

CA Gen

Distributed Processing Proxy User Guide

Release 8.5



This Documentation, which includes embedded help systems and electronically distributed materials (hereinafter referred to as the "Documentation"), is for your informational purposes only and is subject to change or withdrawal by CA at any time.

This Documentation may not be copied, transferred, reproduced, disclosed, modified or duplicated, in whole or in part, without the prior written consent of CA. This Documentation is confidential and proprietary information of CA and may not be disclosed by you or used for any purpose other than as may be permitted in (i) a separate agreement between you and CA governing your use of the CA software to which the Documentation relates; or (ii) a separate confidentiality agreement between you and CA.

Notwithstanding the foregoing, if you are a licensed user of the software product(s) addressed in the Documentation, you may print or otherwise make available a reasonable number of copies of the Documentation for internal use by you and your employees in connection with that software, provided that all CA copyright notices and legends are affixed to each reproduced copy.

The right to print or otherwise make available copies of the Documentation is limited to the period during which the applicable license for such software remains in full force and effect. Should the license terminate for any reason, it is your responsibility to certify in writing to CA that all copies and partial copies of the Documentation have been returned to CA or destroyed.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CA PROVIDES THIS DOCUMENTATION "AS IS" WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT WILL CA BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, LOST INVESTMENT, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF CA IS EXPRESSLY ADVISED IN ADVANCE OF THE POSSIBILITY OF SUCH LOSS OR DAMAGE.

The use of any software product referenced in the Documentation is governed by the applicable license agreement and such license agreement is not modified in any way by the terms of this notice.

The manufacturer of this Documentation is CA.

Provided with "Restricted Rights." Use, duplication or disclosure by the United States Government is subject to the restrictions set forth in FAR Sections 12.212, 52.227-14, and 52.227-19(c)(1) - (2) and DFARS Section 252.227-7014(b)(3), as applicable, or their successors.

Copyright © 2015 CA. All rights reserved. All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

CA Technologies Product References

This document references the following CA Technologies products:

- CA Gen
- COOL: Gen

Contact CA Technologies

Contact CA Support

For your convenience, CA Technologies provides one site where you can access the information that you need for your Home Office, Small Business, and Enterprise CA Technologies products. At <http://ca.com/support>, you can access the following resources:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- CA Support policies and guidelines
- Other helpful resources appropriate for your product

Providing Feedback About Product Documentation

If you have comments or questions about CA Technologies product documentation, you can send a message to techpubs@ca.com.

To provide feedback about CA Technologies product documentation, complete our short customer survey which is available on the CA Support website at <http://ca.com/docs>.

Contents

Chapter 1: Working With Proxies	11
Audience	11
Prerequisite Knowledge	12
Concepts and Definitions	12
.NET Proxy	13
Java Proxy	13
Java Proxy (Classic Style)	14
COM Proxy	14
C Proxy	15
Installing the Proxy Software	15
Install Considerations for .NET Proxy	16
Install Considerations for Java Proxy	17
Install Considerations for Java Proxy (Classic Style)	18
Install Considerations for COM Proxy (Windows)	19
Install Considerations for C Proxy (Windows)	20
Install Considerations for C Proxy (UNIX)	21
Common Features of the Proxies	21
Optional Features of the Proxies	22
Asynchronous Flow Support	22
XML Support	23
Selecting a Proxy	29
Platform	29
Language	29
Programming Style	30
Next Generation	30
Related Information	30
Chapter 2: Generating the Proxy	31
Before You Start	31
Install Proxy Support	31
Complete Modeling Activities	31
Associate Character Names	32
Toolset Generation	33
CSE Generation	38

Chapter 3: .NET Proxy 49

.NET Proxy Generated Code	49
.NET Proxy Interface	50
.NET Proxy Object.....	50
View Objects	56
Using a .NET Proxy.....	62
Synchronous Processing.....	62
Asynchronous Processing.....	63
Security Processing	64
Preparing for Execution.....	65
Configuring the .NET Proxy Runtime.....	65
Deploying a .NET Proxy	67
Executing the Sample Applications	68
ASP.NET Sample	69
Web Service Sample.....	69
Stand-alone .NET Application.....	70
XML Test Application (Optional)	71

Chapter 4: Java Proxy 73

Java Proxy Generated Code	73
Java Proxy Interface	74
Java Proxy Object	75
View Objects	79
Using a Java Proxy	84
Synchronous Processing.....	85
Asynchronous Processing.....	85
Security Processing	86
Preparing for Execution.....	87
Configuring the Java Proxy Runtime	87
Deploying a Java Proxy.....	92
Executing the Sample Applications	94
JSP Sample.....	94
Stand-alone Application	95
XML Test Application (Optional)	95

Chapter 5: Java Proxy (Classic Style) 97

Java Proxy Generated Code	97
Java Proxy (Classic Style) Interface.....	98
Common Properties and Methods.....	99
Unique Attribute Properties	101

Special String Methods	103
Date, Time, and Timestamp Properties	103
Decimal Precision Properties	103
Repeating Group Property Handling	104
Java Proxy Code	105
Using a Java (Classic Style) Proxy	106
Synchronous Processing.....	107
Asynchronous Processing.....	107
Security Processing	108
Preparing for Execution.....	109
Configuring the Java Proxy Runtime	109
Deploying a Java Proxy.....	114
Support Files in Java Proxy (Classic Style)	115
Executing the Sample Applications	117
Applet/Servlet JavaBean Sample	117
Java Application Bean Sample.....	117
Application Bean JSP Pages Sample	118
XML Test Application (Optional)	118

Chapter 6: COM Proxy 119

COM Proxy Generated Code	119
COM Proxy Interface	120
Common Properties and Methods.....	120
Unique Attribute Properties	123
Special Text Permitted Value Property Handling	124
Special Timestamp Property Handling	125
Special Date Property Handling	125
Repeating Group Property Handling.....	126
Using a COM Proxy	127
Synchronous Processing.....	128
Asynchronous Processing.....	128
Security Processing	129
Preparing for Execution.....	130
Trace Log and Configuration File Locations	130
Configuring the COM Proxy Runtime	134
Deploying the COM Proxy	138
Registering the Proxy DLLs	138
Supporting Files in COM Proxy.....	139
Executing the VB and ASP Sample Code	140
Calling the COM Proxy from a Web Server	141
Calling the COM Proxy from Visual Basic	141

Calling a COM Proxy from the XML Test Application (Optional)	142
COM Proxy Server Testing	143
Use Diagram Trace Utility	143
Multi-User Diagram Trace Utility Support	144
Override the Default Trace Environment Variables	144
Regenerating Remote Files After Testing	145

Chapter 7: C Proxy 147

Setting Up the C Proxy API Environment	147
Setting Up Windows Platforms	148
Setting Up UNIX Systems	148
Visual Studio Support	149
64-bit Windows Support	149
Calling the C Proxy API from a C Program	149
Variable Name Case Sensitivity	150
Overview of the Generated Header File	150
Example Generated Header File	150
Overview of an Example C Program	159
Tuxedo C Proxy Support	163
Native vs. Workstation Client Applications	163
Overview of the Generated Tuxedo View File (.tvf)	164
Setting Windows Platform Environment Variables	165
Setting UNIX Platform Environment Variables	166
Building User-written Applications Targeting Tuxedo Server Environments	167
Sample Makefile for C Proxy Targeting Tuxedo	167
Proxy Function Prototypes	171
API Variable Types	172
Function Calls	175
ProxyAllocateParamBlock	177
ProxyCheckAsyncResponse	178
ProxyClearClientPassword	179
ProxyClearClientUserid	180
ProxyClearCommand	181
ProxyClearDialect	182
ProxyClearExitMsgType	183
ProxyClearExitStateMsg	183
ProxyClearExitStateNum	184
ProxyClearNextLocation	185
ProxyClearParamBlock	186
ProxyConfigureComm	187
ProxyDeleteParamBlock	188

ProxyExecute.....	189
ProxyExecuteAsync	190
ProxyGetAsyncResponse.....	191
ProxyGetClientPassword.....	194
ProxyGetClientUserid.....	194
ProxyGetCommand	195
ProxyGetDialect	196
ProxyGetExitMsgType	196
ProxyGetExitStateMsg.....	197
ProxyGetExitStateNum.....	198
ProxyGetNextLocation	199
ProxyIgnoreAsyncResponse	199
ProxySetClientPassword	201
ProxySetClientUserid	202
ProxySetCommand.....	203
ProxySetDialect	203
ProxySetNextLocation	204
ProxyStartTracing.....	205
ProxyStopTracing	206
ProxyTraceOut	207
ProxySetViewBlob	208
ProxyGetViewBlob	208
ProxyClearViews.....	209
Proxy.c.....	210
Using a C Proxy	224
Security Processing	224
Overview of an Example Makefile for Windows	225
Overview of an Example Makefile for UNIX	226
Executing the User-written C Proxy Application	227
Configuring the C Proxy at Runtime	227
 Chapter 8: Using the application.ini File With Proxies	 233
Environment Variables in the application.ini File.....	233
 Index	 235

Chapter 1: Working With Proxies

A CA Gen Distributed Processing (DP) application is made up of many pieces of software, each contributing to the overall processing of a cooperative flow occurring from a Distributed Processing Client (DPC) to a Distributed Processing Server (DPS).

Note: For information about the various parts that make up a DP application, see the *Distributed Processing - Overview Guide*.

A CA Gen application developer often encounters the requirement for code that is not generated from a CA Gen model to interact with one or more of their generated DPS applications. In general, a user-written DPC application must communicate to the generated servers in the same manner as a generated DPC application. CA Gen provides the ability to generate a variety of proxy interfaces to assist the application developer in this effort.

A user-written DPC application can use a generated proxy to facilitate a cooperative flow to its target DPS. Each proxy is generated for a specific DPS. CA Gen provides support for the following types of generated Proxies:

- .NET Proxy
- Java Proxy
- Java Proxy (Classic Style)
- COM Proxy
- C Proxy

Audience

This guide is for CA Gen developers who need to access one or more (new or existing) generated DPS applications from a user-written application. A large variety of user-written applications can use the proxies described in this guide. The application developers must write their code to interface with the generated proxy. Some potential uses include:

- Using a generated proxy in a stand-alone C, Visual Basic, .NET, or Java application
- Using a generated proxy as part of a Java, JSP, ASP, or ASP.NET Web application
- Using a generated proxy as part of a COM application

This guide is intended for application developers that need to understand the following:

- The details of a proxy
- The features offered by each type of proxy
- How an application developer would generate a proxy
- The code generated for a proxy
- The environment in which proxies can be used
- The runtimes that support the execution of a proxy

Prerequisite Knowledge

The reader of this guide should be familiar with:

- Developing CA Gen Distributed Processing applications
- The CA Gen Toolset
- The environment for which, the user-written application is being written. For example, the reader should be familiar with Microsoft IIS if they intend to deploy COM proxies into that environment.

Concepts and Definitions

A CA Gen Proxy is an interface that enables a user-written client application to communicate with a generated CA Gen server. Proxies make it possible for applications that are not generated using CA Gen, to use the procedure step logic provided as part of a Distributed Processing Server (DPS). In general, proxies provide application developers additional flexibility in creating multi-tier applications.

Proxies also make it possible for applications that are not generated using CA Gen, to access the interfaces and methods defined as components within a CBD-compliant model.

An application developer can create proxies of the following types:

- .NET Proxy
- Java Proxy

- Java Proxy (Classic Style)
- COM Proxy
- C Proxy

Apart from creating proxies that interface with the client code and the servers, you can also customize the proxy.

The following are the changes you can make on a proxy:

- Specify meaningful view names and attribute names.
- Use fewer attributes than defined in the model (views).
- Specify default values for omitted attributes.
- Pass default values as attributes.

Such proxies are named Custom Proxies.

Note: You cannot create custom proxies in Java (Classic Style) and C.

Note: For more information about customizing proxies, see the *CA Gen Studio Overview Guide*.

.NET Proxy

The .NET Proxy is a generated .NET object that provides an object-oriented interface to CA Gen servers. The .NET Proxy and associated CA Gen runtimes are 100-percent managed .NET code. Providing a .NET managed object interface enables any .NET aware application to access CA Gen servers. Typical user applications are stand-alone programs (VB.NET or C#), and ASP.NET web applications.

More information:

[.NET Proxy](#) (see page 49)

Java Proxy

The Java Proxy is a generated Java object that provides an object-oriented interface to CA Gen servers. The Java Proxy and associated CA Gen runtimes are 100-percent Java code. Providing a Java object interface enables any Java application to access CA Gen servers. Typical user applications are Stand-alone programs, JSP web applications, Servlet web applications, and EJB components. The proxy can be used in virtually any component of a J2SE or J2EE application.

More information:

[Java Proxy](#) (see page 73)

Java Proxy (Classic Style)

The classic style Java proxy is a Java-based interface that enables Java applets, applications, and servlets to access the CA Gen servers. There are two forms of the classic style Java Proxy.

The first form consists of three parts generated from a CA Gen model:

- A Client Java Bean Proxy, which resides in the Java application ,or Java applet
- A Servlet Proxy, which resides on the Web Server
- A user interface applet, that can be used to test the Client Java Bean, and as an example applet

The second form consists of several parts generated from a CA Gen model:

- An application java bean proxy, which resides in a Java application or Java Servlet/EJB
- A test user interface application
- JSP pages
- Possibly an XML Test Application, which can be used to test the application Java Bean, and used as examples

More information:

[Java Proxy \(Classic Style\)](#) (see page 97)

COM Proxy

The ActiveX/COM Proxy is a generated COM object that provides an automation interface to CA Gen servers. Providing a COM automation interface enables Web applications with Active Server Pages and OLE-enabled clients to access CA Gen servers. The generated product consists of the following two objects:

- an ActiveX object
- a low-level COM object

More information:

[COM Proxy](#) (see page 119)

C Proxy

The C Proxy is a generated C language header file that defines a C language interface to a server procedure step. The application developer can use the interface to write user-written code that is able to call the associated CA Gen server procedure. A C proxy is used by a C language application, or any application that supports a C language interface.

The CA Gen Proxy Generators create a C Language header file and a TVF (Tuxedo View File). The TVF is used if the target server is a Tuxedo application.

Note: The C Proxy does not supply routines for checking permitted values of data flowing between the user-written application and the DPS.

More information:

[C Proxy](#) (see page 147)

Installing the Proxy Software

The proxy software is a separately licensed product, available for install using Custom Install. Ensure that you review the *Technical Requirements* document. The *Technical Requirements* document describes the execution environment supporting each type of proxy. It identifies, for each type of proxy, the set of transport protocols offered on a given platform. The *Technical Requirements* document provides a description of the requirements for building and running the proxies.

The following sections discuss the install considerations for each type of proxy offered by CA Gen. Each section describes the set of items needed on development machines and on deployment machines that executes a proxy.

Note: For information on product licensing, see the *Distributed Systems Installation Guide*.

Install Considerations for .NET Proxy

Development Machine

You must install the following products to generate and build a .NET proxy:

- Workstation Development Toolset
- Workstation Construction Toolset
- Implementation Toolset
- Proxy Programming Interface .NET
- Depending on which DP Server the .NET Proxy is flowing to, you must install one or more of the following products:

Products	CICS	IMS	TE	Tuxedo	EJB	Net
TCP/IP Middleware	X 1,2	X 2	X 1,2	X 1,2	X3	n.a
MQSeries Middleware	X	X	n.a	n.a	n.a	n.a
.NET Servers (includes .NET remoting)	n.a	n.a	n.a	n.a	n.a	X
CFB Converter Services to EJBs (includes EJBRMI)	n.a	n.a	n.a	n.a	X	n.a
Web Services Middleware	n.a	n.a	X	n.a	X	n.a

1. TCP/IP can be used to flow directly to the target server environment.
2. TCP/IP can be used to flow to the target server environment using CA Gen Communications Bridge product.
3. TCP/IP can be used to flow the target EJB environment using the CA Gen CFB Converter Services to EJBs component.

Deployment Machine

You must install the .NET Proxy runtime components on the deployment machine. You can do this by building an installation .msi file on the development machine and then transferring and executing the .msi file on to the deployment machine. The .msi file is built on the development machine using the Assemble capability of the CA Gen Build Tool. The *Build Tool User Guide* provides further details regarding creation of .NET .msi installation files.

Note: You must install the .NET Framework on the computer that runs the generated .NET proxy. For currently supported .NET Framework version information, see the *Technical Requirements* documentation. You must also install the communications protocol software on the computer.

Install Considerations for Java Proxy

Development Machine

You must install the following components to generate and build a Java proxy:

- Workstation Development Toolset
- Workstation Construction Toolset
- Implementation Toolset
- Proxy Programming Interface Java
- Depending on which DP server the Java Proxy is flowing to, you must install one or more of the following products:

Products	Distributed Processing Server						
	CICS	IMS	MQSeries	TE	Tuxedo	EJB	.Net
TCP/IP Middleware	X 1,2	X 1,2	n.a	X 1,2	X 1,2	X3	n.a
ECI Middleware	X	n.a	n.a	n.a	n.a	n.a	n.a
MQSeries Middleware	X	X	X	n.a	n.a	n.a	n.a
Tuxedo Middleware	n.a	n.a	n.a	n.a	X	n.a	n.a
Enterprise Java Beans (includes EJB RMI)	n.a	n.a	n.a	n.a	n.a	X	n.a
CFB Converter Services to EJBs (includes EJB RMI)	n.a	n.a	n.a	n.a	n.a	X	n.a
Web Services Middleware	n.a	n.a	n.a	X	n.a	X	n.a

1. TCP/IP can be used to flow directly to the DP server environment.
2. TCP/IP can be used to flow to the target server environment using CA Gen Communications Bridge product.
3. TCP/IP can be used to flow the EJB environment using the CA Gen CFB Converter Services to EJBs component.

Deployment Machine

You must install the Java Proxy runtime components on the deployment machine. You can do this by building a single runtime JAR file. This JAR file contains all the pieces of the runtime needed by the proxy. You can use the batch file named `mkjavart.bat`, found in the CA Gen classes directory, to accomplish this. For an explanation of the available execution options for `mkjavart.bat`, execute the command `mkjavart.bat usage`.

Install Considerations for Java Proxy (Classic Style)

Development Machine

You must install the following products to generate and build a Java Proxy (Classic Style):

- Workstation Development Toolset
- Workstation Construction Toolset
- Implementation Toolset
- Proxy Programming Interface Java
- Depending on which DP server the Java Proxy (Classic Style) is flowing to, you must install one or more of the following products:

Products	Distributed Processing Server						
	CICS	IMS	MQSeries	TE	Tuxedo	EJB	.Net
TCP/IP Middleware	X 1,2	X 1,2	n.a	X 1,2	X 1,2	X3	n.a
ECI Middleware	X	n.a	n.a	n.a	n.a	n.a	n.a
MQSeries Middleware	X	X	X	n.a	n.a	n.a	n.a
Tuxedo Middleware	n.a	n.a	n.a	n.a	X	n.a	n.a
Enterprise Java Beans (includes EJB RMI)	n.a	n.a	n.a	n.a	n.a	X	n.a
CFB Converter Services to EJBs (includes EJB RMI)	n.a	n.a	n.a	n.a	n.a	X	n.a
Web Services Middleware	n.a	n.a	n.a	X	n.a	X	n.a

1. TCP/IP can be used to flow directly to the DP server environment.
2. TCP/IP can be used to flow to the DP server environment using CA Gen Communications Bridge product.
3. TCP/IP can be used to flow the EJB environment using the CA Gen CFB Converter Services to EJBs product.

Deployment Machine

You must install the Java Proxy runtime components on the deployment machine. You can do this by building a single runtime JAR file. This JAR file contains all the pieces of the runtime needed by the proxy. You can use the batch file named `mkjavart.bat`, found in the CA Gen classes directory, to accomplish this. For an explanation of the available execution options for `mkjavart.bat`, execute the command `mkjavart.bat usage`.

Install Considerations for COM Proxy (Windows)

Development Machine

You must install the following products to generate and build a COM proxy:

- Workstation Development Toolset
- Workstation Construction Toolset
- Implementation Toolset
- Proxy Programming Interface COM

Deployment Machine

- You must install the following products on the proxy deployment machines:
- Proxy Programming Interface COM
- Depending on which DP server the COM Proxy is flowing to, you must install one or more of the following products:

Products	Distributed Processing Server						
	CICS	IMS	MQSeries	TE	Tuxedo	EJB	.Net
TCP/IP Middleware	X 1,2	X 1,2	n.a	X 1,2	X 1,2	X3	n.a
ECI Middleware	X	n.a	n.a	n.a	n.a	n.a	n.a
MQSeries Middleware	X	X	X	n.a	n.a	n.a	n.a
.NET Servers (includes .NET remoting)	n.a	n.a	n.a	n.a	n.a	n.a	X
Tuxedo Middleware	n.a	n.a	n.a	n.a	X	n.a	n.a
Enterprise Java Beans (includes EJBRMI)	n.a	n.a	n.a	n.a	n.a	X	n.a
CFB Converter Services to EJBs (includes EJBRMI)	n.a	n.a	n.a	n.a	n.a	X	n.a
Web Services Middleware	n.a	n.a	n.a	X	n.a	X	n.a

1. TCP/IP can be used to flow directly to the DP server environment.
2. TCP/IP can be used to flow to the DP server environment using CA Gen Communications Bridge product.
3. TCP/IP can be used to flow the EJB environment using the CA Gen CFB Converter Services to EJBs product.

Note: You must install the COM runtime files on the computer that runs the generated COM proxy, usually the Web server or Automation Client. You must also install the communications protocol software on this computer.

Install Considerations for C Proxy (Windows)

Development Machine

You must install the following products to generate a C proxy:

- Workstation Development Toolset
- Workstation Construction Toolset
- Proxy Programming Interface C

Deployment Machine

You must install the following products on the proxy deployment machines:

- Proxy Programming Interface C
- Depending on which DP server the C Proxy is flowing to, you must install one or more of the following products:

Products	Distributed Processing Server						
	CICS	IMS	MQSeries	TE	Tuxedo	EJB	.Net
TCP/IP Middleware	X 1,2	X 1,2	n.a	X 1,2	X 1,2	X3	n.a
ECI Middleware	X	n.a	n.a	n.a	n.a	n.a	n.a
MQSeries Middleware	X	X	X	n.a	n.a	n.a	n.a
.NET Servers (includes .NET remoting)	n.a	n.a	n.a	n.a	n.a	n.a	X
Tuxedo Middleware	n.a	n.a	n.a	n.a	X	n.a	n.a
Enterprise Java Beans (includes EJBRMI)	n.a	n.a	n.a	n.a	n.a	X	n.a
CFB Converter Services to EJBs (includes EJBRMI)	n.a	n.a	n.a	n.a	n.a	X	n.a
Web Services Middleware	n.a	n.a	n.a	X	n.a	X	n.a

1. TCP/IP can be used to flow directly to the DP server environment.
2. TCP/IP can be used to flow to the DP server environment using CA Gen Communications Bridge product.
3. TCP/IP can be used to flow the EJB environment using the CA Gen CFB Converter Services to EJBs component.

Note: You must also install the C runtime files and selected middleware runtime software on this computer.

Install Considerations for C Proxy (UNIX)

Development Machine

You must install the Proxy Programming Interface C to generate a C proxy.

Deployment Machine

Install the following products on the proxy deployment machines:

- Proxy Programming Interface C
- Depending on which DP server the C Proxy is flowing to, you must install one or more of the following products:

Products	Distributed Processing Server						
	CICS	IMS	MQSeries	TE	Tuxedo	EJB	.Net
TCP/IP Middleware	X 1,2	X 1,2	n.a	X 1,2	X 1,2	X3	n.a
MQSeries Middleware	X	X	X	n.a	n.a	n.a	n.a
Tuxedo Middleware	n.a	n.a	n.a	n.a	X	n.a	n.a
Enterprise Java Beans (includes EJBRI)	n.a	n.a	n.a	n.a	n.a	X	n.a
CFB Converter Services to EJBs (includes EJBRI)	n.a	n.a	n.a	n.a	n.a	X	n.a
Web Services Middleware	n.a	n.a	n.a	X	n.a	X	n.a

1. TCP/IP can be used to flow directly to the DP server environment.
2. TCP/IP can be used to flow to the DP server environment using CA Gen Communications Bridge product.
3. TCP/IP can be used to flow the EJB environment using the CA Gen CFB Converter Services to EJBs component.

Note: You must also install the C runtime files and selected middleware runtime software on this computer.

Common Features of the Proxies

Depending on your application, you need to use the most appropriate proxy. However, the .NET, Java, COM, and C Proxies share common features listed as follows:

- Allow access to CA Gen servers.
- Provide additional ways of building multi-tier applications.

- Provide a supporting interface for use with user-written code.
- Support multiple communication protocols. Each proxy uses a transport mechanism to communicate with a server execution environment.
- By default, each generated proxy provides a synchronous programming interface that can be accessed by user-written code. A CA Gen application developer can create proxies containing additional programming interface calls that support asynchronous flows to the target DPS. The communication protocol used must support asynchronous operation. Oracle Tuxedo does not support asynchronous communications.
- By default, each generated proxy provides a language specific definition to the data being exchanged, between the user-written client and server. A CA Gen application developer can create proxies that provide an XML interface. The XML interface can be used to provide a more flexible mechanism of exchanging data with the end user of a user-written application.

Note: The XML interface is not available for C proxy.

Optional Features of the Proxies

A .NET, Java, Java Classic, or COM proxy can be generated to include optional programming interfaces. These interfaces expose support that can be used by the user-written application. These optional proxy interfaces include:

- APIs supporting asynchronous flows
- APIs supporting the use of XML

More information:

[Generating the Proxy](#) (see page 31)

Asynchronous Flow Support

Using CA Gen proxies, a user-written client application can make asynchronous requests to a generated server. The set of additional proxy interfaces includes APIs that provide a user-written client application with the ability to communicate with a target DPS in an asynchronous fashion.

Similar to the action language for generated client applications, asynchronous support for a proxy only influences the handling of a response to a request. Initiating a request to a target server is no longer synchronously linked with obtaining its response. Once a request is accepted for processing by the Proxy Runtime, the user-written client application can engage itself in other processing. The server processing proceeds asynchronously to the client processing. The client application is able to obtain (or ignore) a given outstanding response when it is best suited in the application logic.

More information:

[.NET Proxy](#) (see page 49)

[Java Proxy](#) (see page 73)

[Java Proxy \(Classic Style\)](#) (see page 97)

[COM Proxy](#) (see page 119)

[C Proxy](#) (see page 147)

XML Support

The .NET, Java, Java Classic, and COM proxies are capable of supporting the importing and exporting of view data using XML. An optional API can be generated as part of a generated proxy to support this capability. The API exposes extra methods that a user-written application can use to execute a flow where the import and export views are XML byte streams.

Note: The XML interface is not available for C proxy.

More information:

[.NET Proxy](#) (see page 49)

[Java Proxy](#) (see page 73)

[Java Proxy \(Classic Style\)](#) (see page 97)

[COM Proxy](#) (see page 119)

XML Schema Support

CA Gen proxies can perform XML schema validation of XML data streams. To do this, CA Gen generates an XML Schema Definition file (XSD) for each method on the proxy. This file defines and enforces all the data validation supported by the proxies, and the overall structure of the XML data itself.

To perform schema validation, the XML data streams need to refer to this XSD file. This can be done by using `noNamespaceSchemaLocation` setting. The sample XML file contains an example of how this can be done.

To disable schema validation, do not use the `noNamespaceSchemaLocation` setting in the XML data streams.

XML Data Structure

The XML schema requires one of the three types of data to be contained within XML. The three types are described as follows:

- Request - Holds the import data
- Response - Holds the export data
- Error - Holds any error information

Request XML

A request for a server named MYPSTEP1 in an XML document looks like this:

```
<?xml version="1.0"?>
<MYPSTEP1      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:noNamespaceSchemaLocation="MYPSTEP1.xsd">
    <request ...>
        <id>1</id>
    </request>
</MYPSTEP1>
```

The XML Schema is referenced in the topmost element, MYPSTEP1. The contents on the `<request>` element shown by “...” are the control/system attributes. If a system attribute is specified, the syntax is:

```
<request command="LIST" nextLocation="TEST">
```

XML Request Attribute Mappings for Java Proxy and .NET Proxies

The following table lists the XML attribute names and their corresponding properties, on the Java Proxy and .NET Proxy generated import view object.

XML Attribute Name	Java Proxy Property	.NET Proxy Property
command	Command	Command
clientId	ClientId	ClientId
clientPassword	ClientPassword	ClientPassword
nextLocation	NextLocation	NextLocation
exitState	ExitState	ExitState
dialect	Dialect	Dialect

XML Request Attribute Mappings for Java Proxy (Classic Style) and COM Proxies

The following table lists XML attributes and their corresponding Java Proxy (Classic Style) and COM Proxy properties.

XML Attribute Name	Java Proxy (Classic Style) Property	COM Proxy Property
command	CommandSent	CommandSent
clientId	ClientId	ClientId
clientPassword	ClientPassword	ClientPassword
nextLocation	NextLocation	Location
exitState	ExitStateSent	ExitStateSent
dialect	Dialect	n.a
comCfg	ComCfg	ComCfg
serverLocation	ServerLocation	ServerLocation
servletPath	ServletPath	n.a
fileEncoding	FileEncoding	n.a
tracing	Tracing	n.a

The comCfg XML attribute does not directly correspond to an import view object property. Instead, it is equivalent to a parameter present on the various method calls on the proxies for specifying the encoding setting. The method parameter overrides the XML and any XML overrides the defaults.

Specifying the Tracing XML attribute, results in the equivalent of executing the start and stop tracing methods on the proxy object.

Response XML

A response for a server named MYPSTEP1 in an XML document would look like this:

```
<?xml version="1.0"?>
<MYPSTEP1      xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
               xsi:noNamespaceSchemaLocation="MYPSTEP1.xsd">
  <response      command="LIST"
                 exitState="123456789"
                 exitStateType="OK"
                 exitStateMsg="Processing Completed OK">
    <name>John Smith</name>
  </response>
</MYPSTEP1>
```

The XML Schema for responses is referenced the same way as it is for the requests. The <request>, <response> and <error> schema information is contained in one XML Schema file. The system attributes are returned as attributes on the <response> element.

The view data is always returned if a <response> element is returned, but its values may or may not reflect the activity that occurred on the server, depending on the application logic of the server. If the ExitStateType or OperationStatus values indicate an error status from the server, the export views may or may not be trusted, depending on how the server was written.

XML Response Attribute Mappings for Java Proxy and .NET Proxies

The following table lists the response XML attributes and their corresponding properties on the Java Proxy and .NET Proxies.

XML Attribute Name	Java Proxy Property	.NET Proxy Property
command	Command	Command
exitState	ExitState	ExitState
exitStateType	ExitStateType	ExitStateType
exitStateMsg	ExitStateMessage	ExitStateMessage

XML Response Attribute Mappings for Java Proxy (Classic Style) and COM Proxies

The following table lists the response XML attributes and their corresponding properties on the Java Proxy (Classic Style) and COM Proxies.

XML Attribute Name	Java Proxy (Classic Style) Property	COM Proxy Property
command	CommandReturned	CommandReturned
exitState	ExitStateReturned	ExitStateReturned
exitStateType	ExitStateType	OperationStatus
exitStateMsg	ExitStateMsg	OperationStatusMsg

Error XML

An error response from a server named “MYPSTEP1” in an XML document looks like this:

```
<?xml version="1.0"?>
<MYPSTEP1      xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
               xsi:noNamespaceSchemaLocation="MYPSTEP1.xsd">
    <error>
        ...
    </error>
</MYPSTEP1>
```

The XML Schema for errors is referenced the same way as for the requests and responses. The error element contains sub-elements that contain specific error information. The following table lists the sub-elements that may be present in an error element.

Note: Not all error elements may be present at any given execution. The proxies return a different set of elements, as different information is available to each programming language when the error is detected.

Sub-element Name	Description	COM Proxy	.Net Proxy	Java Proxy	Java Classic Proxy
number	The error number	X			
type	A brief description/name of the error		X	X	X
description	The error message text	X	X	X	X
trace	Any trace or debug information available		X	X	X

View to XML Mapping

Within a request or response element, import and export view data follow similar rules. For example, entity and work views become sub-elements, attributes become sub-elements of entity views, and group views become sub-elements that contain row elements, which themselves contain entity and work view elements.

This is best explained by the following sample import view:

View (Import)	XML
IMPORTS:	<request>
Entity View in proxy	<inproxy>
text1	<text1>...</text1>
text2	<text2>...</text2>
	</inproxy>
Group View inGV (2)	<inGV>
Entity View ingroup proxy	<row>
text1	<ingroupproxy>
text2	<text1>...</text1>
	<text2>...</text2>
	</ingroupproxy>
	</row>
	<row>
	<ingroupproxy>
	<text1>...</text1>
	<text2>...</text2>
	</ingroupproxy>
	</row>
	</inGV>
	</request>

XML and Standard Proxy Support

For the COM and Java (Classic Style) proxies, XML support is implemented to work on top of the standard support. These two proxies are inherently stateful and thus, an executeXML call modifies the proxy import view, export view, and standard properties. In general, after an executeXML is executed, the error information or export data can be retrieved by passing the export XML or by calling the various get methods on the individual properties.

The .NET and Java proxies are stateless, so the XML methods do not modify any state of the proxy.

Selecting a Proxy

There are five types of proxies to choose from. What are the scenarios to which each should be applied? A comparison of the proxies should help you make an informed decision.

Platform

Your hardware and operating system (the platform) limit the proxies you can use. Does the application need to be portable across operating systems? Can the application be limited to Windows? Is a UNIX or MVS platform required? The following table shows how each proxy lines up with the possible choices:

Proxy Type	Platform Choices
.NET Proxy	Limited to Windows operating systems only, but with a possibility of ports to other platforms
Java Proxy	No real platform limitations
Java Proxy (Classic Style)	No real platform limitations
COM Proxy	Limited to Windows operating systems only
C Proxy	Limited to Windows and UNIX operating systems (HP-UX, AIX, Solaris)

Language

Language is the next most important limiting factor. Each proxy is written with a specific technology (or language) in mind. Although it is possible to write code to call each proxy from any particular language, it would not be useful to do it that way. The following table lists the proxies and the languages from which each proxy can be accessed:

Proxy Type	Target Application Languages
.NET Proxy	Any .NET Framework aware language. Most commonly, Managed C/C++, C#, and VB.NET.
Java Proxy	Java
Java Proxy (Classic Style)	Java
COM Proxy	Any COM or Automation aware language. Historically, C/C++ and Visual Basic. In theory any .NET Framework aware language could also call a COM object.

Proxy Type	Target Application Languages
C Proxy	C or C++

Programming Style

Each proxy has a flavor or style to its API. Each style has different applications, possibly different learning curves, and productivity rates for the application developers. The following table attempts to show the natural programming style of each proxy type:

Proxy Type	Programming Styles
.NET Proxy	Object-oriented, stateless.
Java Proxy	Object-oriented, stateless.
Java Proxy (Classic Style)	Stateful and somewhat procedural.
COM Proxy	Automation, ActiveX, DCOM, stateful, and somewhat procedural.
C Proxy	Stateful and procedural.

Next Generation

The .NET and Java Proxy are second-generation proxy designs compared to the COM and Java Proxy (Classic Style) proxies. The designs were improved and many of the limitations inherent in their first implementation have been removed. It is encouraged that new user-written application development make use of these next generation proxies.

Theoretically, the more intuitive and object-oriented nature of the .NET and Java Proxies results in better programmer productivity, as compared to COM and Java Proxy (Classic Style) proxies.

Related Information

Throughout this document, the subfolder name Gen xx is used.

Also throughout this document, the environment variable GENxx is used.

Note: xx refers to the current release of CA Gen. For the current release number, see the *Release Notes*.

Chapter 2: Generating the Proxy

A generated proxy provides a programmatic interface that user-written applications can use to communicate with a cooperatively packaged Server Manager. An application developer can use the CA Gen Toolset or Client Server Encyclopedia (CSE) to generate a proxy.

The .NET, Java, Java Classic Style, and COM proxies are built using the CA Gen Build Tool. The C proxy is generated but does not need the Build Tool, as the generated source code is compiled by the user.

This chapter describes the process of generating a proxy. The proxy specific chapters of this guide discuss the output of generation and subsequent use of a specific type of proxy by a user-written application.

Although the generation procedures for the .NET, Java, Java (Classic Style), COM, and C proxies are similar, there are differences noted throughout the chapter.

Before You Start

Before you generate a proxy, you must complete the following three tasks:

Install Proxy Support

Before generating a proxy, an application developer must install the proxy software. The proxy software is a separately licensed product and must be installed before generating a proxy.

More information:

[Installing the Proxy Software](#) (see page 15)

Complete Modeling Activities

The application developer must complete all modeling activities necessary to define a cooperatively packaged Server Manager. When generating a proxy, the Server Environment settings of the Server Manager dictate the transport protocol that is generated as part of the proxy. The Server Environment settings become the default configuration of that proxy.

The execution environment settings that are generated as part of the proxy can be overridden at runtime.

Note: For more information about overriding the generation time configuration of a proxy, see the chapter "Overriding Communications Support at Execution Time" in the *Distributed Processing - Overview Guide*.

Associate Character Names

Proxies use information from the model such as pstep and entity names. The file PROXYNMS.DAT is used at generation time to associate alternative names with the model names. The file is created during proxy generation if the environment variable GEN_PROXY_WRITE_NAMES is set.

There are two situations where this file is used:

- Your model uses NLS (National Language Set) characters.
- Valid characters are ASCII A through Z, 0 through 9, and underscore. The proxy generator removes NLS characters from names that would cause compilation errors in the proxies. NLS characters that do not affect compilation are not removed (such as in literal strings and permitted values). This removal can cause problems if the name contained only NLS characters or if the name is now no longer unique. For example, the name Ñ would be an empty string causing problems in the proxy.
- Names used in the model are awkward.

Your model may use names that are non-descriptive, excessively long or otherwise not useful. For example, the pstep name might be STEP3, which is not descriptive. You can use a name such as CREATE_CUSTOMER_RECORD.

Note: Many model names cannot be overridden in this manner, so it is more beneficial to change the model names instead.

All proxies that are generated for the model except the Standard Java Proxy use the file PROXYNMS.DAT located in the <model-name>.ief directory. Regenerating the proxy updates the file with new names while not affecting old names. Be sure to check the file after each generation.

Delete the GEN_PROXY_WRITE_NAMES environment variable if you do not want to create or update the PROXYNMS.DAT file.

If you use the PROXYNMS.DAT file, remember that it has an impact on the generated code. Include this file when you distribute the model or track model changes using a code management tool.

The following is a sample PROXYNMS.DAT file. On the left side of the equal sign are the names used in the model. On the right side of the equal sign are the names used in the proxy. Note the duplicate name PRXLS and the empty string used for ÑĚ.

```
PRŌXÝŇLS=PRXLS

ŎPRŌXÝŇLS=PRXLS
S_PROXY-NLS_ŇÁMĚ=S_PROXY-NLS_M
ŇĚ=
NLS_WŎRKĬNG_LİST=NLS_WRKNG_LST
İD=D
ŎÜT=T
```

To correct these problems, edit the file using an editor that supports NLS characters, and then regenerate the proxy. The proxy generator checks that the names contain valid characters and are the correct length. If necessary, the invalid characters are removed and the name is truncated.

The following is a sample PROXYNMS.DAT file after it has been edited:

```
PRŌXÝŇLS=PRXLS

ŎPRŌXÝŇLS=OPRXLS
S_PROXY-NLS_ŇÁMĚ=S_PROXY-NLS_NAME
ŇĚ=NE
NLS_WŎRKĬNG_LİST=NLS_WRKNG_LST
İD=D
ŎÜT=T
```

Note: Be sure to check the PROXYNMS.DAT file each time the proxy is generated and after the model or PROXYNMS.DAT file is changed.

Toolset Generation

Only cooperatively packaged Server Manager load modules are eligible for proxy generation. A model does not need to have a client defined. A proxy is a generated programmatic interface of a Server Manager.

The following steps describe how to generate a proxy. This example uses Toolset Menu navigation to designate diagrams and dialogs. You can use alternate navigation techniques to achieve the same result.

Follow these steps:

1. Start the CA Gen Toolset.
2. Click Tools, Construction, Generation or Packaging to open the Generation or Packaging Diagram.

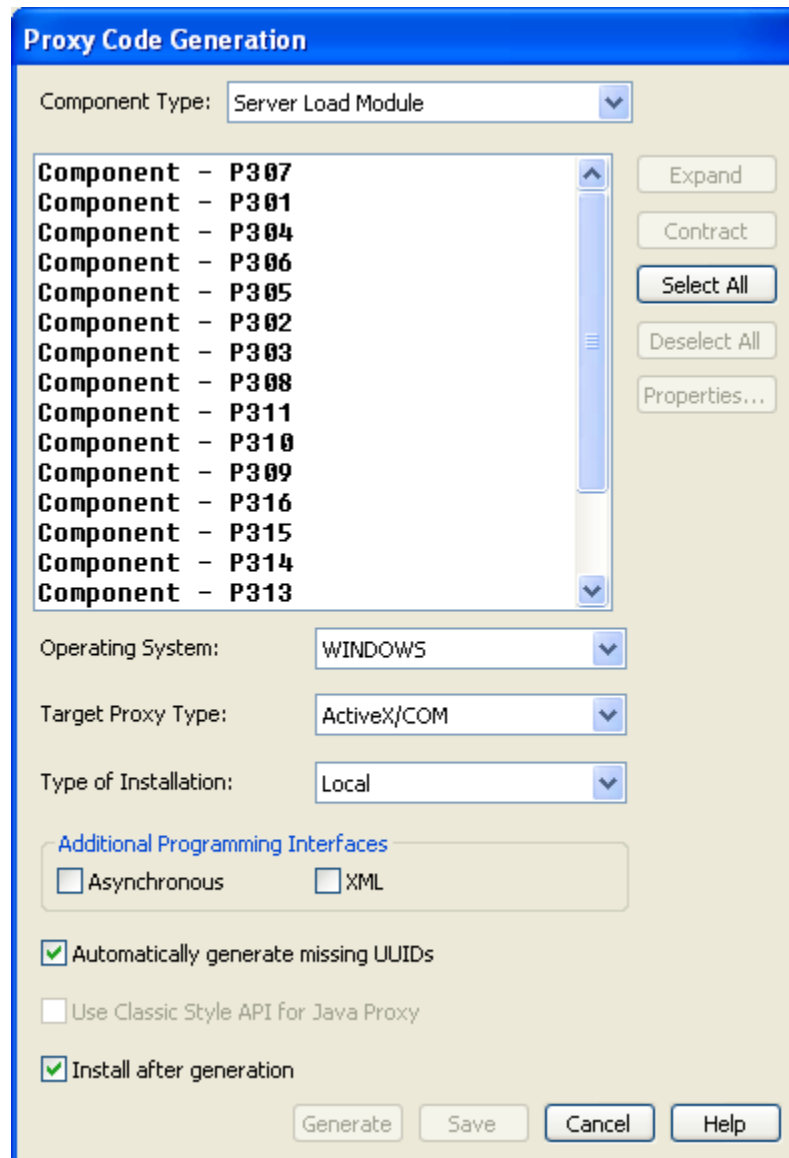
3. To open the Cooperative Code definition, click Diagram, Open, Cooperative Code.
4. To set the parameters of the Server Manager execution environment, select the Business System and click Details, Server Environment.

The Server Environment Parameters dialog opens where you can define the parameters.

Note: Server Environment Parameters can be set at the Business System level or for each individual Server Manager. If the Server Environment Parameters are set at the Business System level, they become the default setting for each Server Manager within the Business System.

5. Click Diagram, Open, Proxy Code.

The Proxy Code Generation dialog opens.



- a. From the Component Type list, select either CBD 96 or Server Load Module to generate proxies for models that are not formatted according to CBD component standards. For components that conform to CBD 3.0 standards. Select Custom Proxies to generate custom proxies designed using CA Gen Studio

Note: When a model does not contain any CBD or CBD 96 component, but contains cooperative server load modules, the Server Load Module component type is automatically selected.

- b. From the Component list, select one or more components you want to generate. When multiple components are defined for a single model, you can select one or more of these components for generation. If you have selected Custom Proxies in the previous list, the custom proxies that were defined in CA Gen Studio are displayed here. For information about defining custom proxies, see *Gen Studio Overview Guide*.

Note: The Generation process always generates all interfaces and methods associated with the selected component.

- c. From the Operating System list, select WINDOWS, JVM, UNIX, or CLR.

Note: The Java proxy generates only for the JVM operating system. The Windows Build Tool builds the generated code, and you can deploy it to any Java-compliant environment.

From the Target Proxy Type list, select the type of generated code you want for the proxy.

- d. From the Type of Installation list, select Local or Remote.

For a Remote install, generated files are packaged into one installation file that you can manually transfer to any target system. For a Local install, generated files are compiled and prepared for copying onto a server running on the same type of machine as the Toolset. The default is Local.

- e. Under Additional Programming Interfaces, select any of the optional APIs that need to be generated into the proxy. These include:

- Support for Asynchronous flow to the Server Manager
- Support for a user-written application to use XML to process import and export views.

Note: The Additional Programming Interfaces option is disabled when generating C Proxies. The C proxy is a generated C language header file. The C proxy runtime provides the supported programming interfaces.

- f. For ActiveX/COM only, select Automatically generate missing UUIDs (Universally Unique identifier) if you want CA Gen to automatically create any missing UUIDs.

By default, this option is selected. ActiveX/COM proxies require a UUID for each component, interface, and method objects.

To specify the missing UUIDs manually, click Properties.

- g. For Java proxy generation only, select Use Classic Style API for Java proxy. By default, the new object oriented Java proxy API is generated. Therefore, by default it is not selected.

Note: This option is disabled if you have selected Custom Proxies option for Component Type.

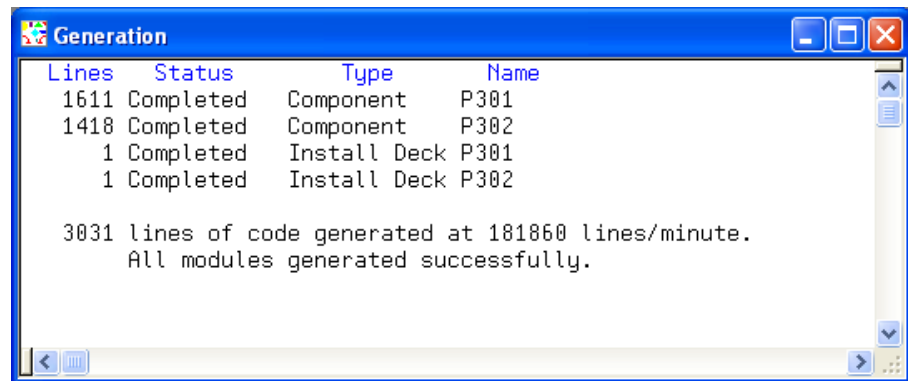
- h. To automatically start the Build Tool following the generation of a proxy, select Install after generation.

Note: The Install after generation option is disabled when generating C Proxies. The C proxy is a generated C language header file. The C proxy does not need the Build Tool, as the generated source code is compiled by the user.

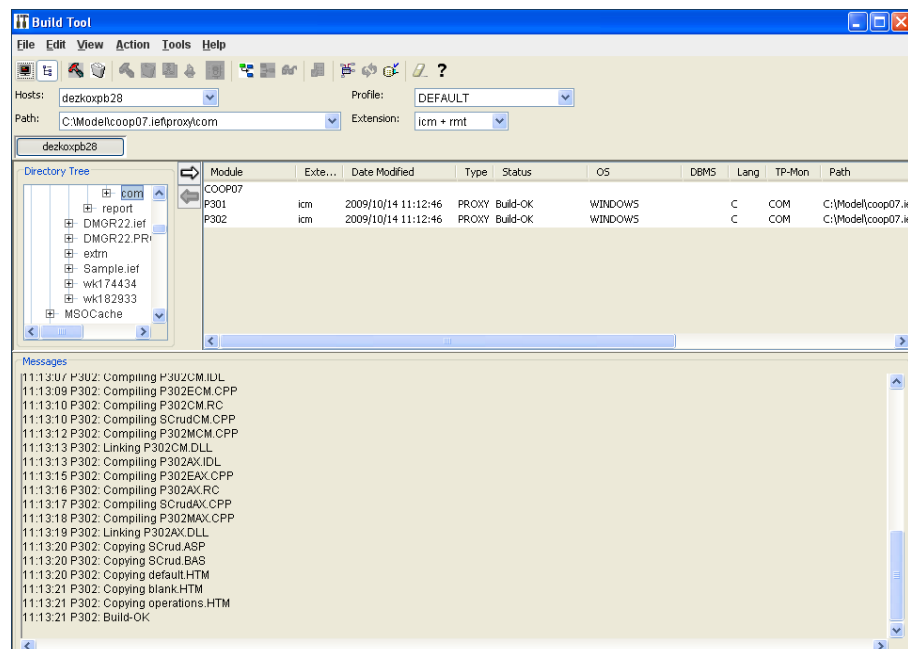
- i. To start generation of the proxy following the selection of all the parameters, click Generate.

Note: Generation fails if proper licensing does not exist for the proxy.

On successful generation, the Generation window displays the following message:



The Build Tool appears and a successful build displays a Build-OK message.



The following table lists the Target Proxy Type options for a given operating system:

Operating System	Target Proxy Type
Windows	ActiveX/COM, C
JVM	Java
CLR	.Net
UNIX	C

More information:

[.NET Proxy](#) (see page 49)

[Java Proxy](#) (see page 73)

[Java Proxy \(Classic Style\)](#) (see page 97)

[COM Proxy](#) (see page 119)

[C Proxy](#) (see page 147)

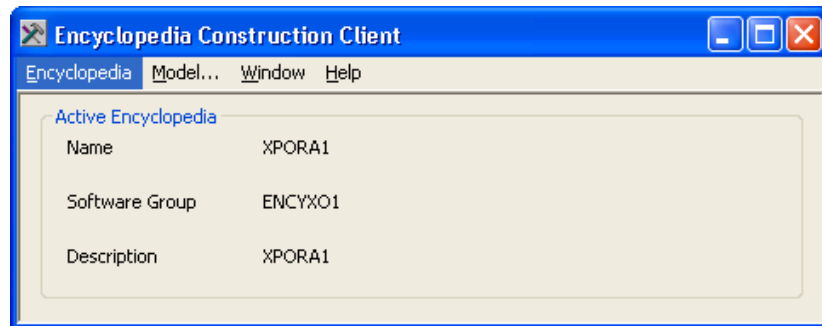
CSE Generation

The following steps describe how to generate a proxy from a Windows CSE. You can use CSE Construction Client Menu navigation to designate diagrams and dialogs. Other navigation techniques can also be used to achieve the same result.

Note: Proxy clients may only be generated from a Windows CSE and is not supported from a UNIX CSE.

Follow these steps:

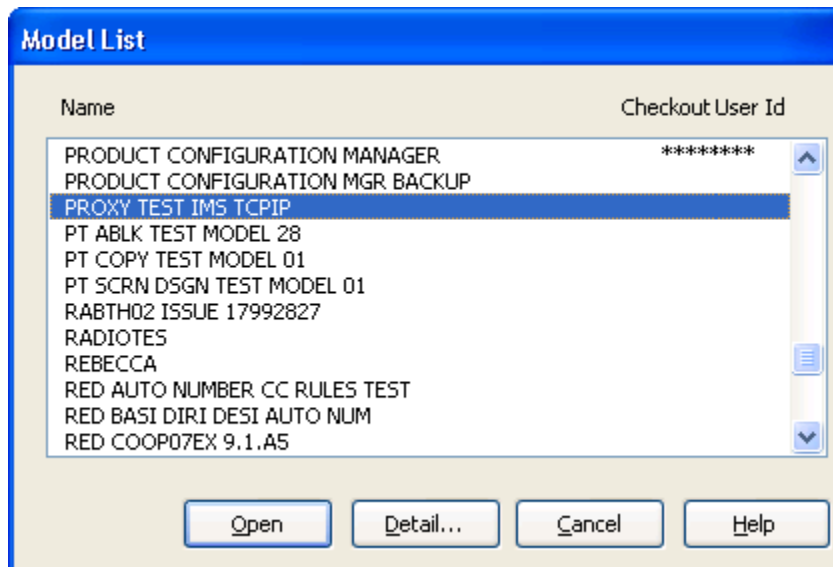
1. Start the CA Gen CSE Construction Client.



2. Click Model, Actions, List.

The Model List dialog opens.

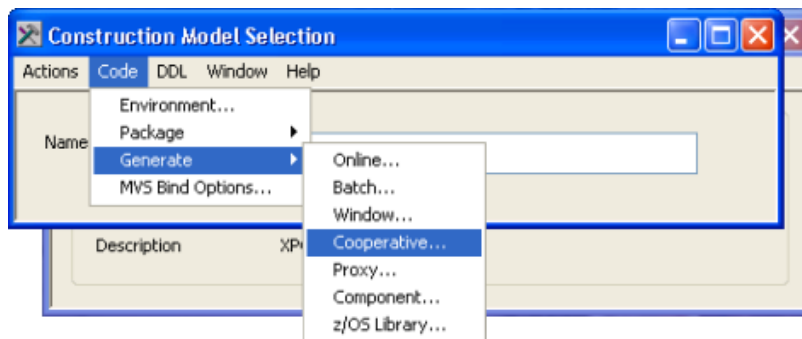
3. Select the model name as shown in the following example:



4. Click Open to open the model that contains the Server Manager for which a proxy is to be generated.

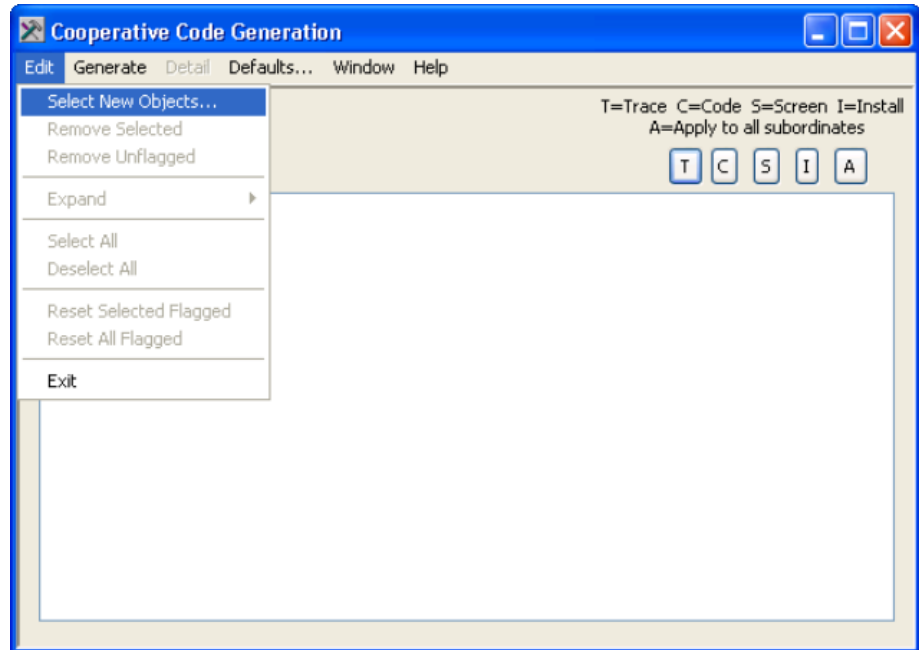
The Construction Model Selection window opens.

5. To open the model to generate Cooperative Code, click Code, Generate, Cooperative.



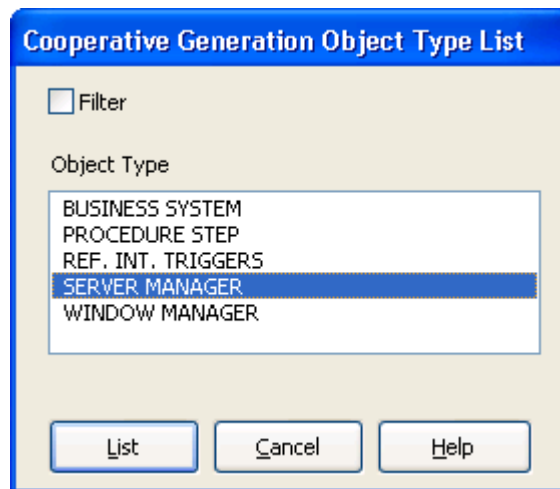
The Cooperative Code Generation window opens.

6. To add the Server Manager, click Edit, Select New Objects.



The Cooperative Generation Object Type List dialog opens.

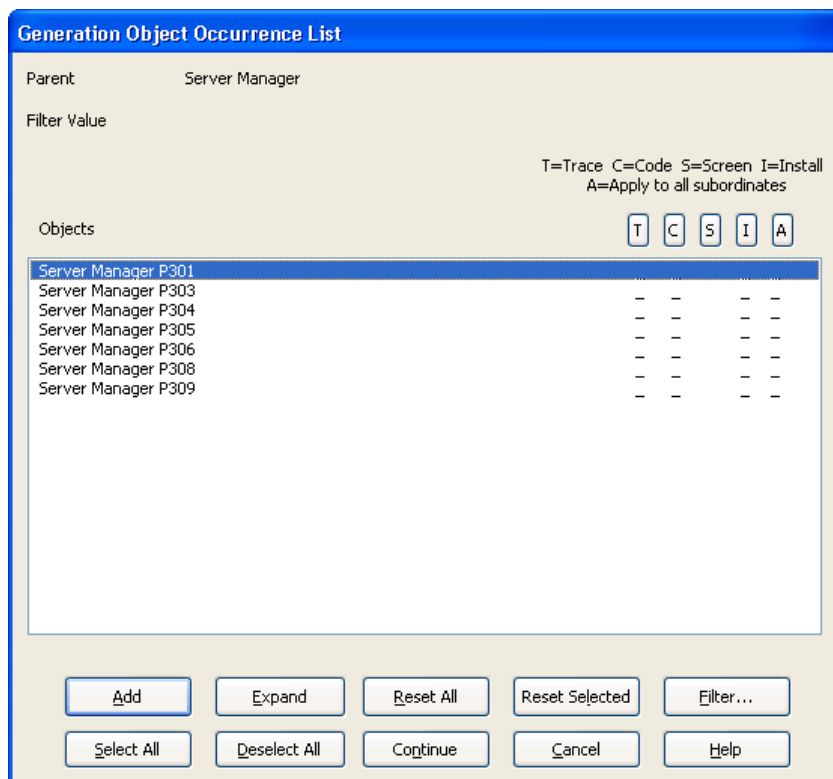
7. Select Server Manager, and click List.



The Generation Object Occurrence List opens. A list of server managers is displayed.

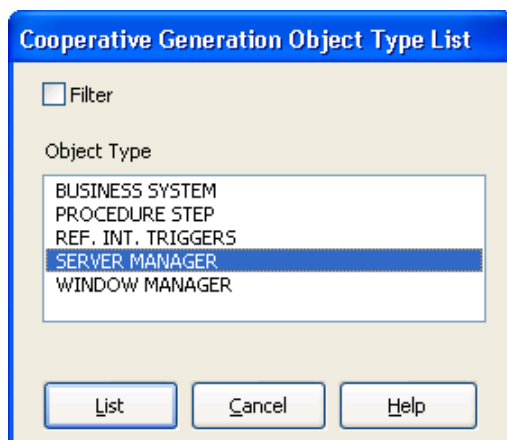
8. Select a Server Manager, click Add, Continue.

This adds the Server Manager to the Cooperative Application Generation, closes the Generation Object Occurrence List, and returns to Cooperative Generation Object Type List.

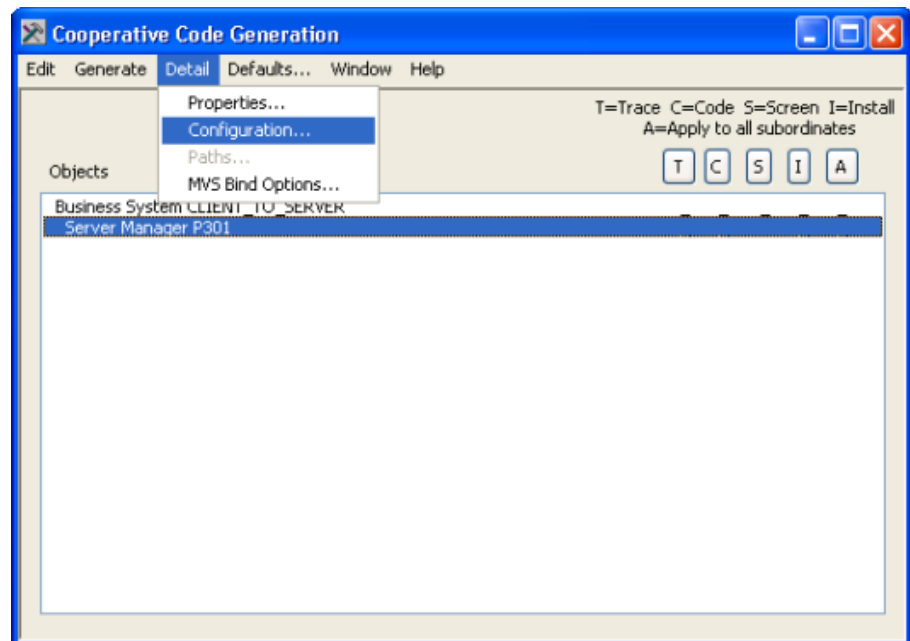


9. Click Cancel on the Cooperative Generation Object Type List dialog.

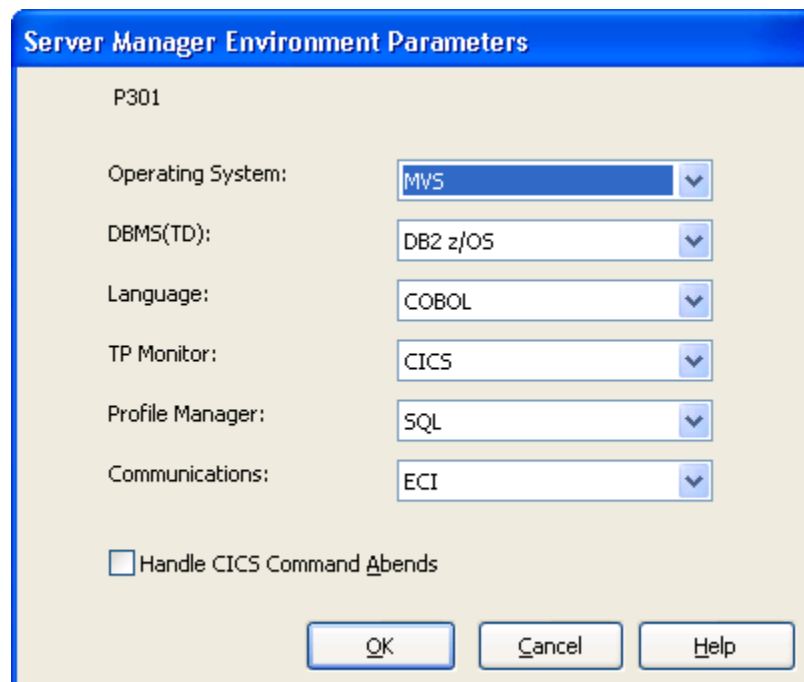
This closes the dialog and returns back to Cooperative Code Generation.



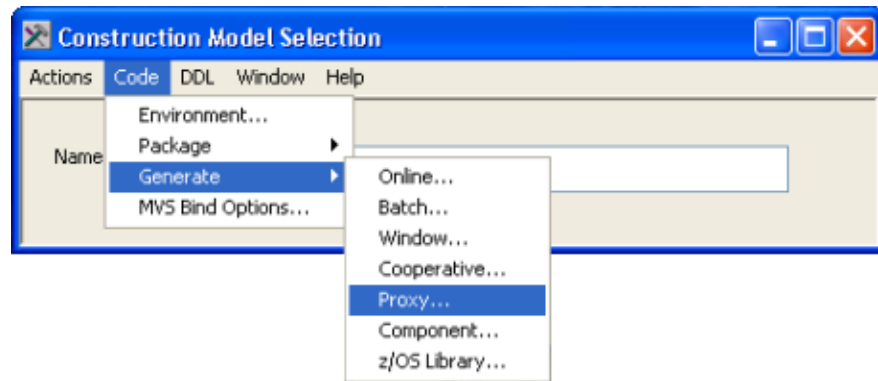
10. Select the Server Manager on the Cooperative Code Generation window and click Detail, Configuration.



The Server Manager Environment Parameters dialog is displayed for the selected Server Manager.

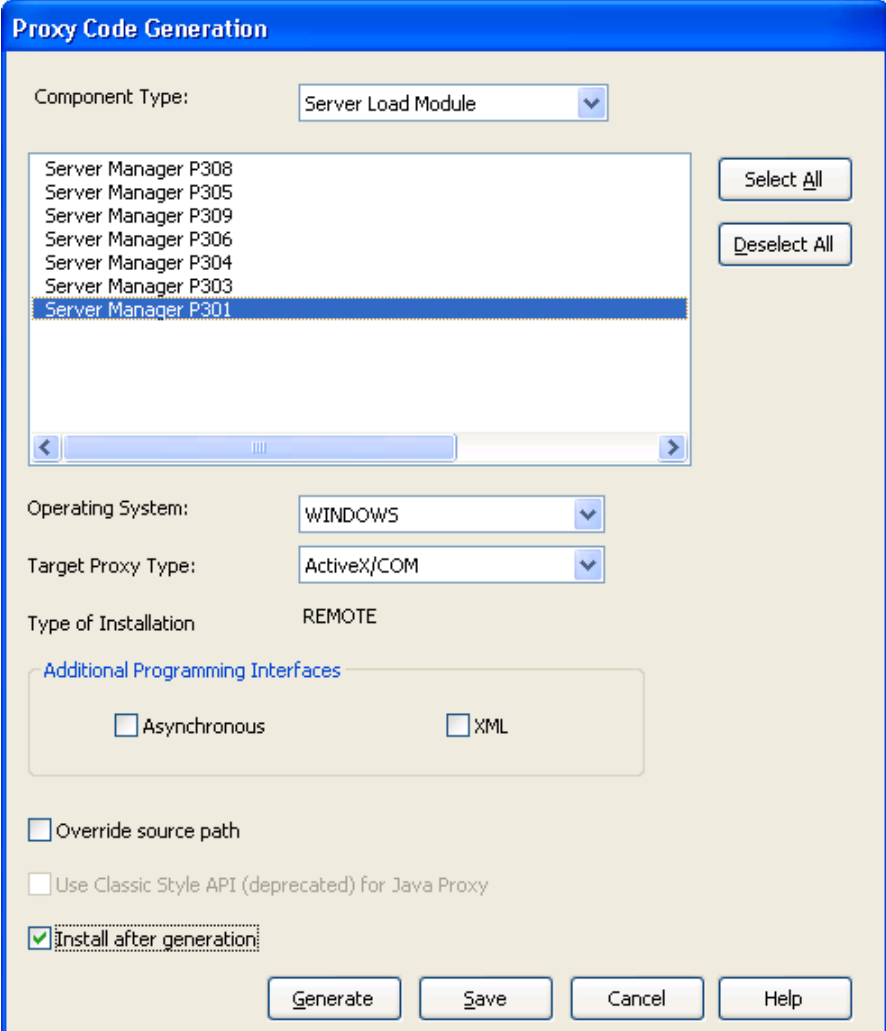


11. Specify the environment for the selected server by setting each parameter. Click OK to accept the specified Server Manager Environment Parameters and exit back to the Cooperative Application Generation dialog.
12. In the Construction Model Selection window, click Code, Generate, Proxy to open the model for Proxy Code Generation.



The Proxy Code Generation dialog opens.

13. Select the Server Manager for which you want to generate a proxy.



The image shows a 'Proxy Code Generation' dialog box. At the top, 'Component Type' is set to 'Server Load Module'. Below this is a list of server managers: P308, P305, P309, P306, P304, P303, and P301. 'Server Manager P301' is selected. To the right of the list are 'Select All' and 'Deselect All' buttons. Below the list is a scrollbar. Further down, 'Operating System' is set to 'WINDOWS' and 'Target Proxy Type' is set to 'ActiveX/COM'. 'Type of Installation' is set to 'REMOTE'. There is a section for 'Additional Programming Interfaces' with checkboxes for 'Asynchronous' and 'XML', both of which are unchecked. Below this are checkboxes for 'Override source path' (unchecked), 'Use Classic Style API (deprecated) for Java Proxy' (unchecked), and 'Install after generation' (checked). At the bottom are 'Generate', 'Save', 'Cancel', and 'Help' buttons.

- a. From the Component Type list, select either CBD 96 or Server Load Module to generate proxies for models that are not formatted according to CBD component standards. For components that conform to CBD 3.0 standards, select Component. Select Custom Proxies to generate custom proxies designed using CA Gen Studio.

Note: When a model contains no CBD or CBD 96 components but does contain cooperative server load modules, the Server Load Module component type is automatically selected.

- b. From the Component list, select one or more components you want to generate. When multiple components are defined for a single model, you can select one or more of these components for generation. If you have selected Custom Proxies in the previous step, a list of custom proxies that were defined in CA Gen Studio are displayed in this list. For more information about defining custom proxies, see *Gen Studio Overview Guide*.

Note: The Generation process always generates all interfaces and methods associated with the selected component.

- c. From the Operating System list, select WINDOWS, JVM, UNIX, or CLR.

Note: The Java proxy generates only for the JVM operating system. The Windows Build Tool builds the generated code and you can deploy it to any Java-compliant environment.

From the Target Proxy Type list, select the type of generated code you want for the proxy.

The following table lists the Target Proxy Type options for a given operating system:

Operating System	Target proxy type
Windows	ActiveX/COM, C
JVM	Java
UNIX	C
CLR	.NET

- d. A CSE only generates Remote install packages. For a Remote install, generated files are packaged into one installation file that you can manually transfer to any target system.
- e. Under Additional Programming Interfaces, select any of the optional APIs that need to be generated into the proxy. These include:
- Support for Asynchronous flow to the Server Manager
 - Support for a user-written application to use XML to process import and export views.

Note: The Additional Programming Interfaces option is disabled when generating C Proxies. The C proxy is a generated C language header file. The C proxy runtime provides the supported programming interfaces. The Override Source Path option specifies whether to override the default destination source path with the value specified in the entry field.

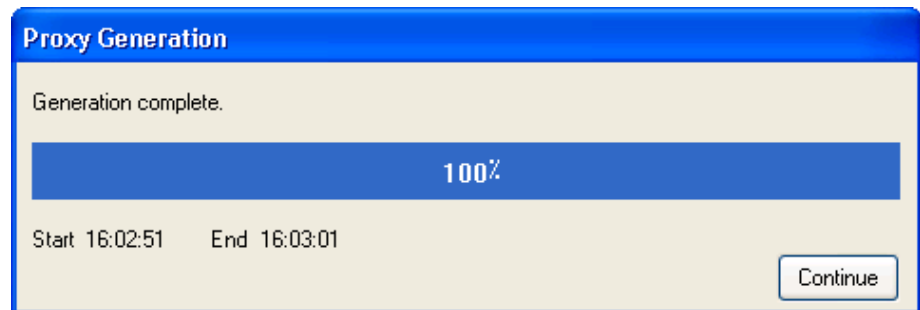
If Override source path is selected, the value you enter in the entry field is used as the destination source path in the target environment.

- f. If Override source path is not selected, the path assigned in the proxy generation is used.

Select Install after generation to automatically package the generated files into a remote file.

Note: The Install after generation option is disabled when generating C proxies. The C proxy is a generated C language header file. The C proxy does not need the Build Tool, as the generated source code is compiled by the user.

The Proxy Generation dialog opens. After the generation completes, it displays a Generation Complete message.



The Proxy Generation Report opens in a new window. The status of proxy generation can be verified from this.



The Proxy Generation produces a list of generated source files. If the option Install after generation is selected on the Proxy Generation dialog, a remote file is also generated.

You can find the source files in the server directory specified in the Override source path on the Proxy Generation dialog. If the source path is not specified, the source is generated in the CSE directory under the userid initiating the generation. Proxy Generation always produces a proxy directory and a subdirectory for the Target Proxy Type.

For the previous example, you can find the source code in:
\\CA\CSE\ENCY\proxy\com\src.

More information:

[C Proxy](#) (see page 147)

Chapter 3: .NET Proxy

The .NET Proxy is a generated programmatic interface that .NET user-written applications can use to access generated CA Gen Distributed Processing Server (DPS) applications. The user-written applications can be either a native .NET application or one created using CA Gen and ASP.NET clients.

.NET Proxy Generated Code

A generated .NET proxy is a set of object-oriented classes that correspond to the set of methods selected during generation. There are three main classes that a developer can access and use. Other classes are generated, but they are not exposed for use by the application. The three exposed classes are:

- <method-name> Referred to as the .NET Proxy object
- <method-name> Import Referred to as the Import object
- <method-name> Export Referred to as the Export object

This chapter discusses the details of the three generated interfaces.

In addition to the generated interface, a .NET Proxy also consists of runtime code and a set of user modifiable exit routines. The user exits influence how the runtime handles security and communications processing.

The creation of an executable .NET Proxy involves both generating the .NET Proxy source code and installing the proxy. You can choose to do these two operations as two independent activities or as a single operation (select Install after generation option on the Proxy Code Generation dialog).

Generating a .NET Proxy results in a \proxy\net directory structure being added to your <model-name>.ief directory. The generated proxy source code is contained in a directory structure under \proxy\net\src. The installed, or built, .NET proxy code is contained in a directory structure \proxy\net\deploy.

The following file map shows the structure of the generated directories:

The generated proxy source code is placed in the \src\<component> directory. After the Build Tool has successfully built the proxy, the \proxy\net directory includes a \deploy\<component> directory. This directory contains a <component>.dll. This dll is the .NET Proxy and the user-written code uses this.

The \deploy\<component> directory includes a \samples directory. Within the \samples directory, there are APP, ASP.NET and an option XML directory.

Select the optional XML API only if you want to generate the XML sample. The ASP.NET, APP, and XML directories contain the generated samples. You can use the sample to test the generated proxy. The generated sample code also demonstrates the possible usage of the proxy. The samples are not considered part of the proxy.

If you select XML API for generation, then the \deploy\<component> directory also includes the XML schema file (.XSD). The XML schema file is considered part of the proxy itself.

More information:

[Generating the Proxy](#) (see page 31)

[User Exits](#) (see page 66)

.NET Proxy Interface

The generated .NET proxy contains three main classes that an application developer can use. Each class exists within a namespace. The generator defines the namespace as either the short model name (default) or as a particular value specified at the business system or model level for the .NET Namespace.

The three exposed classes are:

- <method-name> Referred to as the .NET Proxy object.
- <method-name>Import Referred to as the Import object.
- <method-name>Export Referred to as the Export object.

The following sections discuss these classes in detail.

.NET Proxy Object

The .NET Proxy object is the generated class called <method-name>. It contains the actual execution methods for the proxy. The execution methods are designed to be stateless. All data is stored on the stack (passed as parameters) during execution. Many threads can simultaneously use one instance of the class.

The .NET proxy object contains following two types of methods:

- Trace control methods
- Execution methods

Trace Control Methods

There are five trace control methods. They are always generated with every .NET proxy object. These methods are used to start, stop, and write trace messages using the CA Gen trace capabilities.

Note: When used in this context, Trace refers more to runtime logging capability and you should not confuse this Trace with the Diagram Trace used to debug the generated action block logic.

The following table lists the five trace control methods:

Trace Control Method	Comments
StartTracing()	Turns on tracing with the output going to the file trace- <i><appname>-<procid>.out</i> , where <i><appname></i> is the application name, and <i><procid></i> is the application's process id). This file is located in the %USERPROFILE%\AppData\Local\CA\Genxx\logs\net directory. Note: xx refers to the current release of CA Gen. For the current release number, see the <i>Release Notes</i> .
StartTracing(String filename)	Turns on tracing with the output going to a file specified by the filename parameter.
StartTracing(TextWriter writer)	Turns on tracing with the output going to the TextWriter parameter passed in.
TraceOut(String message)	Writes the message to the trace output if tracing is enabled at that time. If tracing is not enabled, the method simply returns.
StopTracing()	Turns off all tracing.

Tracing in CA Gen runtime is global in scope. If one proxy turns Tracing on, it is enabled for all the proxies running in the same process.

Execution Methods

There are a variety of execution methods. Some are always generated with every .NET proxy object, but others are only generated if one or both of the optional APIs are selected during generation. The following table lists the execution methods, when they are generated, and a summary of what each one does.

Some execution methods make use of a comCfg string. This string takes the same form as is provided in the commcfg.txt file, with the exception that it should not contain the trancode parameter.

For example, for a TCP/IP transport configuration a commcfg.txt specification might be:

```
tranX TCP hostname portNumber
```

The corresponding ComCfg string would then be:

```
TCP hostname portNumber
```

Each supported transport has a slightly different communication string specification. In all cases the ComCfg string is expected to be the same as defined in the commcfg.txt file, but without the trancode parameter at the beginning.

Each execution method is generated as a web method. Microsoft .NET Framework and IIS are able to transform these web methods into Web Services. See the Microsoft documentation for steps to transform .NET classes into web services.

Execution Method	Comments
execute(importView)	Returns an exportView. This method performs a synchronous cooperative flow using the import data provided. It returns the export results and throws GenExceptions for any errors.
execute(importView, String comCfg)	Returns an exportView. This method performs a synchronous cooperative flow configured by the comCfg parameter using the import data provided. It returns the export results and throws GenExceptions for any errors.
xmlExecute(String importView)	(XML API only) Returns a string containing the export view data in XML format. This method performs a synchronous cooperative flow using the import data passed in XML format. It returns the export results and throws GenExceptions for any errors.

Execution Method	Comments
xmlExecute(String importView, String comCfg)	<p>(XML API only)</p> <p>Returns a string containing the export view data in XML format. This method performs a synchronous cooperative flow configured by the comCfg parameter, with the import data passed in XML format. It returns the export results and throws GenExceptions for any errors.</p>
asyncExecute(importView)	<p>(Async API only)</p> <p>Returns an int containing the id used to retrieve the asynchronous results later. This method begins an asynchronous cooperative flow using the import data provided. It throws GenExceptions for any errors.</p>
asyncExecute(importView, boolean noResponse)	<p>(Async API only)</p> <p>Returns an int containing the id used to retrieve the asynchronous results later. This method begins an asynchronous cooperative flow using the import data provided. The noResponse parameter indicates to the runtime whether any results are expected back. It throws GenExceptions for any errors.</p>
asyncExecute(importView, boolean noResponse, String comCfg)	<p>(Async API only)</p> <p>Returns an int containing the id used to retrieve the asynchronous results later. This method begins an asynchronous cooperative flow using the import data provided. The noResponse parameter indicates to the runtime whether any results are expected back. It throws GenExceptions for any errors.</p>

Execution Method	Comments
asyncGetResponse(int id)	(Async API only) Returns an exportView. This method retrieves the results of an asynchronous cooperative flow specified by the id provided. It returns the export results and throws GenExceptions for any errors.
asyncGetResponse(int id, boolean block)	(Async API only) Returns an exportView. This method retrieves the results of an asynchronous cooperative flow specified by the id provided. The call will block waiting for results if they are not available yet, depending on the block parameter. It returns the export results and throws GenExceptions for any errors.
asyncCheckResponse(int id)	(Async API only) Returns a FlowStatus enumeration. This method retrieves the status of an asynchronous cooperative flow specified by the id provided. It throws GenExceptions for any errors.
asyncIgnoreResponse(int id)	(Async API only) This method flags the results of an asynchronous cooperative flow as ignorable. Once ignored, the results become irretrievable. It throws GenExceptions for any errors (Flagging a flow as ignored lets the runtime clean up resources devoted to that flow).

Execution Method	Comments
<code>asyncXMLExecute(String importView)</code>	(Async API and XML API only) Returns an int containing the id used to retrieve the asynchronous results later. This method begins an asynchronous cooperative flow with the import data passed in XML format. It throws <code>GenExceptions</code> for any errors.
<code>asyncXMLExecute(String importView, boolean noResponse)</code>	(Async API and XML API only) Returns an int containing the id used to retrieve the asynchronous results later. This method begins an asynchronous cooperative with the import data passed in XML format. The <code>noResponse</code> parameter indicates the runtime if it expects any results back. It throws <code>GenExceptions</code> for any errors.
<code>asyncXMLGetResponse(int id)</code>	(Async API and XML API Only) Returns a string containing the export view data in XML format. This method retrieves the results of an asynchronous cooperative flow. It returns the export results and throws <code>GenExceptions</code> for any errors.
<code>asyncXMLExecute(String importView, boolean noResponse, String comCfg)</code>	(Async API and XML API Only) Returns an int containing the id used to retrieve the asynchronous results later. This method begins an asynchronous cooperative flow configured by the <code>comCfg</code> parameter, with the import data passed in XML format. The <code>noResponse</code> parameter indicates to the runtime whether any results are expected back. It throws <code>GenExceptions</code> for any errors.

Execution Method	Comments
asyncXMLGetResponse(int id, boolean block)	(Async API and XML API Only) Returns a string containing the export view data in XML format. This method retrieves the results of an asynchronous cooperative flow. The call blocks waiting for results if they are not available yet, depending on the block parameter. It returns the export results and throws GenExceptions for any errors.

More information:

[Configuring the .NET Proxy Runtime](#) (see page 65)

[Web Service Sample](#) (see page 69)

View Objects

One of the areas where the .NET Proxy has improved over the previous generation of proxy technology is in handling of the view data. Previous proxy technology used a flat view of data. This prevented easy manipulation by developers, as there were no row or column operations, no cloning, and no state reuse. Now, the view data is generated into Import and Export Objects. Some benefits of the functionality built into the view objects include:

- Comprehensive-View data and system data together.
- Hierarchical-More intuitive access paths to the data.
- Object-oriented-Allows more sophisticated manipulation of the data.
- Serializable-Allows object state to be saved and restored.
- No Runtime-Allows easier transmission/distribution/serialization.
- Data Validation-(Immediate) Keeps view always in a consistent state.
- Cloneable-Built-in support for cloning.
- Resettable-Multi-level reset ability gives greater control.
- GridView Support-Allows row-level operations or manipulations.

Import Object

Import objects are generated in classes called <method-name>Import based on the import view of the CA Gen server. Import objects contain the actual import data for the execution of a server using the .NET proxy. The Import objects are also responsible for all data validation that is done on the attributes of the export views.

Even if the server does not contain any import views, an Import object is still generated because the system level data (Command, NextLocation) is always present. The developer must instantiate an Import object and then populate it with data. The instance is then passed onto the execution methods as a parameter.

Import objects are stateful. In fact, they exist to hold a state. Therefore, the developer must take care not to use a particular instance of one of these classes between different threads in a multi-threaded application.

Export Object

Export objects are generated in classes called <method-name>Export based on the export view of the CA Gen server. Export objects contain the actual export data for the execution of a server through the .NET proxy. The Export objects are also responsible for all data validation that is done on the attributes on the export views.

Even if the server does not contain any export views, an Export object is still generated because the system level data (Command, ExitState) is always present. The Proxy object execution methods create and populate the Export objects.

The Export objects are stateful. In fact, they exist to hold a state. Therefore, the developer must take care not to use a particular instance of one of these classes between different threads in a multi-threaded application.

View Example

The easiest way to understand view objects is to look at an example. Later sections in this chapter describe how particular mappings occur.

The following example shows how generators map a given import view in the model to a view object.

```
MY_SERVER (procedure step)
IMPORTS:
    Entity View in proxy
        text1
        text2

    Group View inGV (2)
        Entity View ingroup proxy
            text1
            text2
```

From the given import view, five .NET classes are generated. The classes and their properties are shown in the following table:

Class	Attributes
MyServerImport	<ul style="list-style-type: none">■ Command■ NextLocation■ ClientId■ ClientPassword■ Dialect■ ExitState■ InEVProxy-Gets MyServer.InProxy object■ IngvGV-Gets MyServer.Ingv object
MyServer.InEVProxy	<ul style="list-style-type: none">■ Text1Fld■ Text2Fld
MyServer.IngvGV	<ul style="list-style-type: none">■ Capacity (Read-only constant)■ Length■ Rows-Gets a MyServer.IngvGVRow object■ this[]-Implicit indexer same as Rows

Class	Attributes
MyServer.IngvGVRow	IngroupEVProxy-Gets MyServer.IngroupEVProxy object
MyServer.IngroupEVProxy	<ul style="list-style-type: none"> ■ Text1Fld ■ Text2Fld

The number of classes increases in proportion with the number of entity views, work sets, and group views.

To finish the example, it is beneficial to look at how a programmer gets and sets the various pieces of data in the views. The following code fragment is written in C#, but the equivalent VB.NET code looks very similar.

Instantiate the Import View object:

```
MyServerImport importView = new MyServerImport();
```

Set the command system level data:

```
importView.Command = "SEND";
```

Access the InEVProxy text2 attribute:

```
String value = importView.InEVProxy.Text2Fld;
```

Get the maximum capacity of the IngvGV group view:

```
for (int i = 0; i < importView.IngvGV.Capacity; i++)
```

Set the current IngvGV number of rows:

```
importView.IngvGV.Length = 2;
```

Set the text1 attribute of the IngroupEVProxy entity view:

```
importView.Ingv[2].IngroupEVProxy.Text1Fld = "ABC";
```

Reset the InProxy Entity View back to defaults:

```
importView.InProxy.Reset();
```

System Level Properties

Import and export objects contain a set of properties at their top level that is best described as system-level properties. Not all the servers make use of these properties, but they are always present on each cooperative server call. The following table shows the properties of import and export objects:

Objects	Properties
Import Objects	<ul style="list-style-type: none">■ Command■ NextLocation■ ExitState■ Dialect■ ClientId■ ClientPassword
Export Objects	<ul style="list-style-type: none">■ Command■ ExitState■ ExitStateType■ ExitStateMessage

GroupView Objects

Special objects are generated for each group view. The first is the group view object itself. This object holds the Capacity or the maximum number of rows specified in the model. The object also holds the Length or the current number of populated rows in the group view. The Length cannot exceed the Capacity or be negative. If you try to do so, it throws `IndexOutOfRangeException` or `ArgumentOutOfRangeException`.

For import groups, the Capacity property is considered read only, while the Length property is read/write.

Note: It is the responsibility of the user written application to set the import repeating group Length property appropriately before flowing to a server. Failure to do so will result in the view data not being processed properly.

For repeating export groups, the user written application should consider both the Capacity and Length properties to be read only.

To store row data, an object is generated. The object represents a row and contains references to all the entity and work set views within the GroupView. This group view object then stores an array of row objects to hold the data. An individual row can be accessed by indexing the Rows property on the group view, or by using the indexer property built into the group view object itself.

Having a row of data represented as an object enables certain row-level operations to be performed, such as looping, cloning, resetting, and toString.

Data Validation

Import and export views validate attribute data when an attribute is set. The validation checks performed throw `ArgumentException`s if the new value is not valid. The views perform the following validation checks: permitted values, length, precision, and mandatory or optional.

Default Values

Attribute view data in the import and export objects enforce default values, as defined in the model. The default value is applied when the object is created or the attribute is reset programmatically.

Data Type Mappings

Since the import and export objects are designed to have no runtime dependencies, the attribute data types must be native data types supported by the .NET Framework. The following table shows the mappings applied:

CA Gen Attribute Definition	.NET Framework Data Type
Text/Mixed Text/DBCS Text	System.String
Number (no decimals, =< 4 digits)	short
Number (no decimals, =< 9 digits)	int
Number (no decimals, > 9 digits)	double
Number (with decimals)	double
Number (with precision turned on)	System.Decimal
Date	System.DateTime (null represents optional data)
Time	System.DateTime (null represents optional data)
Timestamp	System.DateTime (null represents optional data)
BLOB	byte[]

Using a .NET Proxy

A user-written application can incorporate the use of a .NET Proxy in two different ways:

- Install the proxy as a part of the consuming application.
- Install the proxy in a Global Assembly Cache (GAC).

A .NET proxy includes the .NET runtime dlls.

The generated .NET proxy is capable of supporting both synchronous and asynchronous cooperative flows. The main difference between these processing types is the actual method to perform the cooperative flow.

All sample code supplied with the product (APP, ASP.NET, and XML) demonstrates only the use of synchronous capabilities. However, the following sections provide, in pseudo-code, an outline of the structure of synchronous and asynchronous code.

Note: All non-CA Gen server errors are handled as standard .NET exceptions. Handling of these exceptions is not shown in the following sections.

More information:

[Asynchronous Processing](#) (see page 63)

[Preparing for Execution](#) (see page 65)

Synchronous Processing

The following contains the code for a synchronous process:

```
...
<method-name>Import importView = new <method-name>Import();
<method-name>Export exportView;
<method-name> proxy = new <method-name>();
...
<set up importView properties as desired>
...
exportView = proxy.Execute(importView); //Perform synchronous flow
...
<retrieve exportView data>
...
```

Note: If required, you can substitute the XMLExecute method for the synchronous Execute method.

Asynchronous Processing

The code is broken up into four distinct sections, as you can embed each piece at different locations within the user application.

Send Data

Use the following code to send data:

```
...
<method-name>Import importView = new <method-name>Import();
<method-name> proxy = new <method-name>();
...
<set up importView properties as desired>
...
int id = proxy.AsyncExecute(importView); //Perform asynchronous send
...
```

Note: If required, you can substitute the AsyncXMLExecute for the asynchronous AsyncExecute method.

Check Status

Use the following code to check status:

```
...
switch (proxy.AsyncCheckResponse(id)) //Perform check
{
...
    case FlowStatus.Ready:
        ...
    case FlowStatus.NotReady:
        ...
    default:
        ...
}
...
```

Retrieve Results

Use the following code to retrieve results:

```
...
exportView = proxy.AsyncGetResponse(id) //Perform the retrieval of the results
...
<retrieve export view data>
...
```

Note: If required, you can substitute the `AsyncXMLGetResponse`, for the asynchronous `AsyncGetResponse` method.

Ignore Request

Use the following code to ignore a request:

```
...
proxy.AsyncIgnoreResponse(id)
//Processes the ignore request on the request indicated by the id
...
```

Security Processing

A .NET Proxy provides facilities to implement Distributed Processing Security.

Note: For more information about implementation of Distributed Processing Security, see the *Distributed Processing - Overview Guide*.

By default, the .NET Proxy does not exploit the use of the Proxy Runtime security features. To utilize the security features, the application developer must add code to the .NET application to set the `ClientID` and `ClientPassword` properties of the .NET Proxy Import view object. Depending on the return value of the client security user exit, the security data fields are sent to the target DPS. In addition, if the client security user exit returns `SECURITY_ENHANCED`, the client security user exits can add an optional security token to the data flow. User exits residing in the execution environment of the target DPS validate the collection of security data that is sent as part of the cooperative flow.

Note: For a description of the user exits available within each runtime environment supported by a .NET Proxy, see the *User Exit Reference Guide*.

For a .NET Proxy flowing to a DPS using TCP/IP or MQSeries the supporting runtime lets a portion of the Common Format Buffer (CFB) to be encrypted on the way to the target DPS, and decrypted on the way back from the target DPS. User exits enable the use of encryption and decryption.

More information:

[User Exits](#) (see page 66)

Preparing for Execution

Before you execute an application that uses a generated .NET proxy, address the following items:

- You may need to modify the .NET Proxy Runtime for use in the user application execution environment.
- You must deploy the proxy and Runtime to the applications execution environment.

Configuring the .NET Proxy Runtime

By design, there exist areas within the .NET Proxy Runtime that can be influenced by an application developer. An application developer does many activities that include updating the commcfg.txt file to influence the communications part of the runtime and modifying user exits. The user exits influence security processing and lets the arguments specific to the selected communications processing to be overridden.

Configuring .NET Proxy Communications

The .NET Proxy runtime needs to know where to find the CA Gen servers it needs to access. To specify their locations, use the commcfg.txt file found in the installed CA Gen\ .NET directory.

The file must be placed in a directory where the CA Gen runtime can probe for it. CA Gen probes for files by checking the following directories until the file is located:

- Application Base Directory
- Code base directory of the loaded CA.Gen.odc.dll assembly
- Application private path directory
- All subdirectories under the application base directory

A generated proxy contains communications information as defined in the model. The proxy runtime provides a capability for specifying the communication information at runtime. This information overrides any information specified in the model and built into the proxy during generation.

The proxy runtime is also capable of turning tracing ON and OFF, and provides a capability to control the amount of file caching that takes place at runtime.

Note: Due to the load techniques used by the .NET Framework, multiple copies of a commcfg.txt file might exist in numerous application directories. After creating a new commcfg.txt file, you should manually copy it to all application directories where the new settings are required. Different applications can use different commcfg.txt files, if required.

Note: For specific formatting instructions, see the default file installed with the .NET Proxy software, or see the *Distributed Processing - Overview Guide*.

User Exits

The user exits that are invoked at runtime for a .NET Proxy are common to other C# execution environments.

Note: For more information about each user exit, see the *User Exit Reference Guide*.

The user exits invoked during runtime depend on the selected communication type. The communication type implies the use of a particular process for formatting the import and export message data that is exchanged with the target DPS application.

A .NET Proxy flowing to a DPS using TCP/IP invokes the following user exit entry points:

User Exit Entry Point	Purpose
CFBDynamicMessageSecurityExit	The Client Security user exit class is used to direct the runtime to incorporate the ClientID and ClientPassword Import Object attributes into the CFB. If this exit returns SECURITY_ENHANCED, it can cause an optional Security Token field to be added to the CFB.
CFBDynamicMessageDecryptionExit	The export message encryption user exit class provides the opportunity to decrypt the portion of the CFB that has been encrypted by the target DPS execution environment. This exit must implement the companion to the DPS encryption user exit for the DPS response buffer to be interpreted as a valid response CFB.
CFBDynamicMessageEncryptionExit	The import message encryption user exit class provides an opportunity to encryption algorithm implemented within this user exit to encrypt a portion of the outbound CFB. The target DPS execution environment must implement a companion decryption user exit for the CFB to be interpreted as a valid request CFB.

In addition to the above user exits, a .NET Proxy flowing to a DPS using TCP/IP invokes the following user exit entry point:

User Exit Entry Point	Purpose
TCPIPDynamicCoopFlowExit	This user exit lets certain TCP/IP communication parameters to be overridden at runtime.

A .NET Proxy flowing to a .NET Serviced Component DPS using .NET Remoting invokes the following user exit entry points:

User Exit Entry Point	Purpose
NETDynamicCoopFlowExit	This user exit class lets certain .NET Remoting communication parameters to be overridden at runtime.
NETDynamicCoopFlowSecurityExit	This user exit class lets you specify a user-defined security object to be passed to the server.

In addition to these user exits, a .NET Proxy flowing to a DPS using MQSeries invokes the following user exit entry point:

User Exit Entry Point	Purpose
MQSDynamicCoopFlowExit	This user exit lets certain MQSeries communication parameters to be overridden at runtime.
WSDynamicCoopFlowExit	<p>This class has methods that let you manipulate the target Web Service endpoint URL by modifying the following properties:</p> <p>BaseURL</p> <p>ContextType</p> <p>This class has a method for handling exceptions that occur when performing a Web Service operation. This method lets the failed request to be retried.</p>

Deploying a .NET Proxy

Each \deploy\<component> directory contains a <component>.dll. This DLL is the .NET Proxy. You can deploy the proxy DLL and its runtime in two different ways:

- Install the proxy as a part of the consuming application.
- Install the proxy in a Global Assembly Cache (GAC).

Deploying as Part of an Application

The preferred technique is to deploy the proxy and its runtimes as simply another portion of the consuming application. The Microsoft design for XCOPY installations implies that the proxy DLL and the runtime DLLs must be placed somewhere beneath the APPLICATION BASE directory.

By default, an application can only find DLLs specifically residing in the application base directory. You can add subdirectories to the privatePath settings of the application in the application config file. The APP and XML samples provided by CA Gen put all the DLLs in a subdirectory called bin. privatePath should be set to search the bin subdirectory. Applications installed under IIS as part of ASP.NET automatically search a directory called bin, so the ASP.NET sample also follows this convention.

Deploying Through the Global Assembly Cache

Microsoft provides a mechanism to share assembly DLLs between applications on a machine level. This mechanism is called the Global Assembly Cache (GAC). Using the GAC prevents the easy XCOPY deployment encouraged by Microsoft, but it lets you update and distribute to all consuming applications immediately. Depending on the application architecture, use of the GAC may be considered a benefit or a disadvantage. For more information about how to install and manage assemblies in the GAC, see the Microsoft Documentation.

.NET Proxy Runtime Files

The .NET Proxy requires a set of runtime dll assemblies to be present to execute properly. You can find these runtime dll assemblies in the CA Gen*.NET\bin directory. These assemblies follow:

- CA.Gen.csu.dll
- CA.Gen.localizationrt.dll
- CA.Gen.exits.dll
- CA.Gen.vwrt.dll
- CA.Gen.odc.dll
- commcfg.txt

Note: In addition to the above set of runtime files, the CA Gen runtime files that support a specific communication type will also be required. For example:

CA.Gen.odc.tcpip.dll provides support for the use of TCP/IP from a .NET proxy to its associated target server's execution environment.

CA.Gen.odc.ws.dll provides support for the use of Web Services from a .NET proxy to its associated target EJB Web Services server's execution environment.

Executing the Sample Applications

The following sections describe how you can execute sample applications for ASP.NET, stand-alone .NET, and XML.

ASP.NET Sample

The generation and installation process creates everything you need to invoke the ASP.NET Proxy sample using .NET from IIS. The \samples\ASP.NET directory contains <method-name>.aspx files. The .aspx files may be invoked directly from the Web Browser. For convenience, there are additional files generated (operations.htm, default.htm, and blank.htm).

To view the .aspx files, you must create an alias to the \samples\ASP.NET directory using the IIS Internet Service Manager or the Web Sharing tab on the directory properties. You must enable this directory for Read and Execute access.

After creating the alias to the directory, start IIS and enter the following URL (assumes, the alias created is called myalias):

`http://localhost/myalias`

The default.htm file is loaded automatically. Alternatively, the following URLs are also possible:

`http://localhost/myalias/default.htm`

`http://localhost/myalias/operations.htm`

`http://localhost/myalias/<method-name>.aspx`

When the \samples\ASP.NET directory is created, all of the assemblies and resource files required to execute the sample ASP.NET application are copied into the bin subdirectory. The proxy assembly DLL, the runtime assembly DLLs, and the commcfg.txt file all are copied there. If any one of them changes, you must copy them manually to the bin directory or have the Build Tool rebuild the proxy.

Once the ASP.NET page is displayed, fill in all the import fields appropriately and click Execute. The proxy is executed and any errors or the export view is displayed.

Web Service Sample

The generation and installation process creates everything you need to deploy the .NET proxy as a web service. However, no sample client application is created. Third-party products are available that can be used to create client applications based on the web service WSDL.

Deploying the .NET proxy as a web service requires using Microsoft's IIS to host the proxy. In the same location where the sample .asmx files are generated for the ASP.NET sample are some generated .asmx files. These .asmx files instruct IIS to treat any method within the referenced proxy class marked as a [WebMethod] as a published web service. IIS will then automatically generate the WSDL for the web service.

Follow the instructions under the ASP.NET Sample section to deploy the ASP.NET sample and proxy. Ensure that the .asmx files are also deployed into the virtual directory.

Note: The .aspx and .htm files are optional for deploying the proxy as a web service.

After creating the alias to the directory, start IIS and enter the following URL (assumes the alias created is called myalias):

`http://localhost/myalias/<method-name>.asmx`

IIS will display an information page about the web service. This page can be used to retrieve the WSDL and browse the available web methods. The information typically supports providing a test page to exercise the web service. Unfortunately, IIS can only provide the test page for web services that use primitive data types for parameters. The .NET proxy's import and export data parameters are too complex for IIS test pages to handle.

There are third-party products that are able to provide automatic generation of test harnesses/applications based on a WSDL to verify the web services. It is suggested that some third-party product be utilized to help verify the web services and also generate the client side classes needed to call a web service from an application.

Writing client code to call the web service usually involves using a tool to generate the some client-side proxy code. Microsoft's Visual Studio is capable of consuming a web service and generating the client-side proxy code. There are also other products capable of consuming a web service. Using one of these products should probably be considered a starting point to creating a client application to call the .NET proxy as a web service.

Stand-alone .NET Application

The generation and installation process creates everything you need to invoke the stand-alone Proxy sample application. The \samples\APP directory contains Test<method-name>.exe assemblies.

When the \samples\APP directory is created, all the assemblies and resources files needed to execute the sample application are copied into the bin subdirectory. This means that the proxy assembly DLL, the runtime assembly DLLs, and the commcfg.txt file are copied there. If any one of them changes, you must copy them manually to the bin directory or have the Build Tool rebuild the proxy.

To execute the sample application, launch the executable from Windows Explorer or execute the following from a command line:

`Test<method-name>.exe`

No parameters are required. Once the application starts, enter the import view information as needed. Click Execute to invoke the server. The right side then displays the export view results. Errors are shown in the Results tab at the bottom. If required, you can use the tracing tab to start and stop tracing.

XML Test Application (Optional)

If you select the optional XML programming interface, the generation and installation process creates everything you need to invoke a .NET Proxy sample XML test application. The \samples\XML directory contains a Test<method-name>.exe assembly along with a sample import XML file and the generated XML schema (XSD) file.

When the \samples\XML directory is created, all the assemblies and resources files needed to execute the sample application are copied into the bin subdirectory. This means that the proxy assembly DLL, the runtime assembly DLLs, and the commcfg.txt file all are copied there. If any one of them changes, you must copy them manually to the bin directory or have the Build Tool rebuild the proxy.

To execute the XML sample application, execute the following command from a command line:

```
Test<method-name>.exe <method-name>Sample.xml
```

The parameter is required and must be the input XML file. A sample import XML file is generated that contains the syntax structure, and where possible, default values for the import view. If required, you can modify the sample import XML file for the server. The resulting errors or export view data are the standard outputs of the command.

Note: BLOB data in XML is represented by base 64 format.

Chapter 4: Java Proxy

The Java Proxy is a generated Java object that provides an object-oriented interface to CA Gen servers. The Java Proxy and associated CA Gen runtimes are 100 percent Java code. Providing a Java object interface allows any Java application access CA Gen servers. Typical user applications are stand-alone programs, JSP web applications, Servlet web applications, and EJB components. The proxy can be consumed in virtually any component of a J2SE or J2EE application.

Java Proxy Generated Code

The generated Java proxy is a set of object-oriented classes that correspond to the set of methods selected during generation. There are three main classes that a developer can use and access. Other classes are generated, but they are not exposed for use by the application. The three exposed classes are:

- <method-name> Referred to as the Java Proxy object.
- <method-name>Import Referred to as the Import object.
- <method-name>Export Referred to as the Export object.

The details of the three generated interfaces are discussed later in this chapter.

In addition to the generated interface, the Java Proxy also consists of runtime code and a set of user modifiable exit routines. The user exits influence how the runtime handles security and communications processing. This chapter discusses the Java Proxy user exits.

The creation of an executable Java Proxy involves both generating the Java Proxy source code and installing the proxy. You can choose to do these two operations as two independent activities or as a single operation (check Install after generation option on the Proxy Code Generation dialog).

Generating a Java Proxy results in a \proxy\java directory being added within your <model-name>.ief directory. All proxy code is contained in a series of directories beneath \proxy\java.

Note: The Java Proxy and the Classic Style Java Proxy both generate into the \proxy\java directories and overwrite each other if the names are the same.

After the Build Tool has successfully built the proxy, the \proxy\java directory includes a \deploy\<component> directory, which includes a \samples directory. Within the \samples directory are directories for \APP, \JSP, and \XML. Select the optional XML API only if you want to generate the XML sample. The \JSP, \APP, and \XML directories contain generated samples to test the proxies and demonstrate possible usages of the proxy. The samples are not considered part of the proxy itself.

If you select XML API for generation, then the \deploy\<component> directory also includes the XML schema file (.XSD). The XML schema file is considered part of the proxy itself.

More information:

[Generating the Proxy](#) (see page 31)

Java Proxy Interface

CA Gen generates an object-oriented set of classes for each method selected for generation.

There are many classes generated, but a developer needs to be concerned with three main classes. Other classes are used internally by the proxy and are therefore not discussed here. The three classes are:

- <method-name> Referred to as the Java Proxy object.
- <method-name>Import Referred to as the Import object.
- <method-name>Export Referred to as the Export object.

Each class exists within a package. That package is defined by the generator as either the short-model name (default) or a particular value specified at the business system or model level for the Java Package.

The following sections discuss the three classes in more detail.

More information:

[Java Proxy \(Classic Style\)](#) (see page 97)

Java Proxy Object

The Java Proxy object is the generated class called <method-name>. It contains the actual execution methods for the proxy. The execution methods are designed to be stateless. All data is stored on the stack (passed as parameters) during execution. One instance of the class can be used simultaneously by many threads.

The Java proxy object contains following two types of methods:

- Trace control methods
- Execution methods

Trace Control Methods

There are five trace control methods. They are always generated with every Java proxy object. These methods are used to start, stop and write trace messages using the CA Gen trace capabilities.

Note: When used in this context, Trace refers to runtime logging capability and you should not confuse this Trace with Diagram Trace used to debug generated action block logic.

The following table lists the five trace control methods:

Trace Control Method	Comments
StartTracing()	Turns on tracing with the output going to the file, trace-{program}-{pid}.out, in the "%USERPROFILE%\AppData\Local\CA\Gen xx\logs\{app style directory}". Note: xx refers to the current release of CA Gen. For the current release number, see the <i>Release Notes</i> .
StartTracing(String filename)	Turns on tracing with the output going to a file specified by the filename parameter.
StartTracing(Writer writer)	Turns on tracing with the output going to the TextWriter parameter passed in.
TraceOut(string message)	Writes the message to the trace output if tracing is enabled at that time. If tracing is not enabled, the method simply returns.
StopTracing()	Turns off all tracing.

Tracing in the CA Gen runtime is global in scope. If one proxy turns it on, it is enabled for all the proxies running in the same process.

Execution Methods

There are a variety of execution methods. Some are always generated with every Java proxy object, but others are only generated if one or both of the optional APIs are selected during generation. The following table lists the execution methods, when they are generated, and a summary of what each one does:

Execution Method	Comments
Execute(importView)	Returns an exportView. This method performs a synchronous cooperative flow with the import data provided. It returns the export results and throws CSUExceptions for any errors.
Execute(importView, String comCfg)	Returns an exportView. This method performs a synchronous cooperative flow configured by the comCfg parameter, with the import data provided. It returns the export results and throws CSUExceptions for any errors.
XMLExecute(String importView)	(XML API only) Returns a string containing the export view data in XML format. This method performs a synchronous cooperative flow with the import data passed in XML format. It returns the export results and throws CSUExceptions for any errors.
XMLExecute(String importView, String comCfg)	(XML API only) Returns a string containing the export view data in XML format. This method performs a synchronous cooperative flow configured by the comCfg parameter, with the import data passed in XML format. It returns the export results and throws CSUExceptions for any errors.
AsyncExecute(importView)	(Async API only) Returns an int containing the id used to retrieve the asynchronous results later. This method begins an asynchronous cooperative flow with the import data provided. It throws CSUExceptions for any errors.

Execution Method	Comments
AsyncExecute(importView, boolean noResponse)	(Async API only) Returns an int containing the id used to retrieve the asynchronous results later. This method begins an asynchronous cooperative flow with the import data provided. The noResponse parameter indicates to the runtime whether any results are expected back. It throws CSUExceptions for any errors.
AsyncExecute(importView, boolean noResponse, String comCfg)	(Async API only) Returns an int containing the id used to retrieve the asynchronous results later. This method begins an asynchronous cooperative flow with the import data provided. The noResponse parameter indicates to the runtime whether any results are expected back. It throws CSUExceptions for any errors.
AsyncGetReponse(int id)	(Async API only) Returns an exportView. This method retrieves the results of an asynchronous cooperative flow specified by the id provided. It returns the export results and throws CSUExceptions for any errors.
AsyncGetReponse(int id, boolean block)	(Async API only) Returns an exportView. This method retrieves the results of an asynchronous cooperative flow specified by the id provided. The call will block waiting for results if they are not available yet, depending on the block parameter. It returns the export results and throws CSUExceptions for any errors.
AsyncCheckReponse(int id)	(Async API only) Returns a FlowStatus enumeration. This method retrieves the status of an asynchronous cooperative flow specified by the id provided. It throws CSUExceptions for any errors.

Execution Method	Comments
AsyncIgnoreReponse(int id)	(Async API only) This method flags the results of an asynchronous cooperative flow specified by the id provided as ignorable. Once ignored, the results become unretrievable. It throws CSUExceptions for any errors (Flagging a flow as ignored lets the runtime to clean up resources devoted to that flow).
AsyncXMLExecute(String importView)	(Async API and XML API only) Returns an int containing the id used to retrieve the asynchronous results later. This method begins an asynchronous cooperative flow with the import data passed in XML format. It throws CSUExceptions for any errors.
AsyncXMLExecute(String importView, boolean noResponse)	(Async API and XML API only) Returns an int containing the id used to retrieve the asynchronous results later. This method begins an asynchronous cooperative with the import data passed in XML format. The noResponse parameter indicates to the runtime whether any results are expected back. It throws CSUExceptions for any errors.
AsyncXMLExecute(String importView, boolean noResponse, String comCfg)	(Async API and XML API Only) Returns an int containing the id used to retrieve the asynchronous results later. This method begins an asynchronous cooperative flow configured by the comCfg parameter, with the import data passed in XML format. The noResponse parameter indicates to the runtime whether any results are expected back. It throws CSUExceptions for any errors.

Execution Method	Comments
AsyncXMLGetReponse(int id, boolean block)	(Async API and XML API Only) Returns a string containing the export view data in XML format. This method retrieves the results of an asynchronous cooperative flow specified by the id provided. The call blocks waiting for results if they are not available yet, depending on the block parameter. It returns the export results and throws CSUExceptions for any errors.
AsyncXMLGetReponse(int id)	(Async API and XML API Only) Returns a string containing the export view data in XML format. This method retrieves the results of an asynchronous cooperative flow specified by the id provided. It returns the export results and throws CSUExceptions for any errors.

View Objects

One of the areas where Java Proxy has improved over the previous generation of proxy technology is in the handling of view data. Previous proxy technology used a flat view of data. This prevented easy manipulation by developers, as there were no row or column operations, no cloning, and no state reuse. Now, the view data is generated into Import and Export Objects. Some of the benefits of the functionality built into the view objects are:

- Comprehensive-View data and system data together
- Hierarchical-More intuitive access paths to the data
- Object-oriented-Allows more sophisticated manipulation of the data.
- Serializable-Allows object state to be saved and restored.
- No Runtime-Allows for easier transmission/distribution/serialization
- Data Validation-(Immediate) Keeps view always in a consistent state.
- Cloneable-Built-in support for cloning.
- Resettable-Multi-level reset ability gives greater control.
- GroupView Support-Allows for row-level operations or manipulations.

Import Object

The Import objects are generated in classes called <method-name>Import based on the import view for the CA Gen server. The object contains the actual import data for the execution of a server through the Java proxy. The Import objects are also responsible for all data validation that is done on the attributes of the import views.

Even if the server does not contain any import views, an Import object is still generated because the system level data (Command, NextLocation) is always present. The developer must instantiate an Import object and then populate it with the data. The instance is then passed onto the execution methods as a parameter.

The Import objects are not stateless. They exist to hold a state. Therefore, the developer must take care not to use a particular instance of one of these classes between different threads in a multi-threaded application or use in concurrent asynchronous flows.

Export Object

The Export objects are generated in classes called <method-name>Export based on the export view designed in the model for the CA Gen server. The object contains the actual export data for the execution of a server through the Java proxy. The Export objects are also responsible for all data validation that is done on the attributes on the export views.

Even if the server does not contain any export views, the Export object is still generated because the system level data (Command, ExitState) is always present. The Proxy object execution methods typically create and populate the Export objects.

The Export objects are not stateless. They exist to hold a state. Therefore, the developer must take care not to use a particular instance of one of these classes between different threads in a multi-threaded application or use in concurrent asynchronous flows.

View Example

The easiest way to understand view objects is to look at an example. The later sections in this chapter discuss specific details of particular mappings.

The following example shows how generators map a given import view in the model to a view object.

The import view is shown as follows:

```

MY_SERVER (procedure step)
IMPORTS:
    Entity View in proxy
        text1
        text2

    Group View inGV (2)
        Entity View ingroup proxy
            text1

            text2

```

From the given import view, five Java classes are generated. The classes and their properties are:

Class	Attributes
MyServerImport	<ul style="list-style-type: none"> ■ Command ■ NextLocation ■ ClientId ■ ClientPassword ■ Dialect ■ ExitState ■ InEVProxy-Gets MyServer.InProxy object ■ IngvGV-Gets MyServer.Ingv object
MyServer.InEVProxy	<ul style="list-style-type: none"> ■ Text1Fld ■ Text2Fld
MyServer.IngvGV	<ul style="list-style-type: none"> ■ Capacity (Read-only constant) ■ Length ■ Rows-Gets a MyServer.IngvGVRow object
MyServer.IngvGVRow	<ul style="list-style-type: none"> ■ IngroupEVProxy-Gets MyServer.IngroupEVProxy object
MyServer.IngroupEVProxy	<ul style="list-style-type: none"> ■ Text1Fld ■ Text2Fld

The number of classes increases in proportion with the number of entity views, work sets, and group views.

To finish the sample, it is beneficial to look at how a programmer gets and sets the various pieces of data in the views. The following code snippets are written in Java.

Instantiate the Import View object:

```
MyServerImport importView = new MyServerImport();
```

Set the command system level data:

```
importView.setCommand("SEND");
```

Access the InEVProxy text2Fld attribute:

```
string value = importView.getInEVProxy().getText2Fld();
```

Get the maximum capacity of the IngvGV group view:

```
for (int i = 0; i < importView.getIngvGV().Capacity; i++)
```

Set the current IngvGV number of rows:

```
importView.getIngvGV().setLength(2);
```

Set the text1 attribute of the IngroupEVProxy entity view:

```
importView.getIngvGV().getRows()[2].getIngroupEVProxy().setText1Fld("ABC");
```

Reset the InEVProxy Entity View back to defaults

```
importView.getInEVProxy().Reset();
```

System Level Properties

The import and export objects contain a set of properties at their top level that are best described as system-level properties. Not all servers make use of these properties, but they are always present on each cooperative server call. The following table shows the properties of import and export objects:

Objects	Properties
Import Objects	<ul style="list-style-type: none">■ Command■ NextLocation■ ExitState■ Dialect■ ClientId■ ClientPassword

Objects	Properties
Export Objects	<ul style="list-style-type: none"> ■ Command ■ ExitState ■ ExitStateType ■ ExitStateMessage

GroupView Objects

Special objects are generated for each group view. The first is the group view object itself. This object holds the Capacity or the maximum number of rows specified in the model. It also holds the Length or the current number of populated rows in the group view. The Length cannot exceed the Capacity or be negative. It is impossible to access a row beyond the currently defined Length. If you try to do so, it throws `IndexOutOfRangeException` or `ArgumentOutOfRangeException`.

The Capacity value is accessible via the objects Capacity property. The Length property is set by calling the objects `setLength()` method and read by calling the `getLength()` method.

For import groups, the Capacity property is considered read only, while the Length property is read/write.

Note: It is the responsibility of the user written application to set the import repeating group Length property appropriately prior to flowing to a server. Failure to do so will result in the view data not being processed properly.

For repeating export groups, the user written application should consider both the Capacity and Length properties to be read only.

To store row data, an object is generated. It represents a row and contains references to all the entity and work set views within the GroupView. This group view object stores an array of row objects to hold the data. An individual row can be accessed by indexer property built into the Rows property on the group view.

Since a row of data is an object that lets certain row-level operations to be performed such as looping, cloning, resetting, and toStringing.

Data Validation

The import and export views validate the attribute data when it is set. The validation checks performed throws `IllegalArgumentException`s if the new value is not valid. The views perform the following validation checks-permitted values, length/precision, and mandatory/optional.

Default Values

Attribute view data in the import and export objects enforce the default values, as defined in the model. The default value is applied when the object is created or the attribute is reset programmatically.

Data Type Mappings

Since the import and export objects are designed to have no runtime dependencies, the attribute data types must be the native Data Types supported by Java. The following table shows the mappings applied:

CA Gen Attribute Definition	Java Data Type
Text/Mixed Text/DBCS Text	java.lang.String
Number (no decimals, =< 4 digits)	short
Number (no decimals, =< 9 digits)	int
Number (no decimals, > 9 digits)	double
Number (with decimals)	double
Number (with precision turned on)	java.math.BigDecimal
Date	java.sql.Date (null represents optional data)
Timestamp	java.sql.Timestamp (null represents optional data)
Time	java.sql.Time (null represents optional data)
BLOB	byte[]

Using a Java Proxy

The generated Java proxy is capable of supporting both synchronous and asynchronous cooperative flows. The main difference between these processing types is the actual method used to perform the cooperative flow.

All sample code supplied with the product (APP, JSP and XML) demonstrates only the use of synchronous capabilities. However, the following sections provide, in pseudo-code, an outline of the structure of synchronous and asynchronous code.

Note: All non-CA Gen server errors are handled as standard Java exceptions. Handling of these exceptions is not shown in the following sections.

More information:

[Asynchronous Processing](#) (see page 85)

Synchronous Processing

The following contains the code for a synchronous process:

```
...
<method-name>Import importView = new <method-name>Import();
<method-name>Export exportView;
<method-name> proxy = new <method-name>();
...
<set up importView properties as desired>
...
exportView = proxy.execute(importView); //Perform synchronous flow
...
<retrieve exportView data>
...
```

Note: If required, you can substitute the `xmlExecute` method for the synchronous `execute` method.

Asynchronous Processing

The code is broken up into four distinct sections, as you can embed each piece at different locations within the user application.

Send Data

Use the following code to send data:

```
...
<method-name>Import importView = new <method-name>Import();
<method-name> proxy = new <method-name>();
...
<set up importView properties as desired>
...
int id = proxy.asyncExecute(importView); //Perform asynchronous send
...
```

Note: If required, you can substitute the `asyncXMLExecute` method for the asynchronous `asyncExecute` method.

Check Status

Use the following code to check status:

```
...
switch (proxy.asyncCheckResponse(id)) //Perform check
{
...
    case CoopFlow.DATA_READY:
        ...
    case CoopFlow.DATA_NOT_READY:
        ...
    default:
        ...
}
...
```

Retrieve Results

Use the following code to retrieve results:

```
...
exportView = proxy.asyncGetResponse(id) //Perform the retrieval of the results
...
<retrieve export view data>
...
```

Note: If required, you can substitute the `asyncXMLGetResponse` for the asynchronous `asyncGetResponse` method.

Ignore Request

Use the following code to ignore a request:

```
...
proxy.asyncIgnoreResponse(id)
//Processes the ignore request on the request indicated by the id
...
```

Security Processing

A Java Proxy provides facilities to implement Distributed Processing Security.

Note: For more information about implementation of Distributed Processing Security, see the *User Exit Reference Guide*.

By default, the Java Proxy does not exploit the use of the Proxy Runtime security features. To utilize the security features, the application developer must add code to their Java application that sets the ClientID and ClientPassword properties of the Java Proxy's Import view object. Depending on the return value of the client security user exit, the security data fields are sent to the target Distributed Processing Server (DPS). Additionally, if the client security user exit returns SECURITY_ENHANCED, the client security user exits can add an optional security token to the data flow. The user exits residing in the execution environment of the target DPS validate the collection of security data that is sent as part of the cooperative flow.

Note: For a Java Proxy flowing to a DPS using TCP/IP, MQSeries, or ECI, the supporting runtime lets a portion of the Common Format Buffer (CFB) to be encrypted on the way to the target DPS and decrypted on the way back from the target DPS. User exits enable the use of encryption and decryption.

More information:

[User Exits](#) (see page 88)

Preparing for Execution

Before you execute an application that uses a generated Java proxy, address the following items:

- You may need to modify the Java Proxy Runtime for use in the user application execution environment.
- You must deploy the proxy and Runtime to the applications execution environment.

Configuring the Java Proxy Runtime

By design, there exist areas within the Java Proxy Runtime that can be influenced by an application developer. An application developer does many activities that include updating the commcfg.properties file to influence the communications part of the runtime and modifying user exits. The user exits influence the security processing and allow the arguments specific to the selected communications processing be overridden.

Configuring the Java Proxy Communications

The Java Proxy runtime needs to know where to find the CA Gen servers it needs to access. To specify their locations, use the commcfg.properties file found in the installed CA Gen directory.

A generated proxy contains communications information as defined in the model. The proxy runtime provides capability to specify the communication information at runtime. This information overrides any information specified in the model and was built into the proxy during generation.

Additionally, the proxy runtime is capable of turning the tracing ON and OFF, as well as provides capability to control the amount of file caching that takes place at runtime.

Note: For specific formatting instructions, see the default file installed with the Java Proxy software, or see the *Distributed Processing - Overview Guide*.

User Exits

The user exits that are invoked at runtime for a Java Proxy are common to other Java execution environments.

Note: For more information about each user exit, see *User Exit Reference Guide*.

CA Gen user can modify the behavior of the runtime environment by the way of User Exits. Several user exits are useful during the deployment of the CA Gen Java proxies. The Java user exits apply to either the normal or classic style Java Proxies. The exits that apply to proxies can be grouped into two categories: security and communication.

After modifying Java user exits, the Java class must be recompiled. Then the class must be deployed to replace the old version in the CLASSPATH. If a runtime JAR was built using mkjavart.bat, then it must be rebuilt.

Note: CA Gen runtimes are designed to run in a multi-threaded environment so custom algorithms are thread safe.

The following user exits are bundled together as Java classes. The user exits are installed as part of the Java Proxy runtime.

More information:

[Deploying a Java Proxy](#) (see page 92)

Security User Exits

The user exit source code is located in the following directories under the CA Gen installation directory:

- classes/com/ca/genxx/exits/msgobj/cfb for Common Format Buffer (CFB)
- classes/com/ca/genxx/exits/coopflow/tuxedo for Tuxedo
- classes/com/ca/genxx/exits/coopflow/ejbrmi for EJBRMI

Note: *xx* refers to the current release of CA Gen. For the current release number, see the *Release Notes*.

Flow type: CFB (Common Format Buffer)

User Exit Class	Purpose
CFBDynamicMessageSecurityExit	The Client Security user exit class is used to direct the runtime to incorporate the ClientID and ClientPassword Import Object attributes into the CFB. If this exit returns SECURITY_ENHANCED, it can cause an optional Security Token field to be added to the CFB.
CFBDynamicMessageEncryptionExit	The import message encryption user exit class provides the opportunity for a portion of the outbound CFB to be encrypted using an encryption algorithm implemented within this user exit. The target DPS execution environment must implement a companion decryption user exit for the CFB to be interpreted as a valid request CFB.
CFBDynamicMessageEncodingExit	This user exit returns the message text encoding for the given named transaction.
CFBDynamicMessageDecryptionExit	The export message decryption user exit class provides the opportunity for decrypting the portion of the CFB that has been encrypted by the target DPS execution environment. This exit must implement the companion to the DPS's encryption user exit in order for the DPS's response buffer to be interpreted as a valid response CFB.

Flow type: Tuxedo

User Exit Class	Purpose
TUXDynamicSecurityExit	The Client Security user exit is used to provide security data to the runtime for use when processing a flow request. The security data is used while creating a Oracle Jolt session.

Flow type: EJBRMI

User Exit Class	Purpose
EJBRMISecurityExit	This user exit allocates a security object using the user runtime object. This security object contains all of the security information that is passed from the client to the server.

Communications User Exits

The following user exit classes are invoked by Java Proxies flowing to their target DPSs using a supported communications type. The user exit source code is located in a directory under `classes/com/ca/genxx/exits/coopflow` in the CA Gen installation directory. These classes are installed as part of the specific communications runtime (for example: `classes/com/ca/genxx/exits/coopflow/tcpip`).

Note: *xx* refers to the current release of CA Gen. For the current release number, see the *Release Notes*.

User Exit Class	Purpose
TCPIPDynamicCoopFlowExit	The methods in this class let you override the host name and the port number. Additionally, this class contains a method that exposes that a TCP/IP processing exception has occurred. This user exit method indicates if the processing of the flow should be retried. For example, retry in the event of a Connection Reset by Peer error.
MQSDynamicCoopFlowExit	The methods in this class let you override the following MQSeries processing data items: Local Queue Manager name Remote Queue Manager name Put Queue name Reply to Queue name Local Queue Manager name Remote Queue Manager name Put Queue name Reply to Queue name

User Exit Class	Purpose
ECIDynamicCoopFlowExit	<p>The methods in this class let you override the following CICS Transaction Gateway processing data items:</p> <p>The ECI client security class name</p> <p>The port address of the CICS Transaction Gateway</p> <p>The protocol to be used to communicate with the CICS Transaction Gateway (example: local, TCP, auto, http, https, and SSL)</p> <p>The ECI server security class name</p> <p>The CICS system name, as known within the Transaction Gateway (or supporting Universal Client), to which the cooperative request should be sent.</p>
EJBRMIDynamicCoopFlowExit	<p>The methods in this class lets you to override the following EJB RMI processing data items:</p> <p>The provider URL</p> <p>Additionally, this class has a method that permits an optional User object to be sent as part of the EJB RMI flow to the EJB server.</p> <p>This class has a method for handling exceptions that occur when performing an EJB RMI operation (open, read, write, and so on). This method lets the failed request to be retried.</p>
TUXDynamicCoopFlowExit	<p>This class contains two methods that expose the data flowing to and from the server. The OutData method provides the ability to examine or modify the request data being sent to the server. The InData method provides the ability to examine or modify the response data received from the server.</p>
WSDynamicCoopFlowExit	<p>This class has methods that let you manipulate the target Web Service endpoint URL by modifying the following members:</p> <p>baseURL</p> <p>contextType</p> <p>This class has a method for handling exceptions that occur when performing a Web Service operation. This method lets the failed request to be retried.</p>

Note: If you specified the server address by name, try entering the decimal IPv4 address instead (for example, 1.1.1.1) or hexadecimal IPv6 address (for example FE80:0:0:1::1). Your machine may not be configured properly to work with your local name server

Conversion User Exit

This user exit class is used to modify data saved to and retrieved from a database. It currently handles Strings only and is sometimes required for languages where strings stored in a database need to be modified for display.

Note: For more information about this user exit, see DataConversionExit-Java Data Conversion Exit in the *User Exit Reference Guide*.

Deploying a Java Proxy

Each \deploy\<component> directory contains a <component>.jar. This JAR file is the Java Proxy. You can deploy the proxy JAR file (and its runtime) for use by an application in various ways. In general, all the techniques ensure that the generated classes and the runtime classes are loadable from within the target application. Depending on the target application, the class files can be handled individually as part of JAR files, an executable JAR file, and a larger WAR or EAR file. The generated samples demonstrate the executable JAR file and the EAR file techniques.

Usually, the concept of loadable means having the class file or the JAR file containing it within the CLASSPATH environment variable. With EAR file deployments, the EAR file is not present in the CLASSPATH, but there is an internal CLASSPATH setting within the EAR file specified in various manifest files.

The development group must treat the proxy JAR file and the runtime JAR files as third party libraries and deploy them appropriately. The following sections explain some concepts that may help in the deployment, but you should not consider them the only way to deploy.

Java Proxy Runtime Files

The Java Proxy requires a set of runtime JAR files to be present to execute properly. You can find the runtime files in the CA Gen\classes directory. The following is the list of JAR files:

- Csuxx.jar
- vwrtxx.jar
- odcxx.jar
- jprrxx.jar
- jprrxx.xml.jar (only required if XML API is generated)
- odcxx.jar
- odcxx.tcpip.jar (only if TCP/IP communication is required)
- odcxx.mqs.jar (only if MQSeries communication is required)

- odcxx.eci.jar (only if ECI communication is required)
- odcxx.ejbrmi.jar (only if Java RMI communication is required)
- odcxx.tuxedo.jar (only if Tuxedo communication is required)
- odcxx.ws.jar (only if Web Services communication is required)

Note: xx refers to the current release of CA Gen. For the current release number, see the *Release Notes*.

You should place the following properties files within the CLASSPATH:

- commcfg.properties
- codepage.properties

These properties files can be found at several places: CA Gen directory, CA Gen\classes directory, and CA Gen\classes\resources directory.

There are a set of Java class files that contain the user exits. These are shipped as Java source and class files and not within a JAR file. Because of this, you must deploy the contents of the CA Gen\classes directory to allow usage of the user exit class files. (particularly the com).

Deploying Java Proxy Runtimes

For the list of all the runtime files that must be deployed for the Java proxy to function, see [Java Proxy Runtime Files](#) (see page 92) in this chapter. You may find the task of deploying and coordinating these files complicated. Therefore, CA Gen provides a BAT file to help consolidate the runtime files into one deployable JAR file.

The mkjavart.bat file located in the CA Gen \classes is designed to build complete 'custom' runtime JAR files. The BAT file is capable of building runtimes customized for Java Proxies, Web Clients, and EJBs. The user exit classes are also packaged within the built JAR file. For more information on types of runtimes that can be constructed, see the usage of the BAT file (execute mkjavart.bat usage).

The easiest way to build a custom runtime JAR file for the Java proxy is by executing the following command:

```
mkjavart.bat runtime.jar JavaProxy commcfg AllComms XML
```

This command builds a file called runtime.jar with the Java proxy, all installed communication runtimes, and the optional XML runtime within it. To make the JAR file smaller, you should specify the particular communication types instead and omit the optional XML runtime if you do not need it.

The commcfg.properties file is designed to be modified after deployment and therefore may not be a good candidate for inclusion within the JAR file. In that case, omit the commcfg parameter. It then becomes necessary for the commcfg.properties to be deployed separately.

More information:

[Configuring the Java Proxy Communications](#) (see page 87)

Executing the Sample Applications

This section discusses the different ways to execute the sample application.

JSP Sample

The generation and installation process creates everything you need to invoke the JSP Proxy sample by a J2EE Application Server. The \samples\JSP directory contains a series of .JSP files for each method in the proxy. For convenience, there are additional files generated (operations.htm, default.htm, and blank.htm).

After the build process, all of these files are packaged into an EAR file for the component. The EAR file also contains all the runtimes necessary to execute the sample JSP application.

To execute the JSP, you must deploy the EAR file to a J2EE Application Server. For more information on how to deploy ear files, see the application server documentation.

After deploying, you can execute the JSPs by entering the following URL in a browser:

`http://localhost/<component-name>`

The component-name portion is considered the context for the JSP sample application. It is case-sensitive, so be careful what you enter there.

The EAR file is configured to automatically cause the default.htm file to be loaded. Alternatively, the following URLs are also possible:

`http://localhost/<component-name>/default.htm`
`http://localhost/<component-name>/operations.htm`
`http://localhost/<component-name>/<method-name>.jsp`

Once the JSP page is displayed, fill in all the import fields appropriately and click Execute button. At that point, the proxy is executed and any errors or the export view is displayed.

Stand-alone Application

The generation and installation process creates everything you need to invoke the stand-alone Proxy sample application. The \samples\APP directory contains a Test<method-name>.jar files.

When the JAR file is created, all of the generated code and runtime files needed to execute the sample application is incorporated into it. This means that the proxy JAR and the runtime JAR files have been copied there. If any one of them changes, the Build Tool rebuilds the proxy and subsequently repackages the sample JAR files.

The JAR file created is considered an executable JAR file. The JAR file can be launched automatically with the -jar parameter to the JVM. To execute the sample Application, launch the executable from Windows Explorer or execute the following from a command line:

```
java -jar Test<method-name>.jar
```

No parameters are required. Once the application starts, simply enter the import view and system information as needed. Click Execute to invoke the server. The export view results are then displayed. Errors are shown in a popup dialog box.

XML Test Application (Optional)

If the optional XML programming interface is selected, the generation and installation process creates everything you need to invoke a Java Proxy sample XML test application. The \samples\XML directory contains a Test<method-name>.jar file along with a sample import XML file and the generated XML schema (XSD) file.

When the JAR file is created, all of the generated code and runtime files needed to execute the sample application are incorporated into it. This means that the proxy JAR, the runtime JAR files are copied there. If any one of them changes, the Build Tool rebuilds the proxy and subsequently repackages the sample JAR files.

The JAR file created is considered an executable JAR file. The JAR file can be launched automatically with the `-jar` parameter to the JVM. Therefore, to execute the sample Application, launch the executable from Windows Explorer or execute the following from a command line:

```
java -jar Test<method-name>.jar <method-name>Sample.xml
```

The parameter is required and must be the input XML file. A sample import XML file is generated and contains the syntax structure and where possible default values for the import view. If needed, the sample import XML file for the server can be modified. The resulting errors or export view data is the standard output of the command.

Note: BLOB data in XML is represented by base 64 format.

Chapter 5: Java Proxy (Classic Style)

The classic style Java Proxy is a Java-based interface that lets Java applets, applications, and servlets to access CA Gen servers. There are two forms of the classic style Java Proxy:

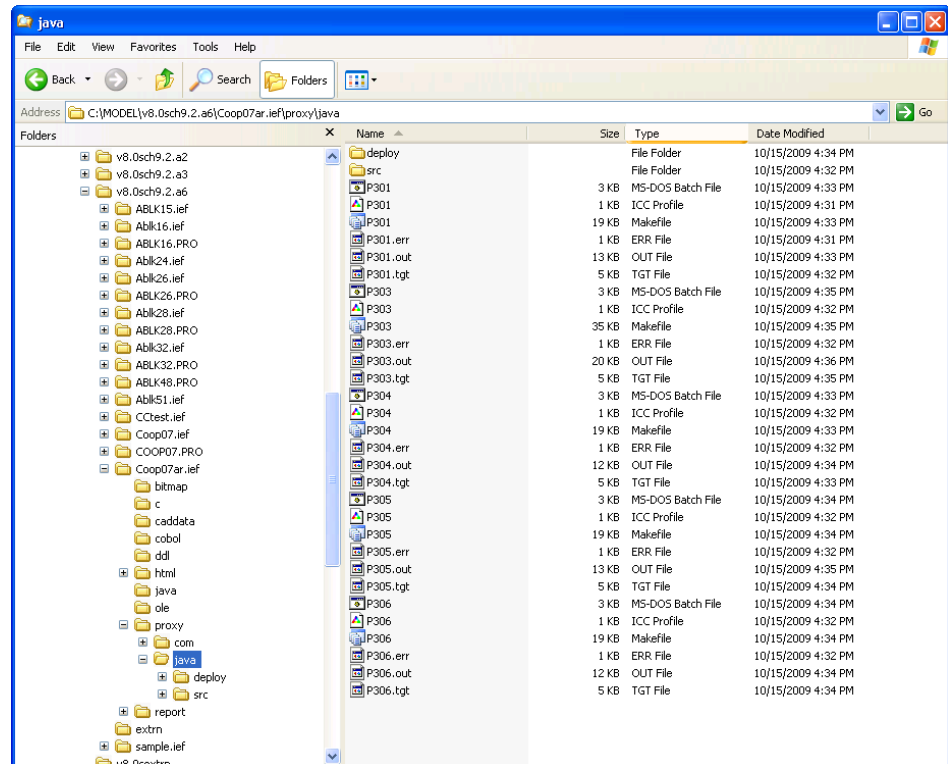
- The first consists of three parts that are generated from A CA Gen model: a Client JavaBean Proxy that resides in the Java application or Java applet, a Servlet Proxy that resides on the Web Server, and a test user interface applet that can be used to test the Client Java Bean and also as an example applet.
- The second consists of several parts that are generated from a CA Gen model: an application Java Bean Proxy that resides in a Java application or Java Servlet/EJB, a test user interface application, JSP pages, and possibly an XML Test Application that can be used to test the application Java Bean and as examples.

Java Proxy Generated Code

After generating your Classic Style Java Proxy, a \proxy\java directory exists within your <model-name>.ief directory. All proxy code is contained in a series of subdirectories beneath \proxy\java.

Note: The Java Proxy and the Classic Style Java Proxy both generate into the \proxy\java directories and overwrite each other if the names are the same.

The following file map shows the structure of the generated directories:



After the Build Tool has successfully built the proxy, the \proxy\java directory includes a \deploy directory. The \deploy directory includes \client, \clientUI, \Abean, \mainUI, \servlet, and \jsp directories, and optionally contains a \xml directory. The \clientUI and \mainUI directories contain generated HTML code, an applet with a test UI, and an application with a test UI. This code is not part of the proxy itself, but is provided as a sample for quick tests. The file <bean package name> in the \src\client directory contains a Javadoc of the API.

Java Proxy (Classic Style) Interface

This section describes the interface of the Java Proxy (Classic Style). This interface is generated into a client or application JavaBean.

The interface consists of a set of properties and methods. The properties can be accessed using methods of the form setXXX and/or getXXX. For example:

```
myproxy.setClientId("clientid");
```

```
String id = myproxy.getClientId();
```

Common Properties and Methods

The following properties and methods are common to all Java (Classic Style) proxies:

- **Clear**-Clears all import and export attributes to their default values if one is specified in the model or otherwise to 0 for numeric, empty string for string attributes, and null for date, time, and timestamp attributes.
- For attributes in repeating groups, the Clear method sets the repeat count to the max occurrences. In addition to clearing attribute properties, the Clear method also clears the system properties `commandReturned`, `exitStateReturned`, and `exitStateMsg`.
- **ClientID**-Retrieves or sets the client user ID property (String data type) to be sent to the server where the procedure step executable code is installed. A client user ID is usually accompanied by a client user password that can be set with the `clientPassword` property. Security is not enabled until the security user exit is modified to enable it.
- **ClientPassword**-Retrieves or sets the client user password property (String data type), which is sent to the server where the procedure step executable code is installed. A client user password is usually accompanied by a client user id, which can be set with the `ClientId` property.
- **ComCfg**-Set this value to the string value associated with the desired communications configuration. The string should take the form similar to the configuration string described in the `commcfg.properties` file, with the exception that it should not contain the `trancode` parameter. For example, for a TCP/IP transport configuration a `commcfg.properties` specification might be:

```
tranX TCP hostname portNumber
```

The corresponding ComCfg string would then be:

```
TCP hostname portNumber
```

Each supported transport has a slightly different communication string specification. In all cases the ComCfg string is expected to be the same as defined in the `commcfg.properties` file, but without the `trancode` parameter at the beginning. For more information, see the chapter “Overriding Communications Support at Execution Time” in the *Distributed Processing - Overview Guide*.

- **CommandReturned**-Retrieves the command returned property (read only String property), if any, after the server procedure step has been executed.
- **CommandSent**-Retrieves or sets the command sent property (String data type) to be sent to the server where the procedure step executable code is installed. This property should only be set if the procedure step uses case of command construct.
- **Dialect**-Sets or retrieves the dialect property (String property). It has the default value of `DEFAULT`.

- **SAXParser-Optional.** Retrieves or sets the classname of the SAXParser. This method can be used to override the default XML SAX parser used when performing XML parsing.
- **Execute-**This method executes the server operation as a synchronous cooperative flow.
- **ExecuteXML-Optional.** This method executes the server operation as a synchronous cooperative flow. However, unlike the normal execute method, it receives and returns its data as XML streams.
- **ExecuteAsync-Optional.** This method executes the send portion of an asynchronous cooperative flow. An ID is returned containing a unique identifier, which can be used later for retrieving the associated results. If no response is required, a parameter setting can indicate that no results are needed for this flow.
- **ExecuteAsyncXML-Optional.** This method executes the send portion of an asynchronous cooperative flow. However, unlike the normal executeAsync method, it receives its import data as an XML stream. All other parameters behave similarly to the executeAsync method.
- **ExecuteAsyncGetResponse-Optional.** This method executes the retrieve portion of an asynchronous cooperative flow. This method requires the ID of the associated request. It also lets you specify whether the method should block or wait if the results are not immediately available.
- In a non-blocking situation, the method returns before the results are processed. The result value indicates whether results are processed. If they are processed successfully, the export view area is populated with data, the transaction is considered complete, and the ID is no longer valid.
- **ExecuteAsyncGetResponseXML-Optional.** This method executes the receive portion of an asynchronous cooperative flow. However, unlike the normal executeAsyncGetResponse method, it returns its results as an XML stream. All other parameters behave similarly to the executeAsyncGetResponse method.
- **CheckAsyncResponse-Optional.** This method checks the status of an outstanding asynchronous request for a specific ID. The result value indicates whether the status is PENDING or AVAILABLE. There are generated constant values for each proxy that can be checked. This is the same as performing an ExecuteGetResponse in a non-blocking mode, but it usually executes many times faster due to less overhead.
- **IgnoreAsyncResponse-Optional.** This method indicates the runtime that the application is no longer interested in the outstanding request and the results may be discarded. All asynchronous programs should either finish a request by retrieving the results or calling this method. The runtime can save processing time and memory if it is aware that the results are no longer required.
- **ExitStateMsg-**Returns the current status text message (read only String property), if one exists. A status message is associated with a status code and can be returned by a CA Gen exit state.

- **ExitStateReturned**-Read only property (int data type) containing the exit state returned from the server.
- **ExitStateSent**-Retrieves or sets the exit state property (int data type) sent to server procedure step (typically, a 10-digit numeric representation of the exit states).
- **ExitStateType**-Read only property that returns the `exitstatetype` based upon the server procedure step exit state.
- **FileEncoding**-Specifies a specific Codepage that the CA Gen communications runtime uses when translating Unicode data prior to transmitting a flow request to the target server execution environment. By default, the file encoding for the Unicode data will be the system default for the JVM.
- **NextLocation**-Retrieves or sets the location name (NEXTLOC) property (String data type) that may be used by ODC user exits.
- **ServerLocation**-(Only Used by applet/Servlet Bean) Sets or returns the URL (URL data type) for the server hosting the servlet.

Note: Java never lets an applet to connect to any arbitrary server. If the applet is running in a browser, it can only connect to the servlet running on the server from where the applet was downloaded. This property is useful for testing and running an applet outside the browser environment, or for using the bean as part of a Java application. This property must be set before calling the `execute` method.

From an applet:

```
object.setServerLocation(getDocumentBase());
```

Or, for an application:

```
try { object.setServerLocation (new URL ("http://hostname"));}  
catch ( malformedURL Exception e) { }
```

- **Tracing**-Setting this property to one, enables the JavaBean to write trace statements to the standard output (the Java console in a browser).

Unique Attribute Properties

Each class in a Java Proxy has a unique set of properties based upon its procedure step import and export attributes. The import properties are the read and write properties, whereas the export properties are read only. The name of each of these properties is the combination of the attribute entity view name with its entity name and its attribute name.

The attribute domain, length, decimal place value, and precision flag determine the type of these properties. Each attribute has an accessor and setter method defined for it. The names are `GetXXXX` and `SetXXXX`. Some attribute types contain additional accessor and setter methods for backward compatibility. The following table lists them. Numerics may also contain special accessor and setter methods.

CA Gen Type	Java Type	Additional Accessors/Setters	Notes
Text, mixed or DBCS text	String	n.a	
Number (length <= 4 decimals = 0)	short	n.a	
Number (4< length <= 9 decimals = 0)	integer	n.a	
Number (length > 9 decimal = 0)	double	n.a	
Number (decimal > 0)	double	n.a	
Number with C Precision	BigDecimal	SetAsString XXXX GetAsString XXXX	
Date	Calendar	SetAsInt XXXX GetAsInt XXXX	Integer format must be YYYYMMDD (that is, 19990101). Null dates must be specified as null calendar objects.
Timestamp	Calendar	SetAsString XXXX GetAsString XXXX	String format must be YYYYMMDDHHMMSSSSSSSS (19981231123000111111). Null timestamps must be specified as null calendar objects.
Time	Calendar	SetAsInt XXXX GetAsInt XXXX	Integer format must be HHMMSS (that is, 12:01:00). Null times must be specified as null calendar objects. The Calendar or Time objects do not accept 24 as an hour. If 24 is passed as a parameter, it is converted to 00, so the time 24:00:00 retrieved from the server gets modified to 00:00:00 which is effectively same as the NULL time. This is different from the GUI and block mode runtimes which do allow a time of 24:00:00.
BLOB	byte[]		

For example, attributes ID (Number domain, length 3, 0 decimal places) and NAME (Text domain, length 15) of entity view names IMPORT, EXPORT, and entity EMPLOYEE can be used as:

```
oper = (<beanclass>)Beans.instantiate(getClass().getClassLoader(), <bean package
name>);
oper.setImportEmployeeId(123);
oper.setImportEmployeeName("EmployeeName");
oper.execute();
String phoneNumber = oper.getExportEmployeePhoneNumber();
```

Special String Methods

Each of the three attribute properties (short, integer, or double) has two set methods: setXXX(type) is the standard set method and setAsStringXXX(stringvalue).

This additional set method lets you assign the value from a text field into a Bean property without first doing a conversion. The CA Gen Java runtime does the conversion for you and returns a PropertyVetoException if the string cannot be parsed as a valid number of the type expected.

These special String methods provide 6 digit sub second support (microsecond precision). The standard methods only provide 3 digit sub second support (millisecond precision).

Date, Time, and Timestamp Properties

Date, time, and timestamp values are represented as Calendar objects. Methods for setting these properties as integers and strings are also provided for compatibility.

NULLs must be represented as null calendar objects. Pass null to the setter methods. Also, make sure your code handles nulls returned from the accessor methods.

Decimal Precision Properties

Prior to COOL:Gen 5.1, Decimal Precision numbers were stored as strings. They are now stored as Java Big Decimals, which prevents the developers from doing extra conversions.

Methods are still provided for manipulating these properties as strings, if needed.

Repeating Group Property Handling

Attributes that are in repeating groups have the same property names as non-repeating attributes, but with an index (an int index parameter).

Two additional properties are defined for repeating groups. The first property, Max, represents the maximum number of rows in the repeating group. The second property, Count, represents the current number of populated rows within the group.

These property names are formed by the combination of the group view name, and the words "Max" or "Count" respectively. The read-only Max property is accessible via the name <ImportGroupName>Max.

The Count property is set by calling the set<ImportGroupName>Count method and read by calling the get<ImportGroupName>Count method. The Count value cannot be set to a negative value or larger than Max. Attempts to do so will result in a PropertyVetoException.

In all cases, <ImportGroupName> is the name of the repeating group as defined within the CA Gen model.

For repeating import groups, the Max property is considered read only, while the Count property is read/write.

Note: It is the responsibility of the user written application to set the import repeating group Count property appropriately before flowing to a server. Failure to do so will result in the view data not being processed properly.

For repeating export groups, the proxy application should consider both the Max and Count properties to be read only.

Example 1

Repeating import group of group view name LIST for procedure step EMPLOYEE_ADD containing attributes ID and NAME of entity view name IMPORT and entity EMPLOYEE:

```
int i;
for (i = 0; i < oper.ListMax; i++)
{
    oper.setImportEmployeeId(i, i);
    oper.setImportEmployeeName(i, "Name" + i);
}
oper.setListCount((short) i);
```


Example 2

Repeating export group of group view name OUT_LIST containing attributes ID and NAME of entity view name OUT and entity EMPLOYEE:

```
for (int i = 1; i < oper.getOutListCount; i++)
{
    int id = oper.getOutEmployeeId(i);
    String name = oper.getOutEmployeeName(i);
}
```

Java Proxy Code

The generated Java code consists of the following parts for each procedure step. These parts are mapped into directories under the generated source tree by the same name. Java source files are generated and then compiled into class files. The class files are then combined into jar files that are copied into the deployment directories.

Abean

An application JavaBean that contains an API to get/set the properties for the procedure step. These properties are a direct mapping of the import/export views. A Javadoc.html file is produced for each bean generated.

Client

An applet JavaBean that contains an API to get/set the properties for the procedure step. These properties are a direct mapping of the import/export views. A Javadoc.html file is produced for each bean generated. This bean must be used with the corresponding servlet classes.

clientUI

A test applet that uses the applet bean API. It can be loaded either into a web browser or an applet viewer. Its purpose is to test the bean API and provide a model of how the API can be used and, as such, is not suitable for production deployment.

mainUI

A test application that uses the application bean API. It can be loaded using Java runtime execution or from an application debug driver. Its purpose is to test the bean API and provide a model of how the API can be used and is not suitable for production deployment.

common

A set of Java classes shared by the client bean and the servlet. These classes represent the import/export views and are needed for data streaming between the client and the servlet. You do not access these classes directly.

Servlet

The server side piece of the Java equation. Its purpose is to provide the communications bridge between the client bean and the CA Gen server.

JSP

A set of sample HTML and JSP pages that can be used to test the application bean. Its purpose is to test the bean API and provide a model of how the API can be used. As such, it is not suitable for production employment.

XML

Optional. Contains an XML Schema Definition file (XSD) for use within XML data streams, a set of sample XML, and a test application for executing the XML API.

Using a Java (Classic Style) Proxy

The generated Java proxy is capable of supporting both synchronous and asynchronous cooperative flows. The main difference between these processing types is the actual method used to perform the cooperative flow.

All sample code supplied with the product (APP, JSP and XML) demonstrates only the use of synchronous capabilities. However, the following sections provide, in pseudo-code, an outline of the structure of synchronous and asynchronous code.

Note: All non CA Gen server errors are handled as standard Java exceptions. Handling these exceptions is not shown in the following sections.

More information:

[Asynchronous Processing](#) (see page 107)

Synchronous Processing

The following contains the code for a synchronous process:

```
op=Beans.Instantiate(. . .)
...
op.addCompletionListener(. . .)
op.addExceptionListener(. . .)
...
<set up import views and communications/system attribute information>
...
op.Execute() //Perform synchronous flow
...
<check results for errors>
...
<retrieve export view data>
...
```

Note: If required, you can substitute ExecuteXML method for the synchronous Execute method.

Asynchronous Processing

The code is broken up into four distinct sections, as you can embed each piece at different locations within the user application.

Instantiation

Use to following code for instantiation:

```
...
op=Beans.Instantiate(...)
...
```

Send Data

Use the following code to send data:

```
...
<setup import views and communication/system attribute information>
...
id=op.ExecuteAsync(False) //Perform asynchronous send, with a response expected
...
```

Note: If required, you can substitute the ExecuteAsyncXML method for the asynchronous ExecuteAsync method.

Check Status

Use the following code to check status:

```
...
switch(op.checkAsyncResponse(id))
{
...
    case XYZ available
        ...
    case XYZ Pending:
        ...
    default:
        ...
}
...
```

Retrieve Results

Use the following code to retrieve results:

```
...
boolean processed=op.ExecuteGetResponse(id,True)

//Perform the retrieval of the results, with blocking turned on
...
<check results for errors>
...
<retrieve export view data>
...
```

Note: If required, you can substitute the ExecuteGetResponseXML method for the asynchronous ExecuteGetResponse method.

Ignore Request

Use the following code to ignore request:

```
...
op.ExecuteIgnoreResponse(id)
//Processes the ignore request on the request indicated by the id
...
```

Security Processing

A Java Proxy provides facilities to implement Distribute Processing Security, as described in the chapter of the *Distributed Processing - Overview Guide*.

By default, the Java Proxy does not exploit the use of the Proxy Runtime security features. To utilize the security features, the application developer must add code to their Java application that will set the ClientID and ClientPassword properties of the Java Proxy Import view object. Depending on the return value of the client security user exit, the security data fields will or will not be sent to the target Distributed Processing Server (DPS). In addition, if the client security user exit returns SECURITY_ENHANCED, the client security user exit(s) can cause an optional security token to be added to the data flow. The collection of security data that is sent as part of the cooperative flow is validated by user exits residing in the execution environment of the target DPS.

For a Java Proxy flowing to a DPS using TCP/IP, MQSeries, or ECI, the supporting runtime lets a portion of the Common Format Buffer to be encrypted on the way to the target DPS, and decrypted on the way back from the target DPS. The use of encryption and decryption are enabled by way of user exits. See User Exits in this chapter, for a description of the user exits used to enable encryption and decryption processing for a Java Proxy.

More information:

[User Exits](#) (see page 110)

Preparing for Execution

Before you execute an application that uses a generated Java proxy, address the following items:

- You may need to modify the Java Proxy Runtime for use in the user application execution environment.
- You must deploy the proxy and Runtime to the applications execution environment.

Configuring the Java Proxy Runtime

By design, there exist areas within the Java Proxy Runtime that can be influenced by an application developer. An application developer does many activities that include updating the commcfg.properties file to influence the communications part of the runtime and modifying user exits. The user exits influence security processing and lets the arguments specific to the selected communications processing be overridden.

The Java Proxy runtime is common between Java Proxy and Java Proxy (Classic). Both proxies can be used within one application for different DPSs. They are deployed at the runtime.

Configuring Java Proxy Communications

A generated proxy contains communications information as defined in the model. The proxy runtime provides capability to specify the communication information at runtime. This information overrides any information specified in the model and was built into the proxy during generation.

Additionally, the proxy runtime is capable of turning the tracing ON and OFF, as well as provides capability to control the amount of file caching that takes place at runtime.

Note: For specific formatting instructions, see the default file installed with the Java Proxy software, or see the *Distributed Processing - Overview Guide*.

User Exits

The user exits invoked at runtime for a Java Proxy are common to other Java execution environments.

Note: For more information about each user exit, see the User Exit Reference *Guide*.

A User Exit is a way for a CA Gen customer to modify a behavior of the runtime environment. Several user exits are useful during the deployment of the CA Gen Java proxies. The Java user exits apply to either the normal or classic style Java Proxies. The exits that apply to proxies can be grouped into two categories: security and communication.

After modifying any Java user exits, the Java class must be recompiled. Then the class must be deployed to replace the old version in the CLASSPATH. If a runtime JAR was built using mkjavart.bat, then it must be rebuilt.

Note: The CA Gen runtime is designed to run in a multi-threaded environment so that the user-written code in user exits is thread safe.

The following user exits are bundled together as Java classes. The user exits are installed as part of the Java Proxy runtime.

More information:

[Deploying a Java Proxy](#) (see page 114)

Security User Exits

The user exit source code is located in a directory under `classes/com/ca/genxx/exits/msgobj` in the CA Gen installation directory (for example: `classes/com/ca/genxx/exits/msgobj/cfb`).

Note: *xx* refers to the current release of CA Gen. For the current release number, see the *Release Notes*.

Flow type: CFB (Common Format Buffer)

User Exit Class	Purpose
CFBDynamicMessageSecurityExit	The Client Security user exit class is used to direct the runtime to incorporate the ClientID and ClientPassword Import Object attributes into the CFB. If this exit returns SECURITY_ENHANCED, it can cause an optional Security Token field to be added to the CFB.
CFBDynamicMessageDecryptionExit	The export message decryption user exit class provides the opportunity for decrypting the portion of the CFB that has been encrypted by the target DPS execution environment. This exit must implement the companion to the DPS encryption user exit to facilitate the DPS response buffer to be interpreted as a valid response CFB.
CFBDynamicMessageEncryptionExit	The import message encryption user exit class provides the opportunity for a portion of the outbound CFB to be encrypted using an encryption algorithm implemented within this user exit. The target DPS execution environment must implement a companion decryption user exit for the CFB to be interpreted as a valid request CFB.

Flow type: Tuxedo

User Exit Class	Purpose
TUXDynamicSecurityExit	The Client Security user exit is used to provide security data to the runtime for use when processing a flow request. The security data is used while creating a Oracle Jolt session.

Communications User Exits

The following user exit classes are invoked by Java Proxies flowing to their target DPSs using a supported communications type. The user exit source code is located in a directory under `classes/com/ca/genxx/exits/coopflow` in the CA Gen installation directory. These classes are installed as part of the specific communications runtime (for example: `classes/com/ca/gen85/exits/coopflow/tcpip`).

Note: *xx* refers to the current release of CA Gen. For the current release number, see the *Release Notes*.

User Exit Class	Purpose
TCPIPDynamicCoopFlowExit	The methods in this class let you override the host name and port number. Additionally, this class contains a method that exposes that a TCP/IP processing exception has occurred. This user exit method indicates if the processing of the flow should be retried. For example, retry in the event of a Connection Reset by Peer error.
MQSDynamicCoopFlowExit	<p>The methods in this class let you override the following MQSeries processing data items:</p> <ul style="list-style-type: none">Local Queue Manager nameRemote Queue Manager namePut Queue nameReply to Queue nameReply timeout valueAn indication if the put queue must be closed after the put operationAn indication if the get queue must be closed after the get operationA dynamic reply to queue (if the reply to queue name specifies the name of a model queue)

User Exit Class	Purpose
ECIDynamicCoopFlowExit	<p>The methods in this class let you override the following CICS Transaction Gateway processing data items:</p> <ul style="list-style-type: none"> The ECI client security class name The port address of the CICS Transaction Gateway The protocol to be used to communicate with the CICS Transaction Gateway (example: local, TCP, auto, http, https, and SSL) The ECI server security class name The CICS system name, as known within the Transaction Gateway (or supporting Universal Client), to which the cooperative request should be sent. <p>Additionally, this class contains a method that exposes that an ECI processing exception has occurred. This user exit method indicates if the processing of the flow should be retried.</p>
EJBRMIDynamicCoopFlowExit	<p>The methods in this class let you override the following EJB RMI processing data item:</p> <ul style="list-style-type: none"> The provider URL <p>Additionally, this class has a method that permits an optional User object to be sent as part of the EJB RMI flow to the EJB server.</p> <p>This class has a method for handling exceptions that occur when performing an EJB RMI operation (open, read, write, and so on). This method lets the failed request to be retried.</p>
TUXDynamicCoopFlowExit	<p>This class contains two methods that expose the data flowing to and from the server. The OutData method provides the ability to examine or modify the request data being sent to the server. The InData method provides the ability to examine or modify the response data received from the server.</p>
WSDynamicCoopFlowExit	<p>This class has methods that let you manipulate the target Web Service endpoint URL by modifying the following members:</p> <ul style="list-style-type: none"> baseURL contextType <p>This class has a method for handling exceptions that occur when performing a Web Service operation. This method lets the failed request to be retried.</p>

Conversion User Exit

This user exit class is used to modify data saved to and retrieved from a database. It currently handles Strings only and is sometimes required for languages where strings stored in a database need to be modified for display.

Note: For more information about this user exit, see DataConversionExit-Java Data Conversion Exit in the *User Exit Reference Guide*.

Deploying a Java Proxy

There are two types of Classic Style Java proxies that are generated and compiled. They are the Applet/Servlet version and the Application Bean version. The following sections detail the deployment techniques for each of them.

Applet/Servlet Deployment

The Applet/Server Java proxy consists of two JAR files. One file is located in the `deploy\client` directory and is referred to as the client bean JAR. The other one is located in the `deploy\servlet` directory and is referred as the servlet bean JAR.

Client Bean JAR

You should download the client bean JAR to a web browser for execution by a user applet. This JAR file already contains the CA Gen runtime. You must copy the JAR file to a location known by the HTML server, so that it can be referenced by the HTML loading the user applet.

Servlet Bean JAR

The servlet bean JAR is intended to execute on a servlet engine. You should copy it to the machine with the servlet engine, along with all the CA Gen support files. The servlet engine should be configured to have the JAR files and support files in CLASSPATH.

For a listing of the CA Gen support files, see Support Files in this chapter.

Application Bean Deployment

The Application Bean Java proxy is a stand-alone proxy and can be executed in a variety of execution environments. The execution environment may be a stand-alone Java application, a servlet engine, a JSP engine, or an EJB environment. In all cases, the deployment files remain the same. The only difference is in configuring each environment to load properly and execute the files.

The Application Bean proxy is located in the `deploy\Abean` directory. Copy the JAR file and the CA Gen support files to the environment.

Support Files in Java Proxy (Classic Style)

Both types of Classic Style Java proxy require the same set of support files. These files are located in either the CA Gen root directory, or one level below the root directory in the classes directory:

- commcfg.properties
- codepage.properties
- csuxx.jar
- vwrtxx.jar
- odcxx.jar
- jprtxx.jar
- jprtxx.xml.jar (optional)

Note: xx refers to the current release of CA Gen. For the current release number, see the *Release Notes*.

The following exits are also required. They are located in the com\ca\genxx\exits\common directory.

- CompareExit.class
- LowerCaseExit.class
- UpperCaseExit.class

Note: Depending on the communication type selected, additional JAR files and exits may also be required.

mkjavart.bat Batch File

All JAR files must be explicitly referenced in environment configurations. The batch file mkjavart.bat can help with this process. This file automatically builds one runtime JAR file containing all the desired runtime components. This batch file is named mkjavart.bat and is located in the CA Gen classes directory.

For an explanation of the available execution options for mkjavart.bat, execute the command mkjavart.bat usage.

Preparing the Applet/Servlet Java Proxy for Web Access

After you generate a Java Applet/Servlet Proxy, each `\deploy\servlet\<component>` directory contains a JAR file containing the proxy servlet and the required runtimes. For the Web Server to access the Java Proxy code, you should copy this JAR file into a Web Server directory as follows:

Follow these steps:

1. Copy the `\deploy\servlet\<component>`. JAR file into a directory on the Web server.
2. This directory is the one specified by the servlet tool you have installed. See the tool documentation for this location.

The CLASSPATH of the product that provides the servlet support must be modified to include the full path and JAR file name of the servlet, for example:

`C:/myservlets/<component>.jar`

If you have generated more than one proxy, you must repeat this procedure for each one.

Note: It is possible to extract the individual classes and directories from the JAR file. This permits only the directory to be included in the CLASSPATH, rather than each servlet JAR file being included individually.

3. To test the servlet installation, see Deploying a Java Proxy in this chapter.

Note: The Web Server runs as a service on Windows operating systems, so you must ensure that all environment variables are part of the system environment.

4. Insert the client bean JAR file in a location that can be used by the custom applet written by the user.

Note: To test using the sample clientUI applet, see Running the Sample Applet/Servlet JavaBean in this chapter.

Preparing the Application JavaBean Proxy

Insert the application (Abean) JAR file in a location that can be used by the custom application/servlet written by the user. The following sections discuss how to test with the sample MainUI application loadable by the class loader.

Preparing the Application Bean Java Proxy for JSP Access

When you have generated a Java Application Bean Proxy, the `\deploy\Abean` directory contains a JAR file for each component. The `\deploy\JSP` directory also contains a set of four JSP pages for each proxy method.

To execute the JSP sample code, copy the `\deploy\Abean\<component>` JAR file into a directory that is accessible by the product providing the JSP support. The CLASSPATH may need to be modified to include the JAR file.

Note: It is possible to extract the individual classes and directories from the JAR file. This permits only the directory to be included in the CLASSPATH, rather than each individual JAR file.

Copy the JSP and HTML pages to a location that is accessible to the web server. See the web server documentation for a default location, or for steps to customize the locations.

Executing the Sample Applications

This section discusses the different ways to execute the sample application.

Applet/Servlet JavaBean Sample

The JavaBean with a test UI is created in this directory:

<model-name>.ief\proxy\java\deploy\clientUI. There is one JAR file and one HTML file for each <component>. To access the proxy, the applet needs to be downloaded to the Web Browser. Any required data is entered in the input fields and then sent to the servlet using the method activated by the push button provided.

The HTML and JAR files must be copied to a location on the web server to be addressed by a URL. See the specific web server documentation for these procedures.

Java Application Bean Sample

The Java application with the test UI is created in the

<model-name>.ief\proxy\java\deploy\mainUI directory. There is one JAR file for each <component>.

Note: The UI JAR file includes the mainUI and Abean class files.

Set the current directory to the <model-name>.ief\proxy\java\deploy\mainUI directory and execute each <component> that you want to test using the following command:

```
java <component>.MainUI.<method-name>UI
```

The sample UI contains two pages for data and an execute push button. The first page displays all the fields for the import and export views of the proxy. The second page (accessible by clicking Show System Attributes), displays all the fields that can be set on the proxy that control the communications and flow.

More information:

[Java Proxy Interface](#) (see page 74)

Application Bean JSP Pages Sample

From a browser, execute the URL of the <component>.html page that was generated. If the web server and JSP engine are properly configured, then the JSP import page is displayed.

Note: The first reference to a JSP page is slower than normal, as JSP engine performs the Java compiles. Subsequent references are much faster.

XML Test Application (Optional)

Follow these steps::

1. Place the Java Proxy Abean jar file or classes into the CLASSPATH.
Note: Jrun and Java Runtime must be set in the CLASSPATH.
2. Place the CA Gen runtime jar files into the CLASSPATH.
3. Place the model.ief/proxy/java/src/<component>/xml directory into the CLASSPATH.
4. Edit the <method-name>Sample.xml file if necessary.
5. Change to the model.ief/proxy/java/src/<component>/xml directory.
6. Execute the following command:

```
java Test<method-name> <method-name>Sample.xml
```

Note: BLOB data in XML is represented by base 64 format.

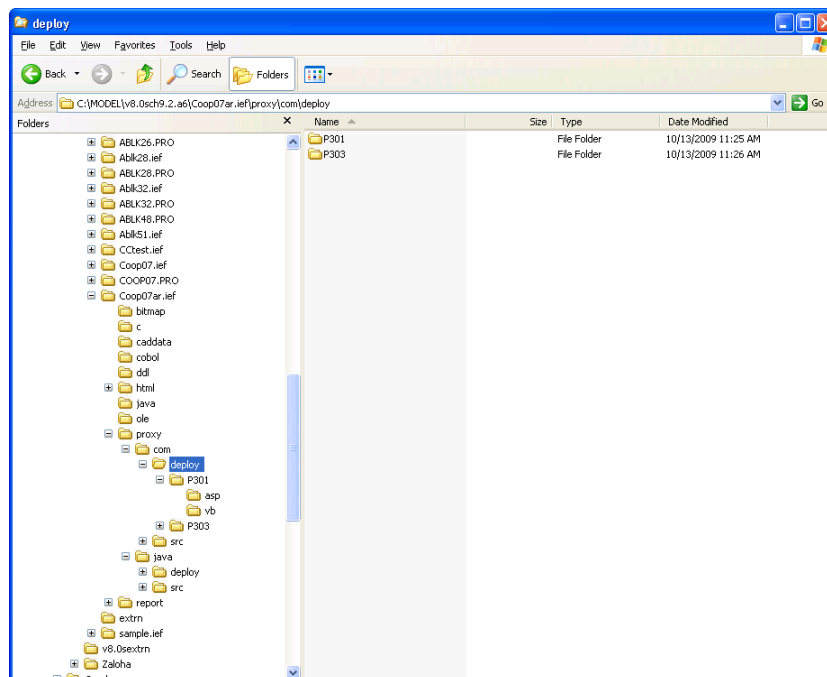
Chapter 6: COM Proxy

The COM Proxy is a generated programmatic interface that OLE automation user-written client application use to process cooperative flows to generated CA Gen Distributed Processing Server (DPS) applications.

COM Proxy Generated Code

After generating your COM Proxy, a `\proxy\com` directory exists within your `<model-name>.ief` directory. All proxy code is contained in subdirectories beneath your `<model-name>.ief \proxy\com` directory.

The following file map shows the structure of the generated directories:



After the Build Tool has successfully built the proxy, the `\proxy\com` directory includes a `\deploy` directory, which includes `\asp` and `\vb` directories. If the XML interface was selected for proxy generation, the `\deploy` directory also contains a `\xml` directory. The `\asp`, `\vb` and `\xml` directories contain generated sample Active Server Page code, a sample Visual Basic template, and a sample XML application, respectively. The Active Server Page and Visual Basic files are not part of the proxy itself, but are provided as samples and quick tests. The XML directory contains an XSD file. The XSD file is part of the proxy, but the sample XML file and test application are not part of the proxy. If you want to use these files, you can modify them as per your requirements.

COM Proxy Interface

Any OLE automation client can make use of a generated COM Proxy. This chapter discusses the interface exposed by a generated COM proxy in terminology consistent with Visual Basic (VB) nomenclature. That is, VB type names are used instead of C++ names.

There is one generated COM proxy for each component. A COM proxy's library name (as it appears in a VB Object Browser) is <component>Typelib. Within each COM Proxy, there is one class (as seen in the VB Object Browser) for each procedure step of the component. For example, if a <component> has the procedure steps READ_EMPLOYEE and UPDATE_EMPLOYEE, there are two classes, ReadEmployee and UpdateEmployee.

Each class in a COM Proxy has a common set of properties and methods that are associated with all COM Proxy classes, and a unique set of properties based upon the procedure step import and export attributes.

See the generated sample code for an example of how an ASP or VB application might make use of a generated proxy. The sample code is placed into the \asp, \vb and \xml subdirectories of your <model-name>.ief \proxy\com directory.

Common Properties and Methods

The following properties and methods are common to all COM proxy classes:

- ClientId-Retrieves or sets the client user id property (String data type) to be sent to the server where the procedure steps executable code is installed. A client user id is usually accompanied by a client user password, which can also be set with the ClientPassword property. Security is not enabled until the security user exit is modified to enable it.
- ClientPassword-Retrieves or sets the client user password property (String data type), which is sent to the server where the procedure step executable code is installed. A client user password is usually accompanied by a client user id, which can also be set or retrieved with the ClientId property.
- CommandSent-Retrieves or sets the command sent property (String data type) to be sent to the server where the procedure step executable code is installed. This property should only be set if the procedure step uses case of command construct.

- **ExitStateSent**-Retrieves or sets the exit state property sent to the server procedure step. If the server does not use exit states, then the data type of this property is Long. Otherwise, the type of this property is the <component>ExitStates enumeration type that contains a value for each exit state with its name prefixed by the component name.

This enumeration also contains a predefined value NONE prefixed by the component name with a value of zero that is the default exit state sent to the server if no ExitStateSent is specified. This property should be set only if the server procedure step refers the exit state sent to control its flow.

Example (of a component EMPLOYEE with exit state PROCESSING_OK):

```
Op.ExitStateSent = EmployeePROCESSING_OK
```

Location-Retrieves or sets the location name (NEXTLOC) property (String data type) that can be used by ODC user exit flow dlls.

- **CommandReturned**-Retrieves the command returned property (read only String property), if any, after the server procedure step has been executed.
- **ExitStateReturned**-Read-only property containing the exit state returned from the server. If the procedure steps server uses exit states, then the type of this property is the <component>ExitStates enumeration type, as described in the ExitStateSent method description. Otherwise, the type is Long.
- **OperationStatus**-Read-only property that returns the operation status based on the server procedure step exit state. This type of property is an enumeration of the following constants:

Enumeration Constant Name	Enumeration Constant Value
<component>NotYetRun	-1
<component>OK	0
<component>Informational	1
<component>Error	3
<component>Warning	2

Example (for a component with the name EMPLOYEE):

```
If Op.OperationStatus > EmployeeOk Then
MsgBox Op.OperationStatus & ": " & Op.OperationStatusMessage
End If
```

- **OperationStatusMessage**-Returns the status text message (read only String property), if one exists. A status message is associated with a status code, and can be returned by a CA Gen exit state.
- **ServerLocation**-Set this value to the name of the server where the remote <trancode>cm.dll has been installed.

- ComCfg-Set this value to the string value associated with the desired communications configuration. The string should take the form similar to the configuration string described in the commcfg.ini file, with the exception that it should not contain the trancode parameter. For example, for a TCP/IP transport configuration a commcfg.ini specification might be:

tranX TCP hostname portNumber

The corresponding ComCfg string would then be:

TCP hostname portNumber

Each supported transport has a slightly different communication string specification. In all cases the ComCfg string is expected to be the same as defined in the commcfg.ini file, but without the trancode parameter at the beginning. For more information, see the chapter “Overriding Communications Support at Execution Time” in the *Distributed Processing - Overview Guide*.

- Clear-This method clears all imports and exports attribute properties to their default values if one is specified in the model or otherwise to 0 for numeric, date and time attributes, empty string for string attributes and 00000000000000000000 for timestamp attributes.

For attributes in repeating groups, the clear method sets the repeat count to 0. In addition to clearing attribute properties, the Clear method also clears the system property CommandReturned, ExitStateReturned, OperationStatus, and OperationStatusMessage.

- Execute-This method executes the server operation as a synchronous cooperative flow.
- ExecuteXML-Optional. This method executes the server operation as a synchronous cooperative flow. However, unlike the normal Execute method, it receives its input and returns its results as an XML stream.
- ExecuteAsync-Optional. This method executes the send portion of an asynchronous cooperative flow. An ID is returned containing a unique identifier that can then be used for retrieving the associated results. If no response is required, a parameter setting can indicate that no results are needed for this flow, allowing fire and forget processing.
- ExecuteAsyncXML-Optional. This method executes the send portion of an asynchronous cooperative flow. However, unlike the normal ExecuteAsync method, it receives its input as an XML stream. All other parameters behave similarly to the ExecuteAsync method.

- **ExecuteGetResponse-Optional.** This method executes the retrieve portion of an asynchronous cooperative flow. This method requires the ID of the associated request. It lets you specify whether the method should block or wait if the results are not immediately available.

In a non-blocking situation, the method may return before the results have been processed. The result value indicates whether results have been processed. If they have been processed successfully, the export view area is populated with data, the transaction is considered complete, and the ID is no longer valid.

- **ExecuteGetResponseXML-Optional.** This method executes the receive portion of an asynchronous cooperative flow. However, unlike the normal **ExecuteGetResponse** method, it returns its output as an XML stream. All other parameters behave similarly to the **ExecuteGetResponse** method.
- **ExecuteCheckResponse-Optional.** This method checks the current status of an outstanding asynchronous request for a specific ID. The result value indicates whether the status is **PENDING** or **AVAILABLE**. There is a generated enumerated value for each proxy that can be checked. This is the same as performing an **ExecuteGetResponse** in a non-blocking mode, but usually executes many times faster due to less overhead.
- **ExecuteIgnoreResponse-Optional.** This method indicates the runtime that the application is no longer interested in the outstanding request, and that the results may be discarded. All asynchronous programs should either finish a request by retrieving the results or calling this method. The runtime can save processing time and memory if it is aware that the results are no longer required.

Unique Attribute Properties

Each class in a COM Proxy has a unique set of properties based upon its procedure step import and export attributes. The import properties are read and write properties, whereas the export properties are read only. The name of each of these properties is the combination of the attribute's entity view name with its entity name and attribute names. The attribute's domain, length, decimal place value, and precision flag determine the type of these properties. See the following table:

CA Gen	VB Type	Note
Number and Length <= 4 and Decimal Places = 0	Integer	
Number and Length >= 5 and Length <= 9 and Decimal Places = 0	Long	
Number and Length <= 15 and (Decimal Places > 0 or Length >= 10)	Double	

CA Gen	VB Type	Note
Number and Length > 15 without Decimal Precision	Double	Precision may be lost
Text or Mixed Text or DBCS Text	String	If attribute has permitted values, see special text permitted value property handling
Date	Date, String, or Long	See special Date Property Handling
Time	Date	
Timestamp	Date	See special timestamp attribute property handling
Number with Decimal Precision	Variant	Variant contains Decimal type
BLOB	Variant	Unsigned char SAFEARRAY

For example, attributes ID (Number domain, length 3, 0 decimal places) and Name (Text domain, length 15) of entity view name IMPORT and entity Employee can be used as:

```
Op.ImportEmployeeId = 999
Op.ImportEmployeeName = "XXX(15)"
```

Special Text Permitted Value Property Handling

If a text attribute has permitted values, then the COM Proxy surfaces two forms of properties for the same internal property: one that uses an enumeration type of the permitted values and another that takes a string.

The name of the former property is the same as given to unique attribute properties (that is, the combination of the attribute's entity view name with its entity name and attribute name) and the name of the later property is this same name with the suffix AsString.

For the former property, the name of its type is the combination of PV with the component name and the attribute entity name and attribute name. To enforce unique enumeration constant names, all permitted value names are prefixed with their enumeration type name. Additionally, if a space is used for a permitted value, its enumeration value name is the enumeration type name combined with the word spaces. For the later property (the AsString version), its type is String, which is the same type of text attributes without permitted values.

For example, text attribute HEIGHT with permitted values HIGH, MEDIUM, and LOW of component EMPLOYEE, entity view name IMPORT and entity EMPLOYEE:

```
Op.ImportEmployeeHeight = PVEmployeeEmployeeHeightHigh ' Setting property to HIGH
```

```
Op.ImportEmployeeHeightAsString = "MEDIUM" ' Resetting same property to MEDIUM
```

Special Timestamp Property Handling

The Data type associated with the timestamp attribute properties do not allow you to specify sub second information as does the CA Gen timestamp that provides for sub second six-digit information. To let you work with sub second timestamp information, there is another format of the timestamp property. This property has the same name for property with AsString suffix.

This format of property has the type String and it takes a twenty-digit string containing the CA Gen timestamp format (four digits for year, two digits for month, two digits for day, two digits for hour, two digits for minute, two digits for second and six sub second digits).

For example, attribute TIME of timestamp domain for entity view name IN and entity ORDER:

```
Op.InOrderTime = #11/20/97 4:56:03 PM#
```

```
Op.InOrderTimeAsString = "19971120165603123456" ' Same time with sub second value of 123456
```

Special Date Property Handling

The COM Proxy provides three interfaces for a Date property.

- The first one uses Automation date type
- The second one uses String (BSTR)
- The third one uses Long

The name of the first property is the same as given to the unique attribute property and the name of the other two properties are the same name with the suffix AsString or AsLong respectively. The COM automation Date type is limited to representing date values between 100.01.01 and 9999.12.31. To deal with date values outside that range, you can use the String or Long interface.

The String interface requires an eight-digit BSTR containing a date value in format of *YYYYMMDD* that is four digits for year, two digits for month and two digits for day.

For example, attribute HireDate of date domain for entity view name Import and entity Employee:

```
Op.ImportEmployeeHireDateAsString = "19921001"; /* Oct. 1st, 1992 */
```

The Long interface requires an eight-digit long integer containing a date value in the format of *YYYYMMDD* that is four digits for year, two digits for month and two digits for day.

For example, attribute HireDate of date domain for entity view name Import and entity Employee:

```
Op.ImportEmployeeHireDateAsLong = 19921001; /* Oct. 1st, 1992 */
```

Repeating Group Property Handling

Attributes that are in repeating groups have the same property names as non-repeating attributes, but with an index (a Long index parameter).

Two additional properties are defined for repeating groups. The first is a constant, *Max*, which represents the maximum number of rows in the repeating group. The second property, *Count*, is of Type Long and represents the current number of populated rows within the group.

The name for the *Max* property is formed by the combination of the procedure step name, group view name, and the word *Max*. The name for the *Count* property is formed by the combination of the group view name and *Count*. Thus, the read-only *Max* property is accessible via the name `<ProcStepNameImportGroupViewName>Max`. The *Count* property is accessed via the `<ImportGroupViewName>Count` property of the instantiated class object.

For repeating import groups, the *Max* property is considered read only, while the *Count* property is read/write.

Note: While it is possible for the user written application to set the *Count* property it will be incremented appropriately by the runtime API when attribute values are set via the instantiated class object as seen in the sample below. This count must be set if the server is to properly process the repeating group views. Undefined behavior will result if the *Count* value is set to a negative value or a value larger than *Max*.

For repeating export groups, the proxy application should consider both the *Max* and *Count* properties to be read only.

Example 1

For repeating import group of group view name LIST for procedure step EMPLOYEE_ADD containing attributes ID and NAME of entity view name IMPORT and entity EMPLOYEE:

```
For i = 1 to EmployeeAddListMax
Op.ImportEmployeeId(i) = i
Op.ImportEmployeeName(i) = "Name" & CStr(i)
Next i
` number of rows in repeating group
Dim integer size = op.ListCount
```

Example 2

For repeating export group of group view name OUT_LIST containing attributes ID and NAME of entity view name OUT and entity EMPLOYEE:

```
For i = 1 to op.OutListCount
Debug.Print CStr(Op.OutEmployeeId(i) & " " & Op.OutEmployeeName(i)
Next i
```

Using a COM Proxy

A COM proxy includes the COM runtime dlls.

The generated proxy is capable of supporting both synchronous and asynchronous cooperative flows. The main difference between these processing types is the actual method used to perform the cooperative flow.

All sample code supplied with the product (VB, ASP and XML) only demonstrates the use of synchronous capabilities. However, the following sections provide, in pseudo-code, an outline of the structure of synchronous and asynchronous code.

Note: All non-CA Gen server errors are handled as standard COM exceptions. Handling of these exceptions is not shown in the following sections.

More information:

[Asynchronous Processing](#) (see page 128)

[Preparing for Execution](#) (see page 130)

Synchronous Processing

The following is the code for a synchronous process:

```
...
op=CoCreateInstance(. . .)
...
<set up import views and communications/system attribute information>
...
op.Execute() [Perform synchronous flow]
...
<check results for errors>
...
<retrieve export view data>
...
```

Note: If required, you can substitute the ExecuteXML method for the synchronous Execute method.

Asynchronous Processing

The code is broken up into four distinct sections, as you can embed each piece at different locations within the user application.

Send Data

Use the following code to send data:

```
...
op=CoCreateInstance(...)
...
<setup import views and communication/system attribute information>
...
op.ExecuteAsync(&id,False) //Perform asynchronous send, with a response expected
...
```

Note: If required, you can substitute the ExecuteAsyncXML method for the asynchronous ExecuteAsync method.

Check Status

Use the following code to check status:

```
...
op.ExecuteCheckResponse(id,&status) //Perform check, result goes in status
switch (status)
{
...
    case XYZ available
    ...
    case XYZ Pending:
    ...
    default:
    ...
}
```

Retrieve Results

Use the following code to retrieve results:

```
...
op.ExecuteGetResponse(id,True,&processed) //Perform the retrieval of the results,
    with blocking turned on
...
<check results for errors>
...
<retrieve export view data>
...
```

Note: If required, you can substitute the ExecuteGetResponseXML method can be substituted for the asynchronous ExecuteGetResponse method.

Ignore Request

Use the following code to ignore request:

```
...
op.ExecuteIgnoreResponse(id) //Processes the ignore request on the request indicated
    by the id
...
```

Security Processing

A COM Proxy provides facilities to implement Distributed Processing Security as described in the *Distributed Processing - Overview Guide*.

By default, the COM Proxy does not exploit the use of the Proxy Runtime security features. To utilize the security features, the application developer must add code to their COM application to set the ClientID and ClientPassword properties of the COM Proxy's Import view object. Depending on the return value of the client security user exit, the security data fields will or will not be sent to the target DPS. Additionally, if the client security user exit returns SECURITY_ENHANCED, the client security user exits can cause an optional security token to be added to the data flow. The collection of security data that is sent as part of the cooperative flow is validated by user exits residing in the execution environment of the target DPS.

For a COM Proxy flowing to a DPS using TCP/IP, MQSeries or ECI the supporting runtime lets a portion of the Common Format Buffer (CFB) to be encrypted on the way to the target DPS, and decrypted on the way back from the target DPS. The use of encryption and decryption are enabled by way of user exits.

More information:

[User Exits](#) (see page 135)

Preparing for Execution

Before you execute an application that uses a generated COM proxy, address the following items:

- You may need to modify the COM Proxy Runtime for use in the user's application execution environment.
- You must deploy the proxy and Runtime to the applications execution environment

Trace Log and Configuration File Locations

By design, CA Gen COM Proxies operate in several configurations. They can be consumed by:

- a standalone COM aware application through a call to Colnitalize
- an IIS web application with the applicable COM Proxy DLL having been registered with the system registry through a call to "regsvr32"
- as a Component Services application where the COM Proxy DLL is registered during the Component Services management process

The name of the consuming application and the user ID that is executing the consuming application will vary depending upon how a given COM Proxy is configured and under which component the COM Proxy is executed.

As COM Proxy DLLs are COM objects, the typical configuration approach is to register the DLLs into the system registry. This can be done either by calling regsvr32 directly or by configuring the COM Proxy using Windows Component Services. Windows Component Services offer a myriad of configuration options not available to regsvr32. The trace log file name and its location are determined by the way a Proxy is configured and executed.

Log File Name

Once registered the COM Proxies are consumed either directly by an application or through a call to a Web application using IIS. It is the combination of how it is registered and how it is consumed that determines the log file name:

- If the COM Proxy is consumed by a standalone COM aware application through a call to CoInitialize, the log file will be named:

trace-<appname>-<procid>.out

appname

Indicates the name of the consuming application.

procid

Indicates the process ID corresponding to that application.

- If the COM Proxy is consumed by an IIS web application that consumes the COM Proxy that was registered through regsvr32, the log file will be named:

trace-w3wp-<procid>.out

w3wp

Specifies the name of the IIS application host process which is IIS6 and later.

procid

Indicates is the process ID corresponding to that application.

- If the COM Proxy is consumed by a standalone or IIS web application accessing a COM Proxy that has been configured using Windows Component Services, the log file will be named:

trace-dllhost-<procid>.out

dllhost

Specifies the name of the Component Services application host process.

procid

Indicates the process ID corresponding to that application.

Configuration File Name

Configuration files are not named dynamically and so the files are required to be named as documented in the product documentation. Configuration files are located in a product area subdirectory which itself is below the above top level "cfg" directory.

File Location

The location of the configuration and log files depends on which user ID the IIS web site and/or Component Services object are configured to use.

Subdirectories

In all cases, subdirectories will be the same, the top level directory will vary depending on the configured user ID.

Note: *xx* refers to the current release of CA Gen. For the current release number, see the *Release Notes*.

The following paths show the common subdirectories:

Log files: %USERPROFILE%\AppData\Local\CA\Gen *xx*\logs\client

Configuration Files: %USERPROFILE%\AppData\Local\CA\Gen *xx*\cfg\client

The following table lists some of the user ID's and top level directory locations used when COM Proxies are configured to run under IIS and/or Windows Component Services.

Consuming User ID	Directory
Anonymous	C:\Windows\ServiceProfiles\NetworkService\AppData\Local\CA\Gen <i>xx</i>
Network Service	C:\Windows\ServiceProfiles\NetworkService\AppData\Local\CA\Gen <i>xx</i>
Local Service	C:\Windows\ServiceProfiles\LocalService\AppData\Local\CA\Gen <i>xx</i>
Local System	C:\Windows\System32\config\systemprofile\AppData\Local\CA\Gen <i>xx</i>
General User (John Doe)	C:\Users\JohnDoe\AppData\Local\CA\Gen <i>xx</i>

Top Level Directories

The following paths show some of the top level directory locations when consuming the COM Proxy as a standalone application.

Note: *xx* refers to the current release of CA Gen. For the current release number, see the *Release Notes*.

Proxy DLLs registered through Component Services

%ALLUSERSPROFILE%

Proxy DLLs registered through regsvr32 by user, example johndoe

%USERPROFILE% which in this case expands to C:\Users\johndoe

Examples:

C:\Windows\ServiceProfiles\NetworkService\AppData\Local\CA\Gen *xx* \logs\client

C:\Windows\ServiceProfiles\NetworkService\AppData\Local\CA\Gen *xx* \cfg\client

C:\Windows\ServiceProfiles\LocalService\AppData\Local\CA\Gen *xx* \logs\client

C:\Windows\ServiceProfiles\LocalService\AppData\Local\CA\Gen *xx* \cfg\client

The USERPROFILE environment variable can also be used. It must correspond to the effective user ID in force at the time of COM Proxy execution.

%USERPROFILE%\AppData\Local\CA\Gen *xx* \logs\client

%USERPROFILE%\AppData\Local\CA\Gen *xx* \cfg\client

Alternate File Locations

There may be cases when a COM Proxy is being executed by a host process which then impersonates a lower level authority user ID when executing the COM Proxy logic. In this case, the effective user ID may not possess the authority to write the trace log file into the desired default directory. In this case, the trace logic will attempt to create the log file using the following root directory hierarchy. In all cases, the lower level subdirectories will be as described previously.

1. The root directory pointed to by the %USERPROFILE% environment variable. This is the default. If this fails, see the next step.

2. The root directory pointed to by the %ALLUSERSPROFILE%. If this fails, see the next step.
3. The root directory pointed to by the users %TEMP% environment variable.

Effective user IDs such as Network Service, Local Service, and Local System will have %TEMP%, which may be set to C:\Windows\TEMP.

Configuring the COM Proxy Runtime

By design, there exist areas within the COM Proxy Runtime that can be influenced by an application developer. An application developer does many activities that include updating the commcfg.ini file to influence the communications part of the runtime and modifying user exits. The user exits influence security processing and lets the arguments specific to the selected communications processing to be overridden.

Configuring COM Proxy Communications

The COM Proxy runtime can use the commcfg.ini file to find the CA Gen servers it needs to access. This is an ASCII text file that allows runtime modifications to the communication configuration for a given set of transactions codes associated with the target DPS applications. The commcfg.ini must be available in directories specified by PATH or in a directory specified by the COMMCFG_HOME environment variable. A default version of this file is available in the installed CA Gen directory.

There is only one copy of the commcfg.ini file, by default. All proxies must use this file. Alternatively, a proxy application searches for the commcfg.ini file in the following order:

1. %COMMCFG_HOME%
2. %USERPROFILE%\AppData\Local\CA\Gen xx\cfg\client
3. %ALLUSERSPROFILE%\Gen xx\cfg\client
4. %PATH%

If the file is not available in these directories, the proxy application assumes that the file does not exist.

If there are multiple copies of the commcfg.ini file, the order of precedence for the files is the same as the order described above.

The following list defines the use of this search order:

%COMMCFG_HOME%

The environment variable COMMCFG_HOME can be set to a directory that contains the commcfg.ini file. This file is used by the invoking proxy application. This variables setting enables customization on a proxy by proxy basis.

%USERPROFILE%\AppData\Local\CA\Gen xx\cfg\client

If the commcfg.ini file is stored in this directory, the file is accessible by all proxies that are executed by a *specific* user on the system.

%ALLUSERSPROFILE%\Gen xx\cfg\client

If the commcfg.ini file is stored in this directory, the file is accessible by all proxies that are executed by *all* users on the system.

%PATH%

If the commcfg.ini file is not available in the above locations, the file is searched in all the directories that are specified by the PATH environment variable. By default, CA Gen is part of the PATH environment variable.

A generated proxy contains communications information as it was defined in the model when the proxy was generated. The proxy runtime provides capability to specify the communication information at runtime. This information overrides any information specified in the model and was built into the proxy during generation.

Additionally, the proxy runtime is capable of turning the tracing ON and OFF, as well as provides capability to control the amount of file caching that takes place at runtime.

Note: For specific formatting instructions, see the default file installed with the CA Gen software, or see the *Distributed Processing - Overview Guide*.

User Exits

The user exits that are invoked at runtime for a COM Proxy are common to other C/C++ execution environments. For more information about each user exit, see the *User Exit Reference Guide*.

There are sets of user exits that are invoked from within the COM Proxy runtime regardless of the selected communication type. There are other user exits that are unique for each communications type.

The following user exit entry points are common to all COM Proxies regardless of the communications type:

User Exit Entry Point	Purpose
WRSECTOKEN	The Client Security user exit is used to direct the runtime to incorporate the ClientUserId and ClientPassword attributes into the CFB. If it returns SECURITY_ENHANCED, it can cause an optional Security Token field to be added to the CFB.

User Exit Entry Point	Purpose
WRSECENCRYPT	The import message encryption user exit provides the opportunity for a portion of the outbound CFB to be encrypted using an encryption algorithm implemented within this user exit. The target DPS's execution environment must implement a companion decryption user exit for the CFB to be interpreted as a valid request CFB.
WRSECDECRYPT	The export message decryption user exit provides the opportunity for decrypting the portion of the CFB that has been encrypted by the target DPS's execution environment. This exit must implement the companion to the DPS's encryption user exit in order for the DPS's response buffer to be interpreted as a valid response CFB.

The following user exit entry points are invoked by COM Proxies flowing to their target DPS using TCP/IP as their communications type:

User Exit Entry Point	Purpose
CI_TCP_DPC_DirServ_Exit	Overrides the destination information used when creating the TCP/IP Socket.
CI_TCP_DPC_setupComm_Complete	Directs the runtime to retry a cooperative flow request that failed during setup.
CI_TCP_DPC_handleComm_Complete	Directs the runtime to retry a cooperative flow request that failed during the non-setup related processing of a cooperative flow.

The following user exit entry points are invoked by COM Proxies flowing to their target DPS using ECI as their communications type:

User Exit Entry Point	Purpose
Ci_eci_get_system_name	Provides the CICS system name, if one has not previously been provided by data obtained from the commcfg.ini file.
n.a	n.a

The following user exit entry points are invoked by COM Proxies flowing to their target DPS using MQSeries as their communications type:

User Exit Entry Point	Purpose
CI_MQS_DPC_Exit	Overrides various pieces of information used to interface with MQSeries.
CI_MQS_DyanmicQName_Exit	Overrides the default structure of Dynamic Reply to Queues.
CI_MQS_DPC_setupComm_Complete	Directs the runtime to retry a cooperative flow request that failed during setup.
CI_MQS_MQShutdownTest	Overrides the default behavior of keeping the connection to the Queue Manager following the completion of a cooperative flow.
CI_MQS_DPC_handleComm_Complete	Directs the runtime to retry a cooperative flow request that failed during the non-setup related processing of a cooperative flow.

The following user exit entry points are invoked by COM Proxies flowing to their target DPS using Tuxedo as their communications type:

User Exit Entry Point	Purpose
ci_c_sec_set	Sets user supplied security data into the security data fields located in Tuxedo's TPINIT structure.
ci_c_user_data_out	Gives the user the opportunity to inspect or modify the cooperative flow request buffer prior to Tuxedo sending the request to the target Tuxedo service.
Ci_c_user_data_in	Gives the user the opportunity to inspect or modify the cooperative flow request buffer on return from the target Tuxedo service. Invoked on return from the TPCALL. Additionally, this exit lets the client to disconnect from the server following each flow.

The following user exit entry points are invoked by COM Proxies flowing to their target EJB DPS using EJBRMI as their communications type:

User Exit Entry Point	Purpose
n.a.	n.a.
n.a	n.a

The following user exit entry points are invoked by COM Proxies flowing to their target .NET Serviced Components DPS using NET as their communications type:

User Exit Entry Point	Purpose
n.a.	n.a.
n.a	n.a

The following user exit entry points are invoked by COM Proxies flowing to their target DPS using Web Services as their communications type:

User Exit Entry Point	Purpose
CI_WS_DPC_Exit	Gives the user an opportunity to modify the Web Service endpoint destination by overriding the base URL and the context type.
CI_WS_DPC_URL_Exit	Gives the user an opportunity to modify the Web Service endpoint destination URL.

Deploying the COM Proxy

Each \deploy\<component> directory contains two DLLs for each method defined in the server. One is the ActiveX automation DLL, which is named <component>ax.dll, and the other is the proxy itself, named <component>cm.dll.

Registering the Proxy DLLs

The DLL files AX.DLL and CM.DLL must be copied to the deployment machine into a directory in the PATH.

You must register both DLLs regardless of whether you want to access the proxy from Visual Basic, an Active Server Page, or a standard COM application. Follow this procedure:

- Register both DLLs using regsvr32. For example:

```
regsvr32 <component>cm.dll  
regsvr32 <component>ax.dll
```

- Unregister by using the following syntax:

```
regsvr32 /u <component>cm.dll  
regsvr32 /u <component>ax.dll
```

Note: You must register the DLLs on each machine that will run the proxy. If you are using ASP, then you only need to register the DLLs on the Web Server. If you are registering 32-bit DLLs on 64-bit Windows, use the following command.

```
\Windows\syswow64\regsvr32.exe
```

Supporting Files in COM Proxy

The CA Gen runtimes must also be placed on the deployment machine, in the same directory as the generated proxy DLLs. The following list identifies the required runtime files.

AX DLL

The generated AX DLL has runtime dependencies on a number of Microsoft and CA Gen runtime dlls..

As the number of dependent Microsoft dlls may change over time a complete list cannot be provided. The Microsoft Visual Studio supplied “dumpbin /dependents” command can be used to list the immediate dll dependencies. A more complete list will require the use of a third party dependency walking tool.

In addition, the AX DLL may require the CA Gen supplied runtime file Xerces-C_#_#_#.DLL and the following files, if the XML API was selected for generation:

CM DLL

The generated CM DLL requires a number of Microsoft runtime dlls as well, but also requires these additional CA Gen runtime files:

- PREXxxN.DLL
- REQTOkxxN.DLL
- CSUMGxxN.DLL
- CSUxxN.DLL
- CPRTxxN.DLL
- IEFDPRTxxN.DLL
- IEFMBT.DLL
- codepage.ini

- commcfg.ini
- CSUVNxxN.DLL
- VVRTxxN.DLL

Note: Depending on the communication runtime selected, additional files will be required. See the documentation for each communication type to get the list of required files.

Note: xx refers to the current release of CA Gen. For the current release number, see the *Release Notes*.

Executing the VB and ASP Sample Code

The VB generated code has sections for:

- Setting import property values to default values
- Executing the operation and examining its status
- Retrieving export values

To use the generated code fragment in a VB program, make sure to reference the <component>1.0 Type Library in the VB project or the VB code fragment does not compile properly. If the <component>1.0 Type Library does not appear in the VB project reference dialog, make sure that the <component>ax.dll and the <component>cm.dll DLLs are properly registered. The generated VB code fragment file has a name based upon the procedure step name with a .bas extension.

Of the two generated client codes, the ASP client code is the most useful. This generated code contains an HTML form with HTML fields for each import attribute (all fields are TEXT fields except for text attributes with permitted values which are SELECT fields) and a SUBMIT button containing the label Run Procedure Step for each procedure step of a component.

When you enter data into the form and click the Run button, the data is posted to the web server using an HTTP POST. At the server, the generated ASP file calls the COM Proxy by first initializing import properties to values passed in the HTTP POST and then calling the COM Proxy Execute method.

If an error occurs with the Execute method call, the generated ASP file displays the HTTP formatted error message. Otherwise, it displays the operation's exports in a similar way to the import's form except for the fact that the export's form fields are non-editable. There are several ASP files created for a component. These files are listed as follows:

- The <component>.asp is generated for each procedure step of a component.

- The generated files default.htm, blank.htm and operations.htm. They provide two frames (default.htm) where the left frame contains links to all procedure steps of a component (operations.htm) and the right frame (blank.htm) displays the operation ASP files when you select an operation link in the left frame.

Note: With IIS, to share the generated ASP files directory, right click the directory in Explorer and choose properties, then select the Internet tab and click Add. Type a virtual directory alias, select read and execute access (ASP files need execute access to work properly), and click OK twice.

Neither the VB nor ASP generated client code files make any reference to COM Proxy common properties ClientId, ClientPassword, CommandSent, ExitStateSent, Location, CommandReturned, ExitStateReturned, and ComCfg. If required, you can modify the generated code to refer these properties.

Calling the COM Proxy from a Web Server

The installation and generation process creates everything you need to invoke the COM Proxy from a web server. The \asp directory contains <method-name>.asp active server page files, which you can invoke directly from the Web Browser, and an operations.htm file, which contains links to the associated .asp files.

To view the .asp files, you must create an alias to these files, using the IIS Internet Service Manager, so that you can copy these files (for each component) into a single directory. You should enable this directory for Read and Execute access.

Calling the COM Proxy from Visual Basic

When you build a COM Proxy, the generation process creates a Visual Basic template with all the code required to invoke the proxy (through an ActiveX automation interface) and to send and receive data. This template is in file <model-name>\proxy\com\deploy\<component>\vb\<operation>.bas

To call this file from Visual Basic, you must add code to let you input the required data and display the outputs.

Follow these steps:

1. To insert the generated module into your Visual Basic project, choose Project from the Visual Basic menu, and click Add Module.
2. To use this code, create an event that invokes the inserted subroutine (for example, a button click event).
3. Make changes to assign values for required import views, or comment out unused import view assignment statements.

4. Change the reading of the export views so that the information is not assigned to temporary variables.
5. Ensure that you add project references to the COM Proxy. To do this, complete the following steps:
 - a. Select Project from the Visual Studio menu.
 - b. Click Add Reference.
The Add Reference dialog appears.
 - c. Select COM on the Add Reference dialog.
 - d. Select the AX Proxy, listed as <component> 1.0 Type Library, in the Component Names drop down.
 - e. Verify if the AX file location is accurate in the Path field.
 - f. Click OK.

The reference is added to the project.

6. If you are using Visual Basic .NET and, add the following statement to the start of the file that contains the generated module:

```
imports <componet>
```

This statement allows you access to the component added in step 5.

Note: The code in the template is commented to help you in adding your own procedures. Pay special attention to the DECLARATIONS section near the top of the file. It contains instructions for adding the ActiveX automation DLL reference to the VB project. You may also need to comment out certain lines in the code (for example, the section on INPUT VALUES).

Calling a COM Proxy from the XML Test Application (Optional)

If the optional XML programming interface was selected, the generation and installation process creates everything you need to invoke the COM Proxy from the sample XML test application.

For each method, the \xml directory contains an XSD (XML Schema Definition) file, a sample Import XML file, and a compiled Test executable.

To execute the test application, you must register the proxy DLLs, change to the \xml directory, and then execute the following command:

```
Test<method-name>.exe <method-name>Sample.xml
```

If required, the input data for the server can be modified by editing the sample XML file.

COM Proxy Server Testing

This section describes the COM Proxy server testing philosophy and the requirements to use the Diagram Trace Utility.

Use Diagram Trace Utility

Use the Diagram Trace Utility to test generated Proxy server applications before moving them to a production environment.

It steps through the application as it executes, allowing you to view the CA Gen model elements used to build the application, such as action diagram statements, as the generated program executes. To view the model elements, you must make certain selections that generate additional code when generating remote files.

When testing an application, the procedure steps and action blocks generated with trace communicate with the Diagram Trace Utility.

Before testing an application, build certain application components with the Build Tool:

- The application database
- RI trigger logic
- Operations libraries
- All load modules

Follow these steps:

1. Build the test Proxy server application generated with trace.
2. Invoke the Diagram Trace Utility on any Windows system, including the same system by clicking Start, All Programs, CA, Gen xx, Diagram Trace Utility.

Note: xx refers to the current release of CA Gen. For the current release number, see the *Release Notes*.

The Diagram Trace Utility is invoked and starts listening on port 4567.

Note: Change the default port in the Diagram Trace Utility, if necessary.

3. Set the trace environment variables in the Proxy server's application.ini file.

Note: For more information about using the Diagram Trace Utility, see the *Diagram Trace Utility User Guide*.

Multi-User Diagram Trace Utility Support

When a remote Proxy server application has been started, one copy of the application.ini file is available for that application. Setting the trace environment variables in the application.ini file only allows one Diagram Trace Utility the ability to trace the servers in the server application.

In a test environment with multiple testers working with the same server application, testers need to debug from multiple client workstations. To do so, the environment needs multiple Diagram Trace Utilities.

Override the Default Trace Environment Variables

A client transaction can transfer the host and port of the client that is executing the transaction. By providing this additional information, the server can establish communication with the Diagram Trace Utility running on that client workstation.

Follow these steps:

1. Build the Proxy server application that has been generated with trace.
2. Leave the trace environment variables within the application.ini file that is located in the server application's model directory commented out.
3. Edit the commcfg.ini file in order to communicate with your remote Proxy servers.
4. Invoke the Diagram Trace Utility on your client workstation.
5. Set the trace environment variables in the command window (for Windows) or the terminal session (for UNIX) in which you will be invoking your Proxy client:

- For Windows:

- set TRACE_ENABLE=1
- set TRACE_HOST=*yourpghost*
- set TRACE_PORT=4567

- For UNIX:

- setenv TRACE_ENABLE 1
- setenv TRACE_HOST *yourpghost*
- setenv TRACE_PORT 4567

These environment variables are picked up by the Proxy client and sent to the Proxy server.

6. Execute your Proxy client application.

The Diagram Trace Utility on the client workstation will be used to trace the Proxy server for each transaction initiated from that client.

Regenerating Remote Files After Testing

The code generated on the CA Gen workstation for testing with trace includes features, such as trace calls, that are only needed during testing. These features increase the size of the Proxy servers' load modules significantly, and impact the application's performance. Even if no changes are required as a result of the tests performed, you must regenerate the load modules without trace before it is placed into production.

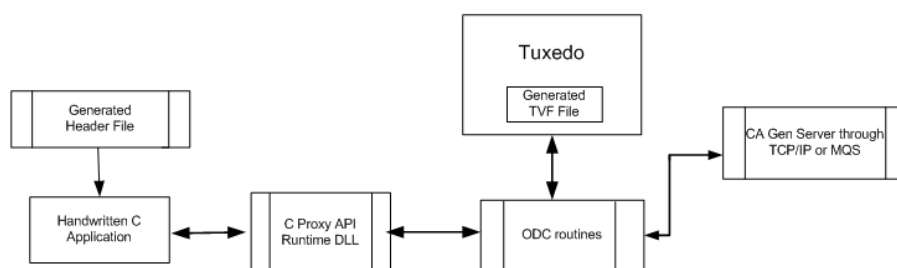
Chapter 7: C Proxy

The C Proxy API is a generated C language header file that provides you the flexibility to call CA Gen Distributed Processing Server (DPS) applications from any C language application or any application that supports a C language interface.

The CA Gen Proxy Generator creates a C language header file that the application programmer uses to define the import and export views passed to and from a generated DPS. In addition, the CA Gen Proxy Generator creates a TVF (Tuxedo View File) for use when Tuxedo is used as the communication type.

This chapter describes how to use the generated header file (and TVF file), a description of the C Proxy API calls, and an overview of how to build a user-written application that includes the generated header and API calls.

The following illustration shows how the C proxy is used:



The CA Gen product installation includes a sample CA Gen model as well as a sample user-written application, which utilizes the C Proxy API. The example user-written code and the associated makefile can be found in the sample subdirectory of the CA Gen product installation directory. The supplied Sample model can be found in the product installation sample.ief subdirectory. The example user-written application is designed to communicate with the Sample model P900 server load module.

Together the user-written application and the generated P900 server load module comprise a complete and usable application demonstrating the use of the C Proxy API.

Setting Up the C Proxy API Environment

To compile and link your C programs, you need to access to the C Proxy API files. You can include your CA Gen path within your makefile, development environment, or environment variables.

Setting Up Windows Platforms

To set up your environment, do the following:

- Add the CA Gen bin directory (for example, %GENxx%Gen) to your PATH environment variable.
- Add the CA Gen include directory (for example, %GENxx%Gen) to your INCLUDE environment variable.
- Add the CA Gen lib directory (for example, %GENxx%Gen) to your LIB environment variable.

Note:

- If working with Visual Studio 32-bit or 64-bit, the CA Gen C Proxy dll and lib directories added to PATH and LIB will use %GENxx%Gen\VSabc or %GENxx%Gen\VSabc\amd64. VSabc refers to the supported version of Visual Studio. Replace VSabc with VS100 for Visual Studio 2010 and VS110 for Visual Studio 2012. xx refers to the current release of CA Gen. For the current release number, see the *Release Notes*.

Setting Up UNIX Systems

To set up your environment, do the following:

- Add the location of the C Proxy directory to your PATH and library path (LD_LIBRARY_PATH, SHLIB_PATH or LIBPATH) environment variables.
- Add the location of the CA Gen proxy runtime support files to the following environment variables:
 - Add the CA Gen bin directory to the PATH environment variable (PATH=\$PATH:\$IEFH/bin)
 - Add the CA Gen lib directory to one of the following library paths depending on the type of UNIX system:
 - HPUX-SHLIB_PATH (SHLIB_PATH=\$SHLIB_PATH:\$IEFH/lib)
 - AIX-LIBPATH (LIBPATH=\$LIBPATH:\$IEFH/lib)
 - Solaris-LD_LIBRARY_PATH (LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:\$IEFH/lib)

Visual Studio Support

CA Gen supports compiling generated C proxy applications on Windows using Visual Studio. Users do not need to regenerate their code to use a different version of the compiler. The %GENxx%Gen\VSabc folder contains a collection of files that have been rebuilt to support Visual Studio. Add %GENxx%Gen\VSabc to PATH when working with C proxy applications.

Note: VSabc refers to the supported version of Visual Studio. Replace VSabc with VS100 for Visual Studio 2010 and VS110 for Visual Studio 2012. xx refers to the current release of CA Gen. For the current release number, see the *Release Notes*.

64-bit Windows Support

CA Gen supports compiling and executing generated C proxy applications as 64-bit images on Windows using Visual Studio. Users will need to regenerate their code to gain access to 64-bit data types used in the Windows X 64 API to create 64-bit images. The %GENxx%Gen\VSabc\amd64 folder contains a collection of files that have been rebuilt for 64-bit support.

If you choose to use the 64-bit runtimes provided with CA Gen to execute your application, you must modify PATH to append %GENxx%Gen\VSabc\amd64 before %GENxx%Gen\VSabc.

Note:

- Prepending %GENxx%Gen\VSabc\amd64 to PATH should only be done in the current command window session, and should not be set in the System Environment variables.
- VSabc refers to the supported version of Visual Studio. Replace VSabc with VS100 for Visual Studio 2010 and VS110 for Visual Studio 2012. xx refers to the current release of CA Gen. For the current release number, see the *Release Notes*.

Calling the C Proxy API from a C Program

To communicate to a CA Gen server, you must create a C program that includes the header file and has code to let you input the required data and display the output.

Note: For information on functions that you must call, see [Proxy Function Prototypes](#) (see page 171). This section contains an example generated header file and an example C program that uses the header file and several of the Proxy functions. This is a simplified example. A more thorough coding example of a C proxy can be found in a subdirectory of the CA Gen install directory. The \sample\CProxy subdirectory contains example proxy application files cproxy.c and cproxy.mak.

Variable Name Case Sensitivity

The C Proxy API uses case sensitivity to create multiple variable names from the procedure step name. The procedure step name is used since it is easy to recognize, is unique to the model, and is not longer than 32 characters (the maximum length supported by some compilers). The disadvantage of using case sensitivity is that it can be confusing when coding.

For example:

- `SERVER_MAINTAIN_DIVISION` (upper case) is used to represent the TranEntry data structure that contains all the information required for a transaction (such as trancode and model name).
- `server_maintain_division` (lower case) is used to represent the transaction specific view data structure that is used to reference import and export views.

If you are using the sample C program as a base for your program, take care to match the case when replacing variable names.

Overview of the Generated Header File

When a C Proxy API is generated, the generation process creates a C header file with all the structures required to invoke the proxy. This header file is:

```
<model-name>\proxy\c\<component>.h
```

This header file is recreated each time you generate the C Proxy API. You can copy the header file to your development area so that you have a stable copy.

Example Generated Header File

Following is an overview of the generated file. Information that is used in the associated C file is shown in **bold** and information specific to the communication type is shown in *italicized bold*. This information has been extracted from proxy generation using the current sample model which is distributed with the CA Gen product. The generated file is not reproduced in its entirety. Important aspects of the file content have been reproduced for discussion purposes.

Description	Generated File
Includes and compiler directives. Some files are included based on the communications method selected during generation. If you change communication types, you will need to regenerate the C Proxy API.	<pre> #ifndef INC_COMPONENT_P900 #define INC_COMPONENT_P900 #include <citranb.h> #include <proxyexe.h> #include <proxytrc.h> #include <proxycfg.h> #include <proxysit.h> #include <tirdp_struct.h> #ifdef __cplusplus extern "C" { #endif /* __cplusplus */ #include <cisrvtcp.h> </pre>
<p>Transaction specific structure that is used to reference the import and export views. In this example, the data in the structure can be referenced in your program as follows:</p> <p>server_detail_division.importRequest_command.value</p> <p>server_detail_division.exportRequest_command.value</p>	<pre> struct s_ServerDetailDivision { /* Entity View: IMPORT_REQUEST */ /* Type: COMMAND */ struct { char value[2]; } importRequest_command; /* Entity View: EXPORT_REQUEST */ /* Type: COMMAND */ struct { char value[21]; } exportRequest_command; }; static struct s_ServerDetailDivision server_detail_division; </pre>

Description	Generated File
Import and Export View tables as used by the CA Gen server. Do not modify these tables since the server is expecting this structure.	<pre>static CViewDefVal import_view_0001835029[] = { {DT_STRING, 1, 1, 2, 0, server_detail_division.importRequest_com mand.value}, ... /* other data definitions */ {DT_END, 0, 0, 0, 0, 0}}; static CViewDefVal export_view_0001835029[] = { {DT_STRING, 1, 1, 2, 0, server_detail_division.exportRequest_com mand.value}, ... /* other data definitions */ {DT_END,0,0,0,0,0}};</pre>
Service Table structure - contains a structure and information specific for the communications type selected during generation. If you change communications types you need to regenerate the C Proxy API.	<pre>static CITCPService comm_service_0001835029 = { "TCP", "", "" };</pre>

Description	Generated File
TranEntry data structure - the type is defined in citrantb.h. Each bulleted item below corresponds to a line item on the right. This structure contains the information needed for a transaction:	static CITranEntry
programid - load module name	SERVER_DETAIL_DIVISION = {
programidalt - an alternate form of the program id (e.g. load module name)	"P900",
trancode - transaction code	"P900",
procName - name of the procedure to be called or advertised	"P900",
procNamealt - an alternate form of the procName	"SERVER_DETAIL_DIVISION",
procSourceName - the source name of the procedure to be called or advertised	"ServerDetailDivision",
modelName - name of the model advertised or used	"SERVERD1",
modelShortName - the short name of the model advertised or used	"GEN SAMPLE MODEL",
importView - pointer to description of import view (defined above)	"sample",
exportView - pointer to description of export view (defined above)	import_view 0001835029,
returnOnExitStates - possible exit states that the server can return on	export_view 0001835029,
returnOnExitCmds - commands associated with returnOnExitStates	""
netApplicationName - the .NET application name	""
netNamespace - the .NET namespace	"com.ca",
netAssemblyVersion - the .NET assembly version	"cfbmo",
javaContext - the Java URL context (not currently used)	"tcpf",
javaPackage - the package name for java	(CIService *)&comm_service_0001835029};
Ending compiler directives.	*/#ifdef __cplusplus } /* extern "C" */ #endif /* __cplusplus */ #endif /* INC_COMPONENT_P900 */ /* End of p900.h

Description	Generated File
msgObjLib - core name of message object shared library such as "cfbmo." If not defined here, it is set by ProxyConfigureComm	
commMethodLib - core name of communication method shared library such as "tcpf." If not defined here, will be set by ProxyConfigureComm	
service - pointer to Service Table structure (defined above) that is a description of communication method services used or advertised	
In this example, the data in the structure is referenced in your program as follows: SERVER_DETAIL_DIVISION.service or SERVER_DETAIL_DIVISION.modelName to change the model name	

Communication-specific Details

The following tables show communication-specific information in the generated header file:

TCP/IP	ECI
#include <cisrvtcp.h>	#include <cisrveci.h>
static CITCPService comm_service_0001835029 = { "TCP", "" , "" };	static CIECIService comm_service_0001835029 = { "ECI"; };

TCP/IP	ECI
<pre>static CTranEntry SERVER_DETAIL_DIVISION = { "P900", "P900", "P900", "SERVER_DETAIL_DIVISION", "ServerDetailDivision", "SERVERD1", "GEN SAMPLE MODEL", "sample", import_view_0001835029, export_view_0001835029, "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "sample", "sample", "", "", "com.ca", "cfbmo", "tcpcf", (CIService *)&comm_service_0001835029};</pre>	<pre>static CTranEntry SERVER_DETAIL_DIVISION = { "P900", "P900", "P900", "SERVER_DETAIL_DIVISION", "ServerDetailDivision", "SERVERD1", "GEN SAMPLE MODEL", "sample", import_view_0001835029, export_view_0001835029, "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "sample", "sample", "", "", "com.ca", "cfbmo", "ecicf", (CIService *)&comm_service_0001835029};</pre>

MQSeries (MQS)	MQSeries (MQI-client)
<pre>#include <cisrvmqs.h> static CIMQSService comm_service_0001835029 = { "MQS", "", "P900", "SYSTEM.DEFAULT.MODEL.QUEUE" };</pre>	<pre>#include <cisrvmqs.h> static CIMQSService comm_service_0001835029 = { "MQS", "", "P900", "SYSTEM.DEFAULT.MODEL.QUEUE" };</pre>

MQSeries (MQS)	MQSeries (MQI-client)
<pre>static CTranEntry SERVER_DETAIL_DIVISION = { "P900", "P900", "P900", "SERVER_DETAIL_DIVISION", "ServerDetailDivision", "SERVERD1", "GEN SAMPLE MODEL", "sample", import_view_0001835029, export_view_0001835029, "" "" "" "" "" "" "" "", "" "" "" "" "" "" "" "", "" "" "" "" "" "" "" "", "sample", "sample", "", "", "com.ca", "cfbmo", "mqscf", (CIService *)&comm_service_0001835029 };</pre>	<pre>static CTranEntry SERVER_DETAIL_DIVISION = { "P900", "P900", "P900", "SERVER_DETAIL_DIVISION", "ServerDetailDivision", "SERVERD1", "GEN SAMPLE MODEL", "sample", import_view_0001835029, export_view_0001835029, "" "" "" "" "" "" "" "", "" "" "" "" "" "" "" "", "" "" "" "" "" "" "" "", "sample", "sample", "", "", "com.ca", "cfbmo", "mqicf", (CIService *)&comm_service_0001835029 };</pre>
Java RMI	Tuxedo
#include <cisrvjvm.h>	#include <cisrvtux.h>
	#include <PROXY1BA.h>

Java RMI	Tuxedo
<pre>static CIJVMService comm_service_0001835029 = { "JVM", "RMI", "" };</pre>	<pre>static CITuxService comm_service_0001835029 = { "TUX", "SERVERD1", "SERVERD1_IM", sizeof(struct SERVERD1_IM), "SERVERD1_EX", sizeof(struct SERVERD1_EX), "SERVERD1_TV", sizeof(struct SERVERD1_TV), TPNOTRAN};</pre>
<pre>static CITranEntry SERVER_DETAIL_DIVISION = { "P900", "P900", "P900", "SERVER_DETAIL_DIVISION", "ServerDetailDivision", "SERVERD1", "GEN SAMPLE MODEL", "sample", import_view_0001835029, export_view_0001835029, "", "", "sample", "sample", "", "", "com.ca", "jvmmo", "jvmcf", (CIService *)&comm_service_0001835029};</pre>	<pre>static CITranEntry SERVER_DETAIL_DIVISION = { "P900", "P900", "P900", "SERVER_DETAIL_DIVISION", "ServerDetailDivision", "SERVERD1", "GEN SAMPLE MODEL", "sample", import_view_0001835029, export_view_0001835029, "", "", "sample", "sample", "", "", "com.ca", #ifdef WS_CLIENT "txmo", "txwcf", #else "txmo", "txcf", #endif (CIService *)&comm_service_0001835029};</pre>

.NET Remoting

```
#include <cisrvnet.h>
```

```
static CINETService
```

```
comm_service_0001835029 = {
```

```
  "NET",
```

```
  "localhost",
```

```
  "80",
```

```
  "B",
```

```
};
```

```
static CITranEntry
```

```
SERVER_DETAIL_DIVISION = {
```

```
  "P900",
```

```
  "P900",
```

```
  "P900",
```

```
  "SERVER_DETAIL_DIVISION",
```

```
  "ServerDetailDivision",
```

```
  "SERVERD1",
```

```
  "GEN SAMPLE MODEL",
```

```
  "sample",
```

```
  import_view_0001835029,
```

```
  export_view_0001835029,
```

```
  "" , "" , "" , "" , "" , "" , "" , "" ,
```

```
  "" , "" , "" , "" , "" , "" , "" , "" ,
```

```
  "" , "" , "" , "" , "" , "" , "" , "" ,
```

```
  "sample",
```

```
  "sample",
```

```
  "" ,
```

```
  "" ,
```

```
  "com.ca",
```

```
  "c2csmo",
```

```
  "c2cscf",
```

```
  (CIService *)&comm_service_0001835029};
```

Overview of an Example C Program

Your C program needs sections for:

- Including the header file generated using the same CA Gen model as was used to generate the target server.
- Setting import values. The generated header file has two structures for the Import view.
 - The Import view table has CA Gen specific information and may be difficult to use directly.
 - The transaction specific structure (for example, `s_ServerDetailDivision`) is a standard C structure that is used to reference import and export data. If the Import view contains a repeating group view, your application needs to populate the cardinality and `active_flag` fields.
- Setting initial control values for the transaction.
 - The function `ProxyAllocateParamBlock` is used to create a parameter block with information used for each transaction.
 - The `ProxySet` functions (for example, `ProxySetClientUserid` and `ProxySetCommand`) are used to set values in the parameter block.
 - The transaction information can be modified directly. For example, use `SERVER_DETAIL_DEPARTMENT.modelName` to change the model name.
 - The function `ProxyConfigureComm` is used to configure communications information.
- Executing the operation. In the following sample C program, the `ProxyExecute` function is used to execute a server call in a synchronous manner. Different commands are necessary to execute the call asynchronously.
- Examining status information from the transaction. If an error occurs during execution, a formatted error message, similar to the one passed to generated CA Gen clients is available. Or else, the operation's exports are available. The `ProxyGet` functions (for example, `ProxyGetExitStateMsg` and `ProxyGetExitMsgType`) are used in this section.
- Retrieving and using export values. The generated header file will have two structures for the export view.
 - The Export view table has CA Gen specific information and may be difficult to use directly.
 - The transaction specific structure (for example, `s_ServerDetailDivision`) is a standard C structure that is used to reference import and export data. The export view may contain a repeating group view.

An overview of an example C program follows. Information referenced in the generated header file is shown in bold. This example is derived from the CA Gen sample proxy source file, but does not include the entire file content. Selected code fragments have been extracted for discussion purposes.

Description	Example C Program
Comments describing program	<p>This sample's purpose is to provide a C programmer a starting point for coding a C proxy routine.</p> <p>This sample builds upon the CA (R) Gen sample model that is included with the CA Gen Toolset. The Gen server module used by this proxy example is packaged within the sample model as load module P900. This proxy example will only exercise a small amount of the sample models capabilities.</p>
Include the CA Gen C Proxy generated header file. This file must be generated using the same CA Gen model used to create the target server load module.	#include "p900.h"
Include other standard libraries as needed	<pre>#include <malloc.h> #include <ctype.h> #include <stdlib.h> #include <stdio.h> #include <string.h></pre>
Declare variables	<pre>/* Contains communication information */ static ProxyParam* psParamBlock; /* Error message length can be from 0-2048. This will determine how much of the error message will be returned by ProxyExecute. A shorter message may be returned depending on the error. */ #define errMsgLen 2048 char errMsg[errMsgLen]; /* Return code from routines {Failure, Success} */ ProxyExecuteReturnValues ReturnVal; /* Integer return code */ int nRC;</pre>

Description	Example C Program
Beginning of main program	<pre>int main (void) { printf ("Beginning of sample C proxy application \n\n");</pre>
ProxyStartTracing opens a file for debug/status information. ProxyStopTracing is used to close the file.	<pre>ProxyStartTracing((char*)"cproxy.out", (char *)"0xFFFFFFFF");</pre>
ProxyTraceOut writes a message to the trace file. These messages are interspersed with standard debugging messages. This function call is optional.	<pre>ProxyTraceOut(NULL, (char *)"cproxy.c - ***** Beginning program");</pre>
ProxyAllocateParamBlock is used to create the parameter block (data structure) that contains information required for the transaction. A NULL return indicates an error in the creation of the parameter block. This function call is required.	<pre>psParamBlock = ProxyAllocateParamBlock(); if (NULL == psParamBlock) { printf("Unsuccessful return - ProxyAllocateParamBlock\n"); return (Failure); }</pre>
ProxyConfigureComm is used to set communication options. This call to ProxyConfigureComm uses the file commcfg.ini.	<pre>nRC = ProxyConfigureComm(&SERVER_DETAIL_DIVISION, SERVER_DETAIL_DIVISION.service, NULL); if (nRC == FALSE) { printf("Unsuccessful return from ProxyConfigureComm\n"); return (Failure); }</pre>
Load example data into the import record.	<pre>strcpy(server_detail_division.importRequest_command.value, Command);</pre>

Description	Example C Program
<p>ProxyExecute executes the transaction to the server as a synchronous, blocked cooperative flow. Export data and messages are returned to the application. This function uses the following parameters:</p> <p>SERVER_DETAIL_DIVISION transaction structure that contains import and export views</p> <p>psParamBlock parameter structure that contain communication information</p> <p>errMsg returns any error message from the server</p> <p>errMsgLen specifies the number of characters to write to errMsg</p> <p>This function call is required.</p>	<pre> If (ProxyExecute(&SERVER_DETAIL_DIVISION, psParamBlock, errMsg, errMsgLen) != Success) { printf("Unsuccessful return from ProxyExecute \n"); printf("errMsg = %s\n",errMsg); ProxyTraceOut(NULL, errMsg); return (Failure); } </pre>
<p>The ProxyClear set of functions is used to clear the parameter block. Each part of the parameter block can be cleared individually, or the ProxyClearParamBlock function can be used to clear the entire parameter block. This function call is optional.</p>	<pre> nRC = ProxyClearParamBlock(psParamBlock); if (RetVal == Failure) { printf("Unsuccessful return from ProxyClearParamBlock\n"); return (Failure); } </pre>
<p>The ProxyDeleteParamBlock function is used to delete the psParamBlock structure. This function call is optional, but recommended.</p>	<pre> nRC = ProxyDeleteParamBlock(psParamBlock); if (RetVal == Failure) { printf("Unsuccessful return from ProxyDeleteParamBlock\n"); return (Failure); } </pre>
<p>Export View processing-displays the export view.</p>	<pre> printf("\n Here is the export \n"); /* Prints the returned division name */ printf("Division Name %s\n", server_detail_division.export_division.name); </pre>
<p>End of program</p>	<pre> return (TRUE); } </pre>

Description	Example C Program
ProxyStopTracing is used to close the trace file opened by ProxyStartTracing. This function is optional, but is recommended if ProxyStartTracing is used.	ProxyStopTracing();

More information:

[Function Calls](#) (see page 175)

[Repeating Group Views](#) (see page 173)

Tuxedo C Proxy Support

CA Gen supports user-written applications targeting a Tuxedo server environment. In general the user-written application code may be identical when targeting Tuxedo or non Tuxedo server environments. A notable exception is the ProxyConfigureComm API. This API is not supported when targeting Tuxedo server environments, thus must not be used when developing an application targeting Tuxedo server environments.

Note: For more information about CA Gen server support within a Tuxedo environment, see the *Tuxedo User Guide*.

Native vs. Workstation Client Applications

User-written client applications targeting a Tuxedo server environment fall into one of two types. The first being native clients, which reside on the same physical machine as do the target servers. The second being Workstation clients which reside on a machine separately from the target servers.

A different set of dependency libraries are required depending upon where the client resides with respect to the server. A compilation define WS_CLIENT must also be defined when building Workstation clients. This enables the correct selection of libraries to support flows to the Tuxedo Server environments which reside on a machine separate from the client applications.

By default, when built on a UNIX platform, the supplied example makefile builds the sample C Proxy application as a native client application. The makefile can be modified to build Workstation clients as well.

Note: Windows platforms will only support Workstation client applications. As mentioned, native clients communicate with servers residing on the same machine. CA Gen Tuxedo servers are not supported on Windows platforms thus it follows that native clients are not supported on Windows platforms.

Overview of the Generated Tuxedo View File (.tvf)

When you generate a C Proxy API for the communication type of Tuxedo, the generation process creates an additional file for each targeted CA Gen Procedure Step. This extra file contains the Procedure Step's view definitions used by the Tuxedo Servers. The file name is typically <ProcStepName>.tvf. The C clients that access the services provided by the Tuxedo Servers require this file.

This .tvf file then must be compiled by a Tuxedo provided utility which will produce two additional files. The first file is an include file, <ProcStepName>.h file, which has already been included via a #include within the generated proxy header file. The second file is a Tuxedo view file, used at runtime, which contains an O.S. specific filename suffix. This view file is named <ProcStepName>.VV on Windows platforms and <ProcStepName>.V on UNIX platforms.

The .tvf file is compiled with the same named Tuxedo command on all supported platforms as follows:

```
viewc32 -n <ProcStepName>.tvf
```

Note: The file {ProcStepName}.h will be included in the generated C proxy header file only if the target server generation parameters are configured to use a communications type of Tuxedo.

Setting Windows Platform Environment Variables

Prior to attempting to build a user-written proxy application (targeting Tuxedo servers) on a Window platform ensure the follow environment variables include paths to the CA Gen and Tuxedo installation directories as specified.

- The TUXDIR environment variable should be set to point to the Tuxedo install area. For example set TUXDIR=C:\oracle\tuxedo
- Add to the PATH variable:

- The CA Gen bin directory
- ```
set PATH=%PATH%;%GENxx%Gen
```

**Note:**

- If working with Visual Studio 32-bit or 64-bit, the PATH environment variable will be appended with %GENxx%Gen\VSabc or %GENxx%Gen\VSabc\amd64 instead of %GENxx%Gen.
- VSabc refers to the supported version of Visual Studio. Replace VSabc with VS100 for Visual Studio 2010 and VS110 for Visual Studio 2012. xx refers to the current release of CA Gen. For the current release number, see the *Release Notes*.

- The Tuxedo installed product bin directory
- ```
set PATH=%PATH%;c:\oracle\tuxedo\bin
```

- Add to the INCLUDE variable

- The CA Gen directory
- ```
set INCLUDE=%INCLUDE%;%GENxx%Gen
```

- Add to the LIB variable

- The CA Gen directory
- ```
set LIB=%LIB%;%GENxx%Gen
```

Note: If working with Visual Studio 32-bit or 64-bit, the LIB environment variable will be appended with %GENxx%Gen\VSabc or %GENxx%Gen\VSabc\amd64 instead of %GENxx%Gen.

Note: VSabc refers to the supported version of Visual Studio. Replace VSabc with VS100 for Visual Studio 2010 and VS110 for Visual Studio 2012. xx refers to the current release of CA Gen. For the current release number, see the *Release Notes*.

- The Tuxedo installed product lib directory
- ```
set LIB=%LIB%;c:\oracle\tuxedo\lib
```

- Set the VIEWDIR32 variable which contains a semicolon-separated list of directories that are searched for .VV VIEW32 files, respectively. For example:

```
set VIEWDIR32=c:\proxy\sample\bin
```

- Set the VIEWFILES32 variable which contains a comma-separated list of allowable filenames for VIEW32 type records. For example:  

```
set VIEWFILES32=ProcStepName1.VV,ProcStepName2.VV
```
- Set the WSNADDR variable which contains the hostname and port number used for connecting to the Tuxedo Workstation listener process that runs on the Target server platform. For example:  

```
set WSNADDR=//Server_hostname:7878
```

## Setting UNIX Platform Environment Variables

Prior to attempting to build a user-written proxy application (targeting Tuxedo servers) on a UNIX platform ensure the follow environment variables include paths to the CA Gen and Tuxedo installation directories as specified.

- The TUXDIR environment variable should be set to point to the Tuxedo install area. For example:  

```
set TUXDIR=/opt/oracle/tuxedo
```
- Add to the PATH variable:  
The CA Gen bin directory  

```
export PATH=$PATH:$IEFH/bin
```

  
The Tuxedo installed product bin directory  

```
export PATH=$PATH:$TUXDIR/bin
```
- Add the CA Gen and Tuxedo product lib directories to one of the following library paths depending on the type of UNIX system:
  - HPUX-SHLIB\_PATH  

```
export SHLIB_PATH=$SHLIB_PATH:$IEFH/lib:$TUXDIR/lib
```
  - AIX-LIBPATH  

```
export LIBPATH=$LIBPATH:$IEFH/lib:$TUXDIR/lib
```
  - Solaris-LD\_LIBRARY\_PATH  

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$IEFH/lib:$TUXDIR/lib
```
- Set the VIEWDIR32 variable which contains a colon-separated list of directories that are searched for .V VIEW32 files, respectively. For example:  

```
export VIEWDIR32=c:/proxy/sample/bin
```

- Set the VIEWFILES32 variable which contains a comma-separated list of allowable filenames for VIEW32 type records. For example:

```
export VIEWFILES32=ProcStepName1.V,ProcStepName2.V
```

- If the user-written application is a Workstation client, set the WSNADDR variable which contains the hostname and port number used for connecting to the Tuxedo Workstation listener process that runs on the Target server platform. Applications written as native clients should not set this variable. For example:

```
export WSNADDR=//Server_hostname:7878
```

## Building User-written Applications Targeting Tuxedo Server Environments

The methods and tools used to compile and link user-written applications targeting Tuxedo server environments differ from those used for non-Tuxedo environments. The Tuxedo product provides several tools and processes which should be used when building Tuxedo applications. The steps involved include:

- Generating the Proxy Client include and Tuxedo support .tvf files. This is done using the CA Gen Workstation to generate a C proxy targeting Tuxedo Servers.
- Transferring these files to the target platform
- Compiling the .tvf files using the Tuxedo viewC32 compiler. This has been detailed in the above .tvf file discussion.
- Building the resulting application executable using the Tuxedo supplied buildclient compile/link tool.
- Building and executing the Tuxedo application server on the target environment.

**Note:** For more information about CA Gen server support within a Tuxedo environment, see the *Tuxedo User Guide*.

It is strongly recommended this tool be used rather than calling the native C language compiler and linker directly. *buildclient* will provide the required compiler flags, include and library paths as well as any extra dependency libraries required to properly build a Tuxedo client application.

## Sample Makefile for C Proxy Targeting Tuxedo

The following is a sample platform generic makefile which can be used to build the sample Proxy application that is delivered with the CA Gen product installation. This makefile as well as the supporting cproxy.c source file can be found the sample subdirectory of the CA Gen installation directory.

```
#####
#
This is a platform generic makefile which can be used to build a
sample C proxy which targets a Tuxedo server environment. The base
name of the executable to be built is "cproxy". To provide a generic
makefile for use on the supported platforms the following make
commands should be executed on the specified platform.
#
To use on Windows:
nmake /f makefile.tux cproxy_win
and
nmake /f makefile.tux clean_win
#
To use on HP UNIX:
make -f makefile.tux cproxy_hp
and
make /f makefile.tux clean_hp
#
To use on Sun Solaris:
make /f makefile.tux cproxy_sol
and
make /f makefile.tux clean_sol
#
To use on IBM AIX:
make /f makefile.tux cproxy_aix
and
nmake /f makefile.tux clean_win
#
#####

Common section

#RELEASE= echo "Set RELEASE to the current release level"
RELEASE=85

all: cproxy

.SUFFIXES: .TVF .V .VV

Suffix rule for UNIX requires .V view files
.TVF.V :
 echo y | viewc32 -n $<

Suffix rule for Windows requires .VV view files
.TVF.VV:
 echo y | viewc32 -n $<

UNIX requires .V files
```



```

UNIX_VFILES=SERVERD1.V SERVERDE.V SERVERM2.V SERVERD2.V SERVERM1.V SERVERMA.V

Windows requires .VV files
WIN_VFILES=SERVERD1.VV SERVERDE.VV SERVERM2.VV SERVERD2.VV SERVERM1.VV SERVERMA.VV

UNIX Compiler Defines
#
for building on HP11 Itanium UNIX
HP_CC="aCC -DUNIX +DD64 -DHPIA64 -g +Z -DTUXEDO"

For building on Sun Solaris UNIX
SOL_CC="CC -m64 -DUNIX -DSYSV -DIEF_SOL -DTCP_THREAD -D_THREAD_SAFE -D_REENTRANT -g
-DTUXEDO"

For building on IBM AIX
AIX_CC="xlC_r -q64 -DUNIX -DAIX -q cpluscmt -DIEF_AIX -g -DTUXEDO"

common UNIX defines
UNIX_INCS= -I. -I$(IEFH)/include -I$(TUXDIR)/include

Native Work-Station clients
#
For building native (local) clients uncomment the following MESSAGE
and UNIX_LIB lines
#
For UNIX, to build a native client uncomment the following lines.
Native clients reside in the same Tuxedo Domain (same machine as Tuxedo
servers) and use the same TUXCONFIG file. Therefore make sure that the
TUXCONFIG environment variable is set to the same value as used by the Tuxedo
servers. In addition set the following env variables prior to executing
the sample application: VIEWDIR32, VIEWFILES32.
VIEWDIR32 is a colon separate list of search directories. Where to
find Tuxedo view (.V) files
export VIEWDIR32=/users/testuser/model.ief/c/proxy/c
VIEWFILES32 is a comma separated list of Tuxedo view (.V) files.
export
VIEWFILES32=SERVERD1.V,SERVERDE.V,SERVERM2.V,SERVERD2.V,SERVERM1.V,SERVERMA.V
#
#
MESSAGE="Building A Native Client"
UNIX_LIB=-L$(IEFH)/lib -lpxrt.$(RELEASE)

#
#

For building Work-Station client's uncomment the following MESSAGE,
UNIX_CFLAGS2, and UNIX_LIB lines
#

```

```
Work-Station clients reside outside the Tuxedo Domain (on a machine
separate from the machine hosting the Tuxedo servers) and use the
WSL to connect to the Tuxedo Servers. In addition set the following
environment variables prior to executing the sample application:
WSNADDR, VIEWDIR32, VIEWFILES32.
VIEWDIR32 is a colon separate list of search directories. Where to
find Tuxedo view (.V) files
export VIEWDIR32=/users/testuser/model.ief/c/proxy/c
VIEWFILES32 is a comma separated list of Tuxedo view (.V) files.
export
VIEWFILES32=SERVERD1.V,SERVERDE.V,SERVERM2.V,SERVERD2.V,SERVERM1.V,SERVERMA.V
WSNADDR contains the hostname and port number assigned to the Tuxedo
WSL process running on the Target server machine
export WSNADDR=//Server_hostname:7878
#
#
#UNIX_CFLAGS2=-DWS_CLIENT
#MESSAGE="Building A Work-Station Client"
#UNIX_LIB=-L$(IEFH)/lib -lpxrt.$(RELEASE) -ltxwcf.$(RELEASE)
#BLDCLNT_OPT=-w

UNIX Compile/link rules
#
#
cproxy_hp : $(UNIX_VFILES)
 echo $(MESSAGE)
 CC=$(HP_CC) CFLAGS="$(UNIX_CFLAGS2) $(UNIX_INCS) $(UNIX_LIB)" buildclient
$(BLDCLNT_OPT) -o cproxy -f cproxy.c

cproxy_aix : $(UNIX_VFILES)
 echo $(MESSAGE)
 CC=$(AIX_CC) CFLAGS="$(UNIX_CFLAGS2) $(UNIX_INCS) $(UNIX_LIB)" buildclient
$(BLDCLNT_OPT) -o cproxy -f cproxy.c

cproxy_sol : $(UNIX_VFILES)
 echo $(MESSAGE)
 CC=$(SOL_CC) CFLAGS="$(UNIX_CFLAGS2) $(UNIX_INCS) $(UNIX_LIB)" buildclient
$(BLDCLNT_OPT) -o cproxy -f cproxy.c

Windows compile/link rules
#
Windows Work-Station clients reside outside the Tuxedo Domain (on a
machine separate from the machine hosting the Tuxedo servers) thus
must use the Tuxedo environment WSL process to connect to the Tuxedo
Servers. The following environment variables must be set prior to
executing the sample application:
WSNADDR, VIEWDIR32, VIEWFILES32.
VIEWDIR32 is a semicolon separate list of search directories.
```

```

where to find Tuxedo view (.V) files
set VIEWDIR32=/users/testuser/model.ief/c/proxy/c
VIEWFILES32 is a comma separated list of Tuxedo view (.V) files.
set
VIEWFILES32=SERVERD1.V,SERVERDE.V,SERVERM2.V,SERVERD2.V,SERVERM1.V,SERVERMA.V
WSNADDR contains the hostname and port number assigned to the
Tuxedo WSL process running on the Target server machine
set WSNADDR=//Server_hostname:7878
#

cproxy_win : $(WIN_VFILES)
 set CFLAGS= -DTUXEDO -DWS_CLIENT -I. -I"$(IEFH)" "$(IEFH)\pxrt85n.lib"
-I"$(TUXDIR)\include"
 buildclient -o cproxy -f cproxy.c

#
clean rules
#
clean_unx :
 rm -f *.o cproxy *.V S*.h

clean_win :
 del s*.h *.obj cproxy.exe *.VV

clean_aix: clean_unx

clean_hp: clean_unx

```

## Proxy Function Prototypes

Proxy function prototypes defined in the various header files which are included in the generated header, are listed in the following table:

| Header File | Proxy Function Prototypes     |
|-------------|-------------------------------|
| Proxytrc.h  | Proxy Trace functions         |
| Proxycfg.h  | Proxy Configuration functions |
| Proxyxit.h  | Proxy Exit functions          |
| Proxyexe.h  | Proxy Runtime functions       |

## API Variable Types

This section lists the API Variable Types mentioned in the following API tables. The characteristics of each type are described.

### Enum Type

Datatypes listed within the enum types each have a set of declared enumerators. Each enumerator represents a numeric or a character value. See the header files for specific enumerators and their values.

### Numeric Type

Numeric types are integer values. They can be classified as double, long, or short. The length for each of these classifications varies depending on the machine and compiler used.

### Character Arrays

Character arrays are text strings. Each type definition has an allowable number of characters for the text and a null terminator.

### Decimal Precision Attributes

The data structure for this attribute is a DPREC array whose size is the length of the attribute plus 3 (for the sign, decimal point, and null terminator). A DPREC is a typedef of char. For example, an attribute that is defined as a number of 18 digits will be implemented as DPREC[21].

The number represented within the DPREC array may consist of the following depending upon the definition of the attribute it implements:

- A minus sign.
- Zero or more decimal digits with a decimal point, one or more decimal digits without a decimal point.
- A decimal point.
- One or more decimal digits.
- A null terminator.

For the import view, all decimal precision attributes should adhere to the following rules:

- The character representation of the number placed in the DPREC array may contain a fewer number of digits than that defined for the attribute. However, it must not contain more digits to the left or right of the decimal than that defined for the attribute.

- The DPrec array must be null terminated.
- If number of digits to the right of the decimal equals the total number of digits, the resulting string must not contain a leading zero prior to the decimal point.

For the export view, a decimal precision attribute may be its simplest form or may contain a plus sign and/or leading and trailing zeros.

## Repeating Group Views

Repeating groups can be in import and export views. The repeating group view has two variables added to the view structure:

- `cardinality`-an integer value that contains either the maximum size of the repeating group or the number of elements that are populated (if less than the maximum). For an import view, your application needs to populate this field with the number of occurrences of actual data at runtime. If you have an array of five elements, but there are only three with data, the `cardinality` variable contains 3 as its value. If you do not set this value correctly on an import view, the server will not process the view.
- `active_flag`-a character array that indicates (with Y or N) which elements of the repeating group view are populated. For an import view, copy the corresponding number of Y characters into the `active_flag` array.

The following is an example of export repeating group view.

```
/* Repeating Group View: ALL_OUTPUT */
/* Repeats: 30 times */
struct {
 long cardinality;
 char active_flag[30];
 /* Entity View: EXPORT */
 /* Type: WI_SELECTION */

 struct {
 char value[30][2];
 } export_wiSelection;

 /* Entity View: EXPORT */
 /* Type: EMPLOYEE */
 struct {
 long number[30];
 char name[30][31];
 } export_employee;

 /* Entity View: EXPORT */
 /* Type: DIVISION */
 struct {
 short number[30];
 char name[30][31];
 } export_division;
} allOutput;
/* End of Group View: ALL_OUTPUT */
```

The following is an example section of code to display the export repeating group values:

```
printf("List of Divisions\n");
if (server_maintain_division.allOutput.cardinality > 0)
{
 printf("\tNumber\tName\n");
 printf(" -----\n");
 for (loop = 0; loop < server_maintain_division.allOutput.cardinality; loop++)
 {
 printf("\t%d",
 server_maintain_division.allOutput.export_division.number[loop]);
 printf("\t%s\n",
 server_maintain_division.allOutput.export_division.name[loop]);
 }
}
else
 printf("\tNo Data Found\n");
}
```

## Function Calls

The following sections list all the function calls available when using a C Proxy API. They are categorized into the function calls that you must use for synchronous processing, asynchronous processing, and for available function calls that are independent of the type of cooperative flow. The detailed description of each function follows.

For synchronous processing, use the function calls listed in the following table:

| Call         | Description                                                           |
|--------------|-----------------------------------------------------------------------|
| ProxyExecute | Sends a transaction to a server and returns export data and messages. |
| n.a.         | n.a.                                                                  |

For asynchronous processing, use the function calls listed in the following table:

| Call                     | Description                                                          |
|--------------------------|----------------------------------------------------------------------|
| ProxyExecuteAsync        | Initiates an asynchronous cooperative flow.                          |
| ProxyCheckAsyncResponse  | Interrogates the state of a specified request.                       |
| ProxyGetAsyncResponse    | Attempts to obtain an asynchronous request's corresponding response. |
| ProxyIgnoreAsyncResponse | Informs the runtime that a specified response is to be ignored.      |

**Note:** The communication type of Tuxedo does not support Asynchronous Processing. Using these calls for Tuxedo results in erroneous behavior.

The functions listed in the following table are not dependent on the type of communication flow being used:

| Functions Independent of Processing | Description                                                                                          |
|-------------------------------------|------------------------------------------------------------------------------------------------------|
| ProxyAllocateParamBlock             | Creates the parameter block (data structure) that contains information required for the transaction. |
| ProxyClearClientPassword            | Clears the password part of the parameter block.                                                     |
| ProxyClearClientUserid              | Clears the userid part of the parameter block.                                                       |

| Functions Independent of Processing | Description                                                                                                                      |
|-------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| ProxyClearCommand.htm               | Clears the command part of the parameter block.                                                                                  |
| ProxyClearDialect                   | Clears the dialect part of the parameter block.                                                                                  |
| ProxyClearExitMsgType               | Clears the Exit Message Type of the parameter block.                                                                             |
| ProxyClearExitStateMsg              | Clears the Exit State message part of the parameter block.                                                                       |
| ProxyClearExitStateNum              | Clears the Exit State Number part of the parameter block.                                                                        |
| ProxyClearNextLocation              | Clears the next location part of the parameter block.                                                                            |
| ProxyClearParamBlock                | Clears the entire parameter block.                                                                                               |
| ProxyConfigureComm                  | Sets communication options.                                                                                                      |
| ProxyDeleteParamBlock               | Delete the psParamBlock structure.                                                                                               |
| ProxyGetClientPassword              | Returns the value of the password part of the parameter block.                                                                   |
| ProxyGetClientUserid                | Returns the value of the Userid part of the parameter block.                                                                     |
| ProxyGetCommand                     | Returns the value of the Command part of the parameter block.                                                                    |
| ProxyGetDialect                     | Returns the value of the Dialect part of the parameter block.                                                                    |
| ProxyGetExitMsgType                 | Returns the value of the Exit Message Type part of the parameter block. See also ProxyGetExitStateMsg and ProxyGetExitStateNum.  |
| ProxyGetExitStateMsg                | Returns the value of the Exit State Message part of the parameter block. See also, ProxyGetExitMsgType and ProxyGetExitStateNum. |
| ProxyGetExitStateNum                | Returns the value of the Exit State Number part of the parameter block. See also, ProxyGetExitMsgType and ProxyGetExitStateMsg.  |
| ProxyGetNextLocation                | Returns the value of the Next Location part of the parameter block.                                                              |



| Functions Independent of Processing | Description                                                                                                                                   |
|-------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| ProxySetClientPassword              | Sets the client user password part of the parameter block that is sent to the server where the procedure step's executable code is installed. |
| ProxySetClientUserid                | Sets the client user id part of the parameter block to be sent to the server where the procedure step's executable code is installed.         |
| ProxySetCommand                     | Sets the value of the Command part of the parameter block.                                                                                    |
| ProxySetDialect                     | Sets the value of the Dialect part of the parameter block.                                                                                    |
| ProxySetNextLocation                | Sets the value of the Next Location part of the parameter block.                                                                              |
| ProxyStartTracing                   | Opens a file for debug/status information.                                                                                                    |
| ProxyTraceOut                       | Writes a message to the trace file.                                                                                                           |
| ProxyStopTracing                    | Closes the trace file opened by ProxyStartTracing.                                                                                            |
| ProxySetViewBlob                    | Set BLOB predicate view to input data, writes input data to BLOB.                                                                             |
| ProxyGetViewBlob                    | Get BLOB predicate view data into output buffer, reads BLOB data into buffer.                                                                 |
| ProxyClearViews                     | Clear view data.                                                                                                                              |

## ProxyAllocateParamBlock

### Function Call

```
ProxyParam *ProxyAllocateParamBlock();
```

### Description

ProxyAllocateParamBlock is used to create the parameter block (data structure) that contains information required for the transaction. This function call is required at least once. Any parameter block allocated should be de-allocated using the function ProxyDeleteParamBlock before exiting the program.

### Inputs

None

### Outputs

Returns a parameter block data structure or a NULL which indicates an error occurred in the creation of the parameter block.

### Related Functions

- ProxyDeleteParamBlock
- ProxySet functions
- ProxyGet functions
- ProxyClear functions

### Example

```
ProxyParam* psParamBlock;
psParamBlock = ProxyAllocateParamBlock();
if (NULL == psParamBlock)
{
 printf("Unsuccessful return from ProxyAllocateParamBlock\n");
 return (Failure);
}
```

## ProxyCheckAsyncResponse

### Function Call

```
ProxyCheckAsyncResponseReturnValue ProxyCheckAsyncResponse(AsyncRequestID
requestID);
```

### Description

Checks the state of a given outstanding asynchronous request. This function does not attempt to obtain a response. ProxyCheckAsyncResponse executes as a non-blocking operation. The supporting runtime interrogates the state of the specified request and then communicates that state to the application by a return code value.

**Note:** This function is not supported by the communication type of Tuxedo.

### Inputs

requestID-An AsyncRequest ID field that contains a unique ID associated with the outstanding asynchronous request for which the status is being checked.

## Return Codes

Returns an enumerated type `ProxyCheckAsyncResponseReturnValue`. Values may be:

- `CheckAsyncAvailable`-The specified request's response can be successfully obtained (using `ProxyGetAsyncResponse`).
- `CheckAsyncPending`-The specified request's response is not immediately available.
- `CheckAsyncInvalidAsyncRequestID`-The request ID cannot be found. This may be because the ID is not valid, or because the response was already completed (that is, retrieved or ignored).
- `CheckAsyncProcessingError`-The proxy runtime was unable to process the check async request, but the error does not fall into one of the above categories.

## Related Functions

- `ProxyExecuteAsync`
- `ProxyGetAsyncResponse`
- `ProxyIgnoreAsyncResponse`

## Example

```
ProxyCheckAsyncResponseReturnValue ReturnVal;
ReturnVal = ProxyCheckAsyncResponse(requestID);
if ((ReturnVal != CheckAsyncAvailable) && (ReturnVal != CheckAsyncPending))
{
 printf("Unsuccessful return from ProxyCheckAsyncResponse\n");
 return (Failure);
}
```

## ProxyClearClientPassword

### Function Call

```
int ProxyClearClientPassword(ProxyParam *param);
```

### Description

The `ProxyClearClientPassword` function clears just the Password part of the parameter block. Alternatively, you can use the `ProxyClearParamBlock` function to clear the entire parameter block. This function call is optional.

### Inputs

The parameter block structure created by a call to `ProxyAllocateParamBlock`.

### Outputs

Returns an integer value that is one of the enumerated types of ProxyExecuteReturnValues. 0==Failure, 1== Success.

### Related Functions

- ProxyClearParamBlock
- ProxySetClientPassword
- ProxyGetClientPassword

### Example

```
int ReturnVal;
ReturnVal = ProxyClearClientPassword(psParamBlock);
if (ReturnVal == Failure)
{
 printf("Unsuccessful return from ProxyClearClientPassword\n");
 return (Failure);
}
```

## ProxyClearClientUserid

### Function Call

```
int ProxyClearClientUserid(ProxyParam *param);
```

### Description

The ProxyClearClientUserid function clears just the Userid part of the parameter block. Alternatively, you can use the ProxyClearParamBlock function to clear the entire parameter block. This function call is optional.

### Inputs

A parameter block structure created by a call to ProxyAllocateParamBlock.

### Outputs

Returns an integer value that is one of the enumerated types of ProxyExecuteReturnValues. 0==Failure, 1== Success.

### Related Functions

- ProxyClearParamBlock

- ProxySetClientUserid
- ProxyGetClientUserid

### Example

```
int ReturnVal;
ReturnVal = ProxyClearClientUserid(psParamBlock);
if (ReturnVal == Failure)
{
 printf("Unsuccessful return from ProxyClearClientUserid\n");
 return (Failure);
}
```

## ProxyClearCommand

### Function Call

```
int ProxyClearCommand(ProxyParam *param);
```

### Description

The ProxyClearCommand function clears just the Command part of the parameter block. Alternatively, you can use the ProxyClearParamBlock function to clear the entire parameter block. This function call is optional.

### Inputs

The parameter block structure created by a call to ProxyAllocateParamBlock.

### Outputs

Returns an integer value that is one of the enumerated types of ProxyExecuteReturnValues. 0==Failure, 1== Success.

### Related Functions

- ProxyClearParamBlock
- ProxySetCommand
- ProxyGetCommand

### Example

```
int ReturnVal;
ReturnVal = ProxyClearCommand(psParamBlock);
if (ReturnVal == Failure)
{
 printf("Unsuccessful return from ProxyClearCommand\n");
 return (Failure);
}
```

## ProxyClearDialect

### Function Call

```
int ProxyClearDialect(ProxyParam *param);
```

### Description

The ProxyClearDialect function clears just the Dialect part of the parameter block. Alternatively, you can use the ProxyClearParamBlock function to clear the entire parameter block. This function call is optional.

### Inputs

The parameter block structure created by a call to ProxyAllocateParamBlock.

### Outputs

Returns an integer value that is one of the enumerated types of ProxyExecuteReturnValues. 0==Failure, 1== Success.

### Related Functions

- ProxyClearParamBlock
- ProxySetDialect
- ProxyGetDialect

### Example

```
int ReturnVal;
ReturnVal = ProxyClearDialect(psParamBlock);
if (ReturnVal == Failure)
{
 printf("Unsuccessful return from ProxyClearDialect\n");
 return (Failure);
}
```

## ProxyClearExitMsgType

### Function Call

```
int ProxyClearExitMsgType(ProxyParam *param);
```

### Description

The ProxyClearExitMsgType function clears just the Exit Message Type part of the parameter block. Alternatively, you can use the ProxyClearParamBlock function to clear the entire parameter block. This function call is optional.

### Inputs

The parameter block structure created by a call to ProxyAllocateParamBlock.

### Outputs

Returns an integer value that is one of the enumerated types of ProxyExecuteReturnValues. 0==Failure, 1== Success.

### Related Functions

- ProxyClearParamBlock
- ProxyGetExitMsgType

### Example

```
int ReturnVal;
ReturnVal = ProxyClearExitMsgType(psParamBlock);
if (ReturnVal == Failure)
{
 printf("Unsuccessful return from ProxyClearExitMsgType\n");
 return (Failure);
}
```

## ProxyClearExitStateMsg

### Function Call

```
int ProxyClearExitStateMsg(ProxyParam *param);
```

### Description

The ProxyClearExitStateMsg function clears just the Exit State Message part of the parameter block. Alternatively, you can use the ProxyClearParamBlock function to clear the entire parameter block. This function call is optional.

### Inputs

The parameter block structure created by a call to ProxyAllocateParamBlock.

### Outputs

Returns an integer value that is one of the enumerated types of ProxyExecuteReturnValues. 0==Failure, 1== Success.

### Related Functions

- ProxyClearParamBlock
- ProxyGetExitStateMsg

### Example

```
int ReturnVal;
ReturnVal = ProxyClearExitStateMsg(psParamBlock);
if (ReturnVal == Failure)
{
 printf("Unsuccessful return from ProxyClearExitStateMsg\n");
 return (Failure);
}
```

## ProxyClearExitStateNum

### Function Call

```
int ProxyClearExitStateNum(ProxyParam *param);
```

### Description

The ProxyClearExitStateNum function clears just the Exit State Number part of the parameter block. Alternatively, you can use the ProxyClearParamBlock function to clear the entire parameter block. This function call is optional.

### Inputs

The parameter block structure created by a call to ProxyAllocateParamBlock.

### Outputs

Returns an integer value that is one of the enumerated types of ProxyExecuteReturnValues. 0==Failure, 1== Success.



## Related Functions

- ProxyClearParamBlock
- ProxyGetExitStateNum

## Example

```
int ReturnVal;
ReturnVal = ProxyClearExitStateNum(psParamBlock);
if (ReturnVal == Failure)
{
 printf("Unsuccessful return from ProxyClearExitStateNum\n");
 return (Failure);
}
```

## ProxyClearNextLocation

### Function Call

```
int ProxyClearNextLocation(ProxyParam *param);
```

### Description

The ProxyClearNextLocation function clears just the Next Location part of the parameter block. Alternatively, you can use the ProxyClearParamBlock function to clear the entire parameter block. This function call is optional.

### Inputs

The parameter block structure created by a call to ProxyAllocateParamBlock.

### Outputs

Returns an integer value that is one of the enumerated types of ProxyExecuteReturnValues. 0==Failure, 1== Success.

## Related Functions

- ProxyClearParamBlock
- ProxySetNextLocation
- ProxyGetNextLocation

### Example

```
int ReturnVal;
ReturnVal = ProxyClearNextLocation(psParamBlock);
if (ReturnVal == Failure)
{
 printf("Unsuccessful return from ProxyClearNextLocation\n");
 return (Failure);
}
```

## ProxyClearParamBlock

### Function Call

```
int ProxyClearParamBlock(ProxyParam *param);
```

### Description

The ProxyClearParamBlock function clears the entire parameter block. Alternatively, you can use the ProxyClear set of functions to clear each part of the parameter block individually. This function call is optional.

### Inputs

The parameter block structure created by a call to ProxyAllocateParamBlock.

### Outputs

Returns an integer value that is one of the enumerated types of ProxyExecuteReturnValues. 0==Failure, 1== Success.

### Related Functions

- ProxySet functions
- ProxyGet functions
- ProxyAllocateParamBlock
- ProxyDeleteParamBlock

### Example

```
int ReturnVal;
ReturnVal = ProxyClearParamBlock(psParamBlock);
if (ReturnVal == Failure)
{
 printf("Unsuccessful return from ProxyClearParamBlock\n");
 return (Failure);
}
```

## ProxyConfigureComm

### Function Call

```
int ProxyConfigureComm(CITranEntry *TranEntry, void *ServiceTable, char *comcfg);
```

### Description

The ProxyConfigureComm function sets the communication options. This function call is required. You can set the communication options in the following ways:

- By calling ProxyConfigureComm with a NULL third parameter to indicate that the file commcfg.ini should be used.
- By calling ProxyConfigureComm with a string containing the Comm Service Type, Host and Port as the third parameter.

### Inputs

The inputs to this function are:

- TranEntry data structure.
- Service Table structure.
- NULL or configuration string. The Trancode is not included since this information is included in the TranEntry data structure.

#### **More information:**

[Overview of the Generated Header File](#) (see page 150)

[Configuring C Proxy Using commcfg.ini](#) (see page 227)

### Outputs

TRUE or FALSE

### Related Functions

None

## Example

```
/* This call to ProxyConfigureComm will use the file commcfg.ini */
nRC = ProxyConfigureComm(&SERVER_DETAIL_DIVISION, SERVER_DETAIL_DIVISION.service,
NULL);
if (nRC == FALSE) {
 printf("Unsuccessful return from ProxyConfigureComm using File\n");
 return (Failure);
}
```

## ProxyDeleteParamBlock

### Function Call

```
int ProxyDeleteParamBlock(ProxyParam *param);
```

### Description

The ProxyDeleteParamBlock function deletes the psParamBlock structure. This function call is optional, but highly recommended to prevent memory leaks.

### Inputs

The parameter block structure created by a call to ProxyAllocateParamBlock.

### Outputs

Returns an integer value that is one of the enumerated types of ProxyExecuteReturnValues. 0==Failure, 1== Success.

### Related Functions

- ProxyAllocateParamBlock
- ProxyClearParamBlock

## Example

```
int ReturnVal;
ReturnVal = ProxyDeleteParamBlock(psParamBlock);
if (ReturnVal == Failure)
{
 printf("Unsuccessful return from ProxyDeleteParamBlock\n");
 return (Failure);
}
```

## ProxyExecute

### Function Call

```
ProxyExecuteReturnValues ProxyExecute(CITranEntry *Tran, ProxyParam *param,
 char *errMsg, int errMsgLen);
```

### Description

ProxyExecute executes the server operation (sends the transaction to the server and returns export data and messages). This function call blocks the client application until the target server operation is complete. That is, this function operates as a synchronous cooperative flow.

### Inputs

The inputs to this function are:

- CITranEntry-Transaction structure that contains import and export views
- ProxyParam (psParamBlock)-The parameter block structure created by a call to ProxyAllocateParamBlock
- errMsg- as input is a pointer to a buffer
- errMsgLen-specifies the number of characters to write to errMsg

### Outputs

The outputs of this function are:

- Returns a value that is of the enumerated type ProxyExecuteReturnValues
- errMsg as output returns any error message from the server
- Exports are set in the ProxyParam structure

### Related Functions

None

### Example

```
if (ProxyExecute(&SERVER_DETAIL_DIVISION, psParamBlock, errMsg, errMsgLen)
 != Success)
{
 printf("Unsuccessful return from ProxyExecute \n");
 printf("errMsg = %s\n",errMsg);
 ProxyTraceOut(NULL, errMsg);
 return (Failure);
}
```

## ProxyExecuteAsync

### Function Call

```
ProxyExecuteAsyncReturnValue
ProxyExecuteAsync (AsyncRequestID * requestID, int noResponse,
 CITranEntry *Tran, ProxyParam *param, char *errMsg,
 int errMsgLen);
```

### Description

This function initiates an asynchronous cooperative flow. If the request is accepted by the supporting runtime, then the request is considered outstanding. Each outstanding asynchronous request is identified by a unique ID value. This ID value is generated by the supporting runtime that places the ID into the address specified by the requestID argument. The returned AsyncRequestID handle lets the requesting application to identify the specific outstanding request on other C Proxy API calls (ProxyGetAsyncResponse, ProxyCheckAsyncResponse, ProxyIgnoreAsyncResponse).

The application should preserve the returned AsyncRequestID value for as long as the request remains outstanding. The application can have concurrent outstanding asynchronous requests, each request referencing a unique value of AsyncRequestID.

**Note:** This function is not supported for communication type of Tuxedo.

### Inputs

The inputs to this function are:

- requestID-A pointer to an AsyncRequest ID field that receives the unique ID associated with the outstanding asynchronous request. The returned AsyncRequestID is used on subsequent API calls to refer to a specific outstanding asynchronous request.
- noResponse-A flag used to inform the supporting runtime if the application wishes to process the response associated with the request being initiated.

A value of NORESPONSE causes the supporting runtime to mark the request as complete before returning control back to the initiating runtime. In this case, the application does not need to explicitly complete the request using either a ProxyGetAsyncResponse or ProxyIgnoreAsyncResponse.

A value of RESPONSE causes the supporting runtime to treat the request as a cooperative flow, such that the application expects to process the request's corresponding response.

- Tran-A pointer to a completed TranEntry data structure. The import view referenced in the specified TranEntry is used by ProxyExecuteAsync to formulate the asynchronous request.
- param-A pointer to an allocated (and completed) ProxyParam data structure.

- `errMsg`-A pointer to a character array buffer that is populated with an error message if the asynchronous request is not accepted by the supporting runtime.
- `errMsgLen`-An integer value that specifies the size of the `errMsg` character array.

## Outputs

Returns an enumerated type `ProxyExecuteAsyncReturnValue`. This may be:

- `ExecuteAsyncRequestAccepted`-The request was accepted.
- `ExecuteAsyncRequestNotAccepted`-The request was not accepted. The character array that is indicated by the `errMsg` argument is populated with an explanation of why the request was not accepted for processing.

## Related Functions

- `ProxyGetAsyncResponse`
- `ProxyCheckAsyncResponse`
- `ProxyIgnoreAsyncResponse`

## Example

```
ProxyExecuteAsyncReturnValue ReturnVal;
ReturnVal = ProxyExecuteAsync(requestID, noResponse, &SERVER_DETAIL_DIVISION,
 psParamBlock, errMsg, errMsgLen);
if (ReturnVal != ExecuteAsyncRequestAccepted)
{
 printf("Unsuccessful return from ProxyExecuteAsync \n");
 printf("errMsg = %s\n", errMsg);
 ProxyTraceOut(NULL, errMsg);
 return (Failure);
}
```

## ProxyGetAsyncResponse

### Function Call

```
ProxyGetAsyncResponseReturnValue
ProxyGetAysncResponse(AsyncRequestID requestID, int blocking,
 CITranEntry *Tran, ProxyParam *param,
 char *errMsg, int errMsgLen);
```

## Description

This function attempts to complete the specified outstanding asynchronous cooperative flow by obtaining the request's corresponding response. The AsyncRequestID argument identifies which outstanding request's response is requested. If the specified AsyncRequestID does not specify an outstanding asynchronous request, the function returns a return code value of InvalidAsyncRequestID.

The ProxyGetAsyncResponse is issued as a blocking or non-blocking call. If the response associated with the specified request is available, both the blocking and non-blocking forms of the call behave the same. If the response is not immediately available, the blocking argument directs the supporting runtime on how it should handle the processing of the ProxyGetAsyncResponse call.

**Note:** This function is not supported for communication type of Tuxedo.

## Inputs

The inputs to this function are:

- requestID-The AsyncRequest ID field contains a unique ID associated with an outstanding asynchronous request.
- blocking-A flag used to inform the supporting runtime that the application wishes to block its processing until the specified request's corresponding response is available.

Values may be:

- BLOCKING-causes the supporting runtime to treat the ProxyGetAsyncResponse call as a blocked call. That is, the ProxyGetAsyncResponse call will not return control to the application until the response to the specified request becomes available.
- NONBLOCKING-causes the supporting runtime to treat the request as a non-blocking call. That is, control is returned to the application, even if the response is not immediately available. In this case, the function returns a return code value of Pending.
- Tran-A pointer to a completed TranEntry data structure. The export view referenced in the specified TranEntry is used by ProxyGetAsyncResponse as the area to be populated with response data from the invoked server.
- param-A pointer to an allocated ProxyParam data structure.
- errMsg-A pointer to a character array buffer that is populated if the asynchronous request encounters a processing problem in either the target server or the supporting runtime.
- errMsgLen-An integer value that specifies the size of the errMsg character array.



## Outputs

Returns an enumerated type `ProxyGetAsyncResponseReturnValue`. The processing of the request determines the return code value that may be:

- `GetAsyncSuccess`-If the supporting runtime determines that the request is satisfied with a response from the server, the call is completed by populating the export data in the specified `CITranEntry`, and the function returns a value of `Success`. The status of the outstanding request is changed to complete.
- `GetAsyncPending`-This code may be returned if the blocking parameter is set to `NONBLOCKING` and the response is not available.
- `GetAsyncInvalidAsyncRequestID`-The specified request identifier does not correspond to an outstanding request. This may be because the ID does not exist, or because the response was already retrieved.
- `GetAsyncServerError` `ServerError`-If the runtime determines that the server encountered an error while processing the request, the call is completed by populating the `errMsg` field with the returned server error message, and the function returning a value of `ServerError`. The status of the outstanding request is changed to complete.
- `GetAsyncCommunicationsError` `CommunicationsError`-If, after the request is accepted, the supporting runtime encounters a communications error servicing the request, the call is completed by populating the `errMsg` field with text describing the communication error and the function returning a value of `CommunicationsError`. The status of the outstanding request is changed to complete.
- `GetAsyncProcessingError`-The proxy runtime was unable to process the request, but the error does not fall into one of the above categories.

## Related Functions

- `ProxyExecuteAsync`
- `ProxyCheckAsync`
- `ProxyIgnoreAsync`

## Example

```
ProxyGetAsyncResponseReturnValue ReturnVal;
ReturnVal = ProxyGetAsyncResponse(requestID, blocking, &SERVER_DETAIL_DIVISION,
 psParamBlock, errMsg, errMsgLen);
if ((ReturnVal != GetAsyncSuccess) && (ReturnVal != GetAsyncPending))
{
 printf("Unsuccessful return from ProxyGetAsyncResponse\n");
 printf("errMsg = %s\n", errMsg);
 ProxyTraceOut(NULL, errMsg);
 return (Failure);
}
```

## ProxyGetClientPassword

### Function Call

```
const char *ProxyGetClientPassword(ProxyParam *param);
```

### Description

This function returns the value of the Password part of the parameter block. This function call is optional.

### Inputs

The parameter block structure created by a call to ProxyAllocateParamBlock.

### Outputs

Pointer to a string containing the Client Password or NULL if not set.

### Related Functions

- ProxyClearParamBlock
- ProxySetClientPassword
- ProxyClearClientPassword

### Example

```
strcpy(OutClientPassword, ProxyGetClientPassword(psParamBlock));
```

## ProxyGetClientUserid

### Function Call

```
const char *ProxyGetClientUserid(ProxyParam *param);
```

### Description

Returns the value of the Userid part of the parameter block. This function call is optional.

### Inputs

The parameter block structure created by a call to ProxyAllocateParamBlock.

## Outputs

Pointer to a string containing the Client Userid or NULL if not set.

## Related Functions

- ProxyClearParamBlock
- ProxySetClientUserid
- ProxyClearClientUserid

## Example

```
strcpy(OutClientUserid,ProxyGetClientUserid(psParamBlock));
```

# ProxyGetCommand

## Function Call

```
const char *ProxyGetCommand(ProxyParam *param);
```

## Description

Returns the value of the Command part of the parameter block. This function call is optional.

## Inputs

The parameter block structure created by a call to ProxyAllocateParamBlock.

## Outputs

Pointer to a string containing the Command or NULL if not set.

## Related Functions

- ProxyClearParamBlock
- ProxySetCommand
- ProxyClearCommand

## Example

```
strcpy(OutCommand, ProxyGetCommand(psParamBlock));
```

## ProxyGetDialect

### Function Call

```
const char *ProxyGetDialect(ProxyParam *param);
```

### Description

Returns the value of the Dialect part of the parameter block. This function call is optional.

### Inputs

The parameter block structure created by a call to ProxyAllocateParamBlock.

### Outputs

Pointer to a string containing the Dialect or NULL if not set.

### Related Functions

- ProxyClearParamBlock
- ProxySetDialect
- ProxyClearDialect

### Example

```
strcpy(OutDialect, ProxyGetDialect(psParamBlock));
```

## ProxyGetExitMsgType

### Function Call

```
MsgType ProxyGetExitMsgType(ProxyParam *param);
```

### Description

Returns the value of the Exit Message Type part of the parameter block. This function call is optional, but recommended. The return value from ProxyExecute indicates basic success or failure. Detailed server errors are returned using ProxyGetExitMsgType, ProxyGetExitStateMsg, and ProxyGetExitStateNum.

### Inputs

The parameter block structure created by a call to ProxyAllocateParamBlock.

## Outputs

Exit Message Type, that is of the enumerated type `MsgType` {`ESInformational`, `ESWarning`, `ESError`, `ESNone`}.

## Related Functions

- `ProxyClearParamBlock`
- `ProxyClearExitMsgType`
- `ProxyGetExitStateMsg`
- `ProxyClearExitStateMsg`
- `ProxyGetExitStateNum`
- `ProxyClearExitStateNum`

## Example

```
OutExitMsgType = ProxyGetExitMsgType(psParamBlock);
```

## ProxyGetExitStateMsg

### Function Call

```
const char *ProxyGetExitStateMsg(ProxyParam *param);
```

### Description

Returns the value of the Exit State Message part of the parameter block. A status message is associated with a status code, and can be returned by A CA Gen exit state. This function call is optional, but recommended. The return value from `ProxyExecute` indicates basic success or failure. Detailed server errors are returned using `ProxyGetExitMsgType`, `ProxyGetExitStateMsg`, and `ProxyGetExitStateNum`.

### Inputs

The parameter block structure created by a call to `ProxyAllocateParamBlock`.

### Outputs

Pointer to a string containing the Exit State Message or `NULL` if not set.

## Related Functions

- `ProxyClearParamBlock`
- `ProxyClearExitStateMsg`

- ProxyGetExitMsgType
- ProxyClearExitMsgType
- ProxyGetExitStateNum
- ProxyClearExitStateNum

### Example

```
strcpy(OutExitStateMsg, ProxyGetExitStateMsg(psParamBlock));
```

## ProxyGetExitStateNum

### Function Call

```
int ProxyGetExitStateNum(ProxyParam *param);
```

### Description

Returns the value of the Exit State Number part of the parameter block. If the server of the procedure step uses exit states, then the exit state is returned from the server. This function call is optional, but recommended. The return value from ProxyExecute indicates basic success or failure. Detailed server errors are returned using ProxyGetExitMsgType, ProxyGetExitStateMsg, and ProxyGetExitStateNum.

### Inputs

The parameter block structure created by a call to ProxyAllocateParamBlock.

### Outputs

An integer value representing an Exit State Number or NULL if not set.

### Related Functions

- ProxyClearParamBlock
- ProxyClearExitStateNum
- ProxyGetExitStateMsg
- ProxyClearExitStateMsg
- ProxyGetExitMsgType
- ProxyClearExitMsgType

### Example

```
OutExitStateNum = ProxyGetExitStateNum(psParamBlock);
```

## ProxyGetNextLocation

### Function Call

```
const char *ProxyGetNextLocation(ProxyParam *param);
```

### Description

Returns the value of the Next Location part of the parameter block. This function call is optional.

### Inputs

The parameter block structure created by a call to ProxyAllocateParamBlock.

### Outputs

Pointer to a string containing the Next Location value or NULL if not set.

### Related Functions

- ProxyClearParamBlock
- ProxySetNextLocation
- ProxyClearNextLocation

### Example

```
strcpy(OutNextLocation, ProxyGetNextLocation(psParamBlock));
```

## ProxyIgnoreAsyncResponse

### Function Call

```
ProxyIgnoreAsyncResponseReturnValue
ProxyIgnoreAsyncResponse(AsyncRequestID
requestID);
```

### Description

Informs the runtime that it wants to ignore the response corresponding to the specified AsyncRequestID. Once ignored, the outstanding request is considered complete.

The resources associated with an ignored request remain allocated as long as the corresponding response of the ignored request remains outstanding. The runtime tracks an ignored request, and only releases the resources when the runtime receives the corresponding response.

**Note:** This function is not supported for communication type of Tuxedo.

### Inputs

*RequestID*-An AsyncRequest ID field that contains a unique ID associated with the outstanding asynchronous request that the application wants ignored.

### Outputs

Returns an enumerated type ProxyIgnoreAsyncResponseReturnValue. The return value may be:

- IgnoreAsyncResponseIgnored-The supporting runtime determines that the request is satisfied with a response from the server and the application is no longer obligated to retrieve the response. The status of the outstanding request is changed to complete.
- IgnoreAsyncInvalidAsyncRequestID-The specified request identifier does not correspond to an outstanding request. This may be because the ID does not exist, or because the response was already retrieved.
- IgnoreAsyncProcessingError-The proxy runtime was unable to process the IgnoreAsync request, but the error does not fall into one of the above categories.

### Related Functions

- ProxyExecuteAsync
- ProxyCheckAsync
- ProxyIgnoreAsyncResponse

### Example

```
ProxyIgnoreAsyncResponseReturnValue ReturnVal;
ReturnVal = ProxyIgnoreAsyncResponse(requested);
If (ReturnVal != IgnoreAsyncResponseIgnored)
{
 printf("Unsuccessful return from ProxyIgnoreAsyncResponse \n");
 return (Failure);
}
```



## ProxySetClientPassword

### Function Call

```
int ProxySetClientPassword(ProxyParam *param, char *ClientPassword);
```

### Description

Sets the client user password part of the parameter block sent to the server where the executable code of the procedure step is installed. A client user password is usually accompanied by a client user ID that is set by the ProxySetClientUserid function. This function call is optional.

**Note:** Security is not enabled by default. For a description of the user exits invoked from the C Proxy Runtime, see the *Distributed Processing - Overview Guide*.

### Inputs

The parameter block structure created by a call to ProxyAllocateParamBlock.

*ClientPassword*-Password that is valid at the server.

### Outputs

Returns an integer value that is one of the enumerated types of ProxyExecuteReturnValues. 0==Failure, 1== Success.

### Related Functions

- ProxyClearParamBlock
- ProxyGetClientPassword
- ProxyClearClientPassword

### Example

```
int ReturnVal;
ReturnVal = ProxySetClientPassword(psParamBlock, ClientPassword);
if (ReturnVal == Failure)
{
 printf("Unsuccessful return from ProxySetClientPassword\n");
 return (Failure);
}
```

## ProxySetClientUserid

### Function Call

```
int ProxySetClientUserid(ProxyParam *param, char *ClientUserid);
```

### Description

Sets the client user ID part of the parameter block sent to the server where the executable code of the procedure step is installed. A client user ID is usually accompanied by a client user password that is set with the ProxySetClientPassword function. This function call is optional.

**Note:** Security is not enabled by default. For a description of the user exits invoked from the C Proxy Runtime, see the *Distributed Processing - Overview Guide*.

### Inputs

The inputs to this function are:

- The parameter block structure created by a call to ProxyAllocateParamBlock.
- Client Userid

### Outputs

Returns an integer value that is one of the enumerated types of ProxyExecuteReturnValues. 0==Failure, 1== Success.

### Related Functions

- ProxyClearParamBlock
- ProxyGetClientUserid
- ProxyClearClientUserid

### Example

```
int ReturnVal;
ReturnVal = ProxySetClientUserid(psParamBlock, ClientUserid);
if (ReturnVal == Failure)
{
 printf("Unsuccessful return from ProxySetClientUserid\n");
 return (Failure);
}
```

## ProxySetCommand

### Function Call

```
int ProxySetCommand(ProxyParam *param, char *Command);
```

### Description

Sets the value of the Command part of the parameter block. This function should only be set if the procedure step uses case of command construct. This function call is optional.

### Inputs

The inputs to this function are:

- The parameter block structure created by a call to ProxyAllocateParamBlock.
- Command to be sent to the server. Must be in uppercase.

### Outputs

Returns an integer value that is one of the enumerated types of ProxyExecuteReturnValues. 0==Failure, 1== Success.

### Related Functions

- ProxyClearParamBlock
- ProxyGetCommand
- ProxyClearCommand

### Example

```
int ReturnVal;
ReturnVal = ProxySetCommand(psParamBlock, Command);
if (ReturnVal == Failure)
{
 printf("Unsuccessful return from ProxySetCommand\n");
 return (Failure);
}
```

## ProxySetDialect

### Function Call

```
int ProxySetDialect(ProxyParam *param, char *Dialect);
```

### Description

Sets the value of the Dialect part of the parameter block. This function call is optional.

### Inputs

The inputs to this function are:

- The parameter block structure created by a call to ProxyAllocateParamBlock.
- Dialect

### Outputs

Returns an integer value that is one of the enumerated types of ProxyExecuteReturnValues. 0==Failure, 1== Success.

### Related Functions

- ProxyClearParamBlock
- ProxyGetDialect
- ProxyClearDialect

### Example

```
int ReturnVal;
ReturnVal = ProxySetDialect(psParamBlock, Dialect);
if (ReturnVal == Failure)
{
 printf("Unsuccessful return from ProxySetDialect\n");
 return (Failure);
}
```

## ProxySetNextLocation

### Function Call

```
int ProxySetNextLocation(ProxyParam *param, char *NextLocation);
```

### Description

Sets the value of the Next Location part of the parameter block. The location name (NEXTLOC) may be used by user exit flow DLLs. This function call is optional.

## Inputs

The inputs to this function are:

- The parameter block structure created by a call to ProxyAllocateParamBlock.
- Next Location

## Outputs

Returns an integer value that is one of the enumerated types of ProxyExecuteReturnValues. 0==Failure, 1== Success.

## Related Functions

- ProxyClearParamBlock
- ProxyGetNextLocation
- ProxyClearNextLocation

## Example

```
int ReturnVal;
ReturnVal = ProxySetNextLocation(psParamBlock, NextLocation);
if (ReturnVal == Failure)
{
 printf("Unsuccessful return from ProxySetNextLocation\n");
 return (Failure);
}
```

## ProxyStartTracing

### Function Call

```
void ProxyStartTracing(char *, char *);
```

### Description

Opens a file for debug and status information. ProxyStopTracing is used to close the file. This function is optional but, if used, must be called before ProxyTraceOut.

### Inputs

The inputs to this function are:

- The first parameter can be either a valid filename or NULL. If a filename is provided it will be used to create the log file if file access permissions allow. If this parameter is NULL then the trace output will be written to a file called "trace-<procname>-<procid>.out" (where <procname> is the name of the proxy application and <procid> is the application's process id). On a Windows platform this file will be written to the %USERPROFILE%\AppData\Local\CA\Genxx\logs\client directory. For UNIX and Linux systems, the file is located in the current directory. An empty string "" for the first parameter will result in no file being created.
- The second parameter is the trace level.
- 0xFFFFFFFF requests the highest (verbose) level of messages.
- NULL indicates that the value of the CMIDEBUG environment variable should be used to determine the trace level.

Functions that return a Failure status, generally write detailed information regarding the failure to the trace file.

### Outputs

None

### Related Functions

- ProxyStopTracing
- ProxyTraceOut

### Example

```
ProxyStartTracing("cproxy.out", "0xFFFFFFFF");
```

## ProxyStopTracing

### Function Call

```
void ProxyStopTracing();
```

### Description

Closes the trace file opened by ProxyStartTracing. This function is optional, but is recommended if ProxyStartTracing is used.

## Inputs

None

## Outputs

None

## Related Functions

- ProxyStartTracing
- ProxyTraceOut

## Example

```
ProxyStopTracing();
```

## ProxyTraceOut

### Function Call

```
void ProxyTraceOut(void *, char *);
```

## Description

Writes a message to the trace file. These messages are interspersed with standard debugging messages. This function call is optional.

## Inputs

The inputs to this function are:

- First parameter is NULL.
- Second parameter is the Message to be written to the trace file.

## Outputs

None

## Related Functions

- ProxyStartTracing
- ProxyStopTracing

## Example

```
ProxyTraceOut(NULL, "CPROXY.C - ***** Beginning program");
```

## ProxySetViewBlob

### Function Call

```
int ProxySetViewBlob(void *, void *, long);
```

### Description

Sets the BLOB predicate view to input data. Writes the input data to BLOB.

### Inputs

The inputs to this function are:

- First parameter is the pointer to the BLOB predicate view.
- Second parameter is the input data buffer.
- Third parameter is the input data length.

### Output

Returns an integer value that is one of the enumerated types of ProxyViewReturnValues. 0==Failure, 1== Success.

## ProxyGetViewBlob

### Function Call

```
int ProxyGetViewBlob(void *, void *, long, long *);
```

### Description

Gets the BLOB predicate view data into output buffer. Reads the BLOB data into buffer.

### Inputs

The inputs to this function are:

- First parameter is the pointer to the BLOB predicate view.
- Second parameter is the output data buffer. This parameter is the output parameter where the output data is stored
- Third parameter is the output data buffer length.
- Fourth parameter is the number of bytes returned in data buffer. This parameter is the output parameter which contains the number of bytes.



## Output

Returns an integer value that is one of the enumerated types of ProxyViewReturnValues. 0==Failure, 1== Success.

## ProxyClearViews

### Function Call

```
int ProxyClearViews (CITranEntry *);
```

### Description

Clears the view data. This function helps ensure that BLOB handles involved in the transaction are properly deallocated. Though this function is optional, it is recommended that this function is called at the end of every transaction.

**Note:** Current implementation only resets the BLOB handles.

### Inputs

The first input parameter of the function is the Transaction Entry data.

## Output

Returns an integer value that is one of the enumerated types of ProxyViewReturnValues. 0==Failure, 1== Success.

## Proxy.c

The following is an example of a C proxy program using Decimal Precision attribute. This code is hand-written not generated.

```
/*
 * This sample's purpose is to provide a C programmer a starting
 * point for coding a C proxy routine.
 *
 * This sample builds upon the CA (R) Gen sample model that is included
 * with the CA Gen Toolset. The Gen server module used by this proxy
 * example is packaged within the sample model as load module P900. This proxy
 * example will only exercise a small amount of the sample models capabilities.
 *
 * This example application performs the following actions:
 * - delDiv - deletes 4 division name records based on data specified in
 * the import view. Implemented in the delete() function.
 * - addDiv - adds four division name records based on data specified in the
 * import view. Implemented in the add() function.
 * - readDiv - retrieves division name records and moves them to the export
 * repeating group view. Implemented in the read function.
 * - check - the check function extracts data from the returned export
 * view depending upon the delete/add/read operation just executed
 *
 * - The import view assignment statements are hard coded. In your
 * application you will probably use a GUI interface to query for
 * values.
 *
 * - The export views are displayed on the screen. In your application
 * you will probably use a GUI interface to display these values.
 *
 * NOTE: Variable Name Case Sensitivity
 *
 * The C Proxy API uses case sensitivity to use the procedure step name
 * as multiple variable names. If you use this sample C program as a basis
 * for your program, take care to "match case" when replacing variable names.
 *
 * For example:
 *
 * - SERVER_DETAIL_DIVISION is used to represent the TranEntry data structure
 * which contains all the information needed for a transaction (such as
 * trancode and model name).
 *
 * - server_detail_division is used to represent the transaction specific
 * view data structure which will be used to reference import and export
 * views.
 *
 * This proxy example is designed to communicate with a server packaged
```

```
* within the CA Gen sample model which is included with the product
* distribution. The sample model's Cooperative Server load module P900
* must be generated and installed within the target environment to which
* this C proxy is to communicate, prior to attempting to execute this
* sample.
*/

/* Include the Gen C Proxy generated header file created from the
* sample model included with the product distribution
*/

#include "p900.h"

/* the following value is one of the returned exit state values. It's
* definition was obtained from server generated code. Additional exit state
* values could be obtained if desired.
*/
#define gui_return_from_link 219941543

/* Include other standard libraries as needed */
#include <ctype.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef WIN32
/* for Windows */
#include <windows.h>
#define SLEEP(i) Sleep(i*1000)
#else
/* for UNIX */
#include <unistd.h>
#define SLEEP(i) sleep(i)
#endif

/* Contains communication information */
static ProxyParam* psParamBlock;

/* Used to store a communication configuration string */
char comcfg[256];

/* Error message length can be from 0-2048 this will determine
* how much of the error message will be returned by ProxyExecute.
* A shorter message may be returned depending on the error.
*/
#define errMsgLen 2048
char errMsg[errMsgLen];
```

```
/* Return code from routines {Failure, Success}
 * ProxyExecuteReturnValues is defined in the proxyexe.h file supplied
 * with the Gen installation.
 */
ProxyExecuteReturnValues ReturnVal;

/* Misc defines */
int nRC; /* Integer return code */

char ClientUserid[ProxyClientUseridLen+1]= "MYUSER";
char ClientPassword[ProxyClientPasswordLen+1]= "123";
char Dialect[ProxyDialectLen + 1] = "DEFAULT";
char NextLocation[ProxyNextLocationLen+1] = "TEST";
/* Command is one of A (add), D (delete), R (read) */
char Command[ProxyCommandLen+1] = " ";
long ExitStateNum;
MsgType ExitMsgType;
char ExitStateMsg[ProxyExitStateMsgLen+1];
char OutClientUserid[ProxyClientUseridLen+1];
char OutClientPassword[ProxyClientPasswordLen+1];
char OutDialect[ProxyDialectLen + 1];
char OutNextLocation[ProxyNextLocationLen+1];
char OutCommand[ProxyCommandLen+1];
long OutExitStateNum;
MsgType OutExitMsgType;
char OutExitStateMsg[ProxyExitStateMsgLen+1];

/* local function prototypes */
ProxyExecuteReturnValues addDiv();
ProxyExecuteReturnValues delDiv();
ProxyExecuteReturnValues readDiv();
void check();

int main (void)
{

 printf ("Beginning of sample C proxy application \n\n");

 /*-----*/
 /* Setup */
 /*-----*/

 /* ProxyStartTracing opens a file for debug/status information.
 * ProxyStopTracing will be used to close the file.
 * This function is optional, but must be called before ProxyTraceOut
 * if used
 */
}
```

```
* - The first parameter must be valid filename, the file will be
* created
* - The second parameter is the trace level.
* 0xFFFFFFFF requests the highest (verbose) level of messages.
* NULL indicates that the value of the CMIDEBUG environment
* variable should be used to determine the trace level
* Functions which return with a Failure status will generally write
* detailed information regarding the failure to the trace file.
*/

ProxyStartTracing((char *) "cproxy.out", (char*)"0xFFFFFFFF");

/* ProxyTraceOut writes a message to the trace file. These
* messages will be interspersed with standard debugging messages.
* This function call is optional.
*/

ProxyTraceOut(NULL, (char *) "CPROXY.C - ***** Beginning program");

/* ProxyAllocateParamBlock is used to create the parameter block
* (data structure) which contains information required for the
* transaction. A NULL return indicates an error in the creation
* of the parameter block.
* This function call is required.
*/

psParamBlock = ProxyAllocateParamBlock();
if (NULL == psParamBlock)
{
 printf("Unsuccessful return from ProxyAllocateParamBlock\n");
 return (Failure);
}

/* ProxySetClientUserid is used to set a userid in the parameter block.
* This function call is required if security on the server is used,
* otherwise it is optional.
*/

nRC = ProxySetClientUserid(psParamBlock, ClientUserid);
if (nRC == Failure)
{
 printf("Unsuccessful return from ProxySetClientUserid\n");
 return (Failure);
}

/* ProxySetClientPassword is used to set a password in the parameter
```

```
* block.
* This function call is required if security on the server is used,
* otherwise it is optional.
*/

 nRC = ProxySetClientPassword(psParamBlock, ClientPassword);
 if (nRC == Failure)
 {
 printf("Unsuccessful return from ProxySetClientPassword\n");
 return (Failure);
 }

/* ProxySetDialect is used to set the dialect in the parameter block.
* This function call is optional.
*/

 nRC = ProxySetDialect(psParamBlock, Dialect);
 if (nRC == Failure)
 {
 printf("Unsuccessful return from ProxySetDialect\n");
 return (Failure);
 }

/* ProxySetNextLocation is used to set the next location in the parameter
* block.
* This function call is optional.
*/

 nRC = ProxySetNextLocation(psParamBlock, NextLocation);
 if (nRC == Failure)
 {
 printf("Unsuccessful return from ProxySetNextLocation\n");
 return (Failure);
 }

/* ProxySetCommand is used to set the command in the parameter block.
* This function call is optional.
*/

 nRC = ProxySetCommand(psParamBlock, Command);
 if (nRC == Failure)
 {
 printf("Unsuccessful return from ProxySetCommand\n");
 return (Failure);
 }

#ifdef TUXEDO
/* NOTE: ProxyConfigureComm() must not be called if targeting a Tuxedo Server
* environment. Tuxedo environment does not support retargeting the server.
```

```

*/

/* ProxyConfigureComm is used to set communication options.
 * This function call is optional, communication options can be set in
 * the following ways:
 * - by specifying the options in the SERVER_DETAIL_DIVISION structure
 * without a call to ProxyConfigureComm
 * - by calling ProxyConfigureComm with a NULL third parameter to
 * indicate that the file commcfg.ini should be used
 * - by calling ProxyConfigureComm with a string containing the
 * Comm Service Type, Host and Port as the third parameter
 */

/* This call to ProxyConfigureComm will use the file commcfg.ini
 * NOTE: the commcfg.ini file must contain a valid override entry
 * if this call is to succeed.
 */

 nRC = ProxyConfigureComm(&SERVER_DETAIL_DIVISION,
SERVER_DETAIL_DIVISION.service, NULL);
 if (nRC == FALSE)
 {
 printf("Unsuccessful return from ProxyConfigureComm using File\n");
 return (Failure);
 }

/* This call to ProxyConfigureComm will use the string containing the
 * Comm Service Type, Host and Port to set communication
 * options. Be sure to include a space between each token.
 */

 strcpy(comcfg, "TCP "); /* Comm Service Type */
 strcat(comcfg, "hostname "); /* Host */
 strcat(comcfg, "port#"); /* Port */

 nRC = ProxyConfigureComm(&SERVER_DETAIL_DIVISION,
SERVER_DETAIL_DIVISION.service, comcfg);

 if (nRC == FALSE)
 {
 printf("Unsuccessful return from ProxyConfigureComm using comcfg string\n");
 return (Failure);
 }
#endif /* ! TUXEDO */

/* The ProxyClear set of functions are used to clear the parameter block.
 * Each part of the parameter block can be cleared individually, or the
 * ProxyClearParamBlock function can be used to clear the entire

```

```
* parameter block.
* These function calls are optional.
* The ProxyClearParamBlock function would replace the other ProxyClear
* functions shown further below.
*/
 nRC = ProxyClearParamBlock(psParamBlock);
 if (nRC == Failure)
 {
 printf("Unsuccessful return from ProxyClearParamBlock\n");
 return (Failure);
 }

/* Function specific to this proxy example
* Deletes the division name records which will be added by the addDiv
* Note: delete will call ProxyClearParamBlock
*/
 delDiv();

 SLEEP(2);

/* Function specific to this proxy example
* add several divisions
* Note: add will call ProxyClearParamBlock
*/
 addDiv();

 SLEEP(2);

/* Function specific to this proxy example
* read back the divisions just added
* Note: read will call ProxyClearParamBlock
*/
 readDiv();

/* For sample purposes, the set of individual ProxyClear functions are shown
* below
*/
 nRC = ProxyClearClientUserid(psParamBlock);
 if (nRC == Failure)
 {
 printf("Unsuccessful return from ProxyClearClientUserid\n");
 return (Failure);
 }

 nRC = ProxyClearClientPassword(psParamBlock);
 if (nRC == Failure)
 {
```



```
 printf("Unsuccessful return from ProxyClearClientPassword\n");
 return (Failure);
 }

 nRC = ProxyClearDialect(psParamBlock);
 if (nRC == Failure)
 {
 printf("Unsuccessful return from ProxyClearDialect\n");
 return (Failure);
 }

 nRC = ProxyClearNextLocation(psParamBlock);
 if (nRC == Failure)
 {
 printf("Unsuccessful return from ProxyClearNextLocation\n");
 return (Failure);
 }

 nRC = ProxyClearCommand(psParamBlock);
 if (nRC == Failure)
 {
 printf("Unsuccessful return from ProxyClearCommand\n");
 return (Failure);
 }

 nRC = ProxyClearExitStateMsg(psParamBlock);
 if (nRC == Failure)
 {
 printf("Unsuccessful return from ProxyClearExitStateMsg\n");
 return (Failure);
 }

 nRC = ProxyClearExitStateNum(psParamBlock);
 if (nRC == Failure)
 {
 printf("Unsuccessful return from ProxyClearExitStateNum\n");
 return (Failure);
 }

 nRC = ProxyClearExitMsgType(psParamBlock);
 if (nRC == Failure)
 {
 printf("Unsuccessful return from ProxyClearExitMsgType\n");
 return (Failure);
 }

 /* The ProxyDeleteParamBlock function is used to delete the psParamBlock
 * structure.
 * This function call is optional, but recommended.
```

```
*/
 nRC = ProxyDeleteParamBlock(psParamBlock);
 if (nRC == Failure)
 {
 printf("Unsuccessful return from ProxyDeleteParamBlock\n");
 return (Failure);
 }

/* ProxyStopTracing is used to close the trace file opened by
 * ProxyStartTracing.
 * This function is optional, but is recommended if ProxyStartTracing is
 * used.
 */
 ProxyStopTracing();

 printf ("\nEnding program cproxy \n");
 return (TRUE);
}

/* addDiv()
 * This function is specific to this proxy example.
 * It attempts to add 4 division records
 */

ProxyExecuteReturnValues addDiv()
{
 /* Add Divisions */
 int count;

 printf("Adding test divisions\n");

 strcpy(Command,"A"); /* Add */
 strcpy(server_detail_division.importRequest_command.value, Command);

 /* add 4 Divisions */
 for (count = 0; count <= 3; count++)
 {
 /* use Numbers 90-93 */
 server_detail_division.import_division.number = 90+count;
 sprintf(server_detail_division.import_division.name, "Test Divison %d",
count);

 /* in case there are no employes in the database use 0*/
 server_detail_division.import_employee.number = 0;

 /* clear the parameter block */
 }
}
```

```

 nRC = ProxyClearParamBlock(psParamBlock);
 if (nRC == Failure)
 {
 printf("Unsuccessful return from ProxyClearParamBlock\n");
 return (Failure);
 }

/* Transaction processing
 * ProxyExecute sends the transaction to the server and returns export
 * data and messages. This function use the following parameters:
 * - SERVER_DETAIL_DIVISION transaction structure which contains
 * import and export views
 * - psParamBlock parameter structure which contain communication
 * information
 * - errMsg will return any error message from the server
 * - errMsgLen specifies the number of characters to write to errMsg
 * This function call is required. The return value is Success or
 * Failure
 */
 if (ProxyExecute(&SERVER_DETAIL_DIVISION, psParamBlock, errMsg, errMsgLen) !=
Success)
 {
 printf("Unsuccessful return from ProxyExecute \n");
 printf("errMsg = %s\n",errMsg);
 ProxyTraceOut(NULL, errMsg);
 return (Failure);
 }
 else /* check results */
 check();
}

return (Success);
}

/* readDiv()
 * This function is specific to this proxy example.
 * read all divisions, returning division name and number
 * The server returns the data in a repeating group view.
 * If the previous add was successful there should be
 * at least four divisions present. The check() function
 * will extract the data read from the repeating group view.
 */
ProxyExecuteReturnValues readDiv()
{
 printf("Reading test divisions\n");

#ifdef TUXEDO
/* NOTE: ProxyConfigureComm() must not be called if targeting a Tuxedo Server

```

```
 * environment. Tuxedo environment does not support retargeting the server.
*/

/* setup to use SERVER_MAINTAIN_DIVISION
 * This read uses a different procedure step than did delDiv/addDiv, so
 * must call ProxyConfigureComm() to initialize for the desired
 * procedure step
 */
nRC = ProxyConfigureComm(&SERVER_MAINTAIN_DIVISION,
 SERVER_MAINTAIN_DIVISION.service, comcfg);

if (nRC == FALSE)
{
 printf("Unsuccessful return from ProxyConfigureComm using comcfg string\n");
 return (Failure);
}

#endif /* !TUXEDO */

strcpy(Command, "R");

/* non space implies server is to return sorted data */
strcpy(server_maintain_division.importSortWrk_sorting.sort_sequence, "S");

/* clear the parameter block */
nRC = ProxyClearParamBlock(psParamBlock);
if (nRC == Failure)
{
 printf("Unsuccessful return from ProxyClearParamBlock\n");
 return (Failure);
}

/* Transaction processing
 * ProxyExecute sends the transaction to the server and returns export
 * data and messages. This function use the following parameters:
 * - SERVER_MAINTAIN_DIVISION transaction structure which contains
 * import and export views
 * - psParamBlock parameter structure which contain communication
 * information
 * - errMsg will return any error message from the server
 * - errMsgLen specifies the number of characters to write to errMsg
 * This function call is required. The return value is Success or
 * Failure
 */
if (ProxyExecute(&SERVER_MAINTAIN_DIVISION, psParamBlock,
 errMsg, errMsgLen) != Success)
{
 printf("Unsuccessful return from ProxyExecute \n");
 printf("errMsg = %s\n", errMsg);
}
```

```
 ProxyTraceOut(NULL, errMsg);
 return (Failure);
 }
 check();

 return (Success);
}

/* delDiv()
 * This function is specific to this proxy example.
 * Attempts to delete the four division records that addDiv() will then add.
 * Hard coded to delete division 90-93, which may or may not exist.
 */
ProxyExecuteReturnValues delDiv()
{
 int count;

 printf("Deleting test divisions\n");
 strcpy(Command, "D"); /* Delete */

 strcpy(server_detail_division.importRequest_command.value, Command);

 for (count = 0; count <=3; count++)
 {
 server_detail_division.import_division.number = 90 + count;
 printf(" Attempting to Delete Division %d\n",
 server_detail_division.import_division.number);

 nRC = ProxyClearParamBlock(psParamBlock);
 if (nRC == Failure)
 {
 printf("Unsuccessful return from ProxyClearParamBlock\n");
 return (Failure);
 }
 }

/* Transaction processing
 * ProxyExecute sends the transaction to the server and returns export
 * data and messages. This function use the following parameters:
 * - SERVER_DETAIL_DIVISION transaction structure which contains
 * import and export views
 * - psParamBlock parameter structure which contain communication
 * information
 * - errMsg will return any error message from the server
 * - errMsgLen specifies the number of characters to write to errMsg
 * This function call is required. The return value is Success or
 * Failure
 */
```

```
 if (ProxyExecute(&SERVER_DETAIL_DIVISION, psParamBlock, errMsg, errMsgLen) !=
Success)
 {
 printf("Unsuccessful return from ProxyExecute \n");
 printf("errMsg = %s\n",errMsg);
 ProxyTraceOut(NULL, errMsg);
 return (Failure);
 }
 }
 printf("\n");
 return (Success);
}

/* check()
 * This function is specific to this proxy example.
 * check the results of the ProxyExecute()
 */
void check()
{
 int loop=0;

 /* The ProxyGet set of functions are used to query the various data items
 * within parameter block.
 * These function calls are optional.
 */
 strcpy(OutClientUserid,ProxyGetClientUserid(psParamBlock));
 strcpy(OutClientPassword, ProxyGetClientPassword(psParamBlock));
 strcpy(OutDialect, ProxyGetDialect(psParamBlock));
 strcpy(OutNextLocation, ProxyGetNextLocation(psParamBlock));
 strcpy(OutCommand, ProxyGetCommand(psParamBlock));
 OutExitStateNum = ProxyGetExitStateNum(psParamBlock);
 OutExitMsgType = ProxyGetExitMsgType(psParamBlock);
 strcpy(OutExitStateMsg, ProxyGetExitStateMsg(psParamBlock));

 /* processing a delete command */
 if (strcmp(Command, "D") == 0)
 {

 /* did server set an exit state? If so, delete failed */
 if (OutExitStateNum != 0)
 {
 printf("OutExitStateNum=%d\n", OutExitStateNum);
 printf("OutExitStateMsg=%s\n", OutExitStateMsg);
 }
 else
 printf("Delete successful\n");
 return;
 }
}
```

```

/* processing the add command */
if (strcmp(Command, "A") == 0)
{
 if (OutExitStateNum == gui_return_from_link)
 {
 printf(" Division successfully added\n");
 printf(" Exit State Number | Division: Number Name\n");
 printf("-----\n");
 printf(" %d", OutExitStateNum);
 printf("\t\t\t\t %d", server_detail_division.export_division.number);
 printf(" %s\n\n", server_detail_division.export_division.name);
 }
 else /* error */
 {
 printf("OutExitStateNum=%d\n", OutExitStateNum);
 printf("OutExitStateMsg=%s\n", OutExitStateMsg);
 return;
 }
}

/* processing the read repeating group view */
if (strcmp(Command, "R") == 0)
{
 printf(" List of Divisions\n");
 if (server_maintain_division.allOutput.cardinality > 0)
 {
 printf(" Number Name\n");
 printf("-----\n");
 for (loop = 0; loop < server_maintain_division.allOutput.cardinality;
loop++)
 {
 printf(" %d",
server_maintain_division.allOutput.export_division.number[loop]);
 printf(" %s\n",
server_maintain_division.allOutput.export_division.name[loop]);
 }
 }
 else
 printf("\tNo Data Found\n");
}
}

```

## Using a C Proxy

A user-written C/C++ application incorporates the use of a C Proxy by using the generated C header file, and calling the various API's provided by the C Proxy Runtime.

A user-written C/C++ application can build applications using the C proxy that are capable of supporting both synchronous and asynchronous cooperative flows. The main difference between these processing types is the actual method used to perform the cooperative flow.

The C Proxy sample code supplied with the product only demonstrates the use of synchronous capabilities.

**More information:**

[Function Calls](#) (see page 175)

## Security Processing

A C Proxy provides facilities to implement Distributed Processing Security as described in the *Distributed Processing - Overview Guide*.

To utilize the security features, the application developer must add code to their user-written C/C++ application that sets the ClientUserid and ClientPassword parameter of the parameter block that is to be sent to the server. Depending on the return value of the client security user exit (WRSECTOKEN), the security data fields will or will not be sent to the target DPS.

If the client security user exit returns SECURITY\_ENHANCED, the client security user exits can cause an optional security token to be added to the data flow. The collection of security data that is sent as part of the cooperative flow is validated by user exits residing in the execution environment of the target DPS.

For user-written C/C++ applications flowing to DPS using TCP/IP, MQSeries, or ECI the supporting runtime lets a portion of the Common Format Buffer to be encrypted on the way to the target DPS, and decrypted on the way back from the target DPS. The use of encryption and decryption are enabled by way of user exits (WRSECENCRYPT and WRSECDECRYPT respectively).

**More information:**

[Configuring the C Proxy at Runtime](#) (see page 227)



## Overview of an Example Makefile for Windows

The steps used to compile and link your C/C++ application vary depending on the environment chosen. The example makefile shown may require modifications to work in your environment.

**Note:** Setup the compiler environment before executing the makefile using %VS100COMNTOOLS%vsvars32.bat for Visual Studio 2010 and %VS110COMNTOOLS%vsvars32.bat for Visual Studio 2012. If building a 64-bit application, you will also need to execute %VSINSTALLDIR%vcvarsall.bat x64.

| Description                                                                                                  | Example Makefile                                                                                                                                                                                                                                                                                                                        |
|--------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Clean up previous files and set command values.                                                              | <pre>ALL : "cproxy.obj" "cproxy.exe" makeok  CLEAN : -@erase "cproxy.obj" -@erase "cproxy.exe" -@erase "cproxy.pdb" -@erase "cproxy.sbr" @echo Example C Proxy program: clean done  CPP=cl.exe CPP_PROJ=/nologo /I"\${IEFH}" -DWIN32 /MD /W3 /Gm /GX /Zi /Od /D "WINDOWS" /D "_DEBUG" /D "_CONSOLE" /D "_MBCS" /FR /Fo /Fd /FD /c</pre> |
| Dependencies - this is definitely not required in the standard makefile, but is shown here for completeness. | <pre>CPP_DEP=\ ".\p900.h"\ "\${IEFH}"\cisrvtcp.h"\ "\${IEFH}"\ciconst.h"\ "\${IEFH}"\citrantb.h"\ "\${IEFH}"\civewdef.h"\ "\${IEFH}"\proxycfg.h"\ "\${IEFH}"\proxyexe.h"\ "\${IEFH}"\proxytrc.h"\ "\${IEFH}"\proxysit.h"</pre>                                                                                                          |
| Compile definition                                                                                           | <pre>.c.obj:: \$(CPP) \$(CPP_PROJ) \$(&lt;R).c</pre>                                                                                                                                                                                                                                                                                    |

| Description                                                                                                                                                                                                                                                                                             | Example Makefile                                                                                                                                                                                       |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Link definition - The libraries you link with are csuxxn.lib and pxrtxxn.lib in the CA Gen directory.<br><b>Note:</b> xx refers to the current release of CA Gen. For the current release number, see the <i>Release Notes</i> .<br><b>Note:</b> YYY refers to either X86 for 32-bit or X64 for 64-bit. | LINK32=link.exe<br><br>LINK32_FLAGS="\$(IEFH)\csuxxn.lib"<br>"\$(IEFH)\pxrtxxn.lib" /nologo\<br>/subsystem:console /incremental:no /debug<br>/machine:YYY/out:"cproxy.exe\<br>LINK32_OBJS="cproxy.obj" |
| Executable definition                                                                                                                                                                                                                                                                                   | "cproxy.exe": \$(LINK32_OBJS) \$(CPP_DEP)<br>\$(LINK32) \$(LINK32_FLAGS) \$(LINK32_OBJS)                                                                                                               |
| Completion message                                                                                                                                                                                                                                                                                      | makeok :<br>@echo C Proxy API program make successful.                                                                                                                                                 |

## Overview of an Example Makefile for UNIX

The steps used to compile and link your C/C++ application are similar on the supported UNIX platforms. Differences do exist in the compiler names used as well as the names of the dependency libraries. The example makefile macros shown may require modifications to work in your environment. Constructs similar to those presented for the Windows makefile above may also be incorporated into makefiles for execution within the UNIX environments but are not presented below.

| Description                                     | Example Makefile                                                                                                                   |
|-------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|
| Compile/link macro for use with IBM AIX         | cproxy: cproxy.c p900.h<br>xlc_r -q64 cplusplus -DUNIX -I\$(IEFH)/include cproxy.c<br>-L\$(IEFH)/lib -l pxrt.xx -o cproxy          |
| Compile/link macro for use with Sun Solaris     | cproxy: cproxy.c p900.h<br>CC -m64 -DUNIX -I\$(IEFH)/include cproxy.c -L\$(IEFH)/lib -l<br>pxrt.xx -o cproxy                       |
| Compile/link macro for use with HP Itanium UNIX | cproxy: cproxy.c p900.h<br>aCC -Y -Aa +Z -Wl,+s -DD64<br>-DUNIX -I\$(IEFH)/include cproxy.c -L\$(IEFH)/lib -l pxrt.xx -o<br>cproxy |

**Note:** xx refers to the current release of CA Gen. For the current release number, see the *Release Notes*.

## Executing the User-written C Proxy Application

Before executing a C application that contains C proxy code, the application's execution environment must be setup. The items needed for the execution environment are contained in the CA Gen install directory.

### Configuring the C Proxy at Runtime

There exist areas within the C Proxy Runtime that can be influenced by an application developer by changing its design. The activities taken by an application developer include:

- Updating the commcfg.ini file to influence the communications part of the runtime.
- Testing applications using the Diagram Trace Utility.
- Modifying user exits. The user exits influence security processing. Additionally, other user exits are invoked as part of the communications runtime. Each communications runtime exit lets arguments specific to that communication runtime to be overridden.

**Note:** For more information about user exits invoked from the C Proxy Runtime, see the *User Exit Reference Guide*.

### Configuring C Proxy Using commcfg.ini

The commcfg.ini file is an ASCII text file that lets runtime modifications to the communication configuration for a given set of transactions codes associated with the target DPS applications.

There is only one copy of the commcfg.ini file, by default. The installed location is the CA Gen installation directory. As there is only one file all proxies must use this same file. Alternatively, a proxy application searches for the commcfg.ini file in the following order:

1. %COMMCFG\_HOME%
2. %USERPROFILE%\AppData\Local\CA\Gen xx\logs\client
3. %ALLUSERSPROFILE%\Gen xx\cfg\client
4. %PATH%

If the file is not available in these directories, the proxy application assumes that the file does not exist.

If there are multiple copies of the commcfg.ini file, the order of precedence for the files is the same as the order described above.

The following list defines the use of this search order:

### **%COMMCFG\_HOME%**

The environment variable COMMCFG\_HOME can be set to a directory that contains the commcfg.ini file. This file is used by the invoking proxy application. This variables setting enables customization on a proxy by proxy basis.

### **%USERPROFILE%\AppData\Local\CA\Gen xx\cfg\client**

If the commcfg.ini file is stored in this directory, the file is accessible by all proxies that are executed by a *specific* user on the system.

### **%ALLUSERSPROFILE%\Gen xx\cfg\client**

If the commcfg.ini file is stored in this directory, the file is accessible by all proxies that are executed by *all* users on the system.

### **%PATH%**

If the commcfg.ini file is not available in the above locations, the file is searched in all the directories that are specified by the PATH environment variable. By default, CA Gen is part of the PATH environment variable.

A generated proxy contains communications information as defined in the model. The proxy runtime provides a capability for specifying the communication information at runtime. This information overrides any information specified in the model and built into the proxy during generation.

In addition, the proxy runtime has the capability of turning tracing ON and OFF, as well as providing a capability for controlling the amount of file caching that takes place at runtime.

**Note:** For specific formatting instructions, see the default file installed with the CA Gen Proxy software, or see the *Distributed Processing - Overview Guide*.

## How to Test Applications

The following process describes the C Proxy server testing philosophy. To test generated applications before moving them to a production environment,

### **Follow these steps:**

1. [Use Diagram Trace Utility](#) (see page 143) to test generated C Proxy server applications.

**Note:** Consider the [multi-user support in Diagram Trace Utility](#) (see page 144).

2. [Override the default trace environment variables](#) (see page 144). This allows the server establish communication with Diagram Trace Utility running on the client workstation.

3. [Regenerate remote files after testing](#) (see page 145). This decreases the size of the C Proxy servers' load modules and prevents a negative impact on the application's performance.

## User Exits

The user exits that are invoked at runtime for a C Proxy are common to other C/C++ execution environments.

**Note:** For more information about the details of each user exit, see the *User Exit Reference Guide*.

A set of user exits is invoked from within the C Proxy runtime regardless of the selected communication type. There are other user exits, which are unique for each communications type.

The following user exit entry points are common to all C Proxies, regardless of communications type:

| User Exit Entry Point | Purpose                                                                                                                                                                                                                                                                                                                              |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| WRSECTOKEN            | The Client Security user exit is used to direct the runtime to incorporate the ClientUserid and ClientPassword attributes into the CFB. If this exit returns SECURITY_ENHANCED, it can cause an optional Security Token field to be added to the CFB.                                                                                |
| WRSECENCRYPT          | The import message encryption user exit provides the opportunity for a portion of the outbound CFB to be encrypted using an encryption algorithm implemented within this user exit. The target DPS's execution environment must implement a companion decryption user exit for the CFB to be interpreted as a valid request CFB.     |
| WRSECDECRYPT          | The export message decryption user exit provides the opportunity for decrypting the portion of the CFB that has been encrypted by the target DPS's execution environment. This exit must implement the companion to the DPS's encryption user exit in order for the DPS's response buffer to be interpreted as a valid response CFB. |

The following user exit entry points are invoked by C Proxies flowing to their target DPS using TCP/IP as their communications type:

| User Exit Entry Point   | Purpose                                                                     |
|-------------------------|-----------------------------------------------------------------------------|
| CI_TCP_DPC_DirServ_Exit | Overrides the destination information used when creating the TCP/IP Socket. |

| User Exit Entry Point          | Purpose                                                                                                                            |
|--------------------------------|------------------------------------------------------------------------------------------------------------------------------------|
| CI_TCP_DPC_setupComm_Complete  | Directs the runtime to retry a cooperative flow request that failed during setup.                                                  |
| CI_TCP_DPC_handleComm_Complete | Directs the runtime to retry a cooperative flow request that failed during the non-setup related processing of a cooperative flow. |

The following user exit entry points are invoked by C Proxies flowing to their target DPS using ECI as their communications type:

| User Exit Entry Point  | Purpose                                                                                                                                                    |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ci_eci_get_system_name | Provides the CICS system name, if one has not previously been provided by data obtained from the commcfg.ini file.                                         |
| ci_eci_get_tpn         | Provides that capability to specify an alternate CICS Mirror Transaction name. The default Mirror Transaction will be CPMT unless overridden by this exit. |

The following user exit entry points are invoked by C Proxies flowing to their target DPS using MQSeries as their communications type:

| User Exit Entry Point          | Purpose                                                                                                                            |
|--------------------------------|------------------------------------------------------------------------------------------------------------------------------------|
| CI_MQS_DPC_Exit                | Overrides various pieces of information used to interface with MQSeries.                                                           |
| CI_MQS_DynamicQName_Exit       | Overrides the default structure of Dynamic Reply to Queues.                                                                        |
| CI_MQS_DPC_setupComm_Complete  | Directs the runtime to retry a cooperative flow request that failed during setup.                                                  |
| CI_MQS_DPC_handleComm_Complete | Directs the runtime to retry a cooperative flow request that failed during the non-setup related processing of a cooperative flow. |
| CI_MQS_MQShutdownTest          | Overrides the default behavior of keeping the connection to the Queue Manager following the completion of a cooperative flow.      |

The following user exit entry points are invoked by C Proxies flowing to their target DPS using Tuxedo as their communications type:

| User Exit Entry Point | Purpose                                                                                                                                                                                                                                                         |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ci_c_sec_set          | Sets user supplied security data into the security data fields located in Tuxedo's TPINIT structure.                                                                                                                                                            |
| ci_c_user_data_out    | Gives the user the opportunity to inspect or modify the cooperative flow request buffer prior to Tuxedo sending the request to the target Tuxedo service.                                                                                                       |
| ci_c_user_data_in     | Gives the user the opportunity to inspect or modify the cooperative flow request buffer on return from the target Tuxedo service. Invoked on return from the TPCALL. Additionally, this exit lets the client to disconnect from the server following each flow. |

The following user exit entry points are invoked by C Proxies flowing to their target DPS using Web Services as their communications type:

| User Exit Entry Point | Purpose                                                                                                                       |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------|
| CI_WS_DPC_Exit        | Gives the user an opportunity to modify the Web Service endpoint destination by overriding the base URL and the context type. |
| CI_WS_DPC_URL_Exit    | Gives the user an opportunity to modify the Web Service endpoint destination URL.                                             |

**More information:**

[.NET Proxy](#) (see page 49)

[Java Proxy](#) (see page 73)

[Java Proxy \(Classic Style\)](#) (see page 97)





# Chapter 8: Using the application.ini File With Proxies

---

In CA Gen Release 8.5, the C and COM Proxy servers use the application.ini file. This file stores application-specific environment variables in a 'token=value' pairing, similar to the GUIEnvironmentVariables.ini file. CA Gen delivers this initialization file with the C Runtime in the CA Gen install directory %IEFH% on Windows and \$IEFH/make on UNIX. When a C or COM Proxy server application is built using the Build Tool, this file is copied, only once, into the model's source directory (referred to by the profile token LOC.CODE\_SRC). This copy of the application.ini file is customizable by the user, and will not be overridden.

When a C or COM Proxy server application executes, the C Runtime library opens and reads the application.ini file in the model's source directory, and sets uncommented environment variables in the application.

## Environment Variables in the application.ini File

The application.ini file contains a set of commented environment variables. To use these environment variables, remove the comment from the variable (";") and modify the value.

The application.ini contains the following environment variables:

### **TRACE\_ENABLE**

Enables communications with the Diagram Trace Utility.

**Default:** 1

### **TRACE\_HOST**

Defines the Windows system running the Diagram Trace Utility.

**Default:** (Windows) localhost; (UNIX) no default value

### **TRACE\_PORT**

Defines the port on which the Diagram Trace Utility listens.

**Default:** 4567

**Note:** To use the Diagram Trace Utility, generate the C application with Trace Enabled.



# Index

---

## .NET proxy • 49

- asynchronous processing • 63
- configure runtime • 65
- deploy • 67
- execution methods • 51
- interface • 50
- object • 50
- prepare execution • 65
- security processing • 64
- synchronous processing • 62
- trace control methods • 51
- user exits • 66
- using • 62
- view objects • 56

## A

- API variable types • 172
- Applet/Server deployment • 114
- Application Bean deployment • 114
- application.ini file • 233
- AsString • 124
- asynchronous operation • 21
- asynchronous processing, .NET • 63

## B

- Build Tool, Java proxy (classic) • 97

## C

- C Precision properties • 103
- C program, calling C proxy API from • 149
- C proxy • 147
  - API variable types • 172
  - application testing • 228
  - calling the API from a C program • 149
  - communication specific details • 154
  - configure proxy at runtime • 227
  - executing user written application • 227
  - functions calls • 175
  - overview of the generated header file • 150
  - overview of the generated Tuxedo view file • 164
  - proxy function prototypes • 171
  - security processing • 224
  - setting up API environment • 147

- setting up UNIX platform • 148
- setting up Windows platform • 148, 165, 166
- user exits • 229
- using • 224
- variable name case sensitivity • 150
- calling the COM proxy from a web server • 141
- calling the COM proxy from Visual Basic • 141
- CBD 96 • 33
- Clear method, Java proxy (classic) • 99
- client bean JAR • 114
- client security user exit • 135
- client security user exit class • 66
- COM proxy • 119
  - application testing
    - multi-user Diagram Trace utility support • 144
    - overriding the default trace environment variables • 144
    - regenerating remote files after testing • 145
    - using Diagram Trace Utility • 143
  - asynchronous processing • 128
  - common properties and methods • 120
  - configure proxy communication • 134
  - configure proxy runtime • 134
  - deploy • 138
  - generated code • 119
  - interface • 120
  - prepare for execution • 130
  - registering the proxy DLLs • 138
  - security processing • 129
  - special text permitted value property handling • 124
  - special timestamp property handling • 125
  - supporting files • 139
  - synchronous processing • 128
  - unique attribute properties • 123
  - user exits • 135
  - using • 127
- commcfg.ini • 227
- Common Format Buffer (CFB) • 64

## D

- data structure, XML • 24
- deploy
  - .NET proxy • 67
  - Applet/Server • 114

---

- Application Bean • 114
- COM proxy • 138
- Java proxy • 92
- Java proxy (classic) • 114

## E

- EJB components • 73
- environment variable, GEN\_PROXY\_WRITE\_NAMES • 32
- error XML • 27
- execution methods, Java proxy • 76
- export message decryption user exit • 135

## F

- function calls • 175

## G

- generating proxy • 31
  - associate character names • 32
  - CSE generation • 38
  - install proxy support • 31
- Global Assembly Cache, deploy .NET proxy • 68

## H

- HTML server • 114

## I

- illustration, how the proxy is used • 147
- import message encryption user exit • 135
  - class • 66
- interface, Java proxy (classic) • 98

## J

- Java Bean proxy • 97
- Java proxy • 73
  - asynchronous processing • 85
  - configure proxy communications • 87
  - configure proxy runtime • 87
  - deploy • 92
  - deploy proxy runtimes • 93
  - generated code • 73
  - interface • 74
  - object • 75
  - prepare execution • 87
  - runtime files • 92
  - sample • 94
  - security processing • 86

- stand alone application • 95
- synchronous processing • 85
- user exits • 88
- using • 84
- view objects • 79
- XML test application • 95

- Java proxy (classic) • 97
  - asynchronous processing • 107
  - C Precision properties • 103
  - code • 105
  - common properties and methods • 99
  - configure proxy communication • 110
  - configure proxy runtime • 109
  - date, time, and timestamp properties • 103
  - deploy • 114
  - generated code • 97
  - interface • 98
  - prepare execution • 109
  - preparing the Applet/Servlet Java proxy for web access • 116
  - preparing the application Bean Java proxy for JSP access • 116
  - preparing the application JavaBean proxy • 116
  - repeating group property handling • 104
  - sample Applet/Servlet JavaBean • 117
  - security processing • 108
  - support files • 115
  - synchronous processing • 107
  - unique attribute properties • 101
  - user exits • 110
  - using • 106
- Java proxy JAR file • 92
- Java Servlet/EJB • 97
- JSP • 97
  - web applications • 73

## M

- mkjavart.bat • 115

## N

- noNamespaceSchemaLocation • 23

## O

- OLE automation • 119
  - client • 120

## P

- preparing

---

- the Applet/Servlet Java proxy for web access • 116
- the Application Bean Java proxy for JSP access • 116
- the Application Java Bean Proxy • 116
- prototypes, proxy function • 171
- proxies
  - asynchronous flow support • 22
  - common features • 21
  - language • 29
  - optional features • 22
  - platforms • 29
  - programming style • 30
  - XML support • 23
- proxy
  - generation • 31
  - installation • 15
- proxy function
  - calls • 175
  - prototypes • 171
- proxy runtime security
  - .NET proxy features • 64
  - COM proxy features • 129
  - Java proxy • 86
- ProxyAllocateParamBlock • 177
- ProxyCheckAsyncResponse • 178
- ProxyClearClientPassword • 179
- ProxyClearClientUserid • 180
- ProxyClearCommand • 181
- ProxyClearDialect • 182
- ProxyClearExitMsgType • 183
- ProxyClearExitStateMsg • 183
- ProxyClearExitStateNum • 184
- ProxyClearNextLocation • 185
- ProxyClearParamBlock • 186
- ProxyConfigureComm • 187
- ProxyDeleteParamBlock • 188
- ProxyExecute • 189
- ProxyExecuteAsync • 190
- ProxyGetAsyncResponse • 191
- ProxyGetClientPassword • 194
- ProxyGetClientUserid • 194
- ProxyGetCommand • 195
- ProxyGetDialect • 196
- ProxyGetExitMsgType • 196
- ProxyGetExitStateMsg • 197
- ProxyGetExitStateNum • 198
- ProxyGetNextLocation • 199
- ProxyIgnoreAsyncResponse • 199

- PROXYNMS.DAT • 32
- ProxySetClientPassword • 201
- ProxySetClientUserid • 202
- ProxySetCommand • 203
- ProxySetDialect • 203
- ProxySetNextLocation • 204
- ProxyStartTracing • 205
- ProxyStopTracing • 206
- ProxyTraceOut • 207

## R

- request XML • 24
- response XML • 25

## S

- sample
  - Applet/Servlet JavaBean • 117
  - ASP.NET • 69
  - executing the VB and ASP • 140
  - Java proxy • 94
- security processing
  - .NET proxy • 64
  - C proxy • 224
  - Java proxy • 86
  - Java proxy ( classic ) • 108
- server environment parameters • 33
- Server Load Module • 33
- Server Manager
  - complete modeling activities • 31
  - Toolset generation • 33
- servlet
  - bean JAR • 114
  - engine • 114
  - web applications • 73
- setting up C proxy API environment • 147
- special string methods, Java proxy (classic) • 103
- special timestamp property handling (COM) • 125
- stand alone application, Java proxy • 95
- system level data • 57
- system level properties, Java proxies • 82

## T

- Toolset generation • 33
- trace control methods, Java proxy • 75
- Tuxedo, C proxy API for • 164
- types, API variable • 172

---

## U

- unique attribute properties, (COM) • 123
- user exits, .NET proxy • 66
- user exits, C proxy • 229
- user exits, COM proxy • 135
- user exits, Java proxy • 88
  - communications • 90
  - security • 89
- user exits, Java proxy (classic) • 110
  - communications • 112
  - security • 111
- using
  - COM proxy • 127
  - Java Proxy • 84

## V

- variable name case sensitivity, C proxy • 150
- variable types, API • 172
- view objects, .NET proxy • 56
  - data validation • 61
  - export objects • 57
  - GroupView objects • 60
  - import objects • 57
  - mapping data type • 61
  - system level properties • 60
- view objects, Java proxy • 79
  - data validation • 83
  - export object • 80
  - group view • 83
  - import object • 80
  - mapping data type • 84

## W

- web access • 116

## X

- XML support • 23
  - data structure • 24
  - error • 27
  - extra methods and files • 23
  - mapping, view to • 27
  - request • 24
  - response • 25
  - schema • 23
  - standard proxy support • 28
- XMLExecute • 62