Sybase web.sql™ Programmer's Guide

Sybase web.sql 1.2
Document ID: 35725-01-0120-01
Last Revised:August 20, 1997

Principal author: W.I.R.E.D. Technical Publications

Document ID: 35725-01-0120

This publication pertains to Sybase web.sql 1.2 of the Sybase database management software and to any subsequent version until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

Document Orders

To order additional documents, U.S. and Canadian customers should call Customer Fulfillment at (800) 685-8225, fax (617) 229-9845.

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the above fax number. All other international customers should contact their Sybase subsidiary or local distributor.

Upgrades are provided only at regularly scheduled software release dates.

Copyright © 1989–1997 by Sybase, Inc. All rights reserved.

No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

Sybase Trademarks

Sybase, the Sybase logo, APT-FORMS, Certified SYBASE Professional, Data Workbench, First Impression, InfoMaker, PowerBuilder, Powersoft, Replication Server, S-Designor, SQL Advantage, SQL Debug, SQL SMART, SQL Solutions, Transact-SQL, VisualWriter, and VQL are registered trademarks of Sybase, Inc. Adaptable Windowing Environment, Adaptive Component Architecture, Adaptive Server, Adaptive Server Monitor, ADA Workbench, AnswerBase, Application Manager, AppModeler, APT-Build, APT-Edit, APT-Execute, APT-Library, APT-Translator, APT Workbench, Backup Server, BayCam, Bit-Wise, ClearConnect, Client-Library, Client Services, CodeBank, Column Design, Connection Manager, DataArchitect, Database Analyzer, DataExpress, Data Pipeline, DataWindow, DB-Library, dbQ, Developers Workbench, DirectConnect, Distribution Agent, Distribution Director, Dynamo, Embedded SQL, EMS, Enterprise Client/Server, Enterprise Connect, Enterprise Manager, Enterprise SQL Server Manager, Enterprise Work Architecture, Enterprise Work Designer, Enterprise Work Modeler, EWA, Formula One, Gateway Manager, GeoPoint, ImpactNow, InformationConnect, InstaHelp, InternetBuilder, iScript, Jaguar CTS, jConnect for JDBC, KnowledgeBase, Logical Memory Manager, MainframeConnect, Maintenance Express, MAP, MDI Access Server, MDI Database Gateway, media.splash, MetaWorks, MethodSet, Net-Gateway, NetImpact, Net-Library, ObjectConnect, ObjectCycle, OmniConnect, OmniSQL Access Module, OmniSQL Toolkit, Open Client, Open ClientConnect, Open Client/Server, Open Client/Server Interfaces, Open Gateway, Open Server, Open

ServerConnect, Open Solutions, Optima++, PB-Gen, PC APT-Execute, PC DB-Net, PC Net Library, Power++, Power AMC, PowerBuilt, PowerBuilt with PowerBuilder, PowerDesigner, Power J, PowerScript, PowerSite, PowerSocket, Powersoft Portfolio, Power Through Knowledge, PowerWare Desktop, PowerWare Enterprise, ProcessAnalyst, Quickstart Datamart, Replication Agent, Replication Driver, Replication Server Manager, Report-Execute, Report Workbench, Resource Manager, RW-DisplayLib, RW-Library, SAFE, SDF, Secure SQL Server, Secure SQL Toolset, Security Guardian, SKILS, smart.partners, smart.parts, smart.script, SQL Anywhere, SQL Central, SQL Code Checker, SQL Edit, SQL Edit/TPU, SQL Modeler, SQL Remote, SQL Server, SQL Server/CFT, SQL Server/DBM, SQL Server Manager, SQL Server SNMP SubAgent, SQL Station, SQL Toolset, Sybase Client/Server Interfaces, Sybase Development Framework, Sybase Gateways, Sybase IQ, Sybase MPP, Sybase SQL Desktop, Sybase SQL Lifecycle, Sybase SQL Workgroup, Sybase Synergy Program, Sybase Virtual Server Architecture, Sybase User Workbench, SybaseWare, SyBooks, System 10, System 11, the System XI logo, SystemTools, Tabular Data Stream, The Architecture for Change, The Enterprise Client/Server Company, The Model for Client/Server Solutions, The Online Information Center, Translation Toolkit, Turning Imagination Into Reality, Unibom, Unilib, Uninull, Unisep, Unistring, Viewer, Visual Components, VisualSpeller, WarehouseArchitect, WarehouseNow, Warehouse WORKS, Watcom, Watcom SQL, Watcom SQL Server, Web.SQL, WebSights, WebViewer, WorkGroup SQL Server, XA-Library, and XA-Server are trademarks of Sybase, Inc. 6/97

All other company and product names used herein may be trademarks or registered trademarks of their respective companies.

Restricted Rights

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

Sybase, Inc., 6475 Christie Avenue, Emeryville, CA 94608.

Table of Contents

About This Book

Audience . . . xiii
How to Use This Book . . . xiii
Related Documents . . . xiii
Other web.sql Documentation . . . xiv
HTML Documentation. . . . xiv
Web Site Administration Documentation . . . xv
Transact-SQL Documentation. . . . xv
Perl Documentation . . . xv
Client-Library Programming Documentation . . . xvi
Conventions . . . xvi

1. web.sql Overview

What Is web.sql? . . . 1-1
Overview of the Components in web.sql . . . 1-1
HyperText Sybase (HTS) File Format . . . 1-3
web.sql Convenience and Client-Library APIs . . . 1-6
Accessing HTS Files . . . 1-6

2. Using SQL in HTS Files

Introduction to Using SQL in HTS Files . . . 2-1
Querying a Database Using SQL . . . 2-2
Performing Other Database Actions Using SQL. . . . 2-3
Using Perl Variables and Form Data in HTS Files . . . 2-4
Using Perl Variables in HTML and SQL Blocks . . . 2-5
Accessing HTML Form Data in HTS Files . . . 2-6

3. Using Perl in HTS Files

Introduction to Using Perl Scripts in HTS Files . . . 3-1
Using Perl Variables and Form Data in HTS Files . . . 3-2
web.sql Convenience and Client-Library APIs . . . 3-2
Using the web.sql Convenience API . . . 3-2
Connecting to a SQL Server with the web.sql Convenience API . . . 3-3
Executing Database Commands with the web.sql Convenience API 3-4

Formatting and Printing Output with the web.sql Convenience API . . 3-6
Handling Errors with the web.sql Convenience API 3-7
Executing Remote Procedure Calls with the web.sql Convenience API 3-7
Using the web.sql Client-Library API 3-10
Connecting to a Server with the web.sql Client-Library API 3-12
Determining Whether To Use ct_connect or ws_connect 3-13
Executing SQL Commands Using the web.sql Client-Library API. . . . 3-14
Sending Database Commands Using the web.sql Client-Library API. . 3-15
Executing Remote Procedure Calls with the web.sql Client-Library API. 3-16
Processing Output Parameters from ct_rpc 3-18
Identifying the Result Set Type with the web.sql Client-Library API . . 3-20
Retrieving Information About Return Data with the web.sql Client-Library API. 3-22
Processing Query Results with the web.sql Client-Library API . . . 3-23
Processing Other SQL Command Results with the web.sql Client-Library API. 3-25
Processing Stored Procedure Results with the web.sql Client-Library API. 3-27
Processing Message Results with the web.sql Client-Library API . 3-28
Canceling Results with the web.sql Client-Library API 3-29
Setting Server Options with the web.sql Client-Library API. 3-29
Handling Errors with Callback Routines in the web.sql Client-Library API 3-30
Returning Non-HTML Data with web.sql 3-33
Using web.sql Features in the Perl File 3-34
Examples of Returning Data Using an HTS File and a Perl Script 3-35
Returning HTTP Header Information 3-36
Example of Setting the Cookie Value. 3-37
Example of URL Redirection. 3-37

4. Sybase web.sql Examples

Transact SQL Statements Within an HTS File 4-1
Examples of SQL in an HTS File. 4-1
A SQL Select Statement. 4-1
SQL Transaction Statements. 4-2
Perl Scripts Within an HTS File 4-2
Examples of Perl Scripts in an HTS File 4-2

Executing SQL Query Statements 4-3
Executing SQL Transaction Statements 4-3
Parsing Form Data 4-5
Using Perl Variables in Form Data 4-6
Accessing Version Information 4-6
Handling Error Messages 4-7
Processing a Perl Script 4-10
Handling Server Results 4-12

A. web.sql Reference Pages

ct_callback A-2
ct_cancel A-5
ct_col_names A-7
ct_col_types A-8
ct_connect A-11
ct_fetch A-14
ct_fetch_parameters A-18
ct_options A-22
ct_res_info A-26
ct_results A-30
ct_rpc A-36
ct_sql A-41
ws_connect A-43
ws_content_type A-45
ws_error A-46
ws_fetch_rows A-47
ws_print A-50
ws_rpc A-51
ws_sql A-54

B. RPC Datatype Summary

Index

List of Figures

Figure 1-1: Typical Web Server Architecture 1-2
Figure 1-2: Web Server Architecture with web.sql 1-2
Figure 2-1: Example Table Produced by a SQL select Statement in an HTS File.......................2-2

List of Tables

Table 2-1: Syntax for Including Variables in HTS Files 2-5
Table 2-2: Predefined Variables for Accessing HTML Form Data 2-6
Table 3-1: web.sql Convenience API Summary 3-3
Table 3-2: web.sql Client-Library API Summary 3-11
Table 3-3: Values of ct_results result type codes 3-20
Table A-1: Values for *$type* (ct_callback) A-2
Table A-2: Values *for $type* (ct_cancel) A-5
Table A-3: Return values of ct_cancel A-5
Table A-4: Values for *$doassoc* (ct_col_types) A-8
Table A-5: Return values of ct_col_types A-8
Table A-6: Column type values of ct_col_types A-9
Table A-7: Values for *$doassoc* (ct_fetch) A-14
Table A-8: Return values of ct_fetch A-14
Table A-9: Values for *$action* (ct_options) A-22
Table A-10: Summary of parameters (ct_options) A-23
Table A-11: Return values of ct_options A-25
Table A-12: Summary of parameters for ct_res_info A-26
Table A-13: Return values of ct_res_info A-27
Table A-14: Values for *$result_type* (ct_results) A-30
Table A-15: Return Values of ct_results A-31
Table A-16: Values for *$result_type* (ct_results loop) A-33
Table A-17: Return Values of ct_sql A-41
Table B-1: Summary of CS_ Datatypes 1

About This Book

This guide describes how to use Sybase web.sql™ to provide access to your organization's databases from World Wide Web browsers.

Audience

This guide is designed primarily for Web site developers or database developers who want to create Web pages for accessing databases. A secondary audience is Web page authors who want a simple method for incorporating Perl scripts into their Web pages.

This guide assumes that you are familiar with HTML authoring, Perl scripting, and database access with Transact-SQL® or the Open Client™ Client-Library. "Related Documents" lists references for these topics.

How to Use This Book

Chapter 1, "web.sql Overview," discusses how web.sql works with a Web server to allow Web browsers to interact with a SQL database server.

Chapter 2, "Using SQL in HTS Files," describes how to use Transact-SQL® for database interactions in your web.sql files.

Chapter 3, "Using Perl in HTS Files," describes how to include Perl scripts in your web.sql files and use the web.sql application programming interface (API) to perform database access.

Chapter 4, "Sybase web.sql Examples," provides several SQL and Perl examples, which show you the capabilities of web.sql and help you determine when to use SQL to access a database and when to use Perl scripts.

Appendix A, "web.sql Reference Pages," contains a full description of the web.sql APIs.

Related Documents

This section lists other documentation that you might find useful for background information on the topics discussed in this book.

Other web.sql Documentation

To learn how to install and configure web.sql on your server, read the *Sybase web.sql Installation and Administration Guide*, which is included in your web.sql distribution from Sybase.

For a description of known problems and work-arounds, documentation updates and clarifications, and other release-specific information, read the *Sybase web.sql Release Notes*, which is included in your web.sql distribution from Sybase.

For a list of frequently-asked questions (FAQ), use the most current version of the FAQ on the Sybase Web site:

```
http://www.sybase.com/products/internet/websql/faq.html
```

E-mail your technical questions, feedback, or comments to websql-help@sybase.com. Please include the platform and version of your Web server and the version and implementation (CGI or NSAPI) of web.sql that you are using.

HTML Documentation

To create web.sql files, you must be familiar with creating standard Web pages in HTML. You can learn HTML from many different books, including:

- *The HTML Sourcebook*, by Ian S. Graham. This book covers a wide range of material suitable for novice as well as advanced users including basic document and text formatting tags; linking documents using URLs; using CGI scripts for interactive pages; and using the wide variety of tools available for creating Web pages and maintaining a Web site. It also contains a complete reference of HTML, including the latest Netscape and Microsoft extensions. Published by John Wiley & Sons, 2nd edition, ISBN 0-471-14242-5.
- *Teach Yourself Web Publishing with HTML 3.0 in a Week*, by Laura Lemay. Topics include background information; presentation and page design; basic, linking, formatting, media and form tags; examples; World Wide Web servers; gateway scripts; and HTML tools. Appendixes list further resources and a summary of commands. Sams Publishing, 2nd edition, ISBN 0-57521-064-9.

Several Web sites include HTML references and tutorials. You can use any of the Web search sites, such as *http://www.yahoo.com,* to obtain lists of such sites.

Web Site Administration Documentation

If you are responsible for installing and configuring web.sql—especially if you are responsible for your Web server's security—you should know how to administer your Web site. You should first read the product documentation supplied with your Web server.

The following books also provide information about Web site administration:

- *How to Set Up and Maintain a World Wide Web Site*, by Lincoln D. Stein. Published by Addison-Wesley Publishing, ISBN 0-20163389-2.
- *Running a Perfect Web Site*, by David M. Chandler, Bill Kirkner, and Jim Minatel. Published by Que Corporation, ISBN 0-7897-0210-X.

Transact-SQL Documentation

You should also be familiar with Transact-SQL if you plan to use it for database interaction in your HTS files. For more information on Transact-SQL, refer to the following Sybase books shipped with the Sybase SQL Server™ product:

- *Transact-SQL User's Guide* documents Transact-SQL®, a Sybase-enhanced version of the SQL relational database language. The *Transact-SQL User's Guide* can help both beginners and those who have experience with other implementations of SQL.
- *SQL Server Reference Manual* contains detailed information on all of the Transact-SQL commands and system procedures.

Perl Documentation

This guide assumes that you are already familiar with Perl 5 if you plan to include Perl scripts in your web.sql files. You can learn more about Perl 5 from the following references:

- *Programming Perl*, by Larry Wall, Randal L. Schwartz, and Stephen Potter. Co-authored by the creator of Perl, this book is regarded as the authoritative guide to Perl programming. The second edition contains information about Perl 5. Published by O'Reilly and Associates, ISBN 1-565921-49-6.

- *Learning Perl* is a step-by-step, hands-on tutorial, but the current edition covers only Perl 4. Written by Randal L. Schwartz. Published by O'Reilly and Associates, ISBN 1-56592-042-2.
- *Teach Yourself Perl 5 in 21 Days*, by Dave Till. Published by Macmillan Computer Publishing, 2nd Edition, ISBN 0-672-30894-0.
- *Perl by Example*, by Ellie Quigley. Published by Prentice Hall, ISBN 0-13-122839-0.

Several Web sites include Perl references and tutorials. You can use any of the Web search sites, such as *http://www.yahoo.com*, to obtain lists of such sites.

Client-Library Programming Documentation

web.sql allows you to interact with databases using Perl scripts by providing a subset of the Open Client Client-Library. Although this guide describes how to use the subset provided, you can read more about the library in the following Sybase books on Client-Library programming:

- *Open Client Client-Library/C Programmer's Guide* contains information on how to write applications using Open Client Client-Library.
- *Open Client Client-Library/C Reference Manual* contains Client-Library reference pages and information on how to accomplish specific programming tasks.

Conventions

This guide uses the following typographical conventions:

- Book titles appear in italics. Example: *Sybase web.sql Installation and Administration Guide*
- Code examples appear in a fixed-space font. Example:

```
$fee += 0.01 * ($shares - 1000);
```

- Text that you should type appears in a bold, fixed-space font. Example:

`vi my_example.hts`

- Program output appears in a fixed-space font. Example:

```
Total amount owed is: $57.29
```

- SQL commands and Perl functions appear in boldface in text. Example: "the **select** statement"
- HTML tags appear in boldface in text. Example: "the **<SYB>** tag"
- File and directory names appear in italics. Example: "the file *my_example.hts*"
- Executable files appear in boldface in text. Example: "Execute **setup** to install and configure web.sql."
- Environment variables appear entirely in upper case. Example: "the SYBASE environment variable"
- Program variables and function parameters appear in italics in text. Example: "to calculate the value of *$foo*".

1 web.sql Overview

This chapter describes how web.sql interacts with Web servers and SQL Servers to make data in your database available on the World Wide Web. This chapter includes the following sections:

- Overview of the Components in web.sql 1-1
- HyperText Sybase (HTS) File Format 1-3
- Accessing HTS Files 1-6

What Is web.sql?

Sybase web.sql provides easy access to relational databases from the World Wide Web and allows you to dynamically generate customized Hypertext Markup Language (HTML) documents.

With web.sql extending your Web server, you can insert database instructions such as SQL statements and Perl scripts into the text of HTML pages. When a client browser requests these pages, web.sql runs the scripts and the resulting output is interpolated into the file.

web.sql recognizes two different file extensions: *.hts* and *.pl*. When your files have an *.hts* file extension, the client browser receives only pure HTML output, since the extensions supported by web.sql are processed on the server side. However, if you want the client browser to receive other document types, such as *.gif* files, then you specify the content type of the document in the *.pl* file before sending the data to the browser.

Overview of the Components in web.sql

Figure 1-1 shows the typical Web server architecture. The Web browser requests pages from the Web server by specifying a Universal Resource Locator (URL). The HTTP server translates this URL into a pathname for a file on the server's host machine. If the file is an *.html*, *.gif*, *.jpeg*, or another file type that the web server understands, the HTTP server sends the file directly to the browser. If the file is a program residing in an authorized directory, the HTTP server executes the program according to the Common Gateway Interface (CGI) and sends the output of the program to the Web browser.

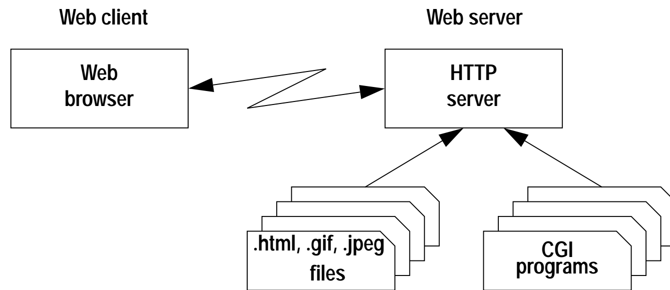

Figure 1-1: Typical Web Server Architecture

Figure 1-2 shows the architecture of a Web server with web.sql. The Web server handles requests for simple HTML files just as it did in Figure 1-1. However, when a browser requests a URL that translates to an *.hts* (HyperText Sybase) file or a *.pl* (Perl) file (that is, a file with only Perl statements and no HTML), the HTTP server passes the request to the web.sql program. The web.sql program reads the specified HTS file, processes the database and Perl requests contained in that file, and then composes HTML output for the HTTP server to pass to the browser. See the *Sybase web.sql Installation and Administration Guide* for more information about the database connections available to the *.hts* and *.pl* files.

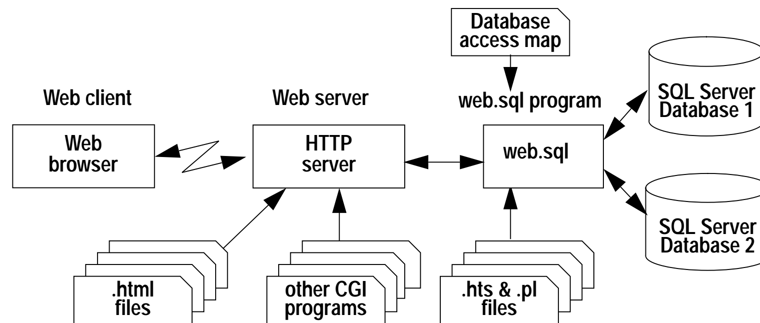

Figure 1-2: Web Server Architecture with web.sql

➤ ***Note***

The Web browser accesses HTS files just as it would any other Web page and receives data in HTML. The browser requires no special extensions or helper applications.

The web.sql program can be used to return dynamic HTML pages to the browser with the help of the HTS file. web.sql can run a Perl script that returns text, pictures, sounds, movies, or other data that the Web browser can display or pass to a helper application. The Perl script can interact with a database just as an HTS file can. For example, you could use a Perl script to retrieve a picture from a database.

There are two versions of the web.sql program available: CGI and NSAPI (Netscape Server Application Programming Interface). In the CGI version web.sql runs as a CGI program. The HTTP server runs the program every time it receives a request for an HTS file.

On UNIX platforms, the NSAPI version of the web.sql program is linked directly into the Netscape HTTP server, which eliminates the overhead that results from starting the web.sql processor for every HTS or Perl request. This high-performance version also allows the web.sql program to cache database connections, further reducing overhead.

➤ ***Note***

The connection-caching feature is not available for the NT version of web.sql.

For more information about the differences between the Netscape Server API and the Common Gateway Interface, see the information provided by Netscape at:

http://home.netscape.com/newsref/std/nsapi_vs_cgi.html

HyperText Sybase (HTS) File Format

The HTS file format is an extension of the standard HTML format. You can include anything in an HTS file that you could in an HTML file. You can also include HTML extensions, such as Java or JavaScript (also known as LiveScript) tags. The web.sql program ignores these tags and passes them to the HTTP server.

HTS files support all HTML 3.0 tags and one additional tag, **<SYB>**, to provide for the additional web.sql functionality. The web.sql program interprets all lines between a **<SYB>** tag and its corresponding **</SYB>** tag as either Transact-SQL statements or Perl code. It inserts the output of those statements into the HTML stream that it sends to the HTTP server.

The **<SYB>** tag has one optional attribute, TYPE. Including "TYPE=SQL" indicates that the **<SYB>** block contains only Transact-SQL statements, whereas "TYPE=PERL" indicates that the **<SYB>** block contains Perl statements. (The Perl statements may contain function calls that execute Transact-SQL statements.) If you do not include a TYPE attribute, the web.sql processor assumes that the **<SYB>** block contains Perl statements.

➤ ***Note***

As with other HTML tags, the <SYB> tag and its attribute are not case-sensitive. You can use upper-, lower-, or mixed case.

The HTS file format returns only HTML content-type output. However, if you want to return output other than the HTML content-type, you need to use a Perl file (*.pl*) and specify the content type using **ws_content_type()**. For example, in a Perl file called *foo.pl*, you could specify the file content as `"image/gif"` then print the file:

```
ws_content_type("image/gif")

print 'cat foo.gif';
```

The following listing shows a simple example of an HTS file. When a browser requests this file, the web.sql processor executes the lines appearing between the **<SYB>** and **</SYB>** tags, which fetch the current time and date on the server and print the results to standard output along with some HTML tags. The web.sql program executes the code within the **<SYB>** blocks and inserts the output into the HTML stream that it sends to the HTTP server.

```
<HTML>

<HEAD>

<TITLE>Sybase web.sql Perl Example</TITLE>

</HEAD>

<BODY>

<H2><CENTER> Perl Example</CENTER></H2>
```

```
This example shows how to include Perl constructs in an HTML
document with Sybase web.sql. The example prints out the date
and time on the machine by having Perl call a function that
returns this information.
<P><HR>

<SYB TYPE=PERL>
   require "ctime.pl";
   print "<P>";
   print "Date: ", "<STRONG>", &ctime(time), "</STRONG>";
   print "<P>";
</SYB>
</BODY>
</HTML>
```

The resulting HTML file sent to the browser is:

```
<HTML>
<HEAD>
<TITLE>Sybase web.sql  Perl Example</TITLE>
</HEAD>
<BODY>
<H2><CENTER> Perl Example</CENTER></H2>

This example shows how to include Perl constructs in an HTML
document with Sybase web.sql. The example prints out the date
and time on the machine by having Perl call a function that
returns this information.
<P><HR>
Date: <STRONG> Thu Feb 29 11:47:08 US/Pacific 1996 </STRONG>
<P>
<HR>
</BODY>
</HTML>
```

web.sql Convenience and Client-Library APIs

web.sql includes two Perl application programming interfaces (APIs) for database access: the web.sql "convenience" API and the web.sql Client-Library API. If you are new to Sybase Open Client, you may want to use the web.sql convenience API; it requires less programming. See "web.sql Convenience and Client-Library APIs" in Chapter 3 for further details.

Accessing HTS Files

You access HTS files through a Web browser using URLs, just as you would any other Web files. Open the web.sql welcome page by using one of the following platform-dependent URLs:

- If you are using the NSAPI version of web.sql on a UNIX platform:

  http://*<servername>*/*<websql.dir>*/welcome.hts

  <servername> is the IP address (hostname) and the port number of the Web server, *<websql.dir>* is the path to the web.sql subdirectory in the Web server's document root directory, and *welcome.hts* is the name of the HTS file that welcomes you to Sybase web.sql.

- If you are using the static (high-performance) CGI version of web.sql on a UNIX platform:

  http://*<servername>*/*<cgi-name>*/ws.exe/*<websql.dir>*/welcome.hts

  <servername> is the IP address (hostname) and the port number of the Web server, *<cgi-name>* is the name of your Web server's CGI script directory, **ws.exe** is the name of the web.sql CGI program, *<websql.dir>* is the path to the web.sql subdirectory in the Web server's document root directory, and *welcome.hts* is the name of the HTS file that welcomes you to Sybase web.sql.

- If you are using the shared-library CGI version of web.sql on a UNIX platform:

  http://*<servername>*/*<cgi-name>*/websql/*<websql.dir>*/welcome.hts

  <servername> is the IP address (hostname) and the port number of the Web server, *<cgi-name>* is the name of your Web server's CGI script directory, **websql** is the name of the web.sql CGI program, *<websql.dir>* is the path to the web.sql subdirectory in the Web server's document root directory, and *welcome.hts* is the name of the HTS file that welcomes you to Sybase web.sql.

- If you are using the NSAPI version of web.sql on the NT platform:

  http://*<servername>*/*<websql>*/welcome.hts

  <servername> is the IP address (hostname) and the port number of the Web server, *<websql>* is the path to the web.sql subdirectory in the Web server's document root directory, and *welcome.hts* is the name of the HTS file that welcomes you to Sybase web.sql.

- If you are using the CGI version of web.sql on the NT platform:

  http://*<servername>*/*<cgi-name>*/ws.exe/*<websql>*/welcome.hts

  <servername> is the IP address (hostname) and the port number of the Web server, *<cgi-name>* is the name of your Web server's CGI script directory, **ws.exe** is the name of the web.sql CGI program, *<websql>* is the path to the web.sql subdirectory in the Web server's document root directory, and *welcome.hts* is the name of the HTS file that welcomes you to Sybase web.sql.

➤ ***Note***

After you specify the name of the web.sql CGI program in web.sql CGI URLs, the document root information and HTS file name are passed to the web.sql CGI program.

2 Using SQL in HTS Files

This chapter describes how to include Transact-SQL statements in a HyperText Sybase (HTS) file. Topics discussed include:

- Introduction to Using SQL in HTS Files 2-1
- Querying a Database Using SQL 2-2
- Performing Other Database Actions Using SQL 2-3
- Using Perl Variables and Form Data in HTS Files 2-4

Introduction to Using SQL in HTS Files

As discussed in “HyperText Sybase (HTS) File Format” on page 1-3, you can include Transact-SQL statements in an HTS file by enclosing them between a set of **<SYB>** and **</SYB>** tags. To identify the block as Transact-SQL statements, the opening **<SYB>** tag must include the attribute `TYPE=SQL`. The example below, which assumes that a database connection to the *pubs2* database has been defined, shows how to include a simple **select** statement in an HTS file:

```
<SYB TYPE=SQL>
    select * from publishers
</SYB>
```

➤ ***Note***

The *pubs2* database is included with your SQL Server. See the *Configuring and Administering Sybase SQL Server* manual for information about installing or locating your *pubs2* database.

In this example, when a client browser requests the HTS file containing this code, the web.sql program sends the request to a SQL Server. The SQL Server evaluates the **select** statement and replaces the **<SYB>** block with the **select** output—in HTML format—in the HTML stream that it sends to the HTTP server. The client browser never sees the SQL statement that produced the output.

You can include multiple SQL statements in a single **<SYB>** block. You can also include multiple **<SYB>** blocks in a single HTS file, and even intersperse Perl **<SYB>** blocks with SQL **<SYB>** blocks. However, you cannot include both SQL statements and Perl statements within the same **<SYB>** block (except when using Perl variables in a SQL block as

discussed in “Using Perl Variables and Form Data in HTS Files” on page 2-4).

Querying a Database Using SQL

From within an HTS file you can use Transact-SQL statements to perform database queries. web.sql converts the results into a table in HTML 3.0 format.

The following examples use the *pubs2* database. For these examples to work, you must associate the HTS files containing the examples with a connection to the *pubs2* database. See Chapter 4 of the *Sybase web.sql Installation and Administration Guide* for information about establishing database connections and associations.

For example, the following simple **select** statement returns an HTML table containing the results of the query:

```
<SYB TYPE=SQL>
    select title, price
    from pubs2..titles
    where type = "business"
</SYB>
```

The results are formatted into an HTML table as shown in Figure 2-1.

title	price
The Busy Executive's Database Guide	19.99
Cooking with Computers: Surreptitious Balance Sheets	11.95
You Can Combat Computer Stress!	2.99
Straight Talk About Computers	19.99

Figure 2-1: Example Table Produced by a SQL select Statement in an HTS File

A query returns an unlimited number of rows. To limit the number of rows returned as a result of a query, you call the **ct_options** routine with the appropriate settings from a Perl **<SYB>** block preceding your SQL **<SYB>** block. For example, to set the maximum number of rows to 200:

```
<SYB TYPE=PERL>
  ct_options($ws_db, CS_SET, CS_OPT_ROWCOUNT, 200, CS_INT_TYPE);
```

```
</SYB>
<SYB TYPE=SQL>
  select * from hugetable
</SYB>
```

See “Setting Server Options with the web.sql Client-Library API” on page 3-29 for more information about **ct_options**.

Performing Other Database Actions Using SQL

In addition to using the **select** command to query a database, you can use any other SQL command in an HTS file. As long as the database user for the HTS file (as specified in the web.sql database access map) has proper permission, the user can insert, delete, and modify rows, create tables, execute stored procedures, and so on.

➤ ***Note***

If the SQL commands you execute in a **<SYB>** block do not produce row output, then the **<SYB>** block produces no output in the resulting HTML stream that web.sql sends to the HTTP server. In particular, web.sql does not print the status of commands such as **insert**, **update**, or **delete**.

For example, the following **<SYB>** block inserts a row into the ***publishers*** table of the ***pubs2*** database:

```
<SYB TYPE=SQL>
    insert into pubs2..publishers
    values ("1622", "Jardin, Inc.", "Camden", "NJ")
</SYB>
```

As another example, the following block increases the price of all titles in the ***pubs2*** database by 10%:

```
<SYB TYPE=SQL>
    update pubs2..titles
    set price = price * 1.1
</SYB>
```

Note that the above examples are “hard-coded,” that is, insertions and updates are fixed strings. In most cases, you will want to allow users to enter information in a form and then use those values for your insertions and updates. You can do so easily with the

predefined Perl variables supported by web.sql. See “Accessing HTML Form Data in HTS Files” on page 2-6 for details on how to do this.

When using the **use** statement, as in the example below, you must place the **use** statement in a separate **<SYB TYPE=SQL>** block preceding the SQL statements you want it to affect. The database you specify in the **use** statement is then in effect until the end of the HTS file or until the next **use** statement.

The following example includes a **use** statement so that it properly affects the SQL statements following it:

```
<HTML>
<HEAD>
<TITLE>Display All Titles</TITLE>
</HEAD>
<BODY>
<H1>Display All Titles</H1>
<SYB TYPE=SQL>
    use pubs2
</SYB>
<SYB TYPE=SQL>
    select * from titles
</SYB>
</BODY>
</HTML>
```

➤ ***Note***

In a SQL **<SYB>** block, a **use** statement has **no effect** on SQL statements following it within the same **ws_sql** call. To ensure that the **use** statement works properly, place the **use** statement in a separate **ws_sql** call or **ct_sql** call.

Using Perl Variables and Form Data in HTS Files

web.sql allows you to define variables within a Perl **<SYB>** block and then use the value of those variables anywhere in your file. Also,

web.sql automatically defines and sets some variables based on any form data passed to the HTS file.

This section describes how to use Perl variables in your HTS file.

Using Perl Variables in HTML and SQL Blocks

In an HTS file, web.sql interprets any string preceded by a dollar sign ($) as a global Perl variable, even if that string appears outside of a Perl **<SYB>** block. You can assign values to variables in a Perl **<SYB>** block and then use them later in your file. (See Chapter 3, "Using Perl in HTS Files," for more information about Perl **<SYB>** blocks.)

Table 2-1 describes the syntax for accessing Perl variables from "plain" HTML and SQL **<SYB>** blocks. Note that web.sql supports only scalar values outside of Perl **<SYB>** blocks; you cannot access an entire array at once outside of a Perl **<SYB>** block.

Table 2-1: Syntax for Including Variables in HTS Files

Format	Meaning
$foo or ${foo}	Scalar variable named "foo"
$values[2]	Third scalar element of array named "values" (arrays start with 0)
$assoc{"foo"}	Scalar element named "foo" of associative array named "assoc"

➤ ***Note***

To force web.sql to interpret a dollar sign literally in an HTS file (that is, to prevent it from interpreting the term following the dollar sign as a Perl variable), precede the dollar sign with a backslash character (\). Note that unlike in Perl, variable substitution is not turned off by single-quoting.

The following example uses the value of a Perl variable in the HTML portion of an HTS file. Notice that the variable is assigned within a Perl **<SYB>** block but is referenced directly in the HTML code.

```
<HTML>
<BODY>
```

```
This example shows how to reference a Perl
variable within HTML text in Sybase web.sql.  Any
Perl variable may be referenced in HTML text in an
HTS (web.sql) file, and the variable's value will
be substituted into the text before the document
is sent to the browser.
<P><HR>

<SYB TYPE=PERL>
    # Assign the system date/time to a variable
    require "ctime.pl";
    $today = &ctime(time);
</SYB>

<STRONG>Today is: ${today}</STRONG>
<P>
</BODY>
</HTML>
```

Accessing HTML Form Data in HTS Files

web.sql automatically parses HTML form data passed to the HTS file and assigns the values to Perl variables. (web.sql supports both the GET and POST methods for passing HTML form data.) You can then use these variables in your SQL statements. Variables can greatly simplify creating forms for database interaction.

Table 2-2 lists the predefined variables that web.sql provides. The format "$foo" is the simplest for use in a SQL statement; the other formats are more useful in Perl blocks where you can use a loop to process all the array values.

Table 2-2: Predefined Variables for Accessing HTML Form Data

Format	Meaning
$foo	The value of the form input field named "foo" if there is a single form input with this name.
@foo	Array of values of the form input fields named "foo" if there are multiple fields with that name.

Table 2-2: Predefined Variables for Accessing HTML Form Data (continued)

Format	Meaning
%ws_form	Associative array of all form input fields and their associated values.
$ws_form{"foo"}	The value of the form input field named "foo." In the case of multiple input fields with the same **NAME** attribute, the value is the concatenation of all the values, separated by null characters.
%ws_multiple	An associative array indicating multiple form input fields with the same name. If $ws_multiple{"foo"} is true, then @foo is defined, and $ws_form{"foo"} contains the concatenation of all the values for the form input field "foo", separated by null characters. If $ws_multiple{"foo"} is false, then $foo is defined, and "foo" is a single-valued form input field name.
$ws_multiple{"foo"}	1 if more than one field named foo, else 0.

As an example of using the Perl HTML form variables, consider a simple form allowing a user to search for stores by state. The following HTML file, *stores.hts*, displays a text field for the user to enter a state:

```
<HTML>
<BODY>
<H2>Search for a store by state</H2>

<FORM ACTION=stores.hts METHOD=POST>
Enter two-letter state abbreviation: <INPUT NAME=state>
<P>
<INPUT TYPE=SUBMIT VALUE=Search>
</FORM>
</BODY>
</HTML>
```

The ACTION attribute of the **<FORM>** tag causes the browser to send the form results to the server for the URL of *stores.hts* when the user clicks the Search button. The following HTS file, *stores.hts*, could then obtain the state name and use it in a **select** statement as shown below:

```
<HTML>
<BODY>
```

```
<H1>Stores</H1>
Stores found in $state:<P>
<SYB TYPE=SQL>
    select stor_name, stor_address, city, state
    from pubs2..stores
    where state = upper("$state")
    order by stor_name
</SYB>
</BODY>
</HTML>
```

3 Using Perl in HTS Files

This chapter describes how to include Perl scripts in a HyperText Sybase (HTS) file. Topics discussed include:

- Introduction to Using Perl Scripts in HTS Files 3-1
- web.sql Convenience and Client-Library APIs 3-2
- Using the web.sql Convenience API 3-2
- Using the web.sql Client-Library API 3-10
- Returning Non-HTML Data with web.sql 3-33
- Returning HTTP Header Information 3-36

Introduction to Using Perl Scripts in HTS Files

As discussed in "HyperText Sybase (HTS) File Format" on page 1-3, you can include a Perl script in an HTS file by enclosing it between a set of **<SYB>** and **</SYB>** tags. You can indicate that a **<SYB>** block contains a Perl script by including the attribute "TYPE=PERL" in the opening **<SYB>** tag; however, if you omit a TYPE attribute, the web.sql program assumes that the block contains a Perl script. The following example shows you how to include a simple Perl script in an HTS file:

```
<SYB TYPE=PERL>
require "ctime.pl";
print "<P>";
print "Date: ", "<STRONG>", &ctime(time), "</STRONG>\n";
print "<P>";
</SYB>
```

In this example, when a client browser requests the HTS file containing this code, the web.sql program evaluates the Perl script and replaces the **<SYB>** block with the printed output in the HTML stream that it sends to the HTTP server. The client browser never sees the Perl script that produced the output.

Note that in a Perl script, you must print anything that you want to include in the final HTML stream web.sql sends to the client browser. Also note that you can include HTML tags in the Perl output. You can use these tags to format the appearance of your output.

You can include multiple <SYB> blocks in a single HTS file. You can also intersperse SQL <SYB> blocks with Perl <SYB> blocks. You can use Perl variables in a SQL block. However, you cannot include both SQL statements and Perl statements within the same <SYB> block.

Using Perl Variables and Form Data in HTS Files

web.sql allows you to define variables within a Perl <SYB> block and then use the value of those variables afterwards in your file. Also, web.sql automatically defines and sets some variables based on form data passed to the HTS file. For more information about including Perl variables and form data in HTS files, see the section "Using Perl Variables in HTML and SQL Blocks" on page 2-5.

web.sql Convenience and Client-Library APIs

web.sql includes two Perl application programming interfaces (APIs) for database access: the web.sql "convenience" API and the web.sql Client-Library API. If you are new to Sybase Open Client, you may want to use the web.sql convenience API; it requires less programming.

All convenience functions are prefixed with "ws_". All CLient-Library functions are prefixed with "ct_".

Use the web.sql convenience API to automatically generate HTML tables of the data returned by the SQL server. Use the Client-Library API to manipulate the data returned by the server on a row-by-row basis. The following sections describe the web.sql convenience and Client-Library APIs.

Using the web.sql Convenience API

The web.sql convenience API provides a simple set of routines for accomplishing the most common tasks in an HTS file. You can use these routines individually to perform most database interaction, or you can combine them with routines described in "Using the web.sql Client-Library API" on page 3-10.

Table 3-1 lists the routines in the convenience API. The rest of this section describes how to use these routines in a Perl <SYB> block.

Table 3-1: web.sql Convenience API Summary

Function	Description
ws_connect	Connects to a SQL server and returns a connection handle.
ws_content_type	Sets the content type for the data returned by a *.pl* file.
ws_error	Prints an error message and an optional string, and terminates processing of the current page.
ws_fetch_rows	Fetches and prints rows returned by a call to **ct_sql**.
ws_print	Prints a string, expanding Perl variable references.
ws_sql	Executes one or more SQL commands and prints the results.
ws_rpc	Executes a stored and registered procedure, updates arguments, and prints the results.

Connecting to a SQL Server with the web.sql Convenience API

web.sql maintains a database access map (*.websql.pl)*, in which it maps the path of HTS files to default database connections. Each entry in the map specifies the server, login, and password to use for an HTS file or a group of HTS files. (The password is encrypted.)

Your HTS file automatically uses the default database connection specified for it in the access map if you do not specify another connection. (The default database connection is the closest connection to the URL. The connection should be unambiguous.) web.sql stores the "handle" for the default database connection in the Perl variable *$ws_db*. Use this handle as an argument to other web.sql routines to specify the database connection.

You can use **ws_connect** or **ct_connect** to establish a database connection other than the default connection. For information about the differences between **ws_connect** and **ct_connect**, see "Determining Whether To Use ct_connect or ws_connect" on page 3-13.

The connection you establish with **ws_connect** must be valid; valid connections are defined in the web.sql Administration pages and stored in the database access map (*.websql.pl)*. The web.sql database access map can contain several valid database connections for an HTS file and its parent directories.

You can specify as an argument to **ws_connect** the logical name of any of these connections. **ws_connect** returns a connection "handle" that you can use as an argument to other web.sql routines rather than the default *$ws_db* handle.

For more information about establishing database connections, please refer to the *Sybase web.sql Installation and Administration Guide*.

Executing Database Commands with the web.sql Convenience API

The **ws_sql** routine executes a SQL command and prints the output. **ws_sql** accepts three arguments: a database connection handle, a string containing the SQL command, and an optional formatting string:

```
ws_sql($db, $sql [, $format])
```

The following sections describe each argument and how to use it.

Database Handle

Typically, you should use *$ws_db* as the database argument (*$db*) to specify the database for the HTS file. But you can also use other database handles that you obtain with the **ws_connect** routine described in "Connecting to a SQL Server with the web.sql Convenience API" on page 3-3 or the **ct_connect** routine described in "Connecting to a Server with the web.sql Client-Library API" on page 3-12.

SQL Command

The *$sql* argument is a string containing one or more Transact-SQL commands. To include multiple commands in a single string, separate the commands with newline characters (that is, include "\n" between commands in the string).

If one SQL command fails, **ws_sql** prints a warning message.

Format String

The format string determines how **ws_sql** formats the rows it returns. The **ws_sql** format string supports the same features as the Perl **printf** format string. The format string is optional.

If you do not include a format string, **ws_sql** formats the returned results as a table in HTML 3.0 format. If you run a long query (1000 rows or greater) from a database, your Web browser may take a long time to display the data in a table format. The browser displays the

table after it formats all of the data. Instead of returning the data in a table format, you can specify a faster format, such as an HTML paragraph or preformatted text.

If you do include a *$format* string, web.sql uses it to format each row returned. You can specify a scalar string, which is used as the **printf** for each row. You can also specify a function reference, which specifies a function to be called for each result-type. For example,

```
ws_sql($ws_db, "select au_lname, au_fname from authors",
\@process_rows())
```

In the previous example, **process_rows()** is the function called to format the SQL rows returned. Users can do any processing they want within that function.

If you supply a function reference as the format string, you can supply the following arguments to that function:

arg1	= $database_handle: if you want to make another query.
arg2	= $result_type: the type of rows being returned.
arg3 - argN	= row elements of the table.

Example 1. Format with <P>.

The following example shows you how to use **ws_sql** to retrieve and format rows from the *pubs2* database with <P>:

```
<SYB TYPE=PERL>
$sql_stmt = qq!select au_fname, au_lname, city, state
            from pubs2..authors!;
$format = "%s %s lives in %s, %s.<P>\n";
ws_sql($ws_db, $sql_stmt, $format);
</SYB>
```

➤ ***Note***

The “qq!...!” syntax quotes everything between the exclamation points (!).

In the previous Perl **<SYB>** block, the *$sql_stmt* variable contains a **select** statement that selects the first name, last name, city, and state of all authors in the *pubs2.authors* table. The format string of the *$format* variable includes the HTML tag **<P>** so that each row appears as a separate paragraph.

The output appears as:

```
Johnson White lives in Menlo Park, CA.
Marjorie Green lives in Oakland, CA.
Cheryl Carson lives in Berkeley, CA.
[And so on...]
```

Example 2. Format with `<PRE>`.

The following example shows you how to use **ws_sql** to retrieve and format the data from the *pubs2* database with `<PRE>`.

```
<SYB TYPE=PERL>
$sql_stmt = qq!select au_fname, au_lname from pubs2..authors!;
print = "<PRE>\n";
$format = "\t%s \t%s \n";
ws_sql($ws_db, $sql_stmt, $format);
print = "</PRE>\n";
</SYB>
```

The output appears as:

```
Johnson     White
Marjorie    Green
Cheryl      Carson
[And so on...]
```

Formatting and Printing Output with the web.sql Convenience API

The **ws_print** routine behaves like Perl's **print**, except that if the string you print contains a reference to a Perl variable, **ws_print** substitutes the value of that variable before printing the string.

For example, suppose that a form has a field named USER_ID and that the form specifies an HTS file as the URL to call when the user submits the form. Now consider the following lines in the HTS file specified by the form:

```
<SYB TYPE=PERL>
$msg = "Field USER_ID does not have a valid value ($user_id).\n";
ws_print("$msg");
</SYB>
```

If the user enters the text “Bad text” into the USER_ID field, **ws_print** substitutes the string “Bad text” for the variable *$user_id*. The code above prints:

```
Field USER_ID does not have a valid value (Bad text).
```

On the other hand, if you use the Perl **print** routine rather than **ws_print**, there is no substitution, and the output is:

```
Field USER_ID does not have a valid value ($user_id).
```

Handling Errors with the web.sql Convenience API

The **ws_error** routine allows you to exit gracefully when you detect an error in your Perl script. It prints an error message, including an optional string that you provide as an argument, and terminates processing of the current HTS page. For example:

```
if (!$name)
{
    ws_error( "You must specify a name in the search field." );
}
```

Executing Remote Procedure Calls with the web.sql Convenience API

The **ws_rpc** routine enables you to call procedures that reside on a remote server. These calls (known as remote procedure calls or RPCs) can be made from an application or another server. In web.sql, remote procedure calls are made to Sybase Open Server applications and stored procedures in SQL Server. RPCs work similarly to the execute command but include additional advantages:

- An RPC command can be used to execute a SQL Server stored procedure or an Open Server registered procedure. (A registered procedure is a procedure that is defined and installed in a running Open Server application.)
- An RPC command passes the stored procedure’s parameters in their native datatypes, as opposed to the execute command, which passes parameters as ASCII characters. The RPC method is faster and more compact than the execute statement because the RPC application does not convert native datatypes to ASCII equivalents.
- It is simpler and faster to accommodate stored procedure return parameters with an RPC than with an execute command.

With an RPC, the parameter's values are automatically returned to the application. However, calling a stored procedure with an execute command does not return constants. Only parameters that use local variables return values.

The **ws_rpc** routine calls the remote procedures with specified parameters, including output parameters. It accepts four arguments: a database connection handle, the RPC name, the RPC parameter list, and an optional formatting string:

```
ws_rpc($handle, $rpc_name, @params or %params [, $format])
```

Database Handle

The *$handle* argument can be either the default database handle (*$ws_db*) for the HTS file or other database handles that you obtain with the **ws_connect** routine or the **ct_connect** routine.

Remote Procedure Name

The *$rpc_name* argument is the name of the RPC you want to call.

Parameter List

The *%params* or *@params* argument is the list of parameters passed to the stored procedure. Stored procedures can be called with either named or positional parameters. *%params* is used for named arguments, and *@params* is used for positional arguments.

Named parameters pass the parameter name and the value assigned to that name to the stored procedure. Positional parameters pass the argument value only. The argument value in the first position, the first parameter, is passed to the stored procedure, the argument value in the second position is passed as the second parameter, the argument value in the third position is passed as the third parameter, and so on.

If you use named parameters, the list of parameters for the RPC passed to the stored procedure is a Perl 5 reference to an associative array. If you use positional parameters, the list is passed as an array.

You can specify output parameters with a reference to a scalar variable. Scalar data can be passed as an integer, a float, or a character string, depending on the Perl datatype.

To pass or receive a particular Sybase datatype other than integer, you must use a datatype-reference structure. For example, if you want to pass an integer as a character string, you must use a

datatype-reference to enforce the behavior. The datatype-references can be written as follows:

```
{
'type'      =>  CS_<datatype>,
'value'     =>  <scalar> or <scalar-ref>,
'scale'     =>  CS_MAX_SCALE or CS_DEF_SCALE,
'precision' =>  CS_MAX_PREC or CS_DEF_PREC
}
```

For a fuller example using named parameters, refer to "ws_rpc" in Appendix A, "web.sql Reference Pages".

➤ ***Note***

If you expect the output to be numeric, call the RPC with the appropriate output parameters to avoid problems.

For a list of the `CS_` datatypes, see Appendix B, "RPC Datatype Summary."

Format Argument

If you specify a *$format*, web.sql uses it to format each row returned. You can specify a scalar string, which is used as the "printf" for each row, or you can specify a function reference, which specifies a function to be called for each result-type. The arguments you can specify include:

```
arg1        = $database_handle
arg2        = $result_type
arg3- argN  = row elements
```

If you do not specify a *$format* argument, the rows returned by the remote procedure, if any, are formatted in an HTML table.

Example

The following example shows you how to use **ws_rpc** to call a remote procedure called *history_proc*. A named parameter is passed to the stored procedure. Note that a function reference is supplied as the format argument.

```
sub my_fmt
    {
```

```
    my $dbh = shift @_;
    my $restype = shift @_;
    my $i;
    if ($restype == CS_ROW_RESULT)
        {
        if (!defined($my_fmt_once))
            {
            my @cols = ct_col_names($dbh);
            for ($i = 0; $i < scalar(@cols); $i++)
                {
                printf "<td align=center> <b> @cols[$i] </b>\n";
                $my_fmt_once = 1;
                }
--....code deleted....

<SYB TYPE=PERL>
ws_rpc($ws_db,"pubs2..history_proc", ["$stor_id"], \&my_fmt);
</SYB>
```

Using the web.sql Client-Library API

The web.sql Client-Library API enables you to manipulate the data returned by the server on a row-by-row basis. It provides an interface similar to the Sybase Open Client Client-Library (CT Lib) API. This interface is more complex than the convenience API, but gives you more control over your SQL Server interaction.

The web.sql Client-Library API differs from the Open Client API in several ways, making web.sql programming easier and reflecting the differences between programming in C and programming in Perl:

- The web.sql API does not include all of the Open Client Client-Library routines. However, it augments several calls into high-level calls (for example, **ct_sql**).
- The web.sql API includes some routines not in the Open Client Client-Library.

- The calling format for the web.sql API differs from that of the Open Client Client-Library.
- Some web.sql routines do not support all the return or parameter values supported by Open Client Client-Library.

➤ ***Note***

You do not need to be an experienced Open Client Client-Library programmer to use the web.sql Client-Library API. This manual contains all the information you need to use the web.sql Client-Library API. However, if you are an experienced Open Client Client-Library programmer, you should review the reference pages in Appendix A, "web.sql Reference Pages," and pay close attention to the discussions in this section to note the differences between the APIs.

You can use the web.sql Client-Library routines in conjunction with the web.sql convenience API described in "Using the web.sql Convenience API" on page 3-2. For example, you might use **ct_connect** to connect to a database, **ws_error** to handle error conditions, and a combination of web.sql Client-Library routines and **ct_fetch** to fetch a row of result data.

Table 3-2 lists the routines in the web.sql Client-Library API. The rest of this section describes how to use these routines in a Perl **<SYB>** block.

Table 3-2: web.sql Client-Library API Summary

Function	Description
ct_callback	Installs or retrieves a callback routine for handling errors.
ct_cancel	Cancels a command or the results of a command.
ct_col_types	Retrieves an array of column types for the current query results.
ct_col_names	Retrieves an array of column names for the current query results.
ct_connect	Establishes a database connection to a server.
ct_fetch	Fetches a single row of result data.
ct_fetch_parameters	Updates output parameter variables supplied to **ct_rpc**.
ct_options	Sets, retrieves, or clears the values of server query-processing options.

Table 3-2: web.sql Client-Library API Summary (continued)

Function	Description
ct_res_info	Retrieves information about the current result set or command.
ct_results	Determines SQL command status and type of results being returned.
ct_rpc	Calls a stored procedure that resides on a remote server.
ct_sql	Sends one or more SQL commands to the database server.

Connecting to a Server with the web.sql Client-Library API

web.sql maintains a database access map, in *.websql.pl*, which maps the path of HTS files to default database connections. Each entry in the map specifies the server, login, and password to use for an HTS file or a group of HTS files. (The password is encrypted.)

Your HTS file automatically uses the default database connection specified for it in the access map unless you specify another connection. web.sql stores the "handle" for the default database connection in the Perl variable *$ws_db*. You use this handle as an argument to other web.sql routines to specify the database connection.

You can use ct_connect or ws_connect to establish a database connection other than the default connection. For information about the differences between ct_connect and ws_connect, see "Determining Whether To Use ct_connect or ws_connect" on page 3-13.

You specify as arguments to ct_connect the server, login, and password that you want to use for the connection and ct_connect returns a connection "handle" that you can use as an argument to other web.sql routines rather than the default *$ws_db* handle. For example:

```
<SYB TYPE=PERL>
$uid = "userid";
$pwd = "passwd";
$srv = "SYB_INET";
$myHandle = ct_connect($uid, $pwd, $srv);
# You can now use $myHandle as a database handle
</SYB>
```

web.sql automatically closes the connection established with ct_connect after the *.hts* or *.pl* script completes.

Determining Whether To Use ct_connect or ws_connect

You can use one of two methods to define the database connections web.sql needs to make in response to user queries.

- Use ct_connect(dblogin, password, server) in every *.hts* file that needs a database connection.

  If you use ct_connect method, you must write the connection name, login, password, and server name into the top of every *.hts* file, for example:

  ```
  <syb>
  $ws_db = ct_connect("joe_user","my_pw","MY_SERVER")
  </syb>
  ```

  This duplicates code and exposes the password to anyone having direct access to the files at the net site.

  The advantages of using ct_connect rather than ws_connect are:

  - You do not have to set up an additional entry in the database access map for the connection you are making.
  - Connections are automatically closed after the *.hts* or *.pl* script completes.

- Use the web.sql Administration page to define which files or directories in the server's document tree connect to a database. In this preferred method for web.sql, the *.hts* file automatically uses ws_connect(connection name).

  When defining a database connection, you provide the connection name, database server name, login, and password (web.sql encrypts the password) so that only a user with that login and password can access the HTS file associated with the connection name. This information is stored in the web.sql database access map (*.websql.pl*) file.

  The advantages of using ws_connect rather than ct_connect are:

  - The NSAPI UNIX version of web.sql can provide considerably improved performance by caching database connections set up with the Administration page or with ws_connect. (Connection caching is not available for web.sql on Windows NT.)
  - Database login IDs and passwords are isolated in the database access map where the password is encrypted.

- By using **ws_connect** to access logical connection names, you can change connection parameters in the database access map; then all affected HTS files automatically pick up the changes.

See "Connecting to a SQL Server with the web.sql Convenience API" on page 3-3 for more information about using **ws_connect**.

➤ ***Note***

Unlike the Open Client Client-Library, the web.sql Client-Library API automatically sets up all necessary data structures for the connection and maintains this information internally. You do not need to allocate and initialize control or context structures or perform any other initialization. Nor do you have access to any of these structures within an HTS file. Furthermore, you do not need to disconnect from the server at the end of your script.

Executing SQL Commands Using the web.sql Client-Library API

The web.sql Client-Library API provides several routines for sending SQL commands to the database server and then processing the results. The general program flow for database interaction is as follows:

```
use ct_sql to send one or more commands to the database server
while ct_results reports result set data available
{
    for each result set returned
    {
        optionally use ct_col_names to determine column names
        optionally use ct_col_types to determine column types
        while data available to fetch for the current result set
        {
            use ct_fetch to get a row of data
            process data
        }
    }
}
```

You can send one or more SQL commands to the database server with **ct_sql**. In turn, each command can generate one or more result sets. For example, a **select** statement can return several regular row result sets and several compute result sets. Stored procedures can generate status and parameter result sets. Even **insert**, **update**, and **delete** statements generate result sets telling you whether or not the statement executed successfully.

You must code your processing loop to handle all result sets that the server can generate in response to your commands. Table A-14 on page A-30 lists the different types of results that can be returned.

Sending Database Commands Using the web.sql Client-Library API

This section describes how to send SQL commands and then process the returned result sets.

You send commands to the database server using **ct_sql**. You can send multiple commands in a single **ct_sql** call by separating the commands with embedded newline characters ("\n"). You can also include Perl variables in the command string; web.sql automatically evaluates these variables before sending the commands to the database server.

ct_sql returns the value CS_SUCCEED if it can send the command to the server successfully. A return value of CS_SUCCEED does not indicate that the server *processed* the SQL commands without errors.

Using **ct_results** ensures that **ct_sql** returns a value of CS_FAIL if it is unable to send the query to the SQL server, and a value of CS_CANCELED if the command is canceled using **ct_cancel**. See "Identifying the Result Set Type with the web.sql Client-Library API" on page 3-20 for information about **ct_results**. See "Canceling Results with the web.sql Client-Library API" on page 3-29 for information about **ct_cancel**.

The following example demonstrates how to send a set of SQL statements to the server:

```
<SYB TYPE=PERL>
$sql_stmt = qq!select au_fname, au_lname, city, state
            from pubs2..authors
            where state = upper("$state")
            select pub_name, city, state
            from pubs2..publishers
```

```
            where state = upper("$state")!;
if (($rc = ct_sql( $ws_db, $sql_stmt ) ) != CS_SUCCEED)
{
    ws_error("Unable to process database request.");
}
# Process the result sets...
</SYB>
```

The sample code above sends two **select** statements to the database server, one that selects all authors in a given state and one that selects all publishers in a given state. The state is determined by the Perl variable *$state*. The *$state* variable may have been a form value, or it may have been set in a prior **<SYB>** block.

Note that the example checks the **ct_sql** return code and calls **ws_error** if it is not successful, which terminates processing of the current HTS page. After confirming that **ct_sql** returns successfully, you are ready to beginning processing the result sets as described in the rest of this section.

➤ ***Note***

ct_sql replaces the **ct_cmd_alloc**, **ct_command**, and **ct_send** routines from the Open Client Client-Library API.

Executing Remote Procedure Calls with the web.sql Client-Library API

The **ct_rpc** routine executes a remote procedure call (RPC). **ct_rpc** accepts three arguments: *$db_handle*, *$rpc_name*, and *@params* or *%params*.

```
ct_rpc($db_handle, $rpc_name, @params or %params)
```

Database Handle

You should use *$ws_db* as the database argument (*$db_handle*) to specify the default database for the HTS file. But you can also use other database handles that you obtain with the **ws_connect** routine or the **ct_connect** routine.

RPC Name

The *$rpc_name* argument is the name of the RPC you want to call.

Parameter List

The *%params* or *@params* argument is the list of parameters passed to the stored procedure. Stored procedures can be called with two styles of parameters: named and positional. *%params* is used for named arguments, and *@params* is used for positional arguments.

Named parameters pass the parameter name and the value assigned to that name to the stored procedure. Positional parameters pass the argument value only.

The argument in the first position is passed as the first parameter to the stored procedure, the argument value in the second position is passed as the second parameter, the argument value in the third is passed as the third parameter, and so on.

If you use the named parameter style, the list of parameters for the RPC is passed as a Perl 5 reference to an associative array. If you use positional parameters, the list is passed as an array.

You can specify output parameters with a reference to a scalar variable. Scalar data can be passed as integer, float, or character string, depending on the Perl datatype.

To pass or receive a particular Sybase datatype other than integer, you must use a datatype-reference structure. For example, if you want to pass an integer as a character string, you must use a datatype-reference to enforce the behavior. The datatype-references can be written as follows:

```
{
'type'       =>  CS_<datatype>,
'value'      =>  <scalar> or <scalar-ref>,
'scale'      =>  CS_MAX_SCALE or CS_DEF_SCALE,
'precision'  =>  CS_MAX_PREC or CS_DEF_PREC
}
```

For a fuller example using named parameters, refer to “ct_rpc” in Appendix A, “web.sql Reference Pages”.

➤ ***Note***

If you expect the output to be numeric, call the RPC with the appropriate output parameters to avoid problems.

For a list of the `CS_` datatypes, see Appendix B, “RPC Datatype Summary.”

Example

The example below shows you how to use the **ct_rpc** routine with named parameters.

```
#....code deleted....


# Calling stored procedure with named parameters (associative
# array)
$status = ct_rpc($ws_db, "tempP2",
      {
      '@result_id' => \$res_id,
      '@result_price' => { "type" => &CS_MONEY_TYPE, "value" =>
$res_price, "precision" => CS_DEF_PREC, "scale" =>
CS_DEF_SCALE },
      '@lastname' => $lname,
      '@firstname' => $first,
      '@price' => $price,
      });


#....code deleted....
```

Processing Output Parameters from ct_rpc

When you call **ct_rpc**, you must call **ct_results** and **ct_fetch** to get rows returned from the **ct_rpc** call. If you supply output parameters when calling **ct_rpc**, **ct_results** will set its result type code parameter to CS_PARAM_RESULT. In this case, there is one row result available that contains the output parameters. When you detect a CS_PARAM_RESULT result type, use **ct_fetch** to fetch the output parameter row.

If you supplied output parameters to an RPC (with the **ct_rpc** routine call), the output cannot be updated until the results row information has been returned. You can use **ct_fetch_parameters** to update the output parameter variables supplied to **ct_rpc**. **ct_fetch_parameters** fills in the variables for any references that were passed in the parameter list.

➤ ***Note***

If output parameters are not supplied with **ct_rpc**, then you do not need to call the **ct_fetch_parameters** routine.

Example

```
#....code deleted....

if ($status  != &CS_SUCCEED)
 {
    print "<H3>Error: Bad status($status) returned from CT_RPC
      tempP2</H3>\n";
     die "ct_rpc failed";
 }
else
 {
   print "<H3>Success: ct_rpc succeeded. Now check for output
      parameters.</H3>\n";

# setup result loop

 }
while (($ret = ct_results($ws_db, $result_type)) == &CS_SUCCEED)
{
      if ($result_type == &CS_PARAM_RESULT)

# Get return parameter from RPC

      {
        $rc = ct_fetch_parameters($ws_db);
        if ($rc != &CS_SUCCEED)
        {
          die "ct_fetch_parameters failed \n";
        }
        if ($userid == $res_id)
```

```
        {
              print "<P><H3>ct_fetch_parameters returns=
$res_id<P></H3>";
        }
          next;
      }
      if ($result_type == &CS_STATUS_RESULT)
      {
        $status = ct_fetch($ws_db);
        ct_fetch($ws_db);
          next;
# ....code deleted....
```

Identifying the Result Set Type with the web.sql Client-Library API

After using **ct_sql** to send one or more SQL commands, your code should enter a loop that processes result sets as long as **ct_results** indicates that there are results to process, which is the case if **ct_results** returns CS_SUCCEED.

When **ct_results** returns a value of CS_SUCCEED, it also sets the value of its second parameter to indicate the type of result set available. Table 3-3 lists the different result type codes and what they indicate.

Within the processing loop, you should test for the different result type codes and process the results appropriately. The following sections provide specific instructions for handling the different types of result sets.

Table 3-3: Values of ct_results result type codes

Values of result type code:	What it indicates:	Result set contains:
CS_CMD_DONE	The server has completed processing a SQL command.	No results.
CS_CMD_FAIL	The server encountered an error while executing a SQL command.	No results.
CS_CMD_SUCCEED	The server successfully executed a SQL command that returns no data (for example, an **insert** command).	No results.

Table 3-3: Values of ct_results result type codes (continued)

Values of result type code:	What it indicates:	Result set contains:
CS_ROW_RESULT	One or more regular rows are available from a **select** command, or a stored procedure call that executes **select** commands.	One or more rows of tabular data.
CS_COMPUTE_RESULT	A compute row is available from a **select** command.	A single row of compute results.
CS_STATUS_RESULT	The return status of a stored procedure is available.	A single row containing a single status value.
CS_PARAM_RESULT	The output parameters of a stored procedure are available.	A single row of output parameter values.
CS_MSG_RESULT	The server sent a message.	No fetchable results. Your application can call **ct_res_info** to get the message's ID. Parameters associated with the message, if any, are returned as a separate parameter result set.
CS_END_RESULTS	There are no more result sets to process.	No results.

The general form of the **ct_results** loop is:

```
# Call ct_sql to send commands to the server.
# Process the results from the server.
while (($ret = ct_results($ws_db, $result_type)) == CS_SUCCEED)
{
    # Equivalent of switch statement in Perl.
    RESULTS:
    {
        # Test for each result type code
        if ($result_type == ... )
            {
                 # Process the result for that return code
                 last RESULTS; # Jump to end of RESULTS block.
            }
        # Etc...
    }
```

```
}
if ($ret == CS_FAIL)
{
    # A connection error occurred. Clean up connection state.
    ct_cancel($ws_db, CS_CANCEL_ALL);
    ws_error("A database connection error occurred");
}
```

When there are no more result sets to process, **ct_results** returns a value of CS_END_RESULTS. If a problem occurs, such as a network failure during processing, **ct_results** returns CS_FAIL. Note that a return value of CS_FAIL does not mean that a command has failed; command failure is indicated by a result type code of CS_CMD_FAIL. If you receive a CS_FAIL return value, you should execute **ct_cancel($ws_db, CS_CANCEL_ALL)** to clean up the state of your connection.

Retrieving Information About Return Data with the web.sql Client-Library API

Once **ct_results** indicates that result set data is available, you can use one of several routines to obtain information about that data.

ct_col_names and **ct_col_types** are useful for determining information about all types of row results.

ct_col_names returns an array containing the column names of the return data currently available. Typically, you use this command to retrieve column names so that you can print them as headers to a table of returned data.

ct_col_types returns an array containing the column data types of the result set data currently available. This routine can be useful for determining if a column contains a money value or some other type that you would like to format specially.

ct_res_info allows you to determine several types of information about return data, including the number of items in a result set, the number of rows affected by the current command, and the number of compute clauses in the current command. See the **ct_res_info** reference page in Appendix A, “web.sql Reference Pages,” for more information.

Processing Query Results with the web.sql Client-Library API

A **select** statement can produce several result sets that you need to process. In general, a successful query produces one or more row result sets and one or more compute result sets. **ct_results** returns several result type codes to indicate the progress of a query and the result sets returned.

When row results (that is, data from tables) are available from the server, **ct_results** returns the CS_ROW_RESULT result type parameter. When you detect this value, you should use either **ct_fetch** or **ws_fetch_rows** to fetch the rows.

Each call to **ct_fetch** retrieves a single row, returning it as a Perl array. You can then print or manipulate the values as you like. **ct_fetch** returns an empty array when there are no more rows. On the other hand, **ws_fetch_rows** fetches all of the rows and prints them according to a formatting string that you provide as a parameter or as an HTML 3.0 table.

➤ ***Note***

By default, a query can return an unlimited number of rows. To specify the number of rows returned as a result of a query, you can use a **ct_options** call. See “Setting Server Options with the web.sql Client-Library API” on page 3-29 for an example.

When a compute row is available, **ct_results** returns the CS_COMPUTE_RESULT result type parameter. When you detect this value, you should use either **ct_fetch** or **ws_fetch_rows** to fetch the compute row. When a **select** command contains a **COMPUTE** clause, the server returns alternating batches of result sets of types CS_ROW_RESULT and CS_COMPUTE_RESULT until the query is complete.

If there is a problem executing the command (for example, permission or rule violations), **ct_results** returns the CS_CMD_FAIL result type parameter. In this case, there is no result set for you to retrieve. If you like, you can test for this value and print an error message.

Once the server has finished processing a single command (that is, a single **select** statement)—whether successful or unsuccessful— **ct_results** returns the CS_CMD_DONE result type parameter.

This sequence of result type parameters and result sets repeats for the **select** statements you include in the command string sent to the server.

➤ ***Note***

Remember that a result type parameter value of CS_CMD_DONE indicates only that a single SQL command is complete. If you sent multiple commands to the server, you must continue to call **ct_results** and process result sets until **ct_results** returns a value of CS_END_RESULTS or CS_FAIL.

The following example of an HTS file shows how to process a simple database query using the web.sql Client-Library API. Note that this example demonstrates using both **ct_fetch** and **ws_fetch_rows** to fetch the rows.

```
<SYB TYPE=PERL>
# Build the SELECT statement
$sql_stmt = qq!select type, price
            from pubs2..titles
            order by type
            compute sum(price) by type!;
# Send the command
if (($rc = ct_sql( $ws_db, $sql_stmt ) ) != CS_SUCCEED) {
    ws_error ("Unable to process database request.");
}
# Process the results from the server.
while (($ret = ct_results($ws_db, $result_type)) == CS_SUCCEED) {
    # Equivalent of switch statement in Perl.
    RESULTS: {
        # Simply ignore report of command completion
        if ($result_type == CS_CMD_DONE) {
            last RESULTS; # Jump to end of RESULTS block.
        }
        # Simply ignore notification of unsuccessful completion
        if ($result_type == CS_CMD_FAIL) {
```

```
            last RESULTS; # Jump to end of RESULTS block.
          }
        # Fetch and print row results
        if ($result_type == CS_ROW_RESULT) {
            while (@row = ct_fetch($ws_db)) {
                print join(" ", @row), "<BR>\n";
             }
             last RESULTS; # Jump to end of RESULTS block.
        }
        # Fetch and print compute results
        if ($result_type == CS_COMPUTE_RESULT) {
            ws_fetch_rows($ws_db);
            last RESULTS; # Jump to end of RESULTS block.
        }
    }
}
if ($ret == CS_FAIL) {
    # A connection error occurred. Clean up connection state.
    ct_cancel(CS_CANCEL_ALL);
    ws_error("A database connection error occurred");
}
</SYB>
```

Processing Other SQL Command Results with the web.sql Client-Library API

SQL commands such as **insert**, **update**, and **delete** are simpler to process because they do not produce any row or compute results.

If the command executes successfully, **ct_results** returns the CS_CMD_SUCCEED result type parameter. However, there are no result sets for you to fetch. If there is a problem executing the command (for example, permission or rule violations), **ct_results** returns the CS_CMD_FAIL result type parameter.

Once the server has finished processing a single command—whether successful or unsuccessful— **ct_results** returns the CS_CMD_DONE result type parameter.

The following example shows the code needed to process a SQL command other than a query. Note the use of **ct_res_info** to determine and print the number of rows affected by the command.

```
<SYB TYPE=PERL>
# Build the SELECT statement
$sql_stmt = qq!insert into pubs2..publishers
            values ('1622', 'Jardin, Inc.', 'Camden', 'NJ')!;
# Send the command
if (($rc = ct_sql( $ws_db, $sql_stmt ) ) != CS_SUCCEED) {
    ws_error ("Unable to process database request.");
}
# Process the results from the server.
while (($ret = ct_results($ws_db, $result_type)) == CS_SUCCEED) {
    # Equivalent of switch statement in Perl.
    RESULTS: {
        # Simply ignore notification of successful completion
        if ($result_type == CS_CMD_DONE) {
            last RESULTS; # Jump to end of RESULTS block.
        }
        # Report command success
        if ($result_type == CS_CMD_SUCCEED) {
            print "<P><B>Insert successful.</B>\n";
            $res_info = ct_res_info($ws_db, CS_ROW_COUNT);
            print "${res_info} row(S) inserted.<P>\n";
            last RESULTS; # Jump to end of RESULTS block.
        }
        # Report command failure
        if ($result_type == CS_CMD_FAIL) {
            print "<P><B>Insert unsuccessful.</B><P>\n";
            last RESULTS; # Jump to end of RESULTS block.
        }
    }
}
```

```
if ($ret == CS_FAIL) {
    # A connection error occurred. Clean up connection state.
    ct_cancel(CS_CANCEL_ALL);
    ws_error("A database connection error occurred");
}
</SYB>
```

Processing Stored Procedure Results with the web.sql Client-Library API

Because a stored procedure can contain many different SQL commands, it can generate many different result types. You should structure the **ct_results** loop in your HTS file to handle all possible result types. To handle the result types and data sets generated by **select** commands, follow the instructions described in “Processing Query Results with the web.sql Client-Library API” on page 3-23. To handle the result types and data sets generated by other SQL commands, follow the instructions described in “Processing Other SQL Command Results with the web.sql Client-Library API” on page 3-25. In addition to the result types discussed so far in this section, stored procedures can generate a return status and output parameters.

ct_results returns CS_STATUS_RESULT to indicate that the server has sent a stored procedure return status. In this case, there is one row result available that contains the return status. When you detect a CS_STATUS_RESULT result type, you should use either **ct_fetch** or **ws_fetch_rows** to fetch the return status row.

ct_results returns CS_PARAM_RESULT to indicate that the server has sent output parameters from a stored procedure. In this case, there is one row result available that contains the output parameters. Use either **ct_fetch** or **ws_fetch_rows** to fetch the output parameter row.

The following code fragment shows you how to process the return status and output parameters from a stored procedure:

```
while (($ret = ct_results($ws_db, $result_type)) == CS_SUCCEED) {
    # Equivalent of switch statement in Perl.
    RESULTS: {
        # ...
        # Process return status
        if ($result_type == CS_STATUS_RESULT) {
           print "<P>Return status: ", ct_fetch($ws_db), "<P>\n";
```

```
                last RESULTS; # Jump to end of RESULTS block.
            }
            # Process output parameter
            if ($result_type == CS_PARAM_RESULT) {
                print "<P>Output Parameter: ",
                      join(",", ct_fetch($ws_db)), "<P>\n";
                last RESULTS; # Jump to end of RESULTS block.
            }
            # ...
        }
    }
```

Processing Message Results with the web.sql Client-Library API

You can also handle **datastream messages** with the web.sql Client-Library API. Datastream messages provide a way for clients and Open Server applications to exchange information.

A datastream message consists of a message ID and zero or more parameters. The client and the Open Server application must be programmed to agree on the meaning of each message ID. User-defined message IDs must be greater than or equal to CS_USER_MSGID and less than or equal to CS_USER_MAX_MSGID. Message IDs SRV_MINRESMSG through SRV_MAXRESMSG are reserved for internal Sybase use. See the *Open Client Client-Library/C Reference Manual* and the *Open Server Server-Library/C Reference Manual* for more information about datastream messages.

ct_results returns CS_MSG_RESULT to indicate that the server has sent a message. In this case, you should call **ct_res_info** requesting the CS_MSGTYPE value to retrieve the message ID number. You should then process the message as appropriate for your application.

Any parameters associated with a message are returned in the form of a parameter result set following the message result set. If a parameter result set follows the message result set, **ct_results** returns CS_PARAM_RESULT to indicate that the server has sent message parameters. When you detect a CS_PARAM_RESULT result type, you should use either **ct_fetch** or **ws_fetch_rows** to fetch the message parameter row.

Canceling Results with the web.sql Client-Library API

The ct_cancel routine allows you to cancel either the current result set or the entire command batch.

You can cancel a result set when you do not need to finish processing the result rows. For example, you might need to process only the first 20 rows of a 100-row result set. After processing 20 rows, you can discard the remaining rows by canceling results. To cancel a current result set, call ct_cancel with the parameter CS_CANCEL_CURRENT. The effect is similar to using ct_fetch to retrieve all the remaining results in the current result set and discarding the results.

When you cancel the entire command batch, your program instructs the server to halt execution of all commands sent in the last call to ct_sql and discard any results already generated. To cancel a command batch, call ct_cancel with the parameter CS_CANCEL_ALL. If you call ct_cancel with the parameter CS_CANCEL_ALL while you are in a ct_results loop, ct_results returns CS_CANCELED the next time you call it.

Example

```
#....code deleted...
if ($ret == CS_FAIL) {
    # A connection error
    ct_cancel(CS_CANCEL_ALL);
    ws_error("A database connection error
occurred");
}
```

Setting Server Options with the web.sql Client-Library API

You can set, retrieve, or clear connection options that take effect at the server. Options alter SQL server processing or modify returns from the server. You can use the ct_options routine to set, retrieve, and clear server options. See the ct_options reference page in Appendix A, "web.sql Reference Pages," for more information about available server options and their effects.

As an example, consider that by default a query can return an unlimited number of rows. To limit the number of rows returned as a result of a query, you can use a ct_options call. For example, you can set the limit to 200 rows using the following line:

```
ct_options($ws_db, CS_SET, CS_OPT_ROWCOUNT, 200, CS_INT_TYPE);
```

Handling Errors with Callback Routines in the web.sql Client-Library API

web.sql supports callback routines for handling callback events, which can occur in either the client or the server. When a callback event occurs, web.sql executes the callback routines that have been installed.

Client callback events can occur when a Client-Library routine encounters trouble, such as running out of memory. Server callback events can occur when the server sends error messages. Client and server callback events are handled separately, and you can install different callback routines for client and server events.

web.sql automatically installs basic error-handling callback routines for both client and server events. These callback routines print error information about the callback event.

You can override these callback routines by installing your own using the **ct_callback** routine. For example, you might want to create a callback routine that logs errors or that sends e-mail to a system administrator.

To install a callback routine, call **ct_callback** with parameters indicating whether it is a client or server routine and the name of the callback routine surrounded by single quotes. For example, the following line installs **client_err** as a client callback routine:

```
ct_callback(CS_CLIENTMSG_CB, 'client_err');
```

The following line installs **server_err** as a client callback routine:

```
ct_callback(CS_SERVERMSG_CB, 'server_err');
```

The callback routines you install must appear in either the same **<SYB>** block as the **ct_callback** call installing them or in an earlier **<SYB>** block.

After handling an event in your callback, if you want to resume processing of your HTS file, include the line:

```
CS_SUCCEED;
```

If you want to abort processing of your HTS file (for example, if the event indicated an unrecoverable error), include the line:

```
CS_FAIL;
```

Client callback routines receive six arguments when called:

- Layer (a number)

- Origin (a number)
- Severity (a number)
- Error number (a number)
- Error message text (a string)
- Operating system message text (a string)

The following example shows the code for the default client callback, **ct_client_callback**:

```
#
# Handle a client-side error
#
sub ct_client_callback {
    my($layer, $origin, $severity, $number, $msg, $osmsg) = @_;
    printf "<P><B>! Open Client Error: %ld (Layer %ld,
        Origin %ld, Severity %ld)</B>\n", $number,
        $layer, $origin, $severity;
    printf "<P><B><blockquote>%s</blockquote></B>\n", $msg;
    if (defined($osmsg)) {
        printf "<P><B>! Operating System Error:<blockquote>
            %s</blockquote></B>\n", $osmsg;
    }
    printf "<P>\n";
    CS_SUCCEED;
}
```

Server callback routines receive eight arguments when called:

- Command
- Error number (a number)
- Severity (a number)
- State (a number)
- Line number (a number)
- Server name (a string)
- Stored procedure (a string, if applicable - otherwise empty)
- Error message text (a string)

The following example shows the code for the default server callback, **ct_server_callback**:

```
sub ct_server_callback {
    my($cmd, $number, $severity, $state, $line,
       $server, $proc, $msg) = @_;
    if ($severity != 10) {
        # This is an error
        printf "<P><B>! Server Error: %ld (Severity %ld,
            State %ld, Line %ld)</B>\n", $number,
            $severity, $state, $line;
        if (defined($server)) {
            printf "<P><B>! Server '%s'</B>\n", $server;
        }
        if (defined($proc)) {
            printf "<P><B>! Procedure '%s'</B>\n", $proc;
        }
        printf "<P><B><blockquote>%s</blockquote></B>\n", $msg;
        printf "<P>\n";
    }
    elsif (! defined $ws__suppress_server_message{$number}) {
        # This is an informational message
        if ($number) {
            print "Server message $number: $msg\n";
        }
        else {
            print "$msg\n";
        }
    }

    CS_SUCCEED;
}
```

Returning Non-HTML Data with web.sql

So far, this guide has discussed how to create HTS files. All HTS files are identified with the suffix *.hts*. When a Web browser requests an *.hts* file, the web.sql processor reads the file, processes the database requests contained in that file, and then composes HTML output for the HTTP server to pass to the browser. *.hts* files return HTML output in HTML format only.

You can also use web.sql to return data in other formats, such as JPEG graphics or WAV audio, by specifying the **ws_content_type** in a *.pl* file. web.sql can execute a Perl file that outputs any data format and then transmits the data to the Web browser. Executing the Perl file through web.sql rather than directly as a CGI script allows the Perl file to use the web.sql database APIs described in this chapter and Appendix A, "web.sql Reference Pages."

Perl files that you want to execute through web.sql must have the extension *.pl*. To access Perl scripts, open the web.sql welcome page by using one of the following platform-dependent URLs:

- If you are using the NSAPI version of web.sql on a UNIX platform:

  http://*<servername>*/*<doc-root-subdir>*/*<scriptname.pl>*

  <servername> is the IP address (hostname) and the port number of the Web server, *<doc-root-subdir>* is the path to your script's subdirectory in the Web server's document root directory, and *<scriptname.pl>* is the name of the Perl script.

- If you are using the shared-library CGI version of web.sql on a UNIX platform:

  http://*<servername>*/*<cgi-name>*/websql/*<doc-root-subdir>*/*<scriptname.pl>*

  <servername> is the IP address (hostname) and the port number of the Web server, *<cgi-name>* is the name of your Web server's CGI script directory, **websql** is the name of the web.sql CGI program, *<doc-root-subdir>* is the path to your script's subdirectory in the Web server's document root directory, and *<scriptname.pl>* is the name of the Perl script.

- If you are using the static CGI version of web.sql on a UNIX platform:

  http://*<servername>*/*<cgi-name>*/ws.exe/*<doc-root-subdir>*/*<scriptname.pl>*

  <servername> is the IP address (hostname) and the port number of the Web server, *<cgi-name>* is the name of your Web server's CGI script directory, **ws.exe** is the name of the web.sql CGI

program, <*doc-root-subdir*> is the path to your script's subdirectory in the Web server's document root directory, and <*scriptname.pl*> is the name of the Perl script.

- If you are using the NSAPI version of web.sql on the NT platform:

  http://<*servername*>/<*doc-root-subdir*>/<*scriptname.pl*>

  <*servername*> is the IP address (hostname) and the port number of the Web server, <*doc-root-subdir*> is the path to your script's subdirectory in the Web server's document root directory, and <*scriptname.pl*> is the name of the Perl script.

- If you are using the CGI version of web.sql on the NT platform:

  http://<*servername*>/<*cgi-name*>/ws.exe/<*doc-root-subdir*>/<*scriptname.pl*>

  <*servername*> is the IP address (hostname) and the port number of the Web server, <*cgi-name*> is the name of your Web server's CGI script directory, **ws.exe** is the name of the web.sql CGI program, <*doc-root-subdir*> is the path to your script's subdirectory in the Web server's document root directory, and <*scriptname.pl*> is the name of the Perl script.

➤ ***Note***

After you specify the name of the web.sql CGI program in web.sql CGI URLs, the document root information and the name of the Perl script are passed to the web.sql CGI program.

Using web.sql Features in the Perl File

When writing a Perl file to return non-HTML data, you can use:

- Any of the predefined Perl variables described in "Using Perl Variables and Form Data in HTS Files" on page 3-2.
- Any of the web.sql convenience routines described in "web.sql Convenience and Client-Library APIs" on page 3-2.
- Any of the web.sql Client-Library routines described in "Using the web.sql Client-Library API" on page 3-10.

➤ ***Note***

Do not use the **<SYB>** tag in *.pl* files. The **<SYB>** tag is for use in *.hts* files only.

Perl files require one additional web.sql routine, **ws_content_type**, which indicates the type of data the script returns. (You do not need to use this routine in an *.hts* file because web.sql automatically includes an HTML data type identifier when it processes an *.hts* file.) You must call **ws_content_type** before your script outputs any data. The parameter you pass to **ws_content_type** should be a string containing a Multipurpose Internet Mail Extension (MIME) type. For example, the following line indicates that the script returns an image in *.jpeg* format:

```
ws_content_type("image/jpeg");
```

➤ ***Note***

Be sure to supply a valid data type with this function. If you supply an invalid data type to **ws_content_type**, the web browser will display an error.

Examples of Returning Data Using an HTS File and a Perl Script

The following examples contrast how return data using an HTS (*.hts*) file and a Perl file (*.pl*) would be created. Both examples return HTML data.

The first example shows an HTS file that is used to call **sp_who** to obtain a list of the users logged on to a database server.

```
<HTML>
<BODY>
<H1><CENTER>Example of web.sql using an .hts file</CENTER></H1>
<SYB>
    ws_sql($ws_db, "sp_who");
</SYB>
<P>
</BODY>
</HTML>
```

The next example shows the same functionality implemented as a *.pl* script:

```
ws_content_type("text/html");  # Print HTTP header info
print qq!
<HTML>
```

```
<BODY>
<H1><CENTER>Example of web.sql using a .pl file</CENTER></H1>
!;
ws_sql($ws_db, "sp_who");
print qq!
<P>
</BODY>
</HTML>
!;
```

Returning HTTP Header Information

web.sql provides a mechanism to output information that can be sent as an HTTP header to the Web browser, like Web server cookies and URL redirection. This section explains how to set a cookie value and URL redirection with web.sql.

A cookie is a small piece of information that a Web server can store with a Web browser and later read back from that browser. This is useful for having the browser remember specific information across several pages.

URL redirection is a mechanism to redirect the Web browser to a different URL from the current HTS file. When the HTS file is accessed, control is transferred to the redirected URL specified by the Location tag in the beginning of the current HTS file.

If you need further information about Web server cookies, see:

http://home.netscape.com/newsref/std/cookie_spec.html

In order to set the HTTP headers from an HTS file you need to prefix the HTS file with the "**wsh-**" prefix. The **wsh-** prefix instructs web.sql to send the appropriate header information to the client, not the default header information. The application programmer should delimit headers in **wsh-** prefixed files with two line feed characters (`\n\n`) to indicate to the server which part of the file is the header and which is the body.

➤ ***Note***

In web.sql, Perl files (*.pl*) do not have a default header.

Example of Setting the Cookie Value

In the following example, *wsh-setcookie.hts* sets the cookie value. The cookie value is read back from *getcookie.hts.*

wsh-setcookie.hts

```
<SYB>
print "Content-type: text/html\n";
print "Set-cookie: mycookie=my-cookie-value\n\n";
</SYB>
```

getcookie.hts

```
<SYB>
print $ENV{'HTTP_COOKIE'};
print "This is the value set in wsh_setcookie.hts.\n";
</SYB>
```

Example of URL Redirection

The following example shows URL redirection. You only need to access the *wsh-locate.hts.*You are automatically redirected to *newlocation.hts.*

wsh-locate.hts

```
<SYB>
print "Content-type: text/html\n";
print "Location: http://hostname:port/newlocation.hts\n\n";
</SYB>
```

newlocation.hts

```
<SYB>
print "You were redirected to the new file.\n";
</SYB>
```

➤ ***Note***

If you use redirection with the CGI implementation of web.sql, you need to include the full CGI path.

4 Sybase web.sql Examples

This chapter contains examples of the types of Transact SQL statements and Perl scripts that you can include in an HTS file. Topics discussed include:

- Transact SQL Statements Within an HTS File 4-1
- Perl Scripts Within an HTS File 4-2

➤ ***Note***

Most of the examples in this chapter are provided with the Sybase web.sql product. You can copy these examples and use them in your HTS files.

Transact SQL Statements Within an HTS File

Generally, you want to use SQL statements to create a table; insert, update, delete, or select data from a table; or execute a stored procedure. Sybase web.sql executes the statement; the results display on the client browser.

Examples of SQL in an HTS File

Any SQL statement can be used in an HTS file that has the attribute TYPE=SQL. This section contains examples of Transact SQL statements that you can include in an HTS file.

A SQL Select Statement

The following code is an example of a **<SYB>** block that contains a simple SQL select statement in which data is selected from the *pubs2* database:

```
<SYB TYPE=SQL>
select au_lname, au_fname, title, price
from pubs2..authors a, pubs2..titleauthor ta, pubs2..titles t
where (a.au_id = ta.au_id
and t.title_id = ta.title_id)
and titles.type = "mod_cook"
```

```
</SYB>
```

The previous query yields the following results:

au_lname	au_fname	title	price
del Castillo	Innes	Silicon Valley Gastronomic Treats	19.99
DeFrance	Michel	The Gourmet Microwave	2.99
Ringer	Anne	The Gourmet Microwave	2.99

SQL Transaction Statements

The following example includes two SQL statements within an HTS file. The first statement inserts values into the following columns in the publisher's table: *pub_id*, *pub_name*, *city*, and *state*. The second statement deletes information from the database.

```
<SYB TYPE=SQL>
insert publishers
values ("9934", "Long Horns", "Dallas", "TX")

delete titles
      from titles t, titleauthor ta, authors a
      where t.title_id = ta.title_id
      and ta.au_id = a.au_id
      and a.au_lname = "Panteley"
      and a.au_fname = "Sylvia"
</SYB>
```

Perl Scripts Within an HTS File

Perl is a flexible scripting language. You can use Perl scripts to access a database and manipulate SQL query results. This section provides examples of ways you can use Perl in your HTS file.

Examples of Perl Scripts in an HTS File

This section contains several examples of Perl scripts, ranging from simple to complex, used within an HTS file.

Executing SQL Query Statements

The following example enables users to select the database they want to query and enter the SQL statement. The **ws_sql** routine is used to execute the SQL statement.

```
<SYB TYPE=PERL>
#Send the SQL statement to the Server.
if ($ws_db && $sql)
{
    ws_sql($ws_db, "use $database");
    ws_sql($ws_db, $sql );
}
</SYB>
```

Executing SQL Transaction Statements

The following example sends a SQL statement to the database that is a transaction rather than a query. The same procedure is used for **insert**, **update**, and **delete** statements. This example updates records in the *pubs2* database *discounts* table, setting customer discounts to 6.0.

Such commands produce results from the server that you, the web.sql programmer, need to handle. For example, you may want to tell the user if the command succeeded and if so, how many rows were affected.

```
<SYB TYPE=PERL>

my $sqlstmt = qq!
update pubs2..discounts
set discount = 6.0
where discounttype = "Customer Discount"!;

    print "Sending the following SQL statement to the server:<P>\n";
    print "<PRE>", $sqlstmt, "</PRE><P>\n";

    my $rc = ct_sql($ws_db, $sqlstmt);
```

```
        if ($rc != CS_SUCCEED) {
              ws_error("ct_sql() call to perform UPDATE failed.");
        }

        # Handle return values
        my $result_type = "";
        while (($rc = ct_results($ws_db, $result_type)) == CS_SUCCEED)
        {
              RES_TYPE: {

                   if ($result_type == CS_CMD_SUCCEED) {

                        print "Command was successful.<BR>\n";
                        my $res_info = ct_res_info($ws_db, CS_ROW_COUNT);
                        print "${res_info} row(s) affected by this \
                         command.<BR>\n";
                        last RES_TYPE;
                   }

                   if ($result_type == CS_CMD_FAIL) {
                        print "Command failed.<BR>\n";
                        last RES_TYPE;
                   }

                   if ($result_type == CS_CMD_DONE) {
                        last RES_TYPE;
                   }

                   print "Unexpected result type returned.<BR>\n";
              }
        }
  </SYB>
```

Parsing Form Data

The following example contains multiple **<SYB>** blocks and illustrates how web.sql can parse form data passed to an HTS file. This example presents an online form in which users select the database to use from a list and then enter the SQL statement in an entry field. web.sql executes the SQL statement entered and displays the results in the client browser.

```
<FORM ACTION=isql_results.hts METHOD=POST>
<P><STRONG>Database: </STRONG>
<SELECT NAME=database>
<SYB TYPE=PERL>
    if ($database ne "") {
        print "<option selected>$database\n";
    }

# Send the T-SQL now.
ws_sql ($ws_db, qq!select name from
master.dbo.sysdatabases
    where name <> "$database"!,
"<option>%s");
</SYB>
</SELECT>
<P>
<STRONG>SQL Command</STRONG>:
<TEXTAREA NAME=SQL ROWS=5 COLS=78>

# Print results.
<SYB TYPE=PERL>
    print $sql;
</SYB>
</FORM>
```

Using Perl Variables in Form Data

The following example is from a document that is activated when a user enters data into a form and submits it. (This causes a “POST” to this document.) All the input data from the form document is received in the form of Perl variables.

For example, a user enters a state abbreviation into an entry field named “state” in the form, which posts to the following HTS file. The Perl variable called ***$state*** can be used to query the ***pubs2*** database for all stores in that state.

```
Stores found in '$state':

<SYB TYPE=SQL>
   select stor_name, stor_address, city, state
   from   pubs2..stores
   where  state = upper("$state")
   order by stor_name
</SYB>
```

Accessing Version Information

The following example shows how a Perl script with the **ws_version** routine can be used in an HTS file to identify versions of a product. Notice the `TYPE` argument is not specified: the default `TYPE` is Perl.

```
<SYB>

      my($rights_n_addr_str, $version_str) = ws_version();

      my($ws_copy_rights, $ws_restricted_rights, $ws_sybase_address)
            = split(/\//, $rights_n_addr_str);

      my($product, $version, $level, $platform, $os_version, $ebf_no,
            $optimized, $build_date, $perl_version, $oc_version,
            $cgi_nsapi)
                = split(/\//, $version_str);

      print "<TABLE BORDER>";
```

```
        print "<TR><TD><B>COPYRIGHTS</B><TD> $ws_copy_rights";
        print "<TR><TD><B>RESTRICTED RIGHTS</B><TD>
        $ws_restricted_rights";
        print "<TR><TD><B>SYBASE ADDRESS</B><TD> $ws_sybase_address";
        print "</TABLE>";
        print "<TABLE BORDER>";
        print "<TR><TD><B>PRODUCT</B><TD>$product";
        print "<TR><TD><B>VERSION</B><TD> $version";
        print "<TR><TD><B>LEVEL</B><TD> $level";
        print "<TR><TD><B>PLATFORM</B><TD> $platform";
        print "<TR><TD><B>OS VERSION</B><TD> $os_version";
        print "<TR><TD><B>EBF #</B><TD> $ebf_no";
        print "<TR><TD><B>COMPILE MODE</B><TD> $optimized";
        print "<TR><TD><B>BUILD DATE</B><TD> $build_date";
        print "<TR><TD><B>PERL VERSION</B><TD> $perl_version";
        print "<TR><TD><B>OPEN CLIENT VERSION</B><TD> $oc_version";
        print "<TR><TD><B>INTERFACE</B><TD> $cgi_nsapi";
        print "</TABLE>";

</SYB>
```

Handling Error Messages

The web.sql routine **ws_error** specifies the text in an error message. When the **ws_error** routine is called, web.sql stops processing or executing the HTS file.

In the example below, a call is made to **ws_sql**. If an error occurs while web.sql is processing **ws_error**, then **ws_error** is executed, and the specified error message is executed. Also, when an error is encountered, all processing stops. In this case, web.sql does not process the stored procedure, nor does it print the remaining HTML text to the browser.

```
<HTML>
<HEAD>
<TITLE>Example of ws_error()</TITLE>
</HEAD>
<BODY>
```

```
ws_sql($ws_db, "sp_who");

ws_error("This is the text of my error.
Processing should stop now.");

</SYB>


<P>

This is more text after the sql block.  It should
not appear in the output since the ws_error() call
should stop processing of the HTS file.

<P>

</BODY>

</HTML>
```

The following example sets up subroutines to handle SQL Server or Open Client error messages. By default, these errors are routed to the default error-handling routines provided with web.sql. However, you can override the defaults and provide custom error-message handling by calling **ct_callback** to point to a custom error handler.

This example contains two custom handlers: one for client error messages, and one for server error messages. An invalid SQL **select** statement is sent to activate the error handler.

```
<SYB TYPE=PERL>

#....code deleted....

      # Set up the callback routines for error messages to point

      # to the custom ones below.  Note that these routines will

      # also be used by any other following <SYB> blocks in this

      # document

   ct_callback(CS_CLIENTMSG_CB, 'client_err');

   ct_callback(CS_SERVERMSG_CB, 'server_err');


   # Send an invalid ct_connect() call for an unknown server so

   # we generate an error


   my $badconn = ct_connect("xxxxx","yyyy","ZZZZZXXX");


#....code deleted....
```

```
sub client_err
   {
      # This routine gets passed 6 arguments related to the error
      local($layer, $origin, $severity, $errno, $msg, $osmsg)
       = @_;

      print "<P><HR><STRONG>CLIENT ERROR OCCURRED:</STRONG>\n";
      print "<P>Client error info: <BR>\n";
      printf "Error: Layer=%ld Origin=%ld Severity=%ld
           Number=%ld <BR>\n",
             $layer, $origin, $severity, $errno;
      printf "Message text: %s <BR>\n", $msg;

      if (defined($osmsg)) {
         printf "OS Message '%s' <BR>\n", $osmsg;
      }

      print "<P><HR>";

      CS_SUCCEED;
   }

   # Callback routine to handle server error messages
   sub server_err
   {
      # This routine gets passed 8 arguments related to the error
      local($cmd, $errno, $severity, $state, $line, $server,
$proc, $msg) = @_;

      print "<P><HR><STRONG>SERVER ERROR OCCURRED:</STRONG>\n";
      print "<P>Server error info: <BR>\n";
      printf "Cmd=%s <BR>\n", $cmd;
```

```
      printf "Error: Number=%ld Severity=%ld State=%ld Line=%ld
<BR>\n",
             $errno, $severity, $state, $line;

      if (defined($server)) {
         printf "Server '%s' <BR>\n", $server;
      }

      if (defined($proc)) {
         printf "Stored procedure '%s' <BR>\n", $proc;
      }

      printf "Message text: %s <BR>\n", $msg;

      print "<P><HR>";

      CS_SUCCEED;
   }

</SYB>
```

Processing a Perl Script

Sybase web.sql can process a straight Perl file that contains valid Perl scripts and web.sql routines. If you do not need to interpolate Perl results into HTML code, you may want to use this approach.

➤ ***Note***

You do not include the <**SYB**> markers in a pure Perl (*.pl*) file. You only use that marker in an *.hts* file.

web.sql expects Perl files to return the proper HTTD header information. The following call prints this header information, and in this case, it sets the MIME type to HTML. If you want to return data for a different MIME type, you can specify the type with the

ws_content_type routine. This is useful if you want to query information from the database and return it to a special helper application that you “attached” to a browser. In that case, the helper application would not have to parse HTML. You could return any format of data to that helper application.

```
ws_content_type("text/html");    # Print HTTP header info

# Since we'll return HTML, let's print some of the standard info it
# expects.  We'll use the "qq" notation so that the carriage return
# characters get inserted into the strings that get printed...

print qq!
<HTML>
<HEAD>
<TITLE>Example of web.sql using a .pl file</TITLE>
</HEAD>
<BODY>
<H2><CENTER>Example of web.sql using a .pl file</CENTER></H2>
!;

# Do a simple web.sql query to get a list of who is logged on
# the database server by calling the "sp_who" stored procedure
# (Note that web.sql has already set up the $ws_db default
# connection for us, as it does for .hts files (assuming you have
# a valid default connection set up in .websql.pl).

ws_sql($ws_db, "sp_who");

# Now let's print the rest of the HTML

print qq!
<P><HR>
</BODY>
</HTML>
```

```
!;
```

Handling Server Results

The following code provides a comprehensive example that demonstrates how web.sql handles various result types returned after SQL queries. The following example uses a subroutine that includes a large switch statement to handle all types of results from different Transact-SQL statements. A number of different SQL statements are sent to show how web.sql handles the results.

In this example the Sybase *pubs2* sample database is used.

```
<SYB TYPE=PERL>

   print "<P><BR><STRONG>First, point to the pubs2\
      database...</STRONG><P>\n";

   my $rc = ct_sql($ws_db, "use pubs2");

   if ($rc != CS_SUCCEED) {
      ws_error("ct_sql() call failed on 'use pubs2' command.");
   }

   process_results();

   print "<P><BR><STRONG>Try a simple SELECT \
      statement...</STRONG><P>\n";

   $rc = ct_sql($ws_db, qq!select type, price
                           from   titles
                           where  type in ("business","mod_cook")
                           order by type!);

   if ($rc != CS_SUCCEED) {
      ws_error("ct_sql() call failed on first SELECT statement.");
   }
```

```
process_results();


print "<P><BR><STRONG>Now a SELECT statement with a
   COMPUTE...</STRONG><P>\n";

$rc = ct_sql($ws_db, qq!select type, price
                        from   titles
                        where  type in ("business","mod_cook")
                        order by type
                        compute sum(price), avg(price) by type!);

if ($rc != CS_SUCCEED) {
   ws_error("ct_sql() call failed on SELECT statement with
         COMPUTE.");
}

process_results();

print "<P><BR><STRONG>Now send an UPDATE statement...</STRONG><P>\n";

$rc = ct_sql($ws_db, qq!update pubs2..discounts
                        set    discount = 6.0
                        where  discounttype = "Customer Discount"!);

if ($rc != CS_SUCCEED) {
   ws_error("ct_sql() call failed on UPDATE statement.");
}

process_results();


print "<P><BR><STRONG>Execute a stored procedure that returns\n"
    . "a status...</STRONG><P>\n";
```

```
   # storename_proc expects one input argument, which is the starting
   # letters of a store name.  It returns all stores that begin with
   # that string and also sets its return status to the number of rows
   # found.

   $rc = ct_sql($ws_db, qq!exec storename_proc "B"!);

   if ($rc != CS_SUCCEED) {
      ws_error("ct_sql() call failed on execution of stored
procedure.");
   }

   process_results();



   print "<P><BR><STRONG>Execute a stored procedure (sp_help) that
returns\n"
       . "results from several SELECT statements...</STRONG><P>\n";

   $rc = ct_sql($ws_db, "exec sp_help");

   if ($rc != CS_SUCCEED) {
      ws_error("ct_sql() call failed on execution of sp_help stored
procedure.");
   }

   process_results();



   print "<P><BR><STRONG>That's it for this example.  You may also
want to \n"
       . "try executing a stored procedure that returns some output \n"
       . "parameters to see how that result type is
handled.</STRONG><P>\n";

sub process_results {
```

```
# Define some local variables used only in this subroutine

my($ret, $result_type, $res_info, $num_cols, @row);

# ct_results() returns a $ret value to tell you if it executed
# ok or not and whether there are any more result types to get,
# and it also sets the argument $result_type to a valid result type
# that was passed from the server.

while (($ret = ct_results($ws_db, $result_type)) == CS_SUCCEED)
{
   RES_TYPE: {

      # CS_CMD_SUCCEED:
      # Returned when a SQL command that returns no rows
      # (i.e., INSERT/UPDATE/DELETE) is successful

      if ($result_type == CS_CMD_SUCCEED) {
         print "<P>Last command executed was successful.<BR>\n";
         $res_info = ct_res_info($ws_db, CS_ROW_COUNT);
         if ($res_info >= 0) {
            print "<P>${res_info} row(s) were affected.\n";
         }
         print "<P>\n";
         last RES_TYPE;
      }

      # CS_CMD_FAIL:
      # Returned when the last SQL command failed to execute

      if ($result_type == CS_CMD_FAIL) {
         print "<P>Execution of the last command FAILED.<P>\n";
         last RES_TYPE;
      }
```

```
# CS_ROW_RESULT:
# Returned when there are regular data rows from a SELECT
# statement or stored procedure available to fetch from the
# server.  You must fetch all these rows (or cancel the
# current result set) before you can retrieve additional
# result info or execute more commands.  The following
# code prints the resulting rows in HTML 3.0 tables, and also
# prints some header information.

if ($result_type == CS_ROW_RESULT) {

   print "<TABLE BORDER>\n";
   # First print the column heading information
   print "<TR><TH>", join("<TH>", ct_col_names($ws_db)), "\n";

   # Then, print the row data
   $num_cols = ct_res_info($ws_db, CS_NUMDATA);
   while (@row = ct_fetch($ws_db)) {
      print "<TR><TD>", join("<TD>", @row), "\n";
   }
   print "</TABLE>\n";
   last RES_TYPE;
}

# CS_COMPUTE_RESULT:
# Returned in between batches of CS_ROW_RESULT rows if a SELECT
# statement containing a "COMPUTE" clause is being processed.
# If there is more than one item being computed (for example,
# "compute sum(price), avg(price) by type"), then each computed
# item will be returned in a different element of the returned
# array.  If you return the results into an associative array,
# you can receive information on what type of operation (i.e.
# sum(), avg()) was performed for the compute, and what column
# number the operation was performed on.  This information is
```

```
# used below to print information related to the computer
# results for the current SELECT statement.  The data is
# printed in HTML 3.0 table format.

if ($result_type == CS_COMPUTE_RESULT) {

   print "<TABLE BORDER>\n";
   while (%row = ct_fetch($ws_db, 1)) {  # fetch into assoc \
   array
      print "<TR><TD>COMPUTE info:<TD>", join("<TD>", %row),\
        "\n";
   }
   print "</TABLE>\n";
   last RES_TYPE;
}

# CS_STATUS_RESULT:
# Returned as a result of executing a stored procedure that
# returns a status value.  The status value is a single
# integer value.  In the code below, the information is
# fetched into the @row array, but only the first array element
# should be set to the status value.

if ($result_type == CS_STATUS_RESULT) {
   while (@row = ct_fetch($ws_db)) {
      print "<P>Stored procedure return status: ", $row[0],\
        "<P>\n";
   }
   last RES_TYPE;
}

# CS_PARAM_RESULT:
# Returned after calling a stored procedure has one or more of
# its parameters defined as an "output" parameter.  A single
# row of an array is returned, with each parameter value
# returned in a different array cell.  In the code below, the
```

```
        # information is fetched into the @row array, but only one
        # row should be returned.

        if ($result_type == CS_PARAM_RESULT) {
           print "<P>Stored procedure output parameter value(s):\n";
           while (@row = ct_fetch($ws_db)) {
              print join("<BR>", @row), "\n";
           }
           print "<P>\n";
           last RES_TYPE;
        }

        # CS_CMD_DONE:
        # Returned after a SQL statement has been executed and its
        # results have been properly retrieved and processed by the
        # client.  Typically, you will just ignore this message and
        # not print any information.  In this example, an informational
        # message is printed so you can see when this result type is
        # returned.

        if ($result_type == CS_CMD_DONE) {
           print "<P>[CS_CMD_DONE result type returned]<P>\n";
           last RES_TYPE;
        }

        printf "<P>Unexpected result type returned: %s<P>\n", \
           $result_type;

     } # End of RES_TYPE
  } # end of ct_results() "while" loop

  return $ret;
}

</SYB>
```

A web.sql Reference Pages

This appendix contains the reference pages for the following web.sql routines:

- ct_callback A-2
- ct_cancel A-5
- ct_col_names A-7
- ct_col_types A-8
- ct_connect A-11
- ct_fetch A-14
- ct_fetch_parameters A-18
- ct_options A-22
- ct_res_info A-26
- ct_results A-30
- ct_rpc A-36
- ct_sql A-41
- ws_connect A-43
- ws_content_type A-45
- ws_error A-46
- ws_fetch_rows A-47
- ws_print A-50
- ws_rpc A-51
- ws_sql A-54

ct_callback

Function

Register a Client-Library message callback routine.

Syntax

```
ct_callback($type,$cb_func)
```

Parameters

$type – The type of callback routine of interest. The following table lists the symbolic values that are valid for *$type*:

Table A-1: Values for *$type* (ct_callback)

Value of *$type*:	Description
CS_CLIENTMSG_CB	**ct_callback** installs a client message callback.
CS_SERVERMSG_CB	**ct_callback** installs a server message callback.

$cb_func – A string specifying the name of a Perl sub-routine.

$cb_func is the callback routine to install.

Comments

- To install a callback routine, an application calls **ct_callback** with *$cb_func* as the callback to install. An application can install a new callback routine at any time. The new callback will automatically replace any existing callback.
- To remove an existing callback routine, an application can call **ct_callback** with *$cb_func* as an empty string (“”).
- The callback routines you install must appear in either the same **<SYB>** block as the **ct_callback** call installing them or in an earlier **<SYB>** block.
- After handling an event in your callback, if you want to resume processing of your HTS file, include as the last line processed:

  ```
  CS_SUCCEED;
  ```

- If you want to abort processing of your HTS file (for example, if the event indicated an unrecoverable error), include as the last line processed:

  ```
  CS_FAIL;
  ```

- Client callback routines receive six arguments when called:
  - Layer (a number)
  - Origin (a number)
  - Severity (a number)
  - Error number (a number)
  - Error message text (a string)
  - Operating system message text (a string)
- Server callback routines receive eight arguments when called:
  - Command
  - Error number (a number)
  - Severity (a number)
  - State (a number)
  - Line number (a number)
  - Server name (a string)
  - Stored procedure (a string, if applicable - otherwise empty)
  - Error message text (a string)

Examples

1. Callback routine to handle client error messages

```
sub ws_client_callback {
    my($layer, $origin, $severity, $number, $msg, $osmsg) = @_;
    printf "<P><B>! Open Client Error: %ld (Layer %ld,
        Origin %ld, Severity %ld)</B>\n", $number,
        $layer, $origin, $severity;
    printf "<P><B><blockquote>%s</blockquote></B>\n", $msg;
    if (defined($osmsg)) {
        printf "<P><B>! Operating System Error:<blockquote>
            %s</blockquote></B>\n", $osmsg;
    }
    printf "<P>\n";
    CS_SUCCEED;
}
```

2. Callback routine to handle server error message:

```
sub ws_server_callback {
    my($cmd, $number, $severity, $state, $line,
       $server, $proc, $msg) = @_;
    if ($severity != 10) {
        # This is an error
        printf "<P><B>! Server Error: %ld (Severity %ld,
            State %ld, Line %ld)</B>\n", $number,
            $severity, $state, $line;
        if (defined($server)) {
            printf "<P><B>! Server '%s'</B>\n", $server;
        }
        if (defined($proc)) {
            printf "<P><B>! Procedure '%s'</B>\n", $proc;
        }
        printf "<P><B><blockquote>%s</blockquote></B>\n", $msg;
        printf "<P>\n";
    }
    elsif (! defined $ws__suppress_server_message{$number}) {
        # This is an informational message
        if ($number) {
            print "Server message $number: $msg\n";
        }
        else {
            print "$msg\n";
        }
    }

    CS_SUCCEED;
}
```

See Also

ct_connect

ct_cancel

Function

Cancel a command or the results of a command.

Syntax

```
$rc = ct_cancel($handle, $type)
```

Parameters

$handle – A pre-defined, valid connection to a database server and login.

$type – The type of cancel. The following table lists the symbolic values that are valid for *$type*

Table A-2: Values *for $type* (ct_cancel)

Value of *$type*:	Description
CS_CANCEL_ALL	**ct_cancel** sends an attention to the server, instructing it to cancel the current commands. Client-Library immediately discards all results generated by the command.
CS_CANCEL_ATTN	**ct_cancel** sends an attention to the server, instructing it to cancel the current command. The next time the application reads from the server, Client-Library discards all results generated by the canceled command.
CS_CANCEL_CURRENT	Cancels (discards) the current result set only.

Return Values

The following table lists the return values of **ct_cancel**.

Table A-3: Return values of ct_cancel

Return Value:	Indicates:
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Comments

- Canceling a command with CS_CANCEL_ALL is equivalent to sending an attention to the server, instructing it to halt execution of the current command. When a command is canceled, any results generated by it are no longer available to an application.
- Canceling the current result set with CS_CANCEL_CURRENT is equivalent to discarding a buffer's worth of results. Once results are canceled, they are no longer available to an application. If all the result sets have not been completely processed, subsequent result sets remain available.

Canceling a Command

To cancel the current commands and all generated results, an application calls **ct_cancel** with *$type* as CS_CANCEL_ALL.

Both types of cancels return CS_SUCCEED immediately, without sending an attention to the server, if no command is in progress.

CS_CANCEL_ALL leaves the internal database handle in a "clean" state, available for use in another operation.

If **ct_cancel** is called, some of the other Client-Library routines will return CS_CANCELED the next time they are called.

The Client-Library routines that can return CS_CANCELED are:

- **ct_cancel**(CS_CANCEL_CURRENT)
- **ct_fetch**
- **ct_options**
- **ct_results**

Example

```
if ($ret == CS_FAIL) {
    # A connection error

    ct_cancel(CS_CANCEL_ALL);
    ws_error("A database connection error
occurred");
}
```

See Also

ct_fetch, **ct_options**, **ct_results**

ct_col_names

Function

Retrieves an array of column names for the current result set.

Syntax

`@names = ct_col_names($handle)`

Parameters

$handle – A pre-defined, valid connection to a database server and login.

Return Values

@names – An array of column names for the current result set.

Comments

- If the current result set is not for a statement that returns row data (for example, a **select** statement), an empty array is returned.

Example

```
#....code deleted....
if ($restype == CS_ROW_RESULT)
        {
        if (!defined($my_fmt_once))
            {
            my @cols = ct_col_names($dbh);
            for ($i = 0; $i < scalar(@cols); $i++)
#....code deleted....
```

See Also

ct_col_types

ct_col_types

Function

Retrieves an array of column types.

Syntax

```
%types | @types = ct_col_types($handle [,$doassoc])
```

Parameters

$handle – A pre-defined, valid connection to a database server and login.

$doassoc – A flag that determines whether the routine returns an associative array or a regular array.

The following table lists the symbolic values that are valid for *$doassoc*:

Table A-4: Values for *$doassoc* (ct_col_types)

Values of *$doassoc*:	Description:
1	Returns an associative array of column types. Array elements are indexed by the column names.
Not specified	Returns an array of column types.

Return Values

The following table lists the return values of ct_col_types.

Table A-5: Return values of ct_col_types

Return Value:	Indicates:
%types	An associative array of column types. The array indices are the column names.
@types	An array of column types, where the possible column type values are shown in Table A-6.

The following table lists the column type values of **ct_col_types**.

Table A-6: Column type values of ct_col_types

Return value	Corresponding Server Data Type
CS_BINARY_TYPE	*binary,* *varbinary*
CS_BIT_TYPE	*boolean*
CS_BOUNDARY_TYPE	*sensitivity,* *boundary*
CS_CHAR_TYPE	*char,* *varchar*
CS_DATETIME_TYPE	*datetime*
CS_DECIMAL_TYPE	*decimal*
CS_FLOAT_TYPE	*float*
CS_ILLEGAL_TYPE	NONE
CS_IMAGE_TYPE	*image*
CS_INT_TYPE	*int*
CS_LONGBINARY_TYPE	NONE
CS_LONGCHAR_TYPE	NONE
CS_MONEY4_TYPE	*smallmoney*
CS_MONEY_TYPE	*money*
CS_NUMERIC_TYPE	*numeric*
CS_REAL_TYPE	*real*
CS_SENSITIVITY_TYPE	*sensitivity*
CS_SMALLINT_TYPE	*smallint*
CS_TEXT_TYPE	*text*
CS_TINYINT_TYPE	*tinyint*
CS_VARBINARY_TYPE	NONE
CS_VARCHAR_TYPE	NONE

Example

```
<SYB TYPE=PERL>

#....code deleted....

print "<P><B>Get an associative array of column types using
ct_col_types(1)...</B></P>\n";

%types = ct_col_types($ws_db,1);


print "<TABLE border> \n";

print "<P>Column Names : Column Types</P>\n";

while(($key, $value) = each(%types)) {

     &prttype($value, $key);

}    # end of while

print "</TABLE>\n";

#....code deleted....
```

See Also

ct_col_names

ct_connect

Function

Connect to a server.

Syntax

```
$handle = ct_connect($userid, $passwd,
           $server,[$appname])
```

Parameters

$userid – The user's database server login name.

$passwd – The user's database server password.

$server – The name of the database server to connect to. *$server* is the name given to the server in the *interfaces* (UNIX) or *sql.ini* (NT) file on the application's host machine. ct_connect looks up *$server* in the *interfaces* or *sql.ini* file to determine how to connect to this server.

➤ ***Note***

An *interfaces* file may not be used on some platforms. For information on whether your platform uses an *interfaces* file, see the *Open Client/Server Supplement* for your platform.

$appname – The name of the application used when web.sql logs on to the server. This is an optional parameter.

Return Values

$handle – A "handle" for the database server connection. You use this handle to specify the database server connection to use in subsequent calls to web.sql routines.

If ct_connect fails, it returns a Perl undefined value.

Comments

- Common reasons for a ct_connect failure include:
  - Server name not found in *interfaces* file.
  - Unknown host machine name.

- SQL Server is unavailable or does not exist.
- Login incorrect.
- Could not open *interfaces* file.

- When a connection attempt is made between a client and a server, there are two ways in which the process can fail (assuming that the system is correctly configured):
  - The machine that the server is supposed to be on is running correctly and the network is running correctly.

    In this case, if there is no server listening on the specified port, the machine that the server is supposed to be on signals the client, via a network error, that the connection cannot be formed. Regardless of the login timeout value, the connection fails.
  - The machine that the server is on is down.

    In this case, the machine that the server is supposed to be on does not respond. Because "no response" is not considered to be an error, the network does not signal the client that an error has occurred. However, if a login timeout period has been set, a timeout error occurs when the client fails to receive a response within the set period.

For more information about **ct_connect**, see "Connecting to a Server with the web.sql Client-Library API" in Chapter 3, "Using Perl in HTS Files".

Multiple QUERY Entries in an *Interfaces* File

It is possible to set up an *interfaces* file so that if **ct_connect** fails to establish a connection with a server, it attempts to establish a connection with an alternate server.

See the **ct_connect** section in the *Sybase Open Client Client-Library Reference Manual* for more information.

Example

```
$uid = "userid";
$pwd = "passwd";
$srv = "SYB_INET";
$myHandle = ct_connect($uid, $pwd, $srv);
# You can now use $myHandle as a database handle
```

See Also

ws_connect, ct_options

ct_fetch

Function

Fetch result data.

Syntax

```
@row | %row = ct_fetch($handle [,$doassoc])
```

Parameters

$handle – A pre-defined, valid connection to a database server and login.

$doassoc – A flag that determines whether the routine retrieves rows as an associative array or as a regular array.

The following table lists the values that are valid for *$doassoc*:

Table A-7: Values for *$doassoc* (ct_fetch)

Values of *$doassoc*:	Description:
1	Row data is returned in an associative array.
Not Specified	Row data is returned in a regular array.

Return Values

The following table lists the return values of ct_fetch.

Table A-8: Return values of ct_fetch

Return Value:	Indicates:
`%row`	An associative array containing a single row of data from the result set. The array indices are the column names.
`@row`	A regular array containing a single row of data from the result set.

Comments

- "Result data" is an umbrella term for all types of data that a server can return to an application. These types of data include:
  - Regular data rows from a **select** statement.
  - Return parameters. Types of data that are returned as parameters include message parameters, stored procedure return parameters, and extended error data.
  - Stored procedure status values.
  - Compute rows from **select** statements.

  ct_fetch is used to fetch all of these types of data.
- Conceptually, result data is returned to an application in the form of one or more rows that make up a "result set".

  Row result sets can contain more than one row. For example, a row result set might contain a hundred rows.
- If an application does not cancel a result set, it must completely process the result set by calling **ct_fetch** as long as **ct_fetch** continues to indicate that rows are available. **ct_fetch** will return an empty array when there are no more rows to fetch in the current result set.

Fetching Return Parameters

Several types of data can be returned to an application as a parameter result set, including:

- Stored procedure return parameters
- Message parameters

A return parameter result set consists of a single row with a number of columns equal to the number of return parameters.

Fetching a Return Status

A stored procedure return status result set consists of a single row with a single column, the return status.

Fetching Compute Rows

Compute rows result from the **compute** clause of a **select** statement.

A compute row result set consists of a single row with a number of columns equal to the number of aggregate operators in the **compute** clause that generated the row.

Each compute row is considered to be a distinct result set.

Example

The following example of an *.hts* file processes a simple database query using the web.sql Client-Library API. Note that this example demonstrates how to use both **ct_fetch** and **ws_fetch_rows** to fetch the rows.

```
<SYB TYPE=PERL>
# Build the SELECT statement
$sql_stmt = qq!select type, price
            frm pubs2..title
            order by type
            compute sum(price) by type!;
# Send the command
if (($rc = ct_sql( $ws_db, $sql_stmt ) ) != CS_SUCCEED) {
    ws_error ("Unable to process database request.");
}
#Process the results from the server.
while (($ret = ct_results($ws_db, $result_type)) == CS_SUCCEED) {
    # Equivalent of switch statement in Perl.
    RESULTS: {
        # Simply ignore report of command completion
        if ($result_type == CS_CMD_DONE) {
            last RESULTS; # Jump to end of RESULTS block.
        }
        # Simply ignore notification of unsuccessful completion
        if ($result_type == CS_CMD_FAIL) {
            last RESULTS; # Jump to end of RESULTS block.
         }
        # Fetch and print row results
        if ($result_type == CS_ROW_RESULT) {
            while (@row = ct_fetch($ws_db)) {
                print join(" ", @row), "<BR>\n";
             }
```

```
                last RESULTS; # Jump to end of RESULTS block.
            }
            #Fetch and print compute results
            if ($result_type == CS_COMPUTE_RESULT) {
                ws_fetch_rows($ws_db);
                last RESULTS; # Jump to end of RESULTS block.
            }
        }
    }
    if ($ret == CS_FAIL) {
        # A connection error occurred. Clean up connection state.
        ct_cancel(CS_CANCEL_ALL);
        ws_error("A database connection error occurred");
    }
    </SYB>
```

See Also

ct_results, ws_fetch_rows

ct_fetch_parameters

Function

Updates the output parameters supplied to **ct_rpc**.

Syntax

```
ct_fetch_parameters($handle)
```

Parameters

$handle – A pre-defined, valid connection to a database server and login.

Comments

- You use **ct_fetch_parameters** to update the output parameter variables supplied to **ct_rpc**. **ct_fetch_parameters** fills in the variables for any references that were passed to **ct_rpc** in the parameter list.
- This routine must be called *after* **ct_results** has returned the result type CS_PARAM_RESULT.
- If output parameters were not passed to **ct_rpc**, the **ct_fetch_parameters** routine is not needed.

Example

```
<SYB TYPE=PERL>


$price = 22.55;

$lname = "Smith";


# Calling stored procedure

$status = ct_rpc($ws_db, "tempP2",

      {

      '@result_id' => \$res_id,

      '@result_price' => { "type" => &CS_MONEY_TYPE, "value" =>
\$res_price, "precision" => CS_DEF_PREC , "scale" =>
CS_DEF_SCALE },

      '@lastname' => $lname,

      '@firstname' => "First",
```

```
      '@price' => $price,
      });
if ($status  != &CS_SUCCEED)
 {
    print "<H3>Error: Bad status($status) returned from CT_RPC
tempP2</H3>\n";
    die "ct_rpc failed";
 }
else
 {
   print "<H3>Success: ct_rpc succeeded. Now check for output
parameters.</H3>\n";
 }
while (($ret = ct_results($ws_db, $result_type)) == &CS_SUCCEED)
{
      if ($result_type == &CS_PARAM_RESULT)
      {
        $rc = ct_fetch_parameters($ws_db);
        if ($rc != &CS_SUCCEED)
        {
          die "ct_fetch_parameters failed \n";
        }
        if ($res_name == $lname && $userid == $res_id && $price
== $res_price)
        {
              print "<P><H3>ct_fetch_parameters returns=
$res_name and $res_id and $res_price <P></H3>";
            print "<H3>SUCCESSFUL testcase</H3>";
        }
          next;
      }
      if ($result_type == &CS_STATUS_RESULT)
      {
        $status = ct_fetch($ws_db);
```

```
         ct_fetch($ws_db);
            next;
        }
        if  ($result_type == &CS_ROW_RESULT)
        {
             while (@row = ct_fetch($ws_db))
              {
              if ($srv_severity > 10)
                  {
                  die "Server failed: $srv_msgtext\n";
                  }
              print "<P>ct_results returns CS_ROW_RESULT: ";
              print @row;
              print "\n";
              }
             next;
        }
        if ($result_type == &CS_CMD_DONE) {
           print  "<P>ct_results returns CS_CMD_DONE\n";
             next;
        }
        if ($result_type == &CS_CMD_SUCCEED)
           {
             print "<P>ct_results returns CS_CMD_SUCCEED\n";
             next;
           }
        if ($result_type == &CS_CMD_FAIL) {
          print "<P>ct_results() server error with result type
CS_CMD_FAIL.</P>\n";
        }
}
if ($ret == CS_FAIL) {
        # A connection error
```

```
        ct_cancel(CS_CANCEL_ALL);
        ws_error("A database connection error occurred");
}

</SYB>
```

See Also

ct_rpc

ct_options

Function

Set, retrieve, or clear the values of server query-processing options.

Syntax

```
ct_options($handle, $action, $option, $param, $type)
```

Parameters

$handle – A pre-defined, valid connection to a database server and login.

$action – One of the following symbolic values:

Table A-9: Values for *$action* (ct_options)

Value of *$action*:	Description:
CS_SET	**ct_options** sets the option. Result data is substituted for *$rc*.
CS_GET	**ct_options** retrieves the option. Result data is substitute for *$param*. An application can use **ct_options** to retrieve options from SQL Server System 10 or 11.
CS_CLEAR	**ct_options** clears the option by resetting it to its default value. Default values are determined by the server to which an application is connected. Result data is substituted for *$rc*.

$option – The server option of interest. Table A-10 on page A-23 lists the symbolic values that are valid for *$option*.

$param – All options take parameters.

When setting an option, *$param* can be an integer value, or a character string. When clearing an option *$param* must be 0.

$type – CS_INT_TYPE if *$param* is an integer; CS_CHAR_TYPE if *$param* is a character string.

Summary of Parameters

The following table lists all the possible parameters for **ct_options**.

Table A-10: Summary of parameters (ct_options)

Value of *$option*:	*$param* is:	Valid values for *$param*:	Defaults to:
CS_OPT_ANSINULL	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_ANSIPERM	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_ARITHABORT	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_ARITHIGNORE	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_AUTHOFF	A string value representing an authority level.	A string value. Possible values include "sa", "sso", and "oper".	Not applicable
CS_OPT_AUTHON	A string value representing an authority level.	A string value. Possible values include "sa", "sso", and "oper".	Not applicable
CS_OPT_CHAINXACTS	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_CURCLOSEON XACT	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_CURREAD	A string value representing a read level label.	A string value.	NULL
CS_OPT_CURWRITE	A string value representing a write level label.	A string value.	NULL
CS_OPT_DATEFIRST	A symbolic value representing the day to use as the first day of the week.	CS_OPT_SUNDAY, CS_OPT_MONDAY, CS_OPT_TUESDAY, CS_OPT_WEDNESDAY, CS_OPT_THURSDAY, CS_OPT_FRIDAY, CS_OPT_SATURDAY	For us_english, the default is CS_OPT_ SUNDAY.
CS_OPT_DATEFORMAT	A symbolic value representing the order of year, month, and day to be used in datetime values.	CS_OPT_FMTMDY, CS_OPT_FMTDMY, CS_OPT_FMTYMD, CS_OPT_FMTYDM, CS_OPT_FMTMYD, CS_OPT_FMTDYM"	For us_english, the default is CS_OPT_ FMTMDY.
CS_OPT_FIPSFLAG	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_FORCEPLAN	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE

Table A-10: Summary of parameters (ct_options) (continued)

Value of *$option*:	*$param* is:	Valid values for *$param*:	Defaults to:
CS_OPT_FORMATONLY	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_GETDATA	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_IDENTITYOFF	A string value representing a table name.	A string value.	NULL
CS_OPT_IDENTITYON	A string value representing a table name.	A string value.	NULL
CS_OPT_ISOLATION	A symbolic value representing the isolation level.	CS_OPT_LEVEL1, CS_OPT_LEVEL3	CS_OPT_ LEVEL1
CS_OPT_NOCOUNT	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_NOEXEC	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_PARSEONLY	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_QUOTED_ IDENT	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_RESTREES	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_ROWCOUNT	The maximum number of regular rows to return.	An integer value. 0 means all rows are returned.	0 means all rows are returned.
CS_OPT_SHOWPLAN	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_STATS_IO	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_STATS_TIME	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_STR_RTRUNC	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_TEXTSIZE	The length, in bytes, of the longest text or image value the server should return.	An integer value.	32,768 bytes.
CS_OPT_TRUNCIGNORE	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE

Return Values

The following table lists the possible return values for **ct_options**.

Table A-11: Return values of ct_options

Return Value:	Indicates:
$rc	The success of the **ct_options** call. CS_SUCCEED indicates that the routine succeeded. CS_FAIL indicates that the routine failed.
$result	The value of *$param* specified in the routine. The result is an empty string if you are setting the parameter.

Comments

- Although query-processing options can be set and cleared through the Transact-SQL **set** command, it is recommended that Client-Library applications use **ct_options** instead. This is because **ct_options** allows an application to check the status of an option, which cannot be done through the **set** command.
- An application can use **ct_options** to change server options only for a single connection at a time. The connection must be open and must not have active commands or pending results.

Example

```
# Set default maximum rowcount
ct_options($PHONE_DB, CS_SET, CS_OPT_ROWCOUNT, 500, CS_INT_TYPE);
```

See Also

ct_connect

ct_res_info

Function

Retrieves information about the current result set or command.

Syntax

```
$res_info = ct_res_info($handle, $info_type)
```

Parameters

$handle – A pre-defined, valid connection to a database server and login.

$info_type – The type of information to return. Table A-12 lists the symbolic values that are valid for *$info_type*.

Table A-12: Summary of parameters for ct_res_info

Value of *$info_type*:	ct_res_info returns:	The information is available after ct_results sets its *$result_type* parameter to:
CS_MSGTYPE	An integer representing the ID of the message that makes up the current result set.	CS_MSG_RESULT
CS_NUM_COMPUTES	The number of compute clauses in the current command.	CS_COMPUTE_RESULT
CS_NUMDATA	The number of items in the current result set.	CS_COMPUTE_RESULT, CS_COMPUTEFMT_RESULT, CS_PARAM_RESULT, CS_ROW_RESULT, CS_ROWFMT_RESULT, CS_STATUS_RESULT
CS_NUMORDER COLS	The number of columns specified in the **order- by** clause of the current command.	CS_ROW_RESULT
CS_ROW_COUNT	The number of rows affected by the current command.	CS_CMD_DONE, CS_CMD_FAIL CS_CMD_SUCCEED

Return Values

Table A-13: Return values of ct_res_info

Return Value:	Indicates:
$res_info	The return of requested information. (See Table A-12 for possible values.)

Comments

- A result set is a collection of a single type of result data. Result sets are generated by commands. For more information on result sets, see the ct_results reference page.
- Typically, an application calls ct_res_info with *$info_type* as CS_NUMDATA, to determine the number of items in a result set.

Retrieving the Number of Rows for the Current Command

To determine the number of rows affected by the current command, call ct_res_info with *$info_type* as CS_ROW_COUNT.

An application can retrieve a row count after ct_results sets its *result_type* parameter to CS_CMD_SUCCEED or CS_CMD_FAIL.

If the command is one that executes a stored procedure, for example a Transact-SQL exec language command or a remote procedure call command, ct_res_info sets *$res_info* to the number of rows affected by the last statement in the stored procedure that affects rows.

Retrieving the Command Number for Current Results

The Client-Library keeps track of the command number by counting the number of times ct_results returns CS_CMD_DONE.

An application's first call to ct_results following a ct_sql call sets the command number to 1. After this, it is incremented each time ct_results is called after returning CS_CMD_DONE.

Retrieving a Message ID

To retrieve a message ID, call ct_res_info with *$info_type* as CS_MSGTYPE.

Servers can send messages to client applications. Messages are received in the form of "message result sets." Message result sets contain no fetchable data, but rather have an ID number.

Messages can also have parameters. Message parameters are returned to an application as a parameter result set, immediately following the message result set.

Retrieving the Number of Compute Clauses

To determine the number of **compute** clauses in the command that generated the current result set, call **ct_res_info** with *$info_type* as CS_NUM_COMPUTES.

A Transact-SQL **select** statement can contain **compute** clauses that generate compute result sets.

Retrieving the Number of Result Data Items

To determine the number of result data items in the current result set, call **ct_res_info** with *$info_type* as CS_NUMDATA.

Results sets contain result data items. Row, and compute result sets contain columns, a parameter result set contains parameters, and a status result set contains a status. The columns, parameters, and status are known as "result data items".

A message result set does not contain any data items.

Retrieving the Number of Columns in an Order-By Clause

To determine the number of columns in a Transact-SQL **select** statement's **order by** clause, call **ct_res_info** with *$info_type* as CS_NUMORDERCOLS.

A Transact-SQL SELECT statement can contain an **order by** clause, which determines how the rows resulting from the SELECT statement are ordered on presentation.

Example

```
#....code deleted....
if ($result_type == CS_CMD_SUCCEED) {
      print "<P>Last command executed was successful.<BR>\n";
      $res_info = ct_res_info($ws_db, CS_ROW_COUNT);
      if ($res_info >= 0) {
            print "<P>${res_info} row(s) were affected.\n";
      }
      print "<P>\n";
```

```
last RES_TYPE;
```

See Also

ct_results

ct_results

Function

Set up result data to be processed.

Syntax

```
$rc = ct_results($handle,$result_type)
```

Parameters

$handle – A pre-defined, valid connection to a database server and login.

$result_type – An integer variable that ct_results sets to indicate the current type of result available from the database server. This parameter is actually an output parameter that is set when ct_results is called.

The following table lists the possible values of *$result_type*:

Table A-14: Values for *$result_type* (ct_results)

	Value of *$result_type*:	Indicates:	Return set contains:
Values that indicate command status	CS_CMD_DONE	The results of a logical command have been completely processed.	No results.
	CS_CMD_FAIL	The server encountered an error while executing a command.	No results.
	CS_CMD_SUCCEED	The success of a command that returns no data, such as a language command containing a Transact-SQL INSERT statement.	No results.

Table A-14: Values for *$result_type* (ct_results) (continued)

	Value of *$result_type*:	Indicates:	Return set contains:
Values that indicate fetchable results	CS_COMPUTE_RESULT	Compute row results.	A single row of compute results.
	CS_PARAM_RESULT	Return parameter results.	A single row of return parameters.
	CS_ROW_RESULT	Regular row results.	Zero or more rows of tabular data.
	CS_STATUS_RESULT	Stored procedure return status results.	A single row containing a single status.
Values that indicate information is available	CS_MSG_RESULT	Message arrival.	No fetchable results. An application can call **ct_res_info** to get the message's ID. Parameters associated with the message, if any, are returned as a separate parameter result set.

Return Values

Table A-15: Return Values of ct_results

Return Value *$rc*	Indicates:
CS_SUCCEED	A result set is available for processing.
CS_END_RESULTS	All results have been completely processed.
CS_FAIL	The routine failed; any remaining results are no longer available. If **ct_results** returns CS_FAIL, an application must call **ct_cancel** with *$type* as CS_CANCEL_ALL before using the affected command structure to send another command.
CS_CANCELED	Results have been canceled.

Comments

- An application calls **ct_results** after sending a command to the server via **ct_sql**, and before reading the results of that command (if any) via **ct_fetch** or **ws_fetch_rows**.
- "Result data" is an umbrella term for all the types of data that a server can return to an application. These types of data include:
  - Regular rows

- Return parameters
- Stored procedure return status numbers
- Compute rows
- Messages

ct_results is used to set up all of these types of results for processing.

➤ ***Note***

Message results are not server error and informational messages. See the **Error and Message Handling** topics page for a discussion of error and informational messages.

- Result data is returned to an application in the form of a "result set." A result set includes only a single type of result data. For example, a regular row result set contains only regular rows, and a return parameter result set contains only return parameters.

The ct_results Loop

Because a command can generate a result set that spans multiple buffers, an application must call ct_results as long as it continues to return CS_SUCCEED, indicating that results are available. The simplest way to do this is in a loop that terminates when ct_results fails to return CS_SUCCEED. After the loop, an application can use a case-type statement to test ct_results' final return code to determine why the loop terminated.

Results are returned to an application in the order in which they are produced. However, this order is not always easy to predict. For example, when an application calls a stored procedure that in turn calls another stored procedure, the application might receive a number of regular row and compute row result sets, as well as a return parameter and a return status result set. The order in which these results are returned depends on how the stored procedures are written.

For this reason, it is recommended that an application's ct_results loop be coded so that control drops into a case-type statement that handles all types of results that can be received. The return parameter *$result_type* indicates symbolically what type of result data the result set contains.

When are the Results of a Command Completely Processed?

ct_results sets *$result_type* to CS_CMD_DONE to indicate that the results of a "logical command" have been completely processed.

For example, suppose a Client-Library language command contains the following Transact-SQL statements:

```
select type, price
    from titles
    order by type, price
    compute sum(price) by type

select type, price, advance
    from titles
    order by type, advance
    compute sum(price), max(advance) by type
```

When calling ct_results to process the results of this language command, an application would see the following *$result_type* values:

Table A-16: Values for *$result_type* (ct_results loop)

CS_ROW_RESULT CS_COMPUTE_RESULT	Row and compute results from the first select, repeated as many times as the value of the type column changes.
CS_CMD_DONE	Indicates that the results of the first query have been processed.
CS_ROW_RESULT CS_COMPUTE_RESULT	Row and compute results from the second select, repeated as many times as the value of the type column changes.
CS_CMD_DONE	Indicates that the results of the second query have been processed.

A *$result_type* of CS_CMD_SUCCEED or CS_CMD_FAIL is immediately followed by a *$result_type* of CS_CMD_DONE.

A connection has **pending results** if it has not processed all of the results generated by a Client-Library command. Usually, an application cannot send a new command on a connection with pending results.

Canceling Results

To cancel all remaining results from a command (and eliminate the need to continue calling ct_results until it fails to return CS_SUCCEED), call ct_cancel with *$type* as CS_CANCEL_ALL.

To cancel only the current results, call **ct_cancel** with *$type* as CS_CANCEL_CURRENT.

Special Kinds of Result Sets

A message result set contains no actual result data. Rather, a message has an "ID". An application can call **ct_res_info** to get a message's ID. In addition to an ID, messages can have parameters. Message parameters are returned to an application as a parameter result set, immediately following the message result set.

ct_results and Stored Procedures

A run-time error on a language command containing an **execute** statement generates a *$result_type* of CS_CMD_FAIL. For example, this occurs if the procedure named in the **execute** statement cannot be found.

A run-time error on a statement inside a stored procedure does *not* generate a CS_CMD_FAIL, however. For example, if the stored procedure contains an **insert** statement and the user does not have insert permission on the database table, the **insert** statement fails, but **ct_results** still returns CS_SUCCEED. To check for run-time errors inside stored procedures, examine the procedure's return status number, which is returned as a return status result set immediately following the row and parameter results, if any, from the stored procedure. If the error generates a server message, it is also available to the application.

Example

The general form of the **ct_results** loop is:

```
# Call ct_sql to send commands to the server.
# Process the results from the server.
while (($ret= ct_results($ws_db, $result_type)) == CS_SUCCEED)
{
    # Equivalent of switch statement in Perl.
    RESULTS:
    {
        # Test for each result type code
        if ($result_type == ... )
            {
                 # Process the result for that return code
```

```
                  last RESULTS; # Jump to end of RESULTS block.
              }
          # Etc...
      }
  }
  if ($ret == CS_FAIL)
  {
      # A connection error occurred. Clean up connection state.
      ct_cancel($ws_db, CS_CANCEL_ALL);
      ws_error("A database connection error occurred");
  }
```

See Also

ct_cancel, ct_fetch, ct_sql, ws_fetch_rows

ct_rpc

Function

Calls a stored procedure that resides on remote servers.

Syntax

ct_rpc($db_handle, $rpc_name, @params or **%params)**

Parameters

$db_handle – A pre-defined, valid connection to a database server and login.

$rpc_name - The name of the RPC you want to call.

%params or *@params*- The list of parameters for the RPC, including output parameters.

You can call stored procedures with two styles of parameters: named and positional.

- If you use the named parameter style, *%params*, you must pass the parameter name and the value assigned to that name to the stored procedure by enclosing them in { }. The list of parameters for the RPC is passed as a Perl 5 reference to an associative (hash) array.
- If you use the positional parameter style, *@params*, you only pass the argument value by enclosing it in [ ]. You pass the argument in the first position as the first parameter to the stored procedure, the argument in the second position as the second parameter, the argument in the third as the third parameter, and so on. The list of parameters for the RPC is passed as a Perl 5 reference to an array.

You can specify output parameters with a reference to a scalar variable. Scalar data can be passed as an integer, a float, or a character string, depending on the Perl datatype.

To pass or receive a particular Sybase datatype other than integer, you must use a datatype-reference structure. For example, if you want to pass an integer as a character string, you must use a datatype-reference to enforce the behavior. The datatype-references can be written as follows:

```
{
'type'       =>  CS_<datatype>,
'value'      =>  <scalar> or <scalar-ref>,
```

```
'scale'      =>  CS_MAX_SCALE or CS_DEF_SCALE,
'precision'  =>  CS_MAX_PREC or CS_DEF_PREC
}
```

For a list of the CS_ datatypes, see Appendix B, “RPC Datatype Summary.”

Comments

- When calling **ct_rpc**, you must also call **ct_results** and **ct_fetch** to get any results that are returned.
- If you supply output parameters to **ct_rpc**, **ct_results** returns CS_PARAM_RESULT. In this case, you should call **ct_fetch_parameters**, which completes the variables for the references that were passed in the parameter list.
- When stored procedures have no arguments, you can leave the argument off. Having an empty argument list is the same as not having an argument list. For example, with **ct_rpc** you could have an argument list, have an empty argument list, or have no arguments, as follows: **ct_rpc($ws_db,"name",[$arg]);**or **ct_rpc($ws_db,"name",[]);** or **ct_rpc($ws_db,"name");**.

Example Using Named Parameters (Associative Array)

```
<SYB TYPE=PERL>


$price = 22.55;

$lname = "Smith";


# Calling stored procedure using named parameters (associative
# array)

$status = ct_rpc($ws_db, "tempP2",

      {

      '@result_id' => \$res_id,

      '@result_price' => { "type" => &CS_MONEY_TYPE, "value" =>
$res_price, "precision" => CS_DEF_PREC , "scale" => CS_DEF_SCALE
},

      '@lastname' => $lname,

      '@firstname' => "First",

      '@price' => $price,
```

```
      });
if ($status  != &CS_SUCCEED)
 {
    print "<H3>Error: Bad status($status) returned from CT_RPC
tempP2</H3>\n";
    die "ct_rpc failed";
 }
else
 {
   print "<H3>Success: ct_rpc succeeded. Now check for output
parameters.</H3>\n";
 }
while (($ret = ct_results($ws_db, $result_type)) == &CS_SUCCEED)
{
      if ($result_type == &CS_PARAM_RESULT)
      {
# Fetching return parameters from RPC

        $rc = ct_fetch_parameters($ws_db);
        if ($rc != &CS_SUCCEED)
        {
          die "ct_fetch_parameters failed \n";
        }
        if ($userid == $res_id)
        {
              print "<P><H3>ct_fetch_parameters returns=
$res_name and $res_id and $res_price <P></H3>";
           print "<H3>SUCCESSFUL testcase</H3>";
        }
          next;
      }
      if ($result_type == &CS_STATUS_RESULT)
      {
        $status = ct_fetch($ws_db);
```

```
        ct_fetch($ws_db);
           next;
      }
      if  ($result_type == &CS_ROW_RESULT)
      {
            while (@row = ct_fetch($ws_db))
             {
             if ($srv_severity > 10)
                 {
                 die "Server failed: $srv_msgtext\n";
                 }
             print "<P>ct_results returns CS_ROW_RESULT: ";
             print @row;
             print "\n";
             }
            next;
      }
      if ($result_type == &CS_CMD_DONE) {
         print  "<P>ct_results returns CS_CMD_DONE\n";
            next;
      }
      if ($result_type == &CS_CMD_SUCCEED)
         {
            print "<P>ct_results returns CS_CMD_SUCCEED\n";
            next;
         }
      if ($result_type == &CS_CMD_FAIL) {
        print "<P>ct_results() server error with result type
CS_CMD_FAIL.</P>\n";
      }
}
if ($ret == CS_FAIL) {
      # A connection error
```

```
      ct_cancel(CS_CANCEL_ALL);
      ws_error("A database connection error occurred");
}

</SYB>
```

Example Using Positional Parameters (Unnamed Array)

```
# Calling stored procedure using positional parameters
# (unnamed array)
#. . . code deleted. . .
$status = ct_rpc($ws_db, "tempProc",
      [
      \$result,
      “Green”
      “Larry”
      { "type" => CS_MONEY_TYPE, "value" => $res_price,
"precision" => CS_MAX_PREC , "scale" => CS_MAX_SCALE }]);
#. . . code deleted. . .
```

See Also

ct_results, ct_fetch, ct_fetch_parameters, ws_rpc

ct_sql

Function

Send one or more SQL statements to the SQL server.

Syntax

```
$rc = ct_sql($handle, $query)
```

Parameters

$handle – A pre-defined, valid connection to a database server and login.

$query – One or more SQL statements.

Return Values

The following table lists the return values for **ct_sql**.

Table A-17: Return Values of ct_sql

Return Value:	Indicates:
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_CANCELED	The operation was canceled. Only a CS_CANCEL_CURRENT type of cancel can be canceled.

Comments

- *$query* may contain Perl variables.
- To send multiple SQL statements to the SQL server in a single batch, use newline characters (“`\n`”) to separate individual statements.
- A **use** command in a command batch does not affect subsequent commands in that batch. Send **use** as a separate batch from following batches for the specified databases.

Example

The following example sends a set of SQL statements:

```
$sql_stmt = qq!select au_fname, au_lname, city, state
```

```
            from pubs2..authors
            where state = upper("$state")
            select pub_name, city, state
            from pubs2..publishers
            where state = upper("$state")!;
if (($rc = ct_sql( $ws_db, $sql_stmt ) ) != CS_SUCCEED)
{
    ws_error ("Unable to process database request.");
}
#Process the result sets...
```

See Also

ct_fetch, ct_results, ws_fetch_rows, ws_rpc

ws_connect

Function

Returns a connection handle based on a database connection that was specified in the database access map (*.websql.pl)*.

Syntax

```
$handle = ws_connect([$connection_name])
```

Parameters

$connection_name – The name of a database connection defined in the database access map in *.websql.pl*.

Return Value

$handle – A connection handle for use in other web.sql API calls.

Comments

- To use a connection name in ws_connect, the name must be associated with an HTS file or a directory that includes the current HTS file. Otherwise, the connection name is not available and ws_connect generates an error.
    - To make a connection name available to all documents in the document tree, associate the database connection name with the document root.
    - To restrict access to a connection name within a directory subtree, associate the connection name with the corresponding subdirectory.
- If no connection name is specified, ws_connect returns the default connection or NULL, if there is no default.

For more information about ws_connect, see “Connecting to a SQL Server with the web.sql Convenience API” in Chapter 3, “Using Perl in HTS Files”.

Example

```
<SYB TYPE=PERL>


$cntname = "qauser1_cn";
$ws_db = ws_connect($cntname);
```

```
ws__debug();
#....code deleted....
</SYB>
```

➤ ***Note***

ws__debug has two underscores.

See Also

ct_connect

ws_content_type

Function

Indicate the type of data returned by a *.pl* file.

Syntax

```
ws_content_type($data_type)
```

Parameters

$data_type – A string specifying the data format in terms of a MIME content-type.

Comments

- Useful only in *.pl* files, not HTS files. Sets the content-type to the string shown. Some examples of content-type strings include “text/html”, “text/plain”, and “image/gif”. This function must be called before any output is done.

➤ ***Note***

Be sure to supply a valid data type with this function. If you supply an invalid data type to **ws_content_type**, the Web browser displays an error.

Example

```
ws_content_type("text/html");
```

ws_error

Function

Prints an error message and terminates the processing of the current page.

Syntax

`ws_error($string)`

Parameters

$string – Any string.

Comments

The printed error message includes *$string*.

Example

```
if (!$name)
{
    ws_error( "You must specify a name in the search field." );
}
```

ws_fetch_rows

Function

Repeatedly fetches data for the current result set that has fetchable data and displays output in a specified format.

Syntax

```
ws_fetch_rows($handle [,$format])
```

Parameters

$handle – A pre-defined, valid connection to a database server and login.

$format – An optional output format for the fetched data, where the syntax of the data is similar to **printf**.

Comments

- **ws_fetch_rows** is a utility routine that can be used with the web.sql Client-Library functions.
- **ws_fetch_rows** can be called after a SQL command has been issued with **ct_sql** and a result set has been returned by **ct_results**, if the result set is one of the following:
  - CS_ROW_RESULT
  - CS_STATUS_RESULT
  - CS_COMPUTE_RESULT
- If *$format* is not specified, **ws_fetch_rows** repeatedly calls **ct_fetch** to fetch the rows and formats the output into a HTML 3.0 table.
- If *$format* is specified, **ws_fetch_rows** formats each row using *$format* as the format string.
- If *$format* is specified, it must be a format string as defined in the definition of Perl 5's **printf()**. This is the same as C's **printf()** except that the "*" character for an indirectly specified length is not supported. This format string is used to print each row returned by the query.

Example

The following example of an HTS file processes a simple database query using **ws_fetch_rows** to fetch the rows.

```
<SYB TYPE=PERL>
# Build the SELECT statement
$sql_stmt = qq!select type, price
            from pubs2..title
            order by type
            compute sum(price) by type!;
# Send the command
if (($rc = ct_sql( $ws_db, $sql_stmt ) ) != CS_SUCCEED) {
    ws_error ("Unable to process database request.");
}
#Process the results from the server.
while (($ret = ct_results($ws_db, $result_type)) == CS_SUCCEED) {
    # Equivalent of switch statement in Perl.
    RESULTS: {
        # Simply ignore report of command completion
        if ($result_type == CS_CMD_DONE) {
            last RESULTS; # Jump to end of RESULTS block.
        }
        # Simply ignore notification of unsuccessful completion
        if ($result_type == CS_CMD_FAIL) {
            last RESULTS; # Jump to end of RESULTS block.
         }
        # Fetch and print row results
        if ($result_type == CS_ROW_RESULT) {
            ws_fetch_rows($ws_db)
             last RESULTS; # Jump to end of RESULTS block.
        }
        # Fetch and print compute results
        if ($result_type == CS_COMPUTE_RESULT) {
            ws_fetch_rows($ws_db);
            last RESULTS; # Jump to end of RESULTS block.
        }
    }
```

```
}
if ($ret == CS_FAIL) {
    # A connection error occurred. Clean up connection state.
    ct_cancel(CS_CANCEL_ALL);
    ws_error("A database connection error occurred.");
}
</SYB>
```

See Also

ct_fetch, ct_results, ct_sql

ws_print

Function

Print a string, expanding Perl variable references.

Syntax

ws_print($string)

Parameters

$string – The string to be printed.

Comments

Executes exactly as the Perl **printf()** function, except that if *$string* contains a reference to a Perl variable, the value of that variable is substituted before the string is printed.

Example

Suppose that a form has a field named USER_ID and that the form specifies an HTS file as the URL to call when the user submits the form. Now consider the following lines in the HTS file specified by the form:

```
<SYB TYPE=PERL>
$msg = "Field USER_ID does not have a valid value ($user_id).\n";
ws_print("$msg");
</SYB>
```

If the user enters the text “Bad text” into the USER_ID field, **ws_print** substitutes the string “Bad text” for the variable *$user_id*. The code above prints:

```
Field USER_ID does not have a valid value (Bad text).
```

On the other hand, if you use the Perl **print** routine rather than **ws_print**, there is no substitution, and the output is:

```
Field USER_ID does not have a valid value ($user_id).
```

ws_rpc

Function

Calls stored procedures that reside on remote servers.

Syntax

```
ws_rpc($db_handle, $rpc_name, @params or %params [, $format])
```

Parameters

$db_handle – A pre-defined, valid connection to a database server and login.

$rpc_name - The name of the RPC you want to call.

%params or *@params* - The list of parameters for the RPC, including output parameters.

You can call stored procedures with two styles of parameters: named and positional.

- If you use the named parameter style, *%params*, you must pass the parameter name and the value assigned to that name to the stored procedure by enclosing them in `{}`. The list of parameters for the RPC is passed as a Perl 5 reference to an associative (hash) array.
- If you use the positional parameter style, *@params*, you only pass the argument value by enclosing it in `[]`. You pass the argument in the first position as the first parameter to the stored procedure, the argument in the second position as the second parameter, the argument in the third as the third parameter, and so on. The list of parameters for the RPC is passed as a Perl 5 reference to an array.

You can specify output parameters with a reference to a scalar variable. Scalar data can be passed as an integer, a float, or a character string, depending on the Perl datatype.

To pass or receive a particular Sybase datatype other than integer, you must use a datatype-reference structure. For example, if you want to pass an integer as a character string, you must use a datatype-reference to enforce the behavior. The datatype-references can be written as follows:

```
{
'type'      =>  CS_<datatype>,
'value'     =>  <scalar> or <scalar-ref>,
```

```
'scale'      =>  CS_MAX_SCALE or CS_DEF_SCALE,
'precision'  =>  CS_MAX_PREC or CS_DEF_PREC
}
```

For a list of the `CS_` datatypes, see Appendix B, “RPC Datatype Summary.”

$format - Optional argument that specifies the format in which you want results returned. If you do not specify a format, the results are returned in an HTML 3.0 table.

Comments

- When stored procedures have no arguments, you can leave the argument off. Having an empty argument list is the same as not having an argument list. For example, with **ws_rpc** you could have an argument list, have an empty argument list, or have no arguments, as follows: **ws_rpc($ws_db,"name",[$arg]),$fmt)**; or **ws_rpc($ws_db,"name")**; or **ws_rpc($ws_db,"name",[$arg])**; or **ws_rpc($ws_db,"name",[],$fmt)**; or **ws_rpc($ws_db,"name",[],\&func)**;.

Example Using Named Parameters (Associative Array)

```
<SYB TYPE=PERL>
$res = ws_rpc($ws_db,"pubs2..insert_salesdetail_proc",
        { '@stor_id' => "$stor_id",
          '@ord_num' => "$ordernum",
          '@title_id' => "$title_id",
          '@qty' => { 'value' => "$qty", 'type' => \
               CS_SMALLINT_TYPE },
          '@discount' => { 'value' => "$discount", 'type' => \
               CS_FLOAT_TYPE }
        });
if ($res == 0)
    {
    printf "<H1>Insert Succeeded</H1>\n";
    ws_sql($ws_db,
        "select * from pubs2..salesdetail where stor_id =
'$stor_id' and ord_num = '$ordernum'");
    }
else
```

```
    {
    printf "<H1>insert failed!</H1>\n";

#....code deleted....
```

ws_sql

Function

Executes one or more SQL statements and prints the output.

Syntax

```
ws_sql($handle, $sql, [,$format])
```

Parameters

$handle – A pre-defined, valid connection to a database server and login.

$sql – A SQL command.

$format - Optional argument that specifies the format in which you want results returned. If you do not specify a format, the results are returned in a HTML 3.0 table.

Comments

- *$sql* may contain Perl variables.
- *$sql* may contain one or more Transact-SQL commands, separated by newline characters ("\n").
- If any command fails, a warning is returned.
- If using a **use** statement for a SQL command in a **ws_sql** routine, you must place the **use** statement in a *separate* **ws_sql** call or in a SQL **<SYB>** block preceding the SQL statements you want it to affect. The database that you specify in the **use** statement is then in effect until the end of the HTS file or until the next **use** statement.
- If *$format* is omitted, HTML 3.0 tables are used to format result sets that are returned.

➤ ***Note***

The tables work well in Netscape version 1.1 and later, and in the latest version of Mosaic on Windows; they may not work on other browsers.

- The *$format* may be a scalar string, which is used as the **printf** for each row. It can also be a function reference, which specifies a function to be called for each result-type.

Example

The example below shows an example of **ws_sql** calling a function reference:

```
<SYB TYPE=PERL>
sub my_fmt
    {
    my $dbh = shift @_;
    my $restype = shift @_;
    my $numcol = scalar(@_);
    my $i;
    if ($restype == CS_COMPUTE_RESULT)
        { printf "<TR> <TD> CS_COMPUTE_RESULT \n"; }
    elsif ($restype == CS_ROW_RESULT)
        { printf "<TR> <TD> CS_ROW_RESULT \n"; }
    else
        { printf "<TR> <TD> unexpected $restype \n"; }
    for ($i = 0; $i < $numcol; $i++)
        {
        printf "<TD> @_[$i] \n";
        }
    }
</SYB>

<BODY>
<H1>Example of ws_sql calling a function ref</H1>

<SYB TYPE=PERL>
ws_sql($ws_db,"select * from pubs2..stores compute
max(stor_id)",\&my_fmt);
</SYB>
```

See Also

ct_sql

B RPC Datatype Summary

The following table lists the Open Client datatypes you can use with ws_rpc and ct_rpc.

Table B-1: Summary of CS_ Datatypes

Open Client Type Constant	Description
CS_BINARY_TYPE	Binary type
CS_LONGBINARY_TYPE	Long binary type
CS_VARBINARY_TYPE	Variable-length binary type
CS_BIT_TYPE	Bit type
CS_CHAR_TYPE	Character type
CS_LONGCHAR_TYPE	Long character type
CS_VARCHAR_TYPE	Variable-length character type
CS_DATETIME_TYPE	8-byte datetime type
CS_DATETIME4_TYPE	4-byte datetime type
CS_TINYINT_TYPE	1-byte datetime type
CS_SMALLINT_TYPE	2-byte integer type
CS_INT_TYPE	4-byte integer type
CS_DECIMAL_TYPE	Decimal type
CS_NUMERIC_TYPE	Numeric type
CS_FLOAT_TYPE	8-byte float type
CS_REAL_TYPE	4-byte float type
CS_MONEY_TYPE	8-byte money type
CS_MONEY4_TYPE	4-byte money type
CS_BOUNDARY_TYPE	Secure SQL Server boundary type
CS_SENSITIVITY_TYPE	Secure SQL Server sensitivity type
CS_TEXT_TYPE	Text type
CS_IMAGE_TYPE	Image type

Index

Symbols

$ in HTS files 2-5

A

Access map. *See* Database access map 3-13
ACTION attribute
  <FORM> tag 2-7
Administration page 3-13

B

Browsers
  requesting HTML pages with 1-1

C

Caching database connection 1-3
Callbacks 3-30, 3-32, A-2
  removing A-2
Canceling
  commands 3-29, A-5 to A-6
  results 3-29, A-5 to A-6, A-33
CGI version, web.sql 1-3
Clearing server options 3-29, A-22
Client callbacks 3-30 to 3-31, A-2 to A-3
Client-Library 1-6, 3-2, 3-10, 3-36
  ct_callback 3-30, 3-32, A-2, A-4
  ct_cancel 3-22, 3-29, A-5 to A-6
  ct_col_names 3-22, A-7
  ct_col_types 3-22, A-8, A-10
  ct_connect 3-12, 3-14, A-11, A-13
  ct_fetch 3-18, 3-23, 3-25, 3-27 to 3-28, A-14, A-17
  ct_fetch_parameters 3-18, A-18
  ct_options 3-29 to 3-30, A-22
  ct_res_info 3-22, 3-26, A-26, A-29
  ct_results 3-20, 3-28, A-30, A-35
  ct_rpc 3-16, A-18, A-36
  ct_sql 3-14 to 3-15, A-41 to A-42
  references xvi
  ws_rpc A-51
Client-Library API
  data manipulation 3-2
Columns
  names, retrieving 3-22, A-7
  types, retrieving 3-22, A-8, A-10
Commands, canceling 3-29, A-5 to A-6
Connection-caching ability 1-3
Connections
  failure to connect A-12
  opening 3-4, 3-12, 3-14, A-11, A-13, A-43 to A-44
  pending results A-33
Content type 1-1
Context structures 3-14
Control structures 3-14
Convenience API 1-6, 3-2, 3-7, 3-11
  ws_connect 3-3 to 3-4, 3-14, A-43 to A-44
  ws_content_type 3-35, A-45
  ws_error 3-7, A-46
  ws_fetch_rows 3-18, 3-23, 3-25, 3-27 to 3-28, A-47, A-49
  ws_print 3-6 to 3-7, A-50
  ws_rpc 3-7
  ws_sql 3-4, 3-6, A-54 to A-55
Cookie location 3-37
Cookies 3-36
CS_CANCEL_ALL parameter 3-29, A-5, A-33
CS_CANCEL_CURRENT parameter 3-29, A-5, A-34
CS_CANCELED return value 3-15, 3-29, A-31, A-41
CS_CLIENTMSG_CB callback type A-2
CS_CMD_DONE result type 3-20, 3-23 to 3-25, A-30, A-33
CS_CMD_FAIL result type 3-20, 3-22 to 3-23, 3-25, A-30, A-33 to A-34

CS_CMD_SUCCEED result type 3-20, 3-25, A-30, A-33
CS_COMPUTE_RESULT result type 3-21, 3-23, A-31
CS_END_RESULT result type 3-21
CS_END_RESULTS return value 3-22, 3-24, A-31
CS_FAIL callback return value 3-30, A-2
CS_FAIL return value 3-15, 3-22, 3-24, A-5, A-31, A-41
CS_MSG_RESULT result type 3-21, 3-28, A-31
CS_MSGTYPE information type 3-28, A-26
CS_NUM_COMPUTES information type A-26
CS_NUMDATA information type A-26
CS_NUMORDERCOLS information type A-26
CS_OPT_ANSINULL option A-23
CS_OPT_ANSIPERM option A-23
CS_OPT_ARITHABORT option A-23
CS_OPT_ARITHIGNORE option A-23
CS_OPT_AUTHOFF option A-23
CS_OPT_AUTHON option A-23
CS_OPT_CHAINXACTS option A-23
CS_OPT_CURCLOSEONXACT option A-23
CS_OPT_CURREAD option A-23
CS_OPT_CURWRITE option A-23
CS_OPT_DATEFIRST option A-23
CS_OPT_DATEFORMAT option A-23
CS_OPT_FIPSFLAG option A-23
CS_OPT_FORCEPLAN option A-23
CS_OPT_FORMATONLY option A-24
CS_OPT_GETDATA option A-24
CS_OPT_IDENTITYOFF option A-24
CS_OPT_IDENTITYON option A-24
CS_OPT_ISOLATION option A-24
CS_OPT_NOCOUNT option A-24
CS_OPT_NOEXEC option A-24
CS_OPT_PARSEONLY option A-24
CS_OPT_QUOTED_IDENT option A-24
CS_OPT_RESTREES option A-24
CS_OPT_ROWCOUNT option A-24
CS_OPT_SHOWPLAN option A-24
CS_OPT_STATS_IO option A-24
CS_OPT_STATS_TIME option A-24
CS_OPT_STR_RTRUNC option A-24
CS_OPT_TEXTSIZE option A-24
CS_OPT_TRUNCIGNORE option A-24
CS_PARAM_RESULT result type 3-18, 3-21, 3-27, A-31
CS_ROW_COUNT information type A-26
CS_ROW_RESULT result type 3-21, 3-23, A-31
CS_SERVERMSG_CB callback type A-2
CS_STATUS_RESULT result type 3-21, 3-27, A-31
CS_SUCCEED callback return value 3-30, A-2
CS_SUCCEED return value 3-15, 3-20, A-5, A-31 to A-32, A-41
ct_callback 3-30, 3-32, A-2, A-4
ct_cancel 3-22, 3-29, A-5 to A-6
ct_col_names 3-22, A-7
ct_col_types 3-22, A-8, A-10
ct_connect 3-12 to 3-14, A-11, A-13
  reasons for failure A-7, A-18, A-37, A-43, A-45 to A-47, A-53
ct_fetch 3-18, 3-23, 3-25, 3-27 to 3-28, A-14, A-17
ct_fetch_parameters 3-18, A-18
ct_options 3-29 to 3-30, A-22
ct_res_info 3-22, 3-26, A-26, A-29
ct_results 3-20, 3-28, A-30, A-35
ct_rpc 3-16, A-18, A-36
ct_sql 2-4, 3-3, 3-14 to 3-15, A-41 to A-42

D

Data, types of 1-1
Database access map 3-3, 3-12, A-43
  securing 3-13
Database connections
  default 3-3, 3-12

specifying in every *.hts* file 3-13
using ct_connect 3-13
using ws_connect 3-13
Datatype references 3-9, 3-17, A-36, A-51
Datatypes 1
Default database connections 3-3, 3-12
Deleting
callbacks A-2
Dollar sign ($) in HTS files 2-5

E

Errors
callbacks, handling with 3-30, 3-32, A-2
handling 3-7, A-46
Executing SQL statements 2-2, 2-4, 3-4, 3-6, 3-14, 3-16, A-41 to A-42, A-54 to A-55
Extensions supported by web.sql 1-1

F

Features of web.sql 1-1
Files
content type of 1-1
.gif 1-1
.hts 1-1
.pl 1-1
$format string 3-5
Form data, accessing 2-6, 2-8
<FORM> tag
ACTION attribute 2-7

G

GET method 2-6
.gif file
accessing via HTTP server 1-1
.gif files 1-1
Graphic files 1-1

H

Handling errors 3-7, A-46
HTML
HTS files, in 1-3
references xiv
<SYB> tag 1-4, 2-1 to 2-2, 3-1 to 3-2
tables, query results formatted as 2-2 to 2-3
HTML extensions
using in HTS files 1-3
HTML file
accessing via Web server 1-1
HTML form data
GET and POST methods 2-6
HTML Form Data Variables
%ws_form 2-7
%ws_multiple 2-7
@foo 2-6
$foo 2-6
$ws_form{"foo"} 2-7
$ws_multiple{"foo"} 2-7
HTML. *See* HyperText Markup Language 1-1
HTML table generation
using convenience API 3-2
HTML table results
example 2-2
HTS file
including Perl script 3-1
HTS files 1-2, 1-6 to 1-7
$ as variable prefix 2-5
aborting execution 3-7, A-46
accessing 1-6
connections, opening 3-4, 3-12, 3-14, A-11, A-13, A-43 to A-44
default database connections 3-3, 3-12
format 1-3, 1-5
form data, accessing 2-6, 2-8
Perl scripts, including 2-1, 3-1, 3-36
SQL statements, including 2-1, 2-8, 3-2
using JavaScript tags 1-3
using Java tags 1-3

using <SYB> tag 1-4
variables 2-4, 2-8, 3-2
.hts files. *See* HyperText Sybase (HTS) files 1-1
HTTP Header Information
returning 3-36
HTTP server 1-1
HTTP server, web.sql version integrated with 1-3
HyperText Markup Language (HTML) 1-1
HyperText Markup Language. *See* HTML xiv
HyperText Sybase (HTS) files 1-1
HyperText Sybase files
See HTS files 2-1
HyperText Sybase files. *See* HTS files 1-2

I

Installing web.sql xiv
interfaces file and ct_connect A-11

J

JavaScript tags
using in HTS files 1-3
Java tags
using in HTS files 1-3
.jpeg file
accessing via HTTP server 1-1

L

Location 3-37

M

Maximum number of rows
setting in <SYB> block 2-2
Maximum number of rows returned by a query 2-2, 3-23, 3-29
Messages, processing results 3-28, A-15, A-34

N

Non-HTML data, returning 3-33, 3-36
NSAPI
web.sql UNIX version 1-3
web.sql version 1-3

O

Open Client Client-Library. *See* Client-Library xvi
Open Client datatypes 1
Opening connections 3-4, 3-12, 3-14, A-11, A-13, A-43 to A-44
Options, server 3-29, A-22
Overview, web.sql 1-1, 1-7, 3-33

P

Parameters
named 3-17
output 3-18
positional 3-17
Pending results A-33
Perl
HTS files, including scripts in 2-1, 3-1, 3-36
non-HTML data returned 3-33, 3-36
"qq!" syntax 3-5
references xv
variables in HTS files 2-4, 2-8, 3-2
Perl file
specifying non-HTML content type 1-4
Perl files 1-1
Perl HTML form variables
example 2-7
Perl script
example of use in HTS file 3-1
running with web.sql 1-3
Perl scripts 1-1

accessing 3-33 to 3-34
Perl statements
including in <SYB> blocks 1-4
Perl <SYB> blocks 3-2
defining variables 3-2
mixing with SQL <SYB> blocks 2-1
.pl files 1-1
See also Perl files 1-4
.pl files *See Also* Perl files 3-36
.pl files *See Also*Perl files 3-33
POST method 2-6
Pre-defined variables 2-6, 2-8
$ws_db 3-3, 3-12
Printing SQL statement output 2-2 to 2-3, 3-4, 3-7, A-47, A-49 to A-50, A-54 to A-55
process_rows() function 3-5
pubs2 database 2-1

Q

"qq!" syntax 3-5
Queries
executing 2-2, 2-4, 3-4, 3-6, 3-14, 3-16, A-41 to A-42, A-54 to A-55
maximum number of rows returned 2-2, 3-23, 3-29
printing output 2-2 to 2-3, 3-4, 3-7, A-47, A-49 to A-50, A-54 to A-55
processing results 3-23, 3-25

R

Redirection 3-37
References xiii, xvi
Relational databases 1-1
Remote Procedure Calls 3-7, A-36, A-51
Removing
callbacks A-2
Result data A-15
Results
canceling 3-29, A-5 to A-6, A-33
ct_results loop 3-20, 3-28, A-30, A-35
information about 3-22, A-26, A-29
pending results A-33
Result types 3-20, 3-22, A-30, A-35
Retrieving
column names 3-22, A-7
column types 3-22, A-8, A-10
server options 3-29, A-22
Return data, information about 3-22, A-26, A-29
Row count A-26
Rows, maximum number returned by a query 2-2, 3-23, 3-29
RPC datatypes 1

S

Security
specifying database connections 3-13
See URL 1-1
select statement
example 2-2
Perl example 2-7
using in HTS file 2-1
Server callbacks 3-30, 3-32, A-3
Server options 3-29, A-22
Setting server options 3-29, A-22
SQL
HTS files, including in 2-1, 2-8, 3-2
references xv
statements, executing 2-2, 2-4, 3-4, 3-6, 3-14, 3-16, A-41 to A-42, A-54 to A-55
statements, printing output 2-2 to 2-3, 3-4, 3-7, A-47, A-49 to A-50, A-54 to A-55
statements, processing results 3-20, 3-28, A-14, A-17, A-30, A-35, A-47, A-49
$sql argument 3-4
SQL statements 1-1
using multiple statements in <SYB> blocks 2-1
SQL <SYB> block
setting maximum rows 2-2
Stored Procedures 3-7, 3-17, A-36, A-51

Stored procedures, processing results 3-27 to 3-28, A-15, A-34
Sybase web.sql Administration page. *See* Administration page 3-13
<SYB> block
  restrictions 3-2
<SYB> tag 1-4, 2-1 to 2-2, 3-1 to 3-2
  using in HTS files 1-4

T

Tables, query results formatted as 2-2 to 2-3
Transact-SQL
  defining in <SYB> blocks 1-4
  using statements in HTS files 2-1
Transact-SQL. *See* SQL xv
Transact-SQL statements
  select 2-1
Troubleshooting web.sql xiv
TYPE=PERL
  using in <SYB> blocks 1-4
TYPE=PERL attribute 3-1
TYPE=SQL
  using in <SYB> block 1-4
TYPE attribute
  using in <SYB> tags 1-4
TYPE attribute in <SYB> tags 1-4, 2-1 to 2-2, 3-1 to 3-2

U

Universal Resource Locator 1-1
URL
  redirection 3-36
URL redirection 3-37
use statement
  example 2-4
  special handling 2-4
use statement, special handling 2-4

V

Variables in HTS files 2-4, 2-8, 3-2

W

Web browsers. *See* Browsers 1-1
Web server
  architecture (typical) 1-1
  architecture (using web.sql) 1-2
Web server cookies 3-36
  detailed explanation 3-36
Web site administration references xv
web.sql 1-3
  CGI version 1-3
  Client-Library API 1-6, 3-2, 3-10, 3-36
  convenience API 1-6, 3-2, 3-7, 3-11
  HTTP server, version integrated with 1-3
  installing xiv
  non-HTML data, returning 3-33, 3-36
  NSAPI version 1-3
  overview 1-1, 1-7, 3-33
  predefined variables 2-6
  running Perl scripts 1-3
  troubleshooting xiv
  version differences 1-3
World Wide Web 1-1
ws_client_callback 3-31
ws_connect 3-3 to 3-4, 3-13 to 3-14, A-43 to A-44
ws_content_type 3-3, 3-35, A-45
ws_content_type() variable 1-4
$ws_db 3-3, 3-12
ws_error 3-3, 3-7, A-46
ws_fetch_rows 3-3, 3-18, 3-23, 3-25, 3-27 to 3-28, A-47, A-49
ws_print 3-6 to 3-7, A-50
ws_rpc 3-7, A-51
ws_server_callback 3-32
ws_sql 2-4, 3-4, 3-6, A-54 to A-55
wsh- prefix 3-36