAGER.
2. Select action: **Test Server**.
3. Select your server entry.
4. Select sub-action: **Ping Test**.
5. Successful result: "Response: Ping Successful!"

2.2.5 Demonstrating Server Lookup Keys: XOBT SAMPLE SERVER Lookup Key

The server lookup XOBT SAMPLE SERVER security key 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 security 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 keys, and avoid hard-coding a particular Web server file entry name.

To associate a server with the XOBT SAMPLE SERVER security 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 security key.
4. Associate it with the server on which you have installed the sample Web service.
5. 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 (#18.12). If empty, you are prompted to select a Web server against which to run the option.

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):

- 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.
5. Update the WEBLOGIC.DIRPATH property to point to a WebLogic home on your system.
6. In the unzipped "**sample-prj**" folder, from the command line, run ANT (no arguments necessary).
7. 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., Service Oriented Architecture Protocol [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 "Manually Modify SOAP Client Proxies to Overcome Length Limits."
- **Length of REST-Style Returned Data—**Because HealtheVet Web Services Client (HWSC) returns the response to a Representational State Transfer (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 "Manually Modify SOAP Client Proxies to Overcome Memory Limitations" 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 can 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 can 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 Web Services Description Language (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 the WEB SERVICE file (#18.02).

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

- Name for the WEB SERVICE file entry to be created.
- File system location of your WSDL file (accessible from the Caché install account).
- Caché package name to use for the compiled proxy classes.
- 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.

REF: For more details on the GENPORT^XOBWLIB API, see the "$$GENPORT^XOBWLIB(): Import/Register Web Service from WSDL" section.

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 entry in the WEB SERVER file (#18.12) on which to call the Web service. For simplicity's sake, in the example in Figure 7 the Web server name ("VHASERVER1") is hard-coded; for production code, you can instead use the $$SNAME4KY^XOBWLIB(): Retrieve Server Name Associated with a Server Lookup Key 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 methods on Web service proxy object to invoke methods on the remote Web service.

For example, the code in Figure 7 invokes a doPing method on the MYAPP WEB SVC Web service, and writes out the result:

Figure 7: Code to invoke a doping method

```
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()
```

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

3.2.2 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.

3.2.3 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.

3.2.4 Large Result Sets (SOAP)

The role in the enterprise architecture currently envisioned for Web services, particularly for M/VistA systems accessing HealtheVet 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 can 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) *cannot* read in the data stream fast enough.

3.2.5 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).

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.

[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.

3.2.6 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:

- 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)
- Post-init routine (in the KIDS transport file) that calls $$GENPORT^XOBWLIB to:
  - Import the WSDL file.
  - Create an entry in the WEB SERVICE file (#18.02).
  - 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.

3.2.7 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 can 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:

1. Change the method return type from %Library.String to %GlobalCharacterStream in the generated port proxy class.
2. Recompile the port proxy class; this causes other generated proxy classes to be re-compiled.
3. 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 can also work for %String input parameter types, which can 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.

3.2.7.1 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 need to remember to manually restore the modifications to the generated classes.
- The application needs 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 file (#18.02) entry needs to be created or 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 entry in the WEB SERVER file (#18.12) on which to call the Web service. 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:

Figure 8: Sample M function that accesses a Ping resource

```
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.

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

3.3.2 Parsing XML Responses (REST)

Suppose the XML response for the example in Figure 8 looks like:

Figure 9: Sample XML response

```
<?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*.
  - 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).
- 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 the REGREST^XOBWLIB(): Register a REST Service Definition API to create an entry in the WEB SERVICE file (#18.02).
  - (Optionally) calls the $$SKEYADD^XOBWLIB(): Add a Server Lookup Key API 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 can depend in part on whether you are 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 a 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 can 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

- 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 Service Oriented Architecture Protocol (SOAP)- or Representational State Transfer (REST)-style Web service. In some cases, one type of call (SOAP or REST) can 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 Web Services Description Language [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

HealtheVet Web Services Client (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 3: 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 HealtheVet Web Services Client (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:

   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 APIs Overview

All the HWSC APIs listed in the Table 4 are tags in the XOBWLIB routine.

Table 4: HWSC APIs

Category	M API	Brief Description
Service Oriented Architecture Protocol (SOAP)	$$GETPROXY	Return Web Service Proxy.
	$$GENPORT	Import/Register Web Service from Web Services Description Language (WSDL).
	REGSOAP	Register Web Service without WSDL.
	$$GETFAC	Return Web Service Proxy Factory.
	ATTACHDR	Add VistaInfoHeader to a Web Service Proxy.
	UNREG	Un-Register/Delete a Web Service.
Representational State Transfer (REST)	$$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.

Category	M API	Brief Description
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

This section describes the HWSC SOAP-related APIs.

5.3.1 $$GETPROXY^XOBWLIB(): Return Web Service Proxy

Reference Type: Supported

Category: HWSC

IA #: 5421

Description: This extrinsic function 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.

Format: `$$GETPROXY^XOBWLIB(web_service_name,web_server_name)`

Input Parameters:

web_service_name:	(required) Name of entry in the WEB SERVICE file (#18.02).
web_server_name:	(required) Name of entry in the WEB SERVER file (#18.12).

Output:

returns:	Web service client proxy object ready to invoke Web service methods on the specified Web server.

5.3.2 $$GENPORT^XOBWLIB(): Import/Register Web Service from WSDL

Reference Type: Supported

Category: HWSC

IA #: 5421

Description: This extrinsic function imports a Web Services Description Language (WSDL) file and runs the Caché WSDL import wizard. It does the following:

- 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 the WEB SERVICE file (#18.02).

Use this call in installation post-init routines.

Format: `$$GENPORT^XOBWLIB(.infoarray)`

Input Parameters:	.infoarray:	(required) Passed by reference. Set up array as follows: • infoarray("WSDL FILE")—WSDL file location on host operating system. • infoarray("CACHE PACKAGE NAME")—Package name in which to place generated Caché classes. • infoarray("WEB SERVICE NAME")—Name to store Web service information in the WEB SERVICE file (#18.02). It's used for lookups and 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.
Output:	returns:	Returns: • **Success:** Positive value • **Failure:** 0^failure description

5.3.2.1 Example

The following example shows a *successful* creation of an entry for the WEB SERVICE file (#18.02):

```
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^XOBWLIB(): Register Web Service without WSDL

Reference Type:	Supported
Category:	HWSC
IA #:	5421
Description:	This API registers 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. Use this call in installation post-init routines.
Format:	`REGSOAP^XOBWLIB(wsname,wsroot,class[,path][,resource])`

Input Parameters:	wsname:	(required) Web Service Name.
	wsroot:	(required) Web Service context root (without trailing "/").
	class:	(required) Caché package + class name of the main class created for the Web service client proxy, as created by the Caché WSDL compiler. **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. **NOTE:** The element names above can be prefaced by a namespace abbreviation (e.g., "wsdl:port", "wsdl:service") depending on the WSDL file.
	path:	(optional) The WSDL file location on the host operating system. The WSDL file is copied into the 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.
Output:	none.	

5.3.3.1 Example

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

5.3.4 UNREG^XOBWLIB(): Un-Register/Delete a Web Service

Reference Type:	Supported
Category:	HWSC
IA #:	5421
Description:	This API un-registers/deletes a Web service entry in the WEB SERVICE file (#18.02). It can be either a SOAP or REST Web service. Also, it removes the service from any Web servers to which it is authorized. Use this call in installation post-init routines.
Format:	`UNREG^XOBWLIB(service_name)`

Input Parameters:	service_name:	(required) SOAP or REST Web service name in the WEB SERVICE file (#18.02).
Output:	none.	

5.3.5 $$GETFAC^XOBWLIB(): Return Web Service Proxy Factory

Reference Type: Supported

Category: HWSC

IA #: 5421

Description: This extrinsic function 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.

Format: `$$GETFAC^XOBWLIB(web_service_name)`

Input Parameters:

web_service_name:	(required) Name of entry in the WEB SERVICE file (#18.02).

Output:

returns:	Returns the Web service request factory object (xobw.WebServiceProxyFactory).

5.3.6 ATTACHDR^XOBWLIB(): Add VistaInfoHeader to a Web Service Proxy

Reference Type: Supported

Category: HWSC

IA #: 5421

Description: This API attaches a "VistaInfoHeader" header block to outgoing Web service request. This header block contains partition and Kernel environment variables as follows:

- duz: User's DUZ value.
- mio: Partition's $IO value.
- mjob: Partition's $JOB value.
- production:
  - **"1"—**If the calling VistA system is a **Production** system
  - **"0"—**If the calling VistA system is a **Test** system.
- 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.

Format: `ATTACHDR^XOBWLIB(client_proxy_object)`

Input Parameters:

client_proxy_object:	Web service client proxy object.

Output: none.

5.4 REST-Related APIs

This section describes the HWSC REST-related APIs.

5.4.1 $$GETREST^XOBWLIB(): Return REST Service Request Object

Reference Type: Supported

Category: HWSC

IA #: 5421

Description: This extrinsic function returns the REST service request object. Use is to make GET, POST, and PUT calls to the specified service and server.

Format: `$$GETREST^XOBWLIB(service_name,server_name)`

Input Parameters:

service_name:	(required) REST Web service name in WEB SERVICE file (#18.02).
server_name:	(required) Web server name in the WEB SERVER file (#18.12).

Output:

returns:	Returns the REST service request object (xobw.RestRequest).

5.4.2 REGREST^XOBWLIB(): Register a REST Service Definition

Reference Type: Supported

Category: HWSC

IA #: 5421

Description: This API registers a REST service by creating an entry in the WEB SERVICE file (#18.02).

Use this call in installation post-init routines.

Format: `REGREST^XOBWLIB(service_name,context_root[,resource])`

Input Parameters:

service_name:	(required) REST Web service name in the WEB SERVICE file (#18.02).
context_root:	(required) Context Root for the REST service (*without* leading or trailing "/" characters).
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.

Output: none.

5.4.3 $$GET^XOBWLIB(): Make HTTP GET Call and Force Error if Problem Encountered

Reference Type: Supported

Category: HWSC

IA #: 5421

Description: This extrinsic function makes a HTTP GET call and (by default) forces an error trap if a problem is encountered.

Format: `$$GET^XOBWLIB(restrequest,resource[,.error][,forceerror])`

Input Parameters:	restrequest:	(required) The xobw.RestRequest object.
	resource:	(required) Resource string to use with GET method.
	.error:	(optional) Where to store any error encountered (pass by reference); errors returned as a xobw.error object.
	forceerror:	(optional) Force error trap (1) or not (0). Defaults to 1.
Output:	returns:	Returns: • **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.

5.4.4 $$POST^XOBWLIB(): Make HTTP POST Call and Force Error if Problem Encountered

Reference Type: Supported

Category: HWSC

IA #: 5421

Description: This extrinsic function makes a HTTP POST call and (by default) force an error trap if problem encountered.

Format: `$$POST^XOBWLIB(restrequest,resource[,.error][,forceerror])`

Input Parameters:	restrequest:	(required) The xobw.RestRequest object.
	resource:	(required) Resource string to use with POST method.
	.error	(optional) Passed by reference. This is where to store any error encountered. Errors are returned as a xobw.error object.
	forceerror:	(optional) Force error trap (1) or not (0). Defaults to 1.
Output:	returns:	Returns: • **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 $$HTTPCHK^XOBWLIB(): Check HTTP Status; if Not OK Create HttpError Object

Reference Type: Supported

Category: HWSC

IA #: 5421

Description: This extrinsic function checks the 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.

Format: `$$HTTPCHK^XOBWLIB(restrequest[,.error][,forceerror])`

Input Parameters:

restrequest:	(required) The xobw.RestRequest object.
.error:	(optional) Passed by reference. This is where to store any error encountered. Errors are returned as a xobw.error object.
forceerror:	(optional) Force error trap (1) or not (0). Defaults to 1.

Output: returns: Returns:

- **True—**If HTTP status is 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.

5.4.6 $$HTTPOK^XOBWLIB(): Is Current HTTP Response Status "OK"?

Reference Type: Supported

Category: HWSC

IA #: 5421

Description: This extrinsic function checks the HTTP status after a GET, POST, or PUT operation has completed; if HTTP status code indicated condition other than success, return false.

Format: `$$HTTPOK^XOBWLIB(http_status_code)`

Input Parameters:

http_status_code:	(required) String containing HTTP status code (e.g., from xobw.RestRequest.HttpResponse.StatusCode).

Output: returns: Returns:

- **True—**If HTTP status is judged OK.
- **False—**If a condition other than success occurred.

5.4.7 $$GETRESTF^XOBWLIB(): Return REST Service Request Factory

Reference Type:	Supported	
Category:	HWSC	
IA #:	5421	
Description:	This extrinsic function returns the REST service request factory (use to create one or more REST service request objects for a particular Web service).	
Format:	`$$GETRESTF^XOBWLIB(service_name)`	
Input Parameters:	service_name:	(required) REST Web Service Name in the WEB SERVICE file (#18.02).
Output:	returns:	Returns the REST service request factory (xobw.RestRequestFactory) object.

5.5 Error Handling APIs

This section describes the HWSC error handling APIs.

5.5.1 $$EOFAC^XOBWLIB(): Error Object Factory

Reference Type:	Supported	
Category:	HWSC	
IA #:	5421	
Description:	This extrinsic function is for use in error trap handlers during SOAP and REST Web services calls, to make it easy to process error conditions. It creates an error object based on the error condition in the partition, representing a SOAP, Caché Object, HWSC dialog, or basic M error. It includes special parsing for <ZSOAP> Web service errors. It is intended for use in an error trap handler (i.e., a known error condition is already present in the partition).	
Format:	`$$EOFAC^XOBWLIB([soap_proxy_object])`	
Input Parameters:	soap_proxy_object	(optional) SOAP proxy object (if making a SOAP call).

Output:	returns:	**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 classes in the "xobw.error" package listed in Table 5.

Table 5: 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 in Table 5.

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^XOBWLIB(): Create ObjectError from Caché Status Object

Reference Type:	Supported	
Category:	HWSC	
IA #:	5421	
Description:	This extrinsic function creates the ObjectError from Caché status (%Library.Status) object.	
Format:	`$$EOSTAT^XOBWLIB(status_object)`	
Input Parameters:	status_object:	(required) Caché %Library.Status object.
Output:	returns:	Returns the xobw.error.ObjectError object.

5.5.3 $$EOHTTP^XOBWLIB(): Create HttpError Object from %Net.Response Object

Reference Type: Supported

Category: HWSC

IA #: 5421

Description: This extrinsic function creates the HttpError object from the Caché %Net.Response object.

Format: `$$EOHTTP^XOBWLIB(response_object)`

Input Parameters:	response_object:	(required) The %Net.HttpResponse object (e.g., from xobw.RestRequest.HttpResponse).
Output:	returns:	Returns the xobw.error.HttpError object.

5.5.4 ERRDISP^XOBWLIB(): Simple Display of Error to Screen

Reference Type: Supported

Category: HWSC

IA #: 5421

Description: This API 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.

Format: `ERRDISP^XOBWLIB(error_object)`

Input Parameters:	error_object:	(required) Any HWSC error object in the xobw.error package.
Output:	none.	

5.5.5 ERR2ARR^XOBWLIB(): Decompose Error Object into M Array

Reference Type: Supported

Category: HWSC

IA #: 5421

Description: This API decomposes an error object into an M array carrying the various components of the error object. "Error Object" should be of the type xobw.error.AbstractError or one of its descendants.

Format: `ERR2ARR^XOBWLIB(error_object,.return_array)`

Input Parameters:	error_object:	(required) Any HWSC error object in the xobw.error package.
	.return_array:	(required) Passed by reference. This is the array in which to return the decomposed components of the error object.

Output Parameters:	.return_array:	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^XOBWLIB(): Check Caché %Library.Status Object

Reference Type: Supported

Category: HWSC

IA #: 5421

Description: This extrinsic function checks the Caché %Library.Status status object (returned by many Caché Object calls); if not OK create ObjectError object and return false.

Format:
```
$$STATCHK^XOBWLIB(status_object[,.error][,forceerror])
```

Input Parameters:	status_object:	(required) Caché %Library.Status object.
	.error:	(optional) This is where to store any error encountered (pass by ref) – errors returned as a xobw.error object.
	forceerror:	(optional) Force error trap (1) or not (0). Defaults to 1.
Output:	returns:	Returns: • **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.

5.5.7 ZTER^XOBWLIB(): Decompose Error Object and Call Error Trap

Reference Type: Supported

Category: HWSC

IA #: 5421

Description: This API 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.

Format:
```
ZTER^XOBWLIB(error_object)
```

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

5.5.8 Example

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

5.6 Server Lookup APIs

This section describes the HWSC server lookup APIs.

5.6.1 $$SKEYADD^XOBWLIB(): Add a Server Lookup Key

Reference Type:	Supported	
Category:	HWSC	
IA #:	5421	
Description:	This extrinsic function adds a new server lookup key, or edits an existing one.	
Format:	`$$SKEYADD^XOBWLIB(key_name[,description][,.error])`	
Input Parameters:	key_name:	(required) Name of server lookup key.
	description:	(optional) Brief description of lookup key.
	.error:	(optional) Passed by reference. This is the location to return error description; returned as array nodes starting at error(1).
Output Parameters:	.error:	Error description nodes are returned in this (optional) error parameter.
Output:	returns:	Returns: • **Success:** IEN of new or existing entry (always > 0). • **Failure:** 0.

5.6.2 $$SNAME4KY^XOBWLIB(): Retrieve Server Name Associated with a Server Lookup Key

Reference Type: Supported

Category: HWSC

IA #: 5421

Description: This extrinsic function retrieves the server name associated with a server lookup key.

Format: `$$SNAME4KY^XOBWLIB(key_name,.retvalue[,.error])`

Input Parameters:

key_name:	(required) Name of server lookup key.
.retvalue:	(required) Passed by reference. This is the storage location to return server name if successful.
.error:	(optional) Passed by reference. This is the location to return error information in if failure.

Output Parameters:

.retvalue:	The matching server name is returned in this parameter.
.error:	An error is returned in the (optional) error parameter: [error format]: error code^error text. Possible errors: • 186008^*description*—Invalid Server Lookup Key • 186009^*description*—Server Lookup Key Missing Association

REF: For a list of HWSC error codes, see Table 6.

Output:

returns:	Returns: • **Success:** IEN of new or existing entry (always > 0). • **Failure:** 0.

5.6.2.1 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!

This section describes the HWSC APIs for developer Test account use only.

5.7.1 $$DISPSRVS^XOBWLIB: Display Server List to Screen

Reference Type: Developer Test Account Use Only!

Category: HWSC

IA #: N/A

Description: This extrinsic function displays, on the current device, a list of all entries defined in the WEB SERVER file (#18.12).

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

Format: `$$DISPSRVS^XOBWLIB`

Input Parameters: none.

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

Web server name, IP Address:Port

5.7.2 $$GETSRV^XOBWLIB: Prompt User to Select Server from List

Reference Type: Developer Test Account Use Only!

Category: HWSC

IA #: N/A

Description: This extrinsic function is interactive and allows the end-user to select an entry in the WEB SERVER file (#18.12).

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.

Format: `$$GETSRV^XOBWLIB`

Input Parameters: none.

Output: returns: Returns:

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

5.7.3 $$SELSRV^XOBWLIB: Display Server List to Screen/Prompt for Selection

Reference Type: Developer Test Account Use Only!

Category: HWSC

IA #: N/A

Description: This extrinsic function provides an interactive display of the WEB SERVER, returning user-selected Web server (combination of $$DISPSRVS^XOBWLIB: Display Server List to Screen and $$GETSRV^XOBWLIB: Prompt User to Select Server from List APIs).

Format: `$$SELSRV^XOBWLIB`

Input Parameters: none.

Output: returns: Returns:

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

6 Appendix A—HWSC Error Codes

Error code entries are contained in the DIALOG file (#.84). Table 6 lists the entries used in HealtheVet Web Services Client (HWSC):

Table 6: HWSC APIs Error Codes

Dialog Number / Error Code	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

Table 7: Glossary

Term	Description
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 (CA)	"A certificate authority (CA) is an entity that creates and then "signs" a document or file containing the name of a user and the user's 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 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	HealtheVet Web Services Client is a support framework that offers VistA M applications real-time, synchronous client access to n-tier (J2EE) Web

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

Term	Description
	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 a CA Intermediate Certificate. The CA Intermediate Certificate is used to sign the peer's (server) certificate. This provides another level of validation-managed Public Key Interface (PKI) for Secure Socket Layer (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—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 resides on a smart card.
Public Certificate	This is the certificate that contains the user's public key. This certificate resides 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

[3] DEA website (http://www.deadiversion.usdoj.gov/ecomm/csos/archive/conops.pdf): "Public Key Infrastructure Analysis Concept of Operations," Section 3.3 "Security."

Term	Description
	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 (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 have the same content when signed by VeriSign; the issuer has 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 Socket 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	An Oracle® product; WebLogic Server 8.1, 9.2, and 10.x 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,

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

Term	Description
	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 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]

REF: For a list of commonly used terms and definitions, see the OI&T Master Glossary VA Intranet Website.

For a list of commonly used acronyms, see the VA Acronym Lookup Intranet Website.

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