

AppBuilder
By Magic Software Enterprises

Magic Software AppBuilder

Version 3.2

Communications Guide

Corporate Headquarters:

Magic Software Enterprises
5 Haplada Street,
Or Yehuda 60218, Israel
Tel +972 3 5389213
Fax +972 3 5389333

© 1992-2013 AppBuilder Solutions

All rights reserved.

Printed in the United States of America.

AppBuilder is a trademark of AppBuilder Solutions. All other product and company names mentioned herein are for identification purposes only and are the property of, and may be trademarks of, their respective owners.

Portions of this product may be covered by U.S. Patent Numbers 5,295,222 and 5,495,610 and various other non-U.S. patents.

The software supplied with this document is the property of AppBuilder Solutions and is furnished under a license agreement. Neither the software nor this document may be copied or transferred by any means, electronic or mechanical, except as provided in the licensing agreement.

AppBuilder Solutions has made every effort to ensure that the information contained in this document is accurate; however, there are no representations or warranties regarding this information, including warranties of merchantability or fitness for a particular purpose. AppBuilder Solutions assumes no responsibility for errors or omissions that may occur in this document. The information in this document is subject to change without prior notice and does not represent a commitment by AppBuilder Solutions or its representatives.

1. Communications Guide	2
1.1 Introduction to Configuring Communications	2
1.1.1 Additional Resources	2
1.2 Configuring and managing communications in window	2
1.2.1 Setting Up Windows Communications	2
1.2.1.1 Launching the Communications Setup	3
1.2.1.2 Understanding the Communications Setup	3
1.2.2 Configuring Systems Files	4
1.2.3 Managing the System Service	9
1.2.4 Managing Computers	10
1.2.5 Managing Servers and Gateways	11
1.2.6 Configuring Clients, Servers, and Gateways	15
1.2.7 Understanding Routing	27
1.2.8 Testing Configurations	37
1.3 Configuring Communications Using MQSeries	37
1.3.1 Understanding MQSeries RPC Operation	38
1.3.2 Meeting Prerequisites for MQSeries RPC	39
1.3.3 Configuring MQSeries RPC	39
1.3.3.1 Configuring the AppBuilder Server	39
1.3.3.2 Configuring the MQ Server	41
1.3.3.3 Configuring the Mainframe MQ Series	41
1.3.3.4 Customizing the Mainframe INI File	43
1.3.4 Handling Errors in MQSeries	45
1.4 Configuring Communications in UNIX	45
1.5 Configuring Communications on the Mainframe	48
1.5.1 Accessing Mainframe Configuration Files	49
1.5.2 Performing Maintenance Operations	50
1.5.3 Configuring LU2 Clients	51
1.5.4 Configuring LU6.2 Clients	51
1.5.5 Configuring LU2 Terminal Emulator	51
1.5.6 Configuring CICS for TCP-IP Listener	52
1.5.7 Configuring CICS HTTP Support	55
1.5.8 Configuring AppBuilder INI Settings for HTTP Support	64
1.5.9 Configuring C HTTP Client	66
1.5.10 Usage Notes	68
1.5.11 DNAINI Settings	69
1.5.12 Understanding Performance Marshalling	74
1.6 Extending Communications	75
1.6.1 Communication Exits in Java	76
1.6.2 C Client and Server Exits	82
1.6.2.1 Authentication	82
1.6.2.2 Authorization	84
1.6.2.3 Encryption	85
1.6.2.4 Open Data Encryption Mechanism	85
1.6.2.5 Setting Up Security Exits	89
1.6.2.6 Security Exits Reference for C	89
1.6.2.7 Directory Services Exits in C	95
1.6.2.8 Compiling and Linking	101
1.6.2.9 Database Exits in C	104
1.6.3 CICS TCP-IP Listener Security Settings	111
1.6.4 Customized Data Types in C	112
1.6.4.1 Using Custom Data Types	112
1.6.4.2 Identifying the Data Type	113
1.6.4.3 Writing the Marshalling Functions	113
1.7 Supported CodePages	115
1.8 Mainframe Logon Script Reference	116

Communications Guide

Introduction to Configuring Communications

AppBuilder is an advanced development environment that uses internal communications for connecting the various tools and services for multi-platform deployment. The platforms and protocols that can be used are covered in the following chapters:

- [Configuring and Managing Communications in Windows](#)
- [Configuring Communications Using MQSeries](#)
- [Configuring Communications in UNIX](#)
- [Configuring Communications on the Mainframe](#)

Interfacing with the communications facilities is covered in [Extending Communications](#). The appendices cover [Supported CodePages](#) and provide a [Mainframe Logon Script Reference](#).

Additional Resources

For complete information about configuration settings, see the *INI Settings Reference Guide*. For information about setting up repositories, see the *Repository Administration Guide for Workgroup and Personal Repositories*. For information about warning messages or error messages related to communications, refer to the *Messages Reference Guide*.

For information on testing and verifying configurations in AppBuilder, see the *IVP User Guide*. That book covers C client and Java client preparation and testing; C server, EJB, webservices, and servlet testing; and CICS, Batch, IMS and other mainframe testing.

Configuring and managing communications in window

The Management Console is the tool for managing and configuring aspects of AppBuilder communications on a local or remote computer. You can use the Management Console to:

- Configure Java and Windows client communications for application execution
- Create multiple AppBuilder application servers and protocol gateways
- Configure application servers and protocol gateways
- Manage application servers and protocol gateways
- View communications error logging and tracing AppBuilder
- Modify the AppBuilder configuration through the AppBuilder system configuration file (hps.ini)
- Modify other AppBuilder-associated configuration (.ini) files

As with any AppBuilder communications application, the Management Console only works between machines where the codepage mapping exists.



Make sure to shut down all AppBuilder communications clients and servers before configuring the host.

For more information about configuring and managing communications, refer to the following sections:

- [Setting Up Windows Communications](#)
- [Configuring System Files](#)
- [Managing the System Service](#)
- [Managing Computers](#)
- [Managing Servers and Gateways](#)
- [Configuring Clients, Servers, and Gateways](#)
- [Understanding Routing](#)

Setting Up Windows Communications

When you install AppBuilder 3.2, once the installation process is complete the AppBuilder Communications Setup program (quickcfg.exe) is launched silently. That silent install sets as defaults the following:

- TCP/IP as the active protocol
- the current Windows system codepage as the active codepage to use
- no database

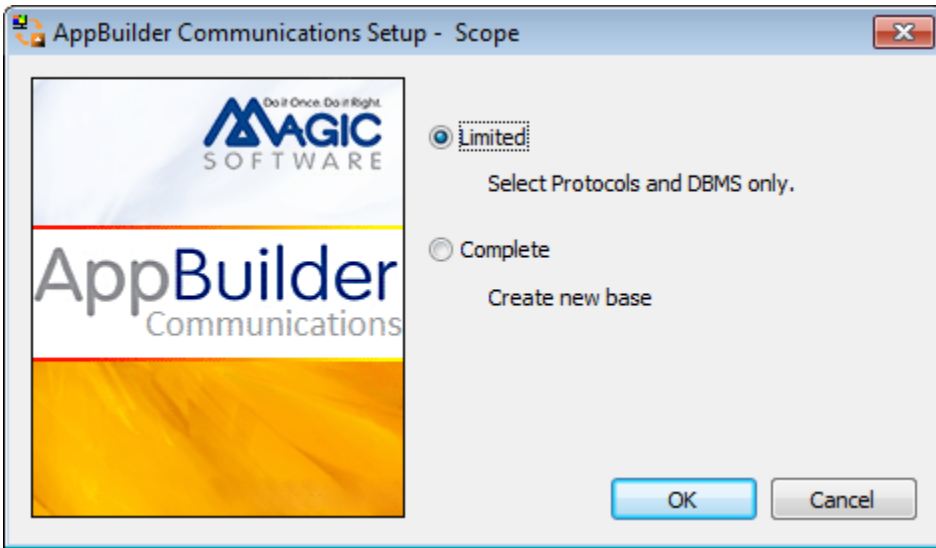
You can use the AppBuilder Communications Setup program to manually configure your system and enable communication between the various tools and services of the AppBuilder system. This includes:

- [Launching the Communications Setup](#)
- [Understanding the Communications Setup](#)

Launching the Communications Setup

From the Start menu, select *Start > Programs > AppBuilder > Configuration > Communications Setup* . This starts the Communications Setup wizard.

Communications Setup dialog



Understanding the Communications Setup

From the Communications Setup wizard, select the scope of the configuration. Select *Limited* for selecting only the protocol or database settings. Select *Complete* to create an entirely new base configuration and to select all the options in the communications setup.

Setup Scope

Setup type	Tasks
Limited	Select protocol Select Database Management System (DBMS)
Complete	Select protocol Select codepage Select Eventing Parent Select Database Management System (DBMS)

If the AppBuilder Communications service is currently running when you attempt to run the setup, the system notifies you and gives you the option to stop the service or stop the setup.

Select protocol

Select one or more protocols from the check boxes on the Protocol screen. Selecting a protocol enables that protocol for AppBuilder communications. You can choose from the following protocols:

- TCP/IP
- Named Pipes
- LU 6.2
- LU 2

TCP/IP and Named Pipes come with the Windows? operating system.



If you select a protocol that you do not have installed (i.e. lu62 without an lu62 installed product), the configuration server will likely crash.

Select codepage

Select the codepage from the drop-down list on the codepage screen. The system codepage is used by communications to specify the local codepage for character encoding. The appropriate tables are enabled for the selected codepage. See [Supported CodePages](#) for a list of supported codepages and codepage conversions.

Select Eventing Parent

Type the name of the eventing parent (host) in the Hostname field on the Eventing Parent screen. This information is required only if global eventing is used.



The use of Global Eventing on the mainframe is restricted to LU6.2 communications. This is the only protocol that is supported on the mainframe.

Select Database Management System (DBMS)

Select a database management system (DBMS) from the drop-down list on the Database screen. If no DBMS is installed on this machine or you are not using it, select *None*. The options in the drop-down list are listed below:

- None
- Oracle
- DB2UDB
- Microsoft SQL Server 2000

When you have finished specifying the settings, click *OK*. The Communications Setup dialog closes.

To check that the configuration is correct, launch the AppBuilder Management Console (see below) and start one of the default servers. If the server starts (turns green), the configuration works.

Configuring Systems Files

There are a number of system files in AppBuilder. They can be accessed through the Management Console. The Management Console provides access to appbuilder.ini, Hps.ini, appbuildercom.ini, and Dna.ini.

Launching the Management Console

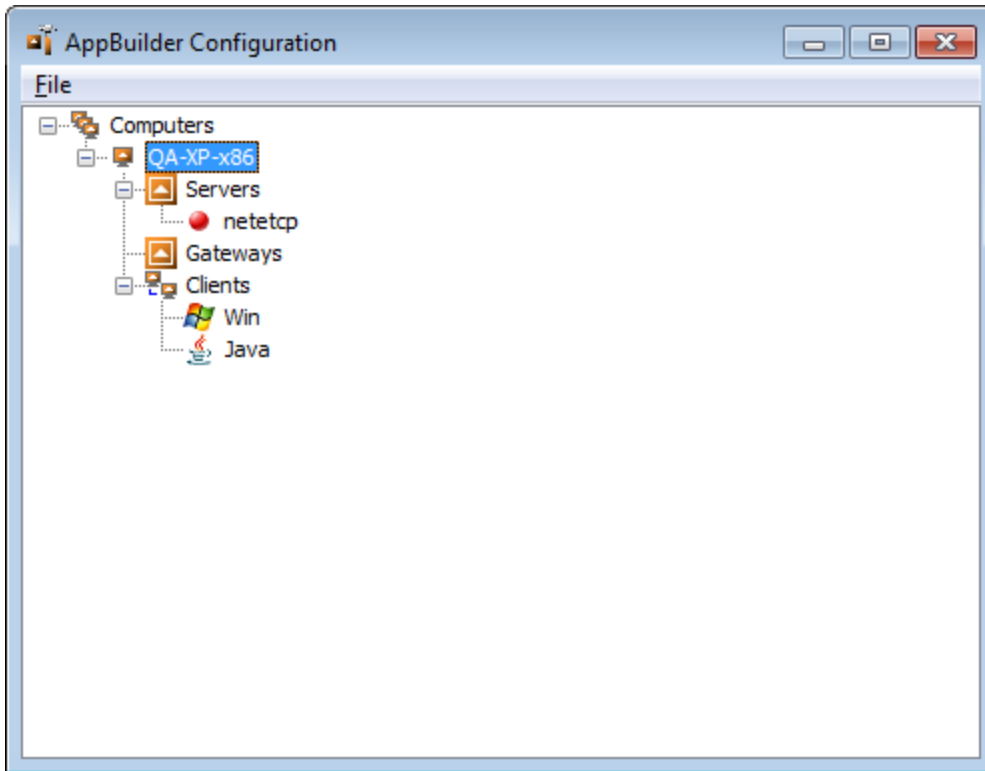
If you are a user of your local machine, you need not log onto it via the Management Console. If you are using a remote machine, however, you have to log on to the Management Console.

To launch the Management Console, select *Start > Programs > AppBuilder > Configuration > Management Console*. Or from the Service Control, select *Launch AppBuilder Management Console*. The AppBuilder Configuration window appears.

Understanding the Management Console

The Management Console window shows a graphical representation of the communications resources defined by the user on Windows machines. From this window, you can select menu options, click resource icons, and manage the resources associated with those icons. The following figure illustrates a typical Management Console window.

Management console window example



From the console, click a machine's icon to select it (client, server, or gateway). Right-click to select the properties. Add a machine or resource by selecting the icon for the type of resource and selecting the *Add* option from the pop-up menu. Delete a resource by selecting its icon and pressing the Delete key.

If the resource icon has been refreshed, you can see whether that resource is active or not.

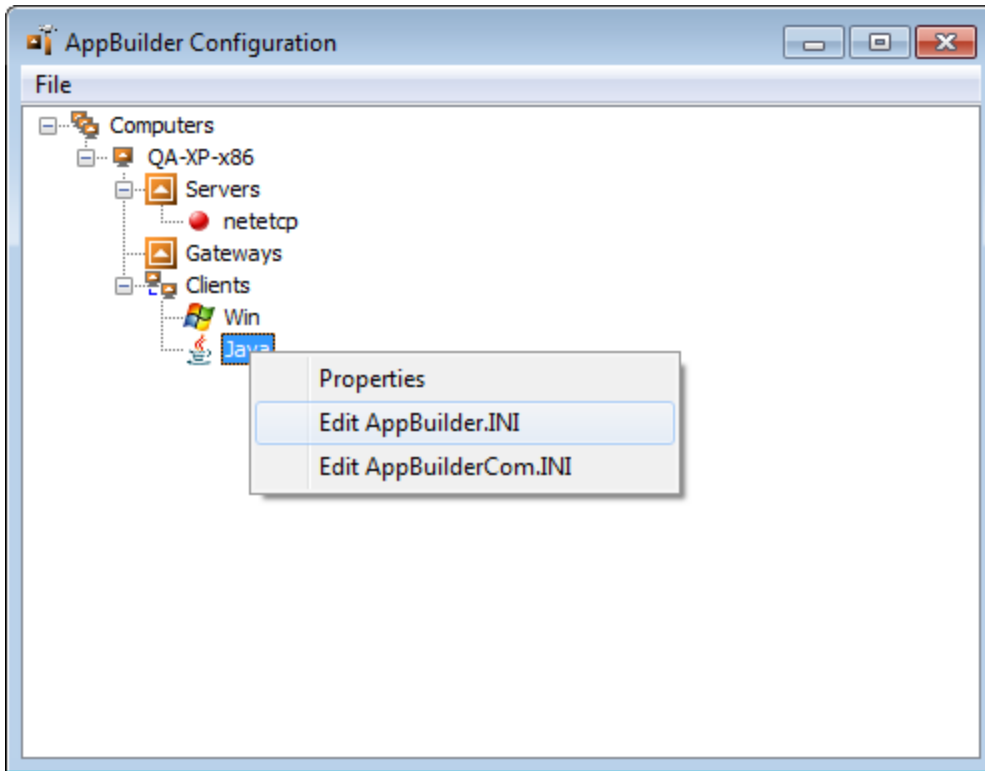
If AppBuilder cannot find a computer (a machine previously added), AppBuilder issues a message and marks the computer icon as inactive. The menu selections are grayed out for that machine, and you can only remove or refresh the computer. If a refresh finds the machine active, reporting returns to normal: the icon and menu item are displayed as active.

Editing Configuration Files

Configuring communications can involve editing the configuration files (.ini) on the workstation. There are configuration files for both Java operations and Windows (C client) operations. The Windows INI settings are specified in dna.ini. Java INI settings are specified in appbuilder.ini and appbuildercom.ini.


To access a configuration file, from the Management Console, select the icon for the client machine and right-click on the appropriate icon. The following figure illustrates how to access the file settings for Java in appbuilder.ini.

Editing appbuilder.ini

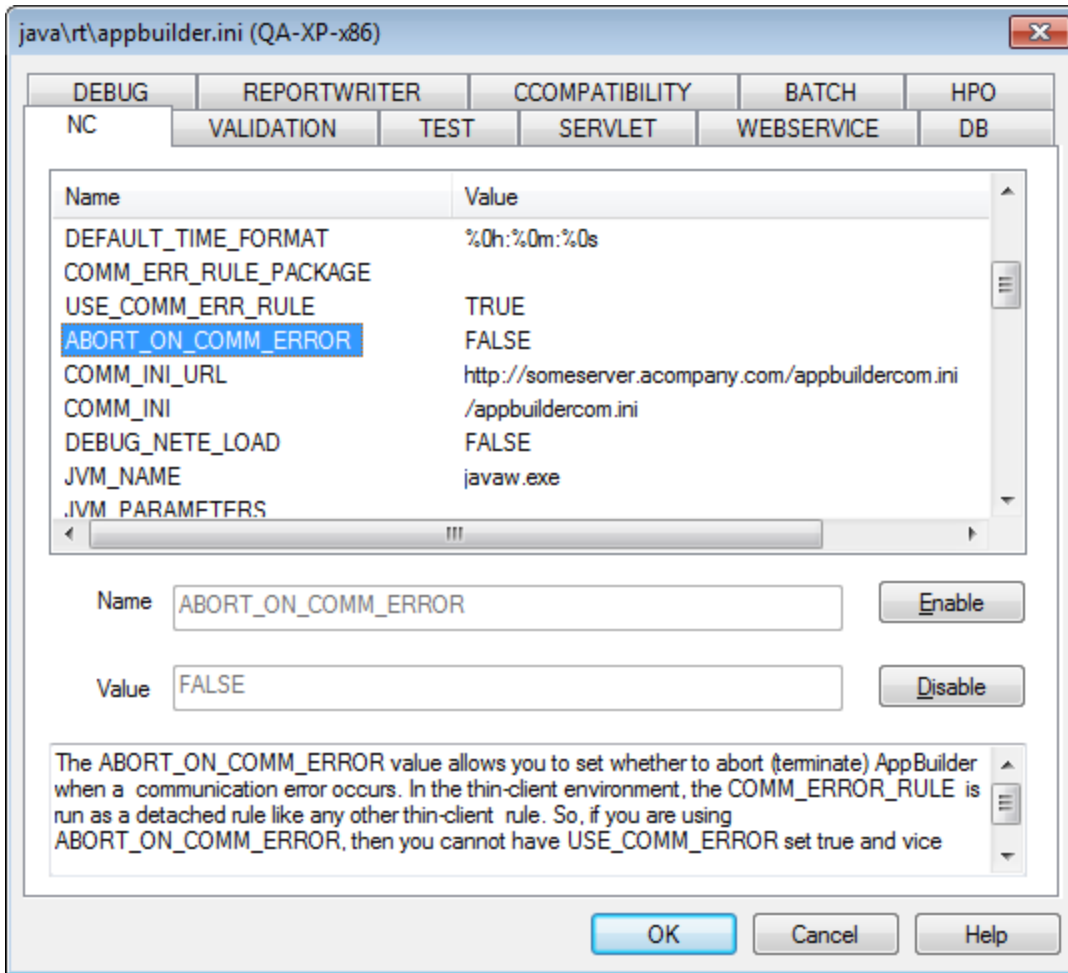


Edit the appbuilder.ini file on the workstation to change the communications configuration in AppBuilder. The sections (tabs) and settings are illustrated in [Option to abort for communication error](#).

For example, the ABORT_ON_COMM_ERROR parameter in the NC section of appbuilder.ini enables you to set whether or not to abort (terminate) AppBuilder when a communication error occurs. (This example is shown in [Option to abort for communication error](#).)

 Check the *INI Settings Reference Guide* for complete information about INI settings. It is possible to specify settings which cause conflicts and there is no warning of possible conflicts. For example, in a thin-client environment, the HPS_COMM_ERROR_RULE is run as a detached rule like any other thin-client rule. Therefore, if you set ABORT_ON_COMM_ERROR to TRUE, setting USE_COMM_ERROR_RULE also to TRUE could cause a conflict and may be a waste of time. If however, it had some logging functionality called prior to the converse or as its only function, then it may be useful to have this rule called, even when the other setting is going to cause the request to be aborted. Even when both settings are set to FALSE, the application can still be informed of a communications error by having a CommError event procedure in the rule making the remote request. So setting both INI settings to FALSE can be a valid option even though they seem to be in conflict.

Option to abort for communication error

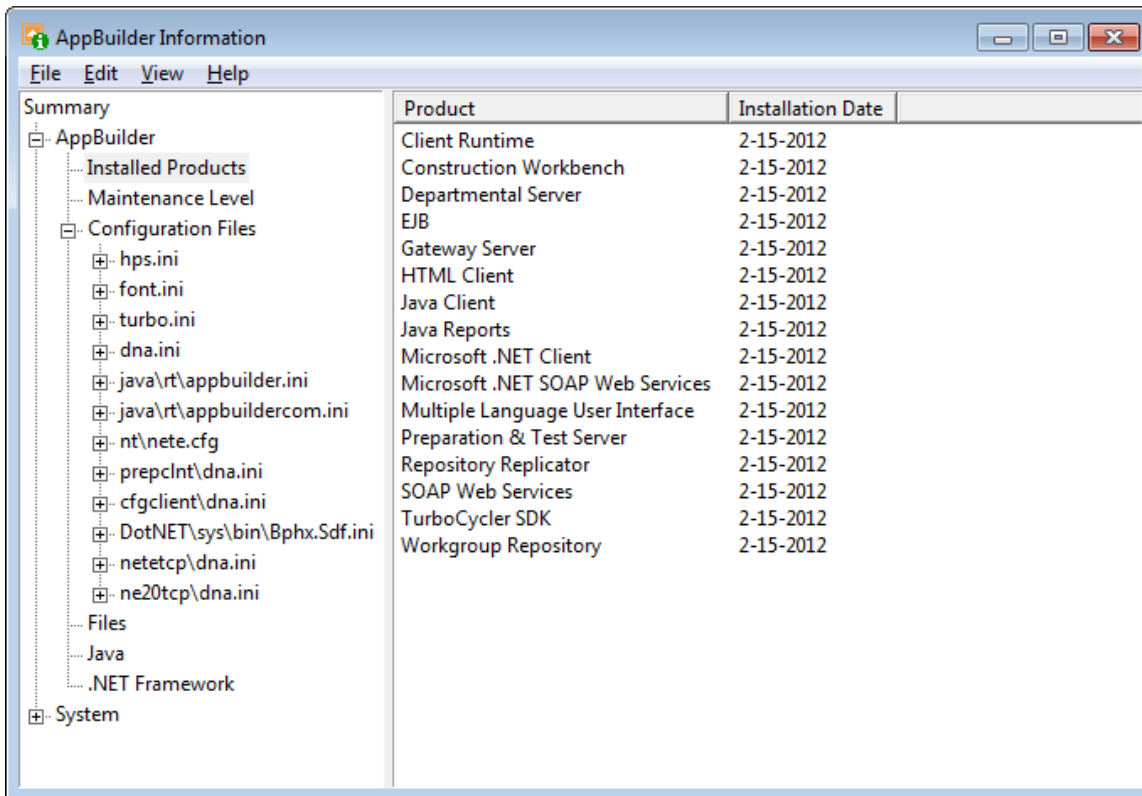


For complete information on settings specified in the appbuilder.ini file, see the *INI Settings Reference Guide*.

Viewing Product Information

You can view information about which AppBuilder products and which versions are installed on a machine. To view AppBuilder Product Information, right-click a computer icon in the Management Console and select *View Product Information*. The AppBuilder Information dialog opens.

Product Information

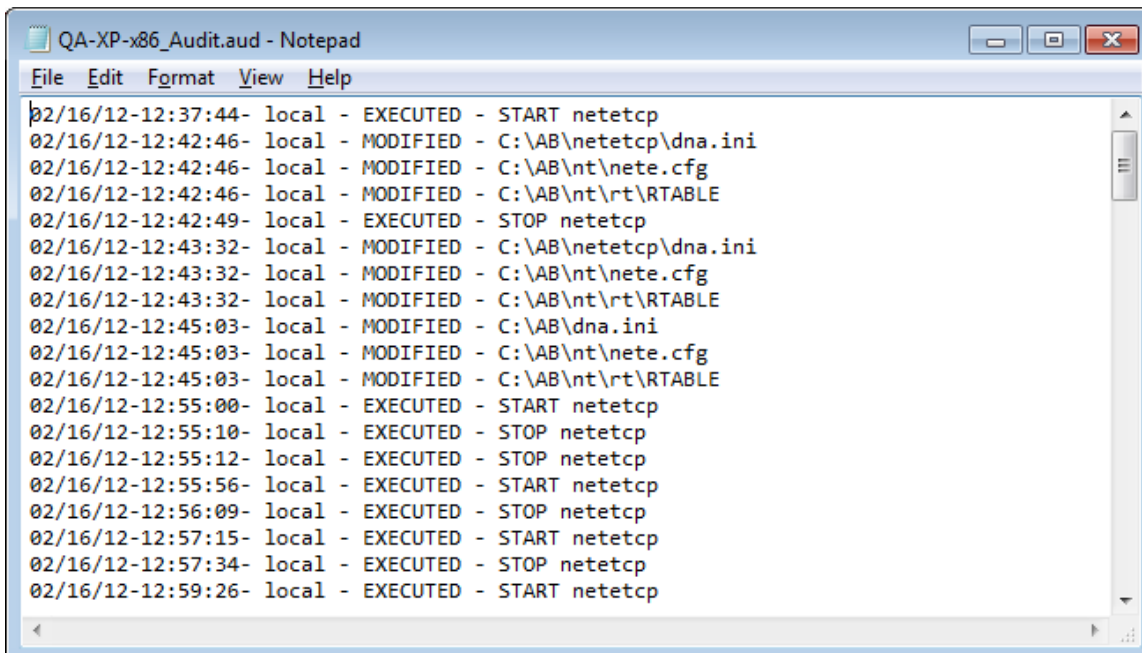


You can then view the installed products, fixpacks installed, and INI settings specified. For more information about viewing installed product versions, refer to the explanation in the *Installation Guide for Windows* . For more information about ini settings, see the *INI Settings Reference Guide*.

Viewing or Clearing Audit Information

AppBuilder keeps track of the actions taken when starting or stopping services or modifying configuration (.ini) files. This information is maintained in an audit file. To view the file, right-click a computer icon and select *View Audit Information* . AppBuilder loads the file from the local machine or a remote machine for you to view it in a text editor, such as Notepad. The figure below shows a sample audit file.

Sample audit file



The audit information contains columns presenting the following information:

- Date and time stamp
- User ID
- Action, whether executed, modified, created, or deleted
- Name of the file edited or service started or stopped

To clear the file and erase all entries, right-click a Computer icon and select *Clear Audit Information* . All entries are erased, except a message that the audit file was deleted. The following is an example of this message:

```
02/17/12-12:03:31 - userID - DELETED - cfgclient\auditfile.out
```

Managing the System Service

The AppBuilder system service (AppBuilder service) is a background process that maintains and coordinates the starting and stopping of the various AppBuilder communications servers, gateways, and agents. The service, an executable program, is a Windows service that can be started at machine boot-up time.

Use the AppBuilder Service Control to start and stop the AppBuilder service and to check on the status of that service.

This topic includes

- [Launching the Service Control](#)
- [Using the Service Control](#)
- [Controlling from the Command Line](#)

The service also automatically starts the AppBuilder service agent that is required for several critical communications tasks. The service agent does the following:

- Receives remote preparation results
- Enables remote configuration
- Handles eventing
- Manages AppBuilder servers and gateways

Launching the Service Control

To launch the Service Control:

Select *Start > Programs > AppBuilder > Configuration > Service Control* .

This selection launches the Service Control and automatically starts the AppBuilder service if it is not already running. An icon is displayed in the Windows task bar (typically the lower right of your workstation screen).

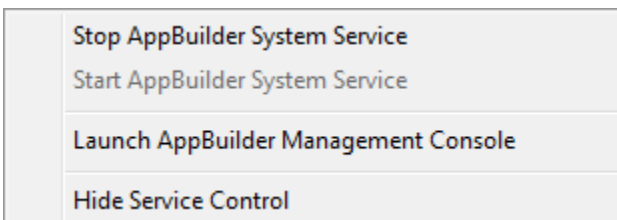
Icon in example task bar



Using the Service Control

Single-click either the right or left mouse button on the Service Control icon to display the Service Control menu.

Service Control menu



Hiding the Service Control Icon

To remove the Service Control icon from the task bar, select *Hide* . The Hide option only hides the icon from view; it does not stop or interrupt the service, and it does not automatically hide the icon for future sessions. To show the icon again, select *Start > Programs > AppBuilder > Configuration > Service Control* .

Starting or Stopping the System Service

From the Service Control menu, you can start or stop the communications service. Select either *Stop AppBuilder System Service* or *Start AppBuilder System Service* from the Service Control menu.

Controlling from the Command Line

In addition to using the Service Control menu, you can perform the following functions using the command-line interface.

Service Control Command

To control the AppBuilder service, run the ABSERVICE command with one or more of three parameters from a command prompt.

Abservice command

Command	Parameter	Function
abservice	-check	Checks whether or not the AppBuilder service is running
	-start	Starts the service if it is stopped
	-stop	Stops the service if it is started

Managing Computers

From the Management Console you can perform administrative functions using the Computer objects, which are at the highest level in the hierarchy. These functions include the following:

- [Adding a Computer](#)
- [Refreshing a Computer](#)
- [Removing a Computer](#)

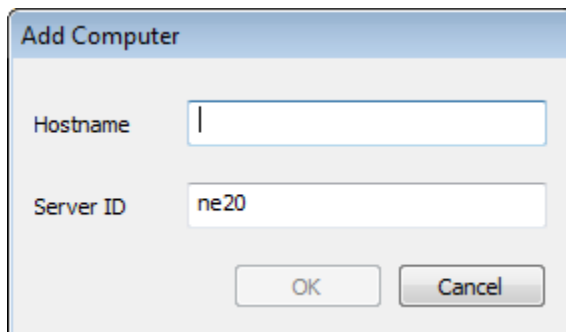
Adding a Computer

Adding a computer to the AppBuilder Management Console indicates that it has a repository to which you have access. Therefore, you cannot add a computer (machine) which has only a Personal Repository installed, since it is not possible to access that remotely. You can access a Workgroup Repository this way. To add a remote computer you must have appropriate access rights to the AppBuilder repository on that machine and system rights to the machine.

To add a remote computer to the management console:

1. Selecting the Computers icon in the console window.
2. Right-click on the icon. A pop-up menu appears.
3. On the pop-up menu, select *Add Computer*. The Add Computer dialog appears as shown in the following figure.
4. Type the host name (the machine name) and the server identifier, which defaults to ne20. Only single-byte host names (computer names) are supported.
5. Click *OK* when done.

Add Computer dialog



You are then prompted for a user name and password that are valid for the target machine. The user name and password must be for a local system account, not a network account, on the target machine.

Refreshing a Computer

To see the latest updates to the hierarchy of machines and services:

1. Right-click a computer icon in the hierarchy.
2. Select *Refresh Computer*.

This selection updates the display and shows any computers that have been added or removed. For a list of information about what actions have been taken on this computer, see [Viewing or Clearing Audit Information](#).

Removing a Computer

To remove any computer other than your own from the display:

1. Right-click on the icon of the machine that you want to delete.
2. Select *Remove Computer*.

Managing Servers and Gateways

From the Management Console, you can perform administrative functions with the servers and gateways. These tasks include the following:

- [Creating a Windows Server](#)
- [Creating a Windows Gateway](#)
- [Starting and Stopping a Server or Gateway](#)
- [Starting a Server with a Database Connection](#)
- [Refreshing a Server or Gateway](#)
- [Pinging a Server or Gateway](#)
- [Deleting a Server or Gateway](#)
- [Viewing and Clearing the Trace Log](#)
- [Viewing and Modifying Properties](#)
- [Using TCP/IP Services](#)

Creating a Windows Server

From the Management Console you can create a server on Windows. To create a new server, do the following:

1. Right-click the *Servers* icon and select *New Server*. The Protocol dialog box is displayed. See [Creating a server - Protocol and subsequent dialogs](#).

Properties specified in the Protocol dialog

Dialog field	Description
Protocol Host name	Defaults to the machine name. The user cannot edit this field.
Protocol	Can be selected by user as either tcpip (TCP/IP) or NPIPE (Named Pipes).
Server ID	Is a user-defined, unique identifier for this server.

2. Enter the information in this dialog box, and click *Next*. The Database dialog box is displayed.
3. Enter the appropriate information in this dialog box. This dialog box requires settings for System DBMS and database name.

Properties specified in the Database dialog

Dialog field	Description
System DBMS	Defaults to the database management system (DBMS) specified when you installed AppBuilder, configured communications, and selected a DBMS. The user cannot edit this field. The System DBMS can be configured using the Communications Setup Wizard. If you previously chose NONE for System DBMS, then the database name is disabled since it is not applicable. When DBMS is other than none, the database name field is enabled and can be modified (only if a database name is not present), and the server cannot use a database, other than DBMS. See also Select Database Management System (DBMS) .
Database name	Is a user-defined, unique identifier for the database.

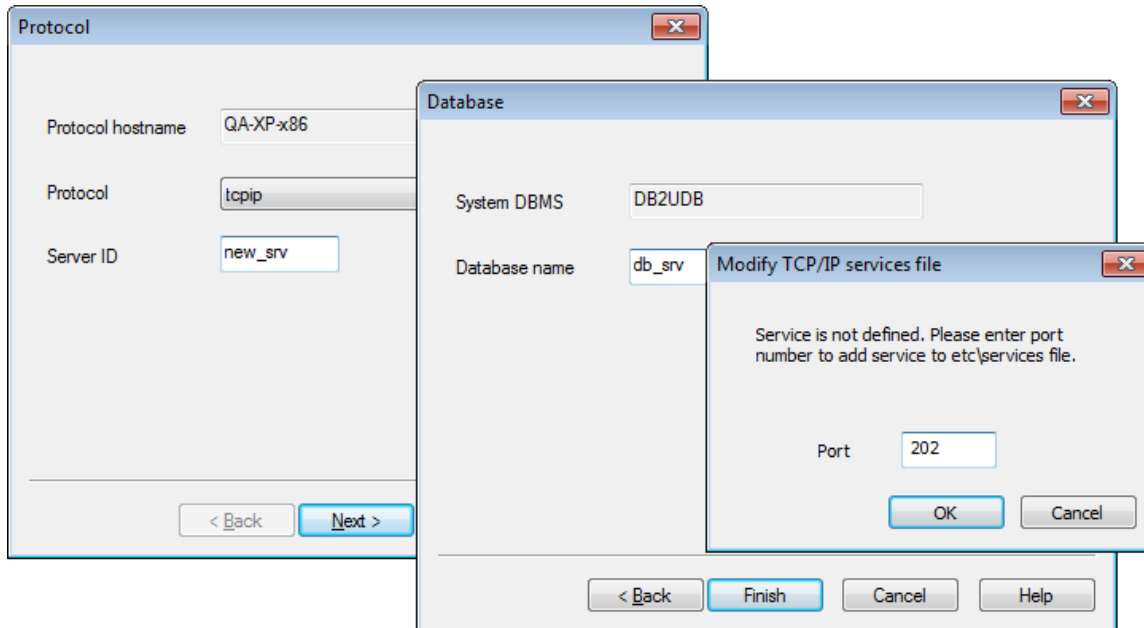
4. Then click *Finish*. At this point the new server is set up to be created. The Modify TCP/IP services dialog box is displayed.
5. Enter the port information in the TCP/IP services dialog box, and click *OK*.

Properties for creating a server

Dialog field	Description
Port	Is a user-defined, unique port number for this server.

6. The system queries the TCP/IP services file for the server ID. If the server ID already exists, an error message is displayed. Click *OK* and re-enter the server, using a different ID.
See [Managing Servers and Gateways](#) for more information.

Creating a server - Protocol and subsequent dialogs



The console then displays an icon for the new server under the Server icon. The name includes either an "np" (named pipes) or "tcp" (TCP/IP), depending on the protocol you have selected.

When a new server is created, AppBuilder checks the product information to see if it is a departmental server. If so, AppBuilder grays out the *Edit HPS.INI* option because an HPS.INI file is not needed for a departmental server.

If you configure a server with a database, then the AppBuilder consrv process needs to have authorization to access the database. By default, consrv runs as a service using a 'Local System' account. This is the account that needs access.

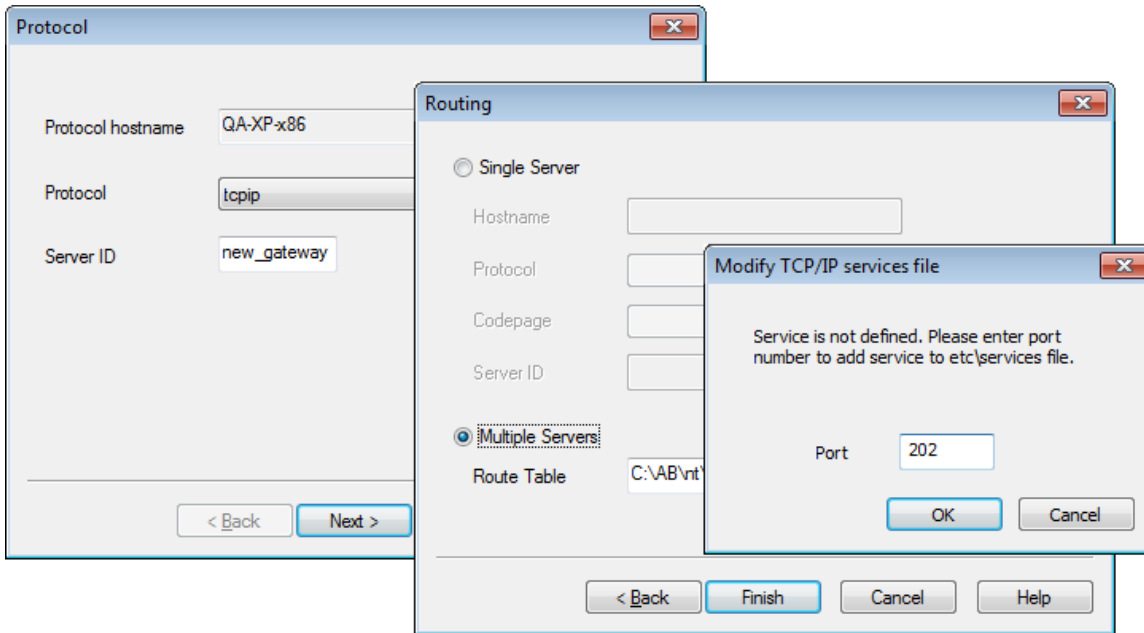
Creating a Windows Gateway

From the Management Console you can also create a gateway. To create a new gateway:

1. Right-click the *Gateways* icon and select *New Gateway*. The Protocol dialog box is displayed. See [Creating a gateway - Protocol and subsequent dialogs](#).
2. Select the protocol and type in the server ID. See [Properties for creating a gateway](#).
3. Click *Next*. The Routing dialog box is displayed.
4. Select the kind of server (single or multiple) in the Routing dialog box and click *Finish*. The Modify TCP/IP services dialog box is displayed. Enter the information in this dialog box, and click *OK*. See [Properties for creating a gateway](#) for fields and descriptions.
5. The system queries the TCP/IP services file for the gateway server ID. If the ID already exists, an error message is displayed. Click *OK* and re-enter the server, using a different ID.

The console displays the new gateway as an icon under the *Gateways* icon. See [Managing Servers and Gateways](#).

Creating a gateway - Protocol and subsequent dialogs



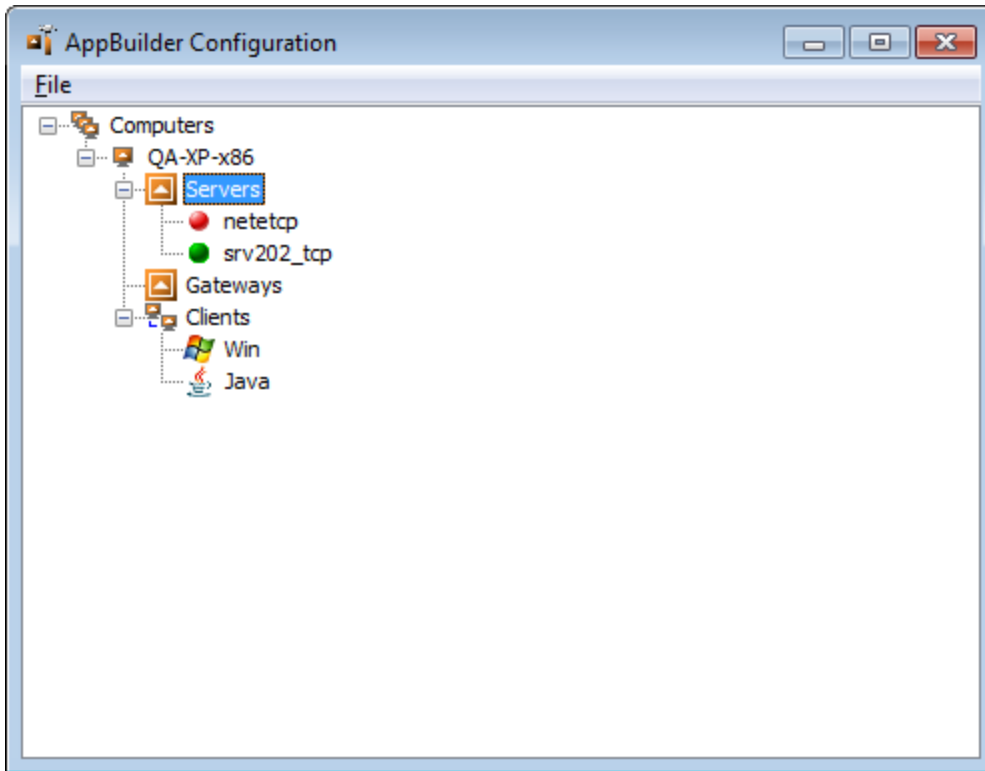
Properties for creating a gateway

Dialog field	Description
Protocol Host name	This defaults to the machine name and is not editable by the user.
Protocol	Select the protocol that this gateway uses for incoming application service requests. This is user-selectable as either tcpip (TCP/IP) or NPIPE (Named Pipes).
Server ID	This is the service entry for the TCP/IP port or Named Pipes pipe name (depending on which protocol was selected) of the gateway.
Single Server	See Configuring for a Single Server .
Single Server Host Name	Host name or machine name (see Host name by protocol)
Single Server Protocol	tcpip (TCP/IP) or NPIPE (Named Pipes)
Single Server codepage	Select from the list of codepages (see Supported CodePages for a list of supported codepages.)
Single Server ID	Server ID
Multiple Servers	See Configuring for Multiple Servers .
Route Table	See Editing the Route Table .
Port	This is a user-defined, unique port number for this server.

Starting and Stopping a Server or Gateway

From the Management Console, you can start or stop an individual server or gateway. When you start a server or gateway, it becomes active, and the icon changes to green in the console window. When a server or gateway is stopped, it becomes inactive, and the icon changes to red in the console window. These color changes are shown in the following figure.

Icon colors



To start a server, right-click the server icon in the console window and select *Start Server* .
To stop a server, right-click the server icon in the console window and select *Stop Server* .
To start a gateway, right-click the gateway icon in the console window and select *Start Gateway* .
To stop a gateway, right-click the gateway icon in the console window and select *Stop Gateway* .

Starting a Server with a Database Connection

If the ABSYSTEMSERVICE is going to start a server (NETETCP, for example), the server must call an authentication exit if the service is to run under the system account. To do this:

1. Update the DNA_EXITS section in the DNA.INI file located in the AppBuilder\CfgClient directory.
A sample exit can be found in the AppBuilder\Samples\C\Exits\Client\Dll directory.
2. Paste the DNAAUTH.DLL and DNAAUTHD.EXE files in your PATH.

Refreshing a Server or Gateway

From the Management Console, you can refresh the display of a resource, such as a server or gateway. The icons in the console window show whether a server or gateway was stopped or started, as shown in [icon colors](#). But this information is not updated dynamically. To view the latest status of a server or gateway, refresh the resource. To refresh the resource, right-click the resource icon in the Management Console window and select *Refresh Server* or *Refresh Gateway* .

Refreshing the server or gateway icon only updates whether or not the server or gateway is started or stopped. It does not show if the server or gateway has been deleted. Perform a refresh on the computer icon to show if a server or gateway has been deleted. Refer to [Refreshing a Computer](#).

Pinging a Server or Gateway

From the Management Console, you can ping a server or gateway to see whether or not it is active. The system displays an information dialog that tells you whether the server or gateway is active (started) or inactive (stopped). To ping a server or gateway, right-click on the icon in the Management Console window and select *Ping Server* or *Ping Gateway*.

Deleting a Server or Gateway

From the Management Console you can delete a server or gateway. To delete a server or gateway, right-click the icon in the Management Console window and select *Delete Server* or *Delete Gateway*.

Refreshing from a server or gateway icon only updates whether or not the server or gateway is started or stopped. It does not show if the server or gateway is deleted. Perform a refresh on the computer icon to show if a server or gateway is deleted. See [Refreshing a Computer](#).

Viewing and Clearing the Trace Log

If a server, gateway, or Windows execution client is configured to produce error logs or trace files, you can view those files using this utility. AppBuilder loads the file from the local or remote machine and enables you to view it in the text editor, Notepad.

Viewing and Modifying Properties

To view or modify the properties of a server, right-click the server icon and select *Properties*. To view or modify the properties of a gateway, right-click the gateway icon and select *Properties*. This is covered in [Configuring a Server for Remote Communications](#) and [Configuring a Gateway for Remote Communications](#).

To view or modify the properties of a Java client, right-click the Java client icon and select *Properties*. This is covered in [Configuring a Java Client for Remote Communications](#).

To view or modify the properties of a Windows client, right-click the Windows client icon and select *Properties*. This is covered in [Configuring a Windows Client for Remote Communications](#).

Using TCP/IP Services

Server Side

On the server side, when you either create a new TCP/IP server or modify the server ID or port number of an existing TCP/IP server, the configurator warns you of any naming conflicts and updates the TCP/IP services file automatically.

Client Side

On the client side, if you use single-server routing to route service requests to a TCP/IP execution server, the configurator adds an entry for the server in the TCP/IP services file.

By contrast, if you use multiple-server routing to route service requests to an execution server other than the default server, the configurator does not update the TCP/IP services file for you. Make sure to add an entry for the server in the services file.

Configuring Clients, Servers, and Gateways

From the Management Console, you can configure clients, servers, and gateways for Windows machines for remote communications. This topic includes information on the following:

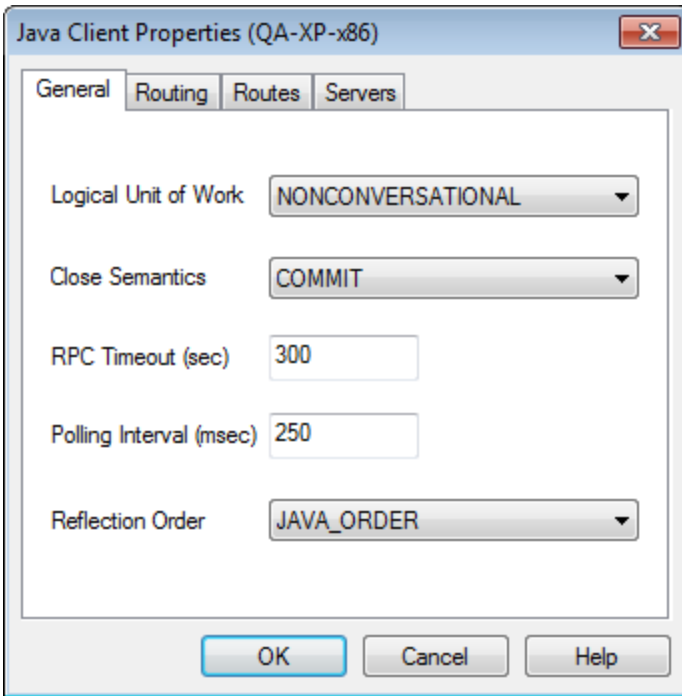
- [Configuring a Java Client for Remote Communications](#)
- [Adding a Server for Client Communications](#)
- [Configuring a Windows Client for Remote Communications](#)
- [Configuring Performance Settings](#)
- [Configuring Security Settings](#)
- [Configuring Trace Log File Settings](#)
- [Configuring a Server for Remote Communications](#)
- [Configuring a Gateway for Remote Communications](#)

You can use the IVP, provided in the AppBuilder installation in the IVP directory, to test both client and server configurations and insure that they are configured correctly. See the *IVP User Guide* for details on installing, configuring, and running Java and C client tests; C server and EJB and Webservices tests; and CICS, Batch, IMS, and other mainframe tests.

Configuring a Java Client for Remote Communications

You can configure a Java client for RPC communication with remote servers. To configure the client, right-click the Java client icon (Java under Clients) and select *Properties*. This invokes the Java Client Properties dialog as shown in the following figure.

Java client communications properties dialog



The tabs and properties for this dialog are described in the following table.

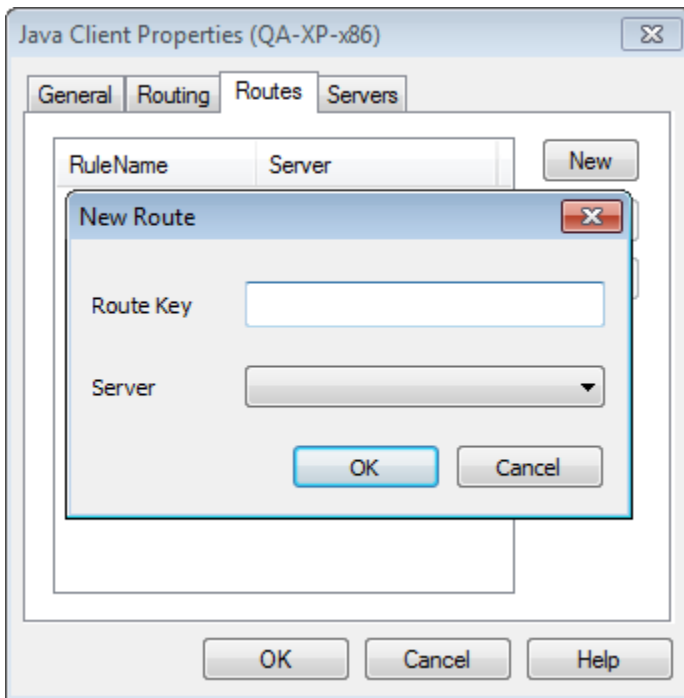
Java Client properties

Tab	Property	Description and Valid Values
General		General parameters relating to Java client communications.
	Logical Unit of Work	Select either nonconversational or conversational.
	Close Semantics	Select commit or abort
	RPC Timeout	Default is 300 (seconds).
	Polling Interval	Default is 250 (milliseconds).
	Reflection Order	Default is Java Order
Routing		Parameters relating to the routing of communications.
	Max Routes	Maximum integer number of allowed routes. Default is 2.
	Number of Retries	Integer number of retries. Default is 0.
	Retry Interval	Integer number of milliseconds before retry. Default is 0.
	Routes Key	Whether to use shortname or longname. Default is shortname.
Routes		For more general background on routing, refer to Understanding Routing .
	Rule Name	Rule name to use as a route key.
	Server	Name of the server
Servers		Parameters relating to the communications with the server.
	Server Name	Name of the server

	Server Type	Choose one: <ul style="list-style-type: none">• Custom• EJB (see EJB Server)• JMS (see JMS Server)• NETE (see NetEssential Server)• RMI (see RMI Server)• Webservice
--	-------------	---

To add a Java route, from the Routes tab, click *New* . In the dialog, type the route key and server name as described in [Java Client properties](#).

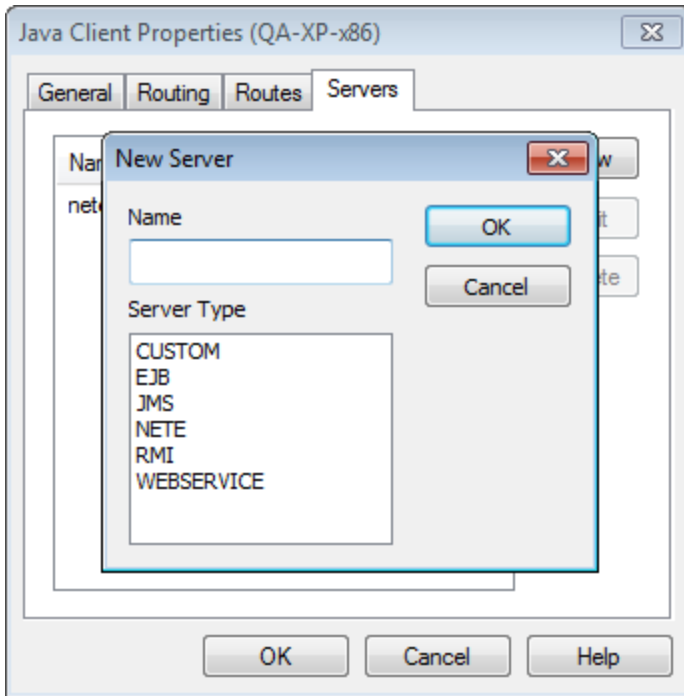
Adding a Java route



Adding a Server for Client Communications

To add a server, from the Servers tab, click *New* . In the dialog, as shown in the following figure, type a server name and select the server type.

Adding a server



This topic includes adding and configuring the following servers:

- [Custom Server](#)
- [EJB Server](#)
- [JMS Server](#)
- [NetEssential Server](#)
- [RMI Server](#)
- [Webservice Server](#)

Custom Server

If you selected a Custom server and typed a name for it, a properties dialog displays. In this dialog you can specify the following:

Summary of Custom server properties

Server Property	Description
CLASS	A Java class implementing AbfCustomTransport interface.

EJB Server

If you selected an EJB (Enterprise Java Bean) server and typed a name for that server, a properties dialog displays. The properties for this server are summarized in the following table:

Summary of EJB server properties

Server Property	Description
INITIAL_CONTEXT_FACTORY	For WebLogic, select weblogic.jndi.WLInitialContextFactory For WebSphere, select com.ibm.ejs.ns.jndi.CNInitialContextFactory For WebSphere 4, select com.ibm.websphere.naming.WsnInitialContextFactory
PROVIDER_URL	iiop://localhost:900 which is < protocol >://< host_name >:< port_number > For example, iiop://localhost:900 for WebSphere t3://localhost:7001 for WebLogic

JMS Server

If you selected a JMS (Java Messaging Service) server and typed a name for that server, a properties dialog displays. The properties for this server are summarized in the following table:

Summary of JMS server properties

Server Property	Description
JMS_INITIAL_CONTEXT_FACTORY	com.sun.jndi.fscontext.ReffSContextFactory Name of initial context factory. For use with JNDI.
JMS_PROVIDER_URL	Provider URL. For use with JNDI. Example: <i>file:/C:/JNDI-Directory</i> .
MQ_QUEUE_MANAGER	Name of MQSeries queue manager.
MQ_CHANNEL	Name of MQSeries channel.
MQ_RECV_QUEUE	Name of MQSeries receiving queue.
MQ_SEND_QUEUE	Name of MQSeries sending queue.
MQ_HOST_NAME	Host name of the remote machine.
MQ_PORT	TCP/IP port to connect.
MQ_CODEPAGE	Server codepage for local conversion.
MQ_SERVERID	Server identifier.
MQ_PROTOCOL	Protocol to use to connect to the server: <ul style="list-style-type: none"> • TCPIP - TCP/IP • PMTCPIP - performance marshalled TCP/IP • POTCPIP - performance marshalled OpenCOBOL TCP/IP
MQ_TIMEOUT	Time to wait before timing out.

NetEssential Server

If you selected a NETE (NetEssential) server and typed a name for that server, a properties dialog appears. The properties for this server are summarized in the following table:

Summary of NetEssential server properties

Server Property	Description
HOST_NAME	Host name of the remote machine
PROTOCOL	Protocol to use to connect to the server: <ul style="list-style-type: none"> • TCPIP - TCP/IP • PMTCPIP - performance marshalled TCP/IP • POTCPIP - performance marshalled OpenCOBOL TCP/IP For information on performance marshalling, refer to Understanding Performance Marshalling
PORT	TCP/IP port to connect
CODEPAGE	Server codepage for local conversion
SERVERID	Server identifier

RMI Server

If you selected a RMI (remote method invocation) server and typed a name for that server, a properties dialog displays. The properties for this server are summarized in the following table:

Summary of RMI server properties

Server Property	Description
REGISTRY_URL	< protocol >://< host_name >:< port_number > For example, iiop://localhost:900.

Webservice Server

If you selected a Webservice server and typed a name for that server, a properties dialog displays. The properties for this server include:

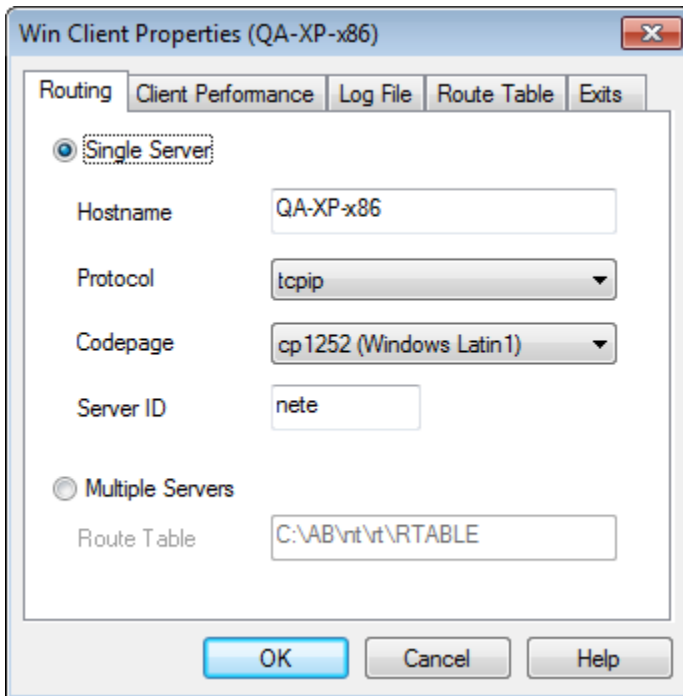
Summary of Webservice properties

Server Property	Description
HOST_URL	This setting specifies the host name of the remote machine. The format for the specification is provided in the drop-down box.

Configuring a Windows Client for Remote Communications

You can configure a Windows client for RPC communication with remote servers. To configure the client, right-click the Windows client icon (Win under Clients) and select *Properties*.

Windows client communications properties dialog



The properties are described in the following table:

Windows client properties

Tab	Property	Description and Valid Values
Routing		Set whether the client routes service request to a single-server or multiple servers.
	Single Server	Check if using single server configuration. Refer to Configuring for a Single Server .
	Single Server Hostname	Host name or machine name (see Host name by protocol).
	Single Server Protocol	tcpip (for TCP/IP) or NPIPE (for named pipes).
	Single Server codepage	Select from the list of codepages. (Refer to Supported CodePages for a list of supported codepages.)
	Single Server ID	Server ID
	Multiple Servers	Check if using multiple server configuration. Refer to Configuring for Multiple Servers .

Client Performance		Set the performance characteristics for client/server interaction. Refer to Configuring Performance Settings .
	Logical unit of work	Default is NONCONVERSATIONAL.
	Close semantics	Default is COMMIT.
	Timeout (Sec.)	Default is 300 ms. This determines how long the client, server, or gateway tries to reach the target server before timing out, in seconds.
	Max number servers	Default is 100.
Log File		Set the kind of information to be written to the client's error log (trace) file. Refer to Configuring Trace Log File Settings .
	Log file name	Default is C:\AppBuilder\trace.out.
	Error level	Default is ERRORS.
	Log file size	Default is 1M.
	Backup file name	Default is C:\AppBuilder\trace.bak.
Exits		Set the security exits used by the client (if any).Refer to Configuring Security Settings .
	Exit ID	The unique identifier of the exit.
	Value	The current value of the exit.
	Status	The status (enabled or disabled) of the exit.
Route Table		For multiple-server routing. Refer to Configuring for Multiple Servers . For more general background about routing, refer to Understanding Routing .

Configuring Performance Settings



Do not change an agent's performance settings without first consulting Customer Support.

You can accept the default settings for performance settings or you can specify this performance setting. *Logical unit of work* determines how AppBuilder handles pending database transactions:

Logical unit of work values

Select	If you want AppBuilder
Nonconversational	To commit pending database transactions after the service request completes and then close the server connection.
Pseudoconversational	To commit pending database transactions after the service request completes but leave the server connection open.
Conversational	To commit or abort pending database transactions when the client context closes or as specified explicitly in the code. In the former case, the value of the Close semantics field (see below) determines whether AppBuilder commits or aborts pending transactions. The server connection closes when the client context closes.

The following table shows the relationship between the logical unit of work values and their values in previous releases.

Logical unit of work values

In AppBuilder	In Previous Releases
NONCONVERSATIONAL	52LOCAL
PSEUDOCONVERSATIONAL	LOCAL
CONVERSATIONAL	REMOTE


Close semantics determines how AppBuilder handles pending database transactions when the client context closes. This field is relevant only when the logical unit of work is set to *CONVERSATIONAL* :

- Select *COMMIT* for AppBuilder to commit pending database transactions when the client context closes

- Select *ABORT* for AppBuilder to roll back pending database transactions when the client context closes

The close semantics setting is irrelevant for gateways and forwarding servers.

Timeout determines how long the client, server, or gateway tries to reach the target server before timing out, in seconds. If you specify 0, no timeout occurs.

 To prevent a client timing out before a gateway, make sure the client has a greater timeout value than the gateway.

Maximum number of servers determines the maximum number of servers with which the client can communicate. The maximum number of servers setting is irrelevant for gateways, forwarding servers, and agents.

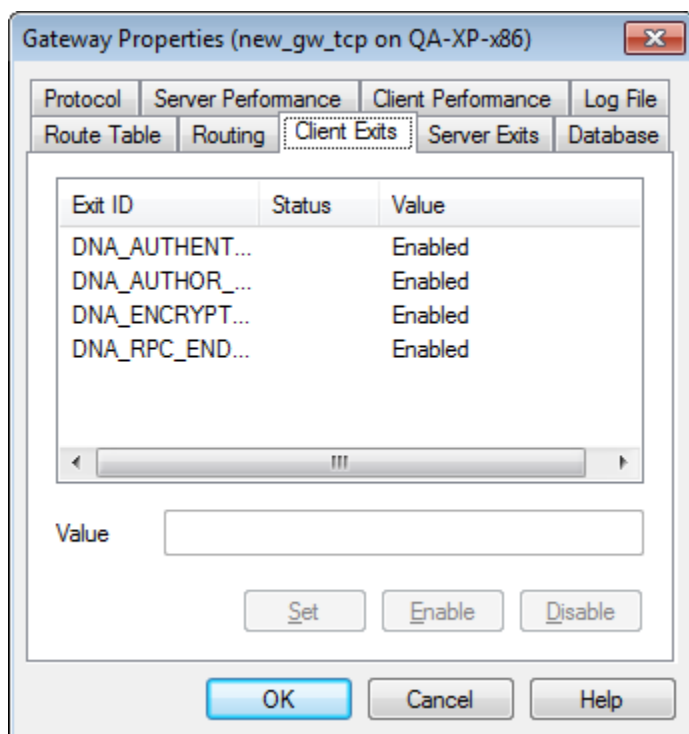
Configuring Security Settings

Use the Security configuration to specify the security exits that the client, gateway, or forwarding server employs to do the following:


- Obtain authentication information (with Authentication exit)
- Obtain authorization information (with Authorization exit)
- Log remote service usage (with RPC-end exit)
- Require encrypt and decrypt passwords (with Encryption exit)

In the Exits tab (on the client or server) or the Client Exits and Server Exits tabs (on the gateway), you can specify an exit by specifying the Value for a selected Exit ID and set, enable, disable the exit. An example of the interface is shown in the following figure:

Client Exits tab example



In the Clients Exits tab, you can specify any of the four types of exits. In the Server Exits tab, you can specify authentication and authorization exits.

 A gateway or forwarding server uses different security exits on the client and server sides. Do not specify the same security exits on each side.

Security configuration also lets you specify an authentication exit for a Windows eventing agent that talks directly over LU6.2 to a security-enabled mainframe environment. See [If Your Windows Host Talks to the Mainframe](#). By default, no security exits are invoked. See [C Client and Server Exits](#) for more information.

On workstation hosts, security exits take the form of a DLL. For example, you might enter

```
C:\AppBuilder\SAMPLES\C\EXITS\CLIENT\DLL\authen.dll
```


for the authentication exit.

If Your Windows Host Talks to the Mainframe

If your parent machine is a security-enabled mainframe listening over LU6.2, your eventing agent must point to a client-side authentication exit. The authentication exit passes the user name and password to LU6.2. There is no need to code a server-side exit.

If you configured the host for remote procedure calls (RPCs) as well as eventing, type the path name of the authentication exit you specified for the RPC client. Ignore the prompts for the other exits; these are irrelevant for an eventing agent.

Configuring Trace Log File Settings

You can accept the default settings or you can specify the following settings about the log file:

Log file name fully specified path name (including the drive letter) of the file to which AppBuilder writes messages.



Errors that occur before initialization are reported in the file `c:\dnatrace.out`.

Error level determines the kind of information that will be written to the log file:

Levels of tracing

Error level	Explanation
Errors	Logs errors only.
Errors and Lengths	Logs input and output view lengths as well as errors.
Errors and Data	Logs input and output view lengths with more data, as well as errors. Lengths are logged with the type of view data and number of bytes written per view determined by the values entered for data dump type, data dump input size, and data dump output size, respectively.
Errors and Traces	Logs trace information for successful and unsuccessful calls as well as errors. For successful calls, it logs the time the function was called, the time the function exited, and input and output view lengths, with the number of bytes to be written per view determined by the values entered for data dump input size and data dump output size. For unsuccessful calls, it also logs errors and status view data. A hex dump is included for both successful and unsuccessful calls.



Select ERRORS AND TRACES only when debugging AppBuilder applications. It can degrade system performance and use excessive disk space.

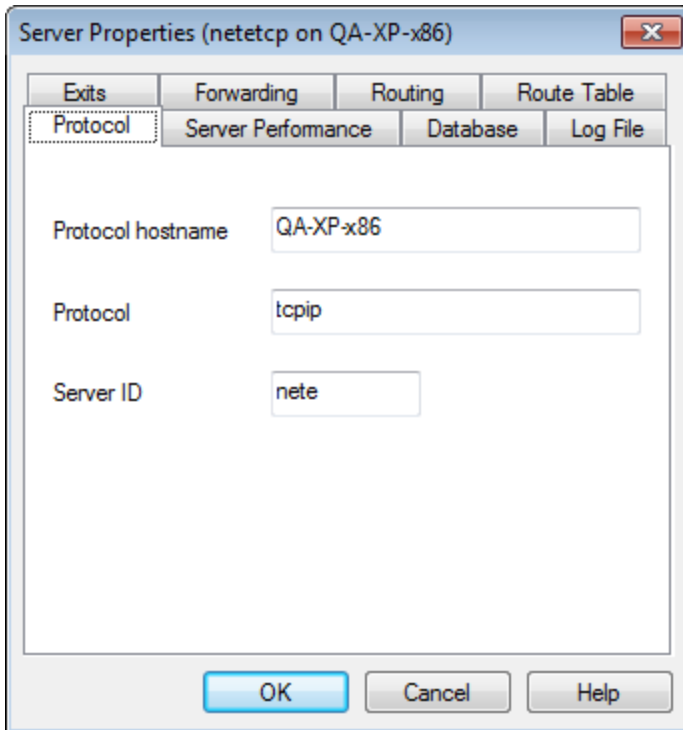
Log file size is the size of the log file, in bytes. Specify an integer followed by an upper-case K (for kilobytes) or M (for megabytes). When the log file exceeds this size, the information is deleted and stored in the backup log file.

Backup file name is the fully specified path name, including the drive letter, of the file to which AppBuilder writes information when the size of the main log file exceeds its maximum size.

Configuring a Server for Remote Communications

You can configure an execution RPC server for use on the local host. You can configure the default execution servers created during installation or create new execution servers. If the server accesses a database, you must also specify the database name. To configure the server, right-click the server icon and select *Server Properties*. Refer to [Creating a Windows Server](#) for information about creating a server.

Server communications properties dialog



The properties are described in the following table:

Server properties

Tab	Property	Description and Valid Values
Protocol		The network protocol for communication with this server.
	Protocol hostname	Not editable; just for viewing.
	Protocol	Not editable; just for viewing.
	Server ID	Not editable; just for viewing.
Server Performance		Select the properties related to the performance of the server.
	Server type	Default is FORKING. Alternate choice is BANKING.
	Timeout (Sec.)	Default is 600 seconds.
	Services Directory	Default includes <AppBuilder>\nt\sys\bin and < AppBuilder >\nt\rt\SERVER
Database		Set the database to use.
	System DBMS	Not editable; just for viewing.
	Database name	Enter the database name if any.
Log File		Set the kind of information to be written to the server's error log (trace) file. Refer to Configuring Trace Log File Settings .
	Log file name	Default is C:\AppBuilder\ <i>machinename</i> \trace.out.
	Error level	Default is ERRORS.
	Log file size	Default is 1M.
	Backup file name	Default is C:\AppBuilder\ <i>machinename</i> \trace.bak.
Exits		Set the security exits used by the gateway (if any).Refer to Configuring Security Settings .
	Exit ID	The unique identifier of the exit.

	Value	The current value of the exit.
	Status	The status (enabled or disabled) of the exit.
Forwarding		Determine forwarding characteristics.
	Use forwarding	Default is not used (unchecked).
	Try remotely first	Venue of Service Execution. If used, select one or the other.
	Try locally first	Venue of Service Execution. If used, select one or the other.
Routing		Set whether the client routes service request to a single-server or multiple servers
	Single Server Hostname	Host name or machine name (see Host name by protocol)
	Single Server Protocol	tcpip (for TCP/IP) or NPIPE (for named pipes)
	Single Server codepage	Select from the list of codepages (Refer to Supported CodePages for a list of supported codepages.)
	Single Server ID	Server ID
	Multiple Servers	Check if using multiple server configuration. Refer to Configuring for Multiple Servers .
Route Table		For multiple-server routing. Refer to Configuring for Multiple Servers . For more general background on routing, refer to Understanding Routing .

Configuring a Gateway for Remote Communications

Gateway configuration prepares an AppBuilder server to transparently reroute client requests across protocol boundaries. The default configuration does not include a gateway. You can use the procedures in this section to create one or to configure an existing gateway. Refer to [Creating a Windows Server](#) for information on creating a gateway.

Gateways have dual personalities: they have characteristics of both server and client. To configure a gateway, you specify its *inbound* and *outbound* characteristics:

- *Inbound* characteristics configure the gateway's server-side features; how the gateway handles requests from remote clients
- *Outbound* characteristics configure the gateway's client-side features; how the gateway directs requests to remote servers

To configure a gateway, right-click the gateway icon and select *Properties* .

Gateway communications properties dialog

The screenshot shows a dialog box titled "Gateway Properties (new_gw_tcp on QA-XP-x86)". It has several tabs: "Route Table", "Routing", "Client Exits", "Server Exits", "Database", "Protocol", "Server Performance", "Client Performance", and "Log File". The "Protocol" tab is currently selected. Inside this tab, there are three text input fields: "Protocol hostname" with the value "QA-XP-x86", "Protocol" with the value "tcpip", and "Server ID" with the value "new_gw_". At the bottom of the dialog, there are three buttons: "OK", "Cancel", and "Help".

The properties are described in the following table:

Gateway properties

Tab	Property	Description and Valid Values
Protocol		The network protocol for communication with this gateway.
	Protocol hostname	Not editable; just for viewing.
	Protocol	Not editable; just for viewing.
	Server ID	Not editable; just for viewing.
Server Performance		Select the properties related to the performance of the server.
	Server type	Default is FORKING. Alternate choice is BANKING.
	Timeout (Sec.)	Default is 600 seconds.
	Services Directory	Default includes < AppBuilder >\nt\sys\bin and < AppBuilder >\nt\rt\SERVER
Client Performance		Set the performance characteristics for client/server interaction.
	Logical unit of work	Default is NONCONVERSATIONAL.
	Close semantics	Default is COMMIT.
	Timeout (Sec.)	Default is 300 seconds.
	Max number servers	Default is 100.
Client Exits		Set the security exits used by the client (if any). Refer to Configuring Security Settings .
	Exit ID	The unique identifier of the exit.
	Value	The current value of the exit.
	Status	The status (enabled or disabled) of the exit.
Server Exits		Set the security exits used by the gateway (if any). Refer to Configuring Security Settings .
	Exit ID	The unique identifier of the exit.
	Value	The current value of the exit.
	Status	The status (enabled or disabled) of the exit.
Database		Set the database to use.
	System DBMS	Not editable; just for viewing.
	Database name	Type the database name.
Log File		Set the kind of information to be written to the gateway's error log (trace) file. Refer to Configuring Trace Log File Settings .
	Log file name	Default is < AppBuilder >\machinename\trace.out.

	Error level	Determines what level of debug information is available in the trace output log file. Default is ERRORS. Choices include the following: <ul style="list-style-type: none"> • ERRORS - logs only error information • ERRORS_AND_DATA - logs error information and reporting on input/output view data on the client side • ERRORS_AND_LENGTH - logs error information and input/output view lengths • ERRORS_AND_TRACING - all the available information.
	Log file size	Default is 1M (for 1 megabyte).
	Backup file name	Specify the path and name of the file to which tracing information is written when the size of the Log file exceeds the Log file size. Default is < AppBuilder >\ machine_name \trace.bak.
Route Table		For multiple-server routing. Refer to Configuring for Multiple Servers . For more general background on routing, refer to Understanding Routing .
Routing		Set whether the client routes service request to a single-server or multiple servers
	Single Server Hostname	Host name or machine name (see Host name by protocol)
	Single Server Protocol	tcpip (for TCP/IP) or NPIPE (for named pipes)
	Single Server codepage	Select from the list of codepages (Refer to Supported CodePages for a list of supported codepages.)
	Single Server ID	server ID
	Multiple Servers	Check if using multiple server configuration. Refer to Configuring for Multiple Servers .

Understanding Routing

This topic includes the following:

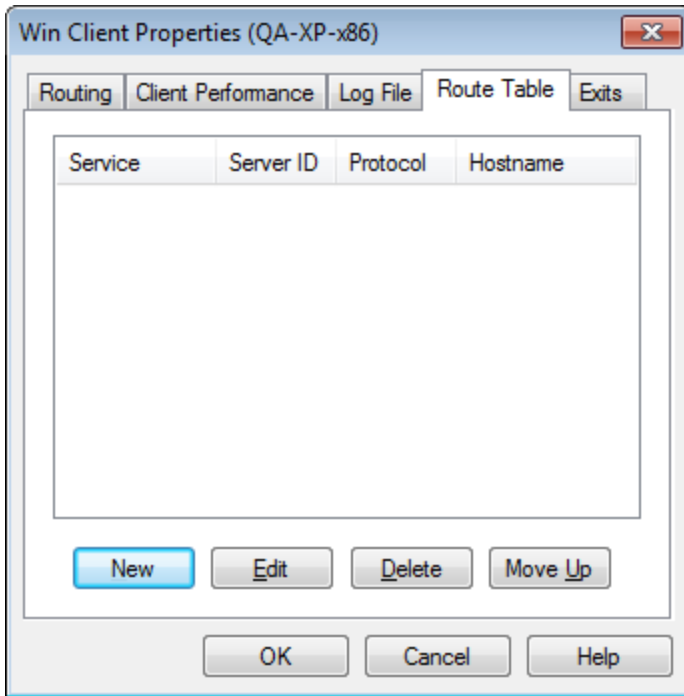
- [Editing the Route Table](#)
- [Using Wild Cards and Priority](#)
- [Understanding Routing Types](#)
- [Configuring Routing](#)

Editing the Route Table

The route table is a simple text file used for RPCs. The route table identifies the routes from the client or gateway to the services the client requests. To edit the route table, from the Management Console:

1. Select the Windows client icon (Win) and from the right-click menu, select *Properties* . The Windows client properties window displays. (See [Windows client communications properties dialog](#).)
2. Select the Route Table tab. It is enabled only when the Multiple Servers option is selected on the Routing tab of the dialog box.

Selecting route table



Route table entry action

Button	Action
New	Evokes a Route Table Entry dialog so that you can add a new entry to the route table. The new entry appears in the list when you close the Route Table Entry dialog.
Edit	After selecting one of the entries in the list, click this button to bring up the Route Table Entry dialog with the values for this entry. You can then edit the values for this entry. The updated entry appears in the list when you close the Route Table Entry dialog.
Delete	After selecting one of the entries in the list, click this button to remove that entry from the list in the route table.
Move Up	After selecting one of the entries in the list that is not at the top, click this button to move the entry up one position in the list.

Route table entry dialog

The screenshot shows a 'Route Entry' dialog box with the following fields and values:

- Rule name: \$ANY
- Hostname: (empty)
- Protocol: tcpip
- Server ID: (empty)
- Code Page: cp1252
- Priority: (empty)
- Data-Dependent Routing
- Static section:
 - Length: (empty)
 - From: (empty)
 - Type: (empty)
 - To: (empty)
 - Offset: (empty)

Buttons: OK, Cancel

Route table entry values

Field	Description
Rule name	Case-sensitive. The name of the rule to be invoked. Use the short name or the system-assigned name, the system ID, not the long name for the rule. Refer to Using Wild Cards and Priority .
Host name	Case-sensitive. The ID of the server machine as known to the protocol for the route, in 32 characters or less. You can use periods in the workgroup implementation name so that you can specify either IP addresses or domain names. <ul style="list-style-type: none"> • For TCP/IP, the host name of the machine in the hosts file. • For TCP/IP, you can obtain the host name with the Network Information Service (NIS) or the Domain Name Server (DNS). You can obtain the Server ID with NIS. • For Named Pipes, the Machine ID or, if a requested service resides on the same machine as the client, LOCAL. • For LU6.2, the client-side Partner LU alias. • For LU2, the value substituted for HOSTNAME in the LU2 logon script? a CICS region name, for example. If HOSTNAME is not in use, any value.
Protocol	Case-sensitive. The network protocol for the route. <ul style="list-style-type: none"> • lu2 - for LU2 • lu62 - for LU6.2 • NPIPE - for Named Pipes • pmlu2 - for performance marshalling LU2 • pmlu62 - for performance marshalling LU6.2 • pmtcpip - for performance marshalling TCP/IP • potcpip - for performance marshalling TCP/IP for OpenCOBOL • tcpip - for TCP/IP

Server ID	<p>Case-sensitive. An alias for an endpoint, in 32 characters or less. The ID of the execution server to which the client routes the service request, as known to the protocol for the route, in 32 characters or less:</p> <ul style="list-style-type: none"> • For UNIX and Windows family servers listening over TCP/IP, the server name in the services file. • For Windows family servers listening over Named Pipes, the server pipe name. • For UNIX and Windows family servers listening over LU6.2, the server-side TP name. <p>For AIX autostart servers listening over LU6.2, specify the file name in the server-side Local transaction program path.</p> <ul style="list-style-type: none"> • For iSeries servers listening over LU6.2, the name of the library in which the service resides, followed by a forward slash and the name of the command language program that starts the server. • For mainframe CICS servers listening over LU2 or LU6.2, the server-side CICS transaction ID. • For IMS servers listening over LU2, the server-side IMS transaction code. • For IMS servers listening over LU6.2, the server-side TP name in the APPC/MVS TP profile.
Codepage	<p>Optional. The codepage in use at the server node. If the codepage specified in the route table differs from the codepage specified in the LOCAL_CODEPAGE entry in the NLS section of the client's dna.ini file, the system converts the data to be transmitted to the codepage specified in the route table. If you enter a hyphen (-) and the codepage specified in the LOCAL_CODEPAGE variable in the NLS section of the client dna.ini differs from the codepage specified in the LOCAL_CODEPAGE variable of the NLS section of the server dna.ini, the system performs codepage conversion on the server. Conversions to double-byte character sets rely on native operating-system routines. Refer to Supported CodePages for a list of supported codepages.</p>
Priority	<p>Optional. An integer greater than or equal to zero specifying the ordinality of a route. Default is zero, which specifies that a route has highest priority. A hyphen or blank is equivalent to zero. As noted in Using Wild Cards and Priority, the keyword \$ANY in the Service field lets you specify the route to every service on a machine that has the values specified in the remaining fields of the route table entry. See the example in Priorities Setting Example.</p>
Data-dependent	<p>Check this box for data-dependent routing or to allow you to edit any of the fields below.</p>
Static	
Length	<p>Optional. For data-dependent routing, the length in bytes of the data to be checked. Length is used only if the data are type char.</p>
Type	<p>Optional. For data-dependent routing, the type of data at the specified offset. Can be:</p> <ul style="list-style-type: none"> • char (for character data) • long (for a four-byte integer) • short (for a two-byte integer) <p>Int is integer and is the same as long.</p>
Offset	<p>Optional. For data-dependent routing, the symbolic name of the data field to be checked, as it appears in the control-structure array for the service request. For subfields, use a format such as the following: view1.view2.field1 You can specify an array index in brackets after the subview name: view1.view2[5].field1 Array element 1 is the first position in the array. Maximum length of any element in the format string is 32 characters. Maximum length of the entire format string is 64 characters. For error-dependent routing, the Offset field must be set to the keyword DNAERROR. The Offset field may be set to the offset from the beginning of the data structure. Offset 0 is the first position in the data.</p>
From	<p>Optional. For data-dependent routing, the lowest value that satisfies the criteria against which the data is checked. Double-quoted strings – "Smith, Chris" – are supported. Maximum length is 128 characters. For error-dependent routing, the From field is the lowest error number in a range of error numbers. To specify multiple individual error numbers, separate the numbers with a vertical bar – 10</p>
To	<p>Optional. For data-dependent routing, the highest value that satisfies the criteria against which the data is checked. Double-quoted strings – "Smith, Chris" – are supported. Maximum length is 128 characters. For error-dependent routing, the To field is the highest error number in the range.</p>

Use a hyphen (-) to specify an unused field, except as described in [Using Wild Cards and Priority](#).

Understanding Route Table Format

A route table is a simple text file with two kind of lines:

Route Table formats

Entry type	Description
------------	-------------

Route entries	<i>Route entries</i> begin in column one and consist of eleven case-sensitive fields separated by white space. The fields include the name of the service, the machine it executes on, the protocol and channel of the server connection, and the priority and data-dependency of the route. An optional or unused field is denoted by a hyphen (-). Route table fields correspond to the settings in the communications configuration file (dna.ini). You can use dna.ini for RPC routing if you prefer, but this restricts you to specifying only a single route from the local host.
Comment lines	<i>Comment lines</i> begin with a pound sign (#) in column one.

The route table for a communications client is initially empty except for commented copyright information.

Initializing the Route Table

By default, an RPC client obtains routing information from the dna.ini file. To enable route table routing, you have two choices:

- Reconfigure the client with the host configuration tool
- Set the following variables in the ROUTING section of the client dna.ini file

To edit the dna.ini file, select the machine icon in the Management Console and select *Edit INI Properties*.

Clients obtain the path to the dna.ini file from an environment variable named DNAINI. Make sure you set the path to the dna.ini in the client environment:

```
SET DNAINI= path.
```

UNIX clients set this variable in their startup script.

Settings in routing section of dna.ini

Variable	Setting for route table routing
NAME_SERVICE	ROUTE_TABLE
DNA_MAX_ROUTES	Maximum number of routes the system attempts for a given service request. The default is 1.
DNA_NO_OF_RETRIES	Number of times the system retries an individual route. The default is 0.
DNA_RETRY_INTERVAL	Number of seconds between the failure of a route and a retry. The default is 0. For each retry, the retry interval increments by a factor of two: if set to 10 seconds, it increments to 20 seconds after the first retry, 40 seconds after the second retry, and so forth.
ROUTETBL	Full pathname of the route table. The default route table name is rtable. The default location is platform-specific: <ul style="list-style-type: none"> • On Windows, the default location is: < AppBuilder >\nt\rt\rtable • On UNIX hosts, the default location is: \$HOME/nes_apps/< client_dir >/rtable • On mainframe hosts, the default location is: <HIGH_LEVEL_QUALIFIER>.rtable

Using Wild Cards and Priority

You can use wild cards in the route table. In place of a rule name entry of a route table, you can use the keyword \$ANY to indicate every route that matches all of the values specified in the remaining fields. For example, the route table entry

```
$ANY - - - - AIX_1 lu62 READLU62 - 1
```

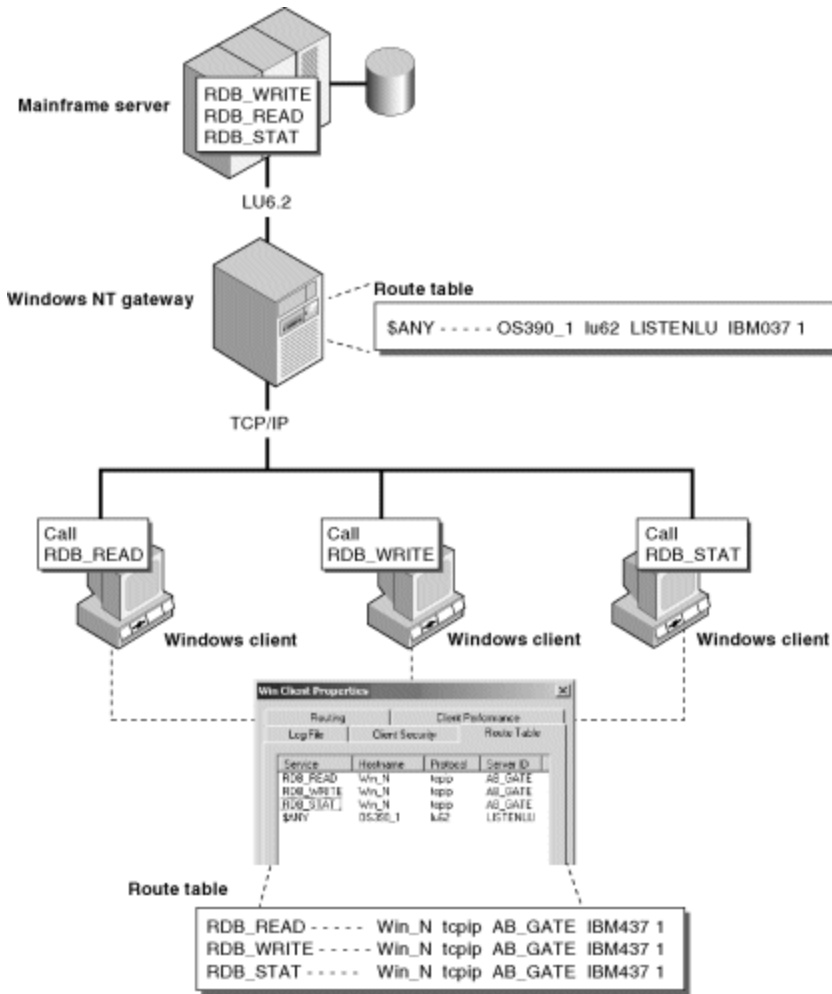
refers to every rule on the AIX_1 machine that listens on the READLU62 channel over LU6.2.

\$ANY enables you to implement complex routing schemes with just a few route table entries?sometimes as few as one. Consider the gateway example in the [Gateway Routing](#) section: two clients call the RDB_READ service and a gateway reroutes the request to another machine. While the example shows how to implement simple gateway routing, it is not realistic. More typically, multiple clients – each calling multiple services – would route requests to the gateway.

Rather than creating a route table with an entry for each service, you can use \$ANY to reroute every request with a single entry. Not only is the route table easier to create, it is easier to maintain. In fact, when new services are introduced into the system, you may not need to update the route table at all.

The following figure recasts the gateway example from the previous section using \$ANY. A single route table entry on the Windows gateway forwards requests from multiple Windows clients to the mainframe server. The route priority determines when the \$ANY route is evaluated.

Wild-card routing



Of course, the clients in this figure could use \$ANY in their route tables, since the service requests are all routed to the same server, namely, the Windows gateway:

```
$ANY - - - - Win_N npipe AB_GATE IBM437 1
```

Priorities Setting Example

For example, the following entry refers to every service on the AIX_1 machine that listens on the READLU62 channel over LU6.2:

```
$ANY - - - - AIX_1 lu62 READLU62 - 1
```

Each route would have a priority of 1, unless the route was explicitly specified with a higher priority:

```
# try RDB_READ on AIX_1 machine first, then use $ANY
# routes to try alternate machines
RDB_READ - - - - AIX_1 lu62 READLU62 - 0
$ANY - - - - AIX_1 lu62 READLU62 - 2
$ANY - - - - Win_1 tcpip READTCP - 1
```

A \$ANY route with a priority equal to that of an explicitly specified but otherwise identical route is ignored:

```
# ignore $ANY route for all RDB_READ service requests
$ANY - - - - AIX_1 lu62 READLU62 - 0
RDB_READ - - - - AIX_1 lu62 READLU62 - 0
```

A \$ANY entry is ignored if it has the same priority as a previous \$ANY entry, even if the entries refer to different routes:

```
$ANY - - - - WinNT_1 tcpip READTCP - 0
$ANY - - - - AIX_1 lu62 READLU62 - 0
```

The second \$ANY entry is ignored. Use a higher priority instead:

```
$ANY - - - - WinNT_1 tcpip READTCP - 0
$ANY - - - - AIX_1 lu62 READLU62 - 1
```

Understanding Routing Types

Programs that use remote procedure calls (RPCs) execute synchronously: clients request a service, wait for a response, and continue processing

only after receiving it. AppBuilder applications can route service requests in their program code, but more typically use initialization file (dna.ini) routing or a route table on the client. This topic describes how you initialize the route table and create route table entries for server and gateway routing. You can modify route tables at runtime without interrupting network service. This topic includes:

- [Alternate Routing](#) - where the repeated failure of a service route results in the use of a lower-priority route
- [Data-Dependent Routing](#) - where a service route is dependent on the data being passed by the client
- [Error-Dependent Routing](#) - where a service route is dependent on the kind of error that the client encountered
- [Gateway Routing](#) - where a service route connects disparate machine types across a network or between networks

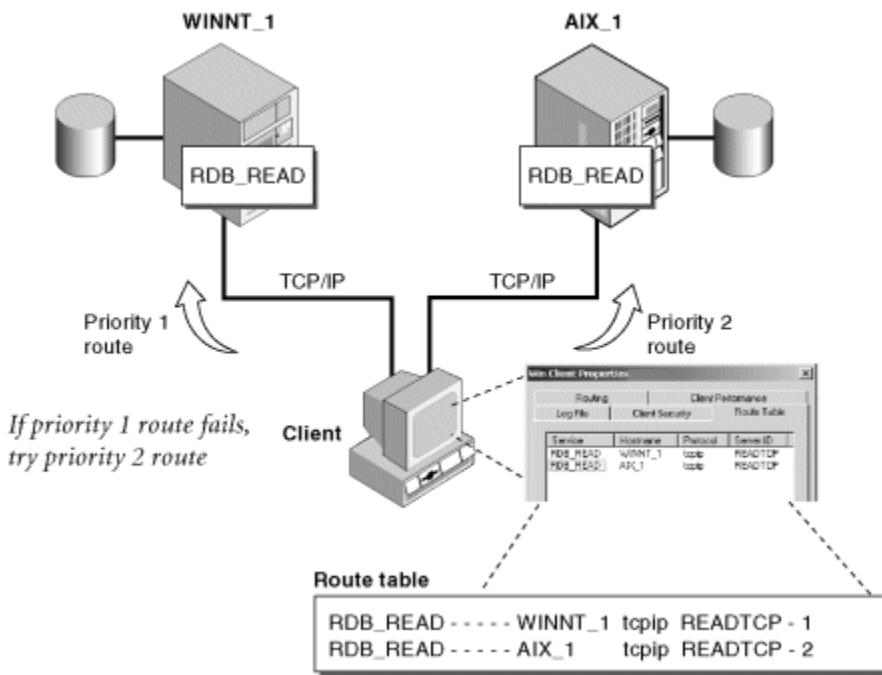
An example of a client-server application may involve several platforms. For example, a client program on a Windows machine may use the database-read service, RDB_READ. Service requests are routed to an HP-UX workstation over a TCP/IP connection and to an AIX workstation over an LU6.2 connection.

Alternate Routing

Alternate routing enables you to direct a service request to another machine or to other machines if the primary server is down or otherwise unreachable. AppBuilder implements alternate routing by assigning a priority to each instance of the service request in the route table. Higher priority routes are tried first, followed by lower-priority routes.

Consider the configuration in the following figure. Suppose you want the service RDB_READ on the IBM AIX machine to be executed only if the call to the service on the Windows machine fails. Also, you want the system to retry the Windows route twice before attempting to use a different route.

Alternate routing



You must specify two routing parameters in the dna.ini file and then create the necessary route table entries. Follow these steps:

1. Specify the following values in the ROUTING section of the client's dna.ini file:
 DNA_MAX_ROUTES=2
 DNA_NO_OF_RETRIES=2
 The first value specifies the maximum number of routes the system attempts for a given request for the RDB_READ service. The second value specifies the number of times the system retries an attempt.
2. Create two entries in the client's route table: one for the first priority route and one for the second priority route. Use the Management Console to Edit the Route Table. (Refer to [Editing the Route Table](#).)
 The last field in the each route table entry specifies the priority:
 RDB_READ - - - - WINNT_1 tcpip READTCP - 1
 RDB_READ - - - - AIX_1 tcpip READTCP ISO438 2
 Fields containing hyphens (-) are unused. The remaining fields have the following meanings:

Route table fields

Field	Meaning
RDB_READ	RDB_READ is the name of the service.

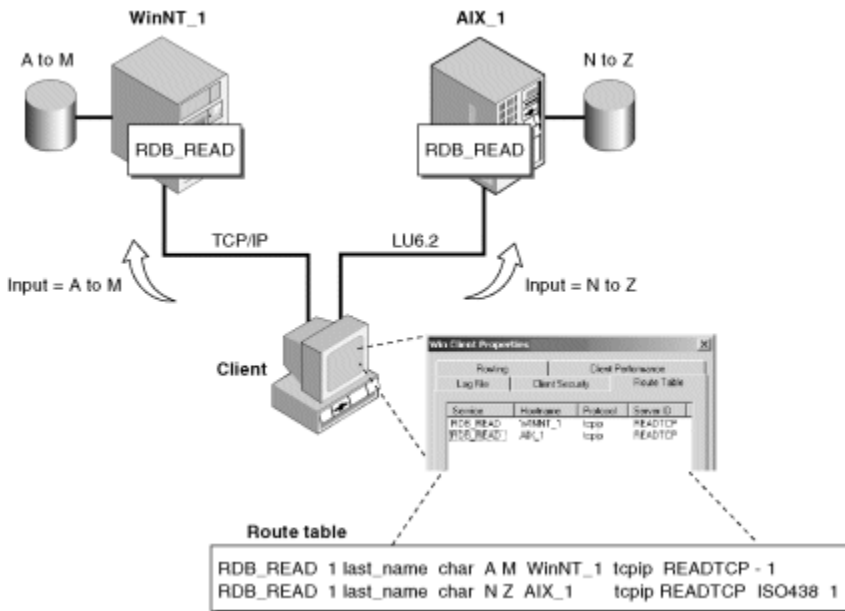
WINNT_1 AIX_1	WINNT_1 and AIX_1 are the IDs of the server machines as known to the protocols they use to communicate with the client: for the Windows machine, use the host name of the server machine in the client \tcip\etc\hosts file; for the AIX machine, use the client-side profile name.
tcip	tcip is the network protocol (TCP/IP) for the routes.
READTCP	READTCP is the channel (or server ID) the server listens on: respectively, the TCP/IP service name in the client \tcip\etc\services file on Windows.
ISO438	ISO438 is the codepage in use at the AIX server node.
1 2	1 and 2 are the relative priorities of the route table entries.

When you run the application, the system retries the TCP/IP route to the Windows server two times after the initial failure (three times total). After the third failure, it attempts to access the service on the AIX machine. It retries that route twice on failure as well.

Data-Dependent Routing

Using data-dependent routing you can choose a service route based on the data being passed by the client. For example, suppose the service routes in our sample application are of equal priority but you want the database on the Windows machine to contain information only on customers whose names begin with the letters A through M and the database on the AIX machine to contain information on all other customers.

Data-dependent routing



To test the input in the route table entry for each server and determine if the data matches your criteria, follow these steps:

1. In the client's dna.ini file, specify the maximum number of routes the system attempts for a given request for the RDB_READ service:
DNA_MAX_ROUTES=2
2. Specify the data-dependency of the service routes in the route table entries for each service request. Use the Management Console to edit the Route Table. (Refer to [Editing the Route Table](#).)
RDB_READ 1 last_name char A M WINNT_1 tcip READTCP - 1
RDB_READ 1 last_name char N Z AIX_1 tcip READTCP ISO438 1
Field 1 and fields 7-11 are the same as before. Fields 2-6 have the following meanings:

Route table fields

Field	Meaning
1	1 is the length of the data to be checked, in bytes.
last_name	last_name is the symbolic name of the data field to be checked, as it appears in the control-structure array for the service request.
char	char (character) is the type of data in the specified field.

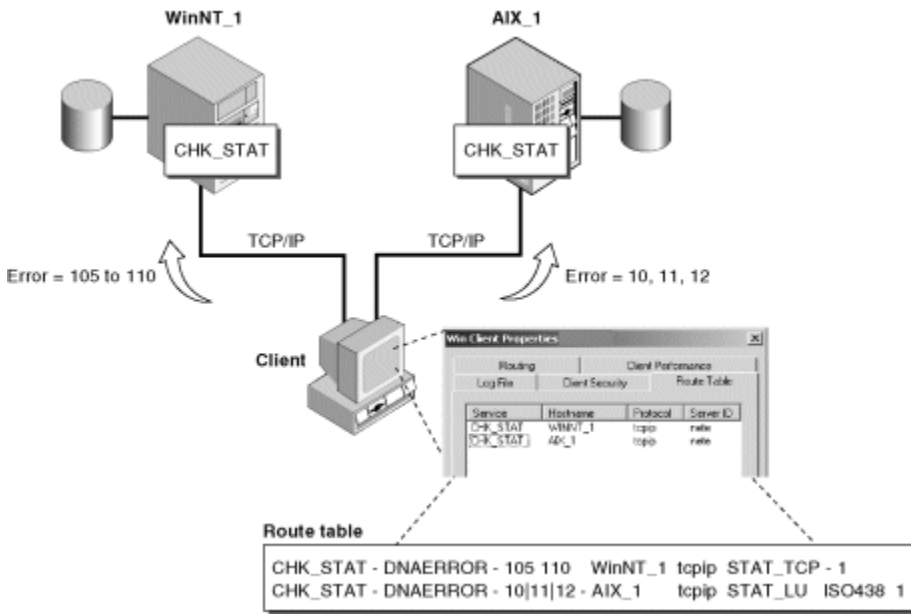
A N	A and N are the lowest values that satisfy the criteria for each route.
M Z	M and Z are the highest values that satisfy the criteria for each route.

You can use periods in the workgroup implementation name so that you can specify either IP addresses or domain names. When you run the application, the system uses the TCP/IP route to the Windows server if the server input begins with the letters A through M, and the TCP/IP route to the AIX server route if the input begins with N through Z. To specify data containing spaces, enclose the string in double quotes, for example, "Smith, Chris." You can include a double-quoted string in either limit field.

Error-Dependent Routing

Error-dependent routing is similar to data-dependent routing, but uses an AppBuilder error code instead of client data to base the routing decision. Suppose our sample application performs error-handling with the CHK_STAT service. The Windows server handles the class of errors denoted by AppBuilder communications error codes 105 through 110, and the AIX server handles the class of errors denoted by error codes 10, 11, and 12.

Error-dependent routing



Like data-dependent routing, you can specify the routing criteria in the route table entry. Error-dependent routing fields replace data-dependent routing fields in these entries: set the data offset field to the keyword DNAERROR, and specify the error number range in the from and to fields. Use the Management Console to Edit the Route Table. (Refer to [Editing the Route Table](#).)

```
CHK_STAT - DNAERROR - 105 110 WinNT_1 tcpip STAT_TCP - 1
CHK_STAT - DNAERROR - 10|11|12 - AIX_1 tcpip STAT_LU ISO438 1
```

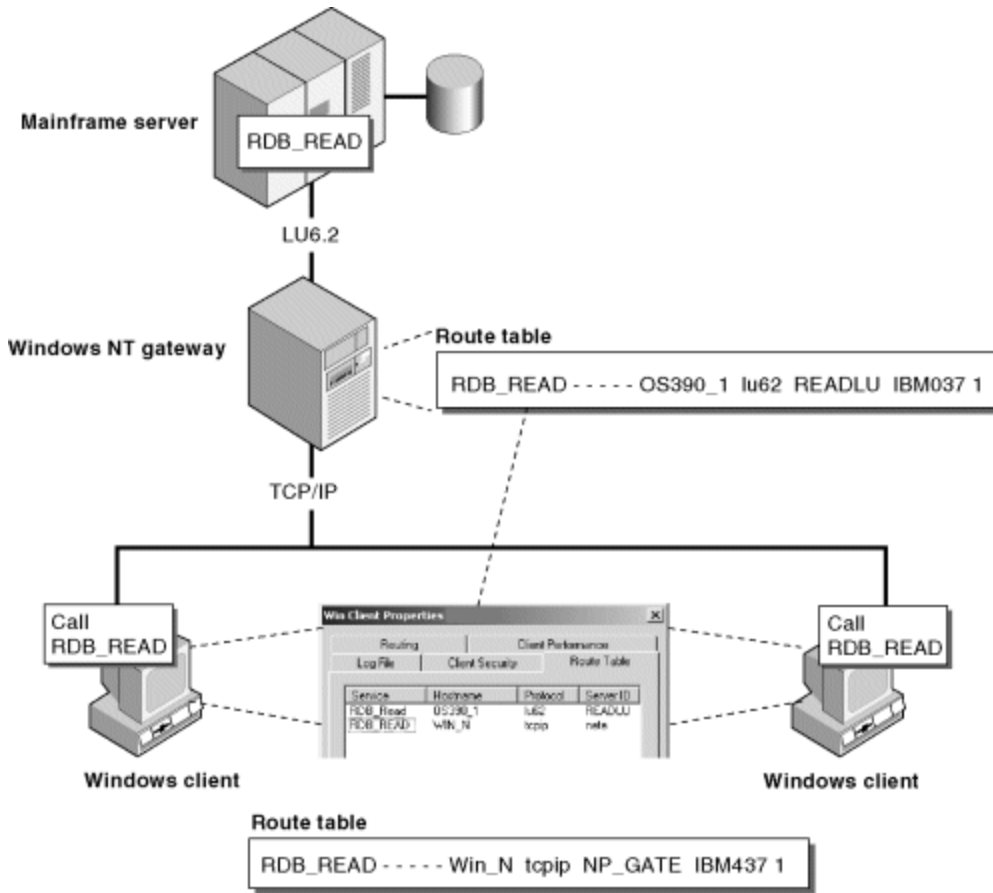
As shown above, you can specify multiple individual error codes in the from field by separating the error numbers with a vertical bar.

Gateway Routing

A gateway is a server that translates protocol information from one network format into another?it reroutes client requests rather than fulfilling them directly. You specify the routing information for a gateway not only on the client machine, but on the gateway machine as well.

Suppose our client program executes on a series of Windows machines in a TCP/IP network, while the RDB_READ service executes on a mainframe running IMS. (While CICS supports TCP/IP, IMS does not support it.) As a result, we need a gateway to connect to the mainframe over another network protocol. The following figure shows how you might accomplish this using a Windows gateway with a connection over LU6.2.

Gateway routing



To set up the gateway, complete the following steps:

- In the dna.ini file on both the client and gateway machines, specify the maximum number of routes the system attempts from either machine:
DNA_MAX_ROUTES=1
- Create an entry for the route to the gateway in the route table on the client machines. Use the Management Console to Edit the Route Table. (Refer to [Editing the Route Table](#).)
RDB_READ - - - - Win_N tcpip READNP IBM437 1
The fields have the following meanings:

Route table fields

Field	Meaning
RDB_READ	RDB_READ is the name of the service.
Win_N	Win_N is the Machine ID for the machine on which the gateway resides.
tcpip	tcpip (for TCP/IP) is the network protocol for the route.
READNP	READNP is the channel (or server ID) the gateway listens on.
IBM437	IBM437 is the codepage in use at the gateway node.
1	1 is the priority of the route entry.

- Create an entry for the route to the mainframe server in the route table on the gateway machine. Use the Management Console to Edit the Route Table. (Refer to [Editing the Route Table](#).)
RDB_READ - - - - MVS_1 lu62 READLU62 IBM037 1
The fields have the following meanings:

Route table fields

Field	Meaning
RDB_READ	RDB_READ is the name of the service.
MVS_1	MVS_1 is the ID of the server machine as known to the protocol for the route: for LU6.2, the client-side Partner LU alias.
lu62	lu62 is the network protocol for the route.
READLU62	READLU62 is the channel (or server ID) the server listens on: for LU6.2, the CICS transaction ID.
IBM037	IBM037 is the codepage in use at the server node.
1	1 is the priority of the route entry.

When you run the application, the gateway reroutes client requests to the mainframe server.

Forwarding Servers

A forwarding server has the characteristics of both an execution sever and a gateway. It services client requests if the service DLL is stored locally; otherwise, it forwards the request to a different server.

In some situations, you can use a forwarding server as an alternative to priority-based routing. Suppose you wanted to test a subset of an application's services on server 1, while continuing to maintain the entire set of services on server 2. The forwarding server acts as a gateway when the DLLs for the larger set are not found. As long as you specify the route to the services in the route table on server 1, the system will look for the services on server 2.

In addition, you can also configure a "reverse forwarding" server. A reverse forwarding server looks for the service at the remote location before it looks for it locally.

Configuring Routing

You can choose to configure the client's routing for either a single server or multiple servers. For complete information, select from:

- [Configuring for a Single Server](#)
- [Configuring for Multiple Servers](#)

Configuring for a Single Server

If you want the client to route service requests to a single server, set the configuration for "Single Server". By default, an AppBuilder client is configured to route service requests to the execution server running on the machine's RPC parent. In single-server routing, the client obtains routing information from the communications configuration file, dna.ini. The host name (or parent name) can be changed. If you want the client to route service requests to a different host, clear the current host name and enter a new one. The form of the name depends on the protocol as defined in the following table:

Host name by protocol

Protocol	Host name
TCP/IP	Host name of the machine in the hosts file
Named Pipes	Named Pipes Machine ID or, if a requested service resides on the same machine as the client, LOCAL

To configure this value, refer to [Configuring a Windows Client for Remote Communications](#) for an explanation of how to get to the Routing Tab in the Client Properties.

Configuring for Multiple Servers

If you want the client to route service requests to multiple servers, select Multiple Servers. In multiple-service routing, the client obtains routing information from a route table. Type the full path name, including drive letter, of the route table in the Route Table field.

Testing Configurations

Magic Software Enterprises provides an application to test and validate common configurations with AppBuilder. The IVP application is provided with the installation of AppBuilder. Directions for importing the application and the procedure for testing many configurations are in the [IVP User Guide](#).

Configuring Communications Using MQSeries

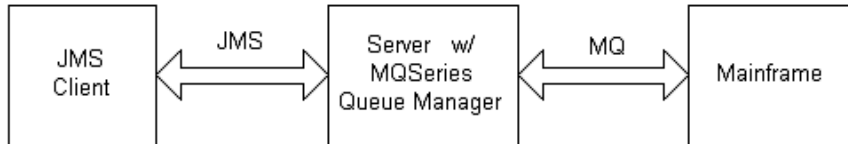
One of the ways that AppBuilder provides extended connectivity for applications that run on multiple architectures in an enterprise is with MQSeries RPC support. This enables AppBuilder-generated Java programs to make remote rule calls to a mainframe running MQSeries using JMS (Java Messaging Service). This topic includes:

- [Understanding MQSeries RPC Operation](#)
- [Meeting Prerequisites for MQSeries RPC](#)
- [Configuring MQSeries RPC](#)
- [Handling Errors in MQSeries](#)

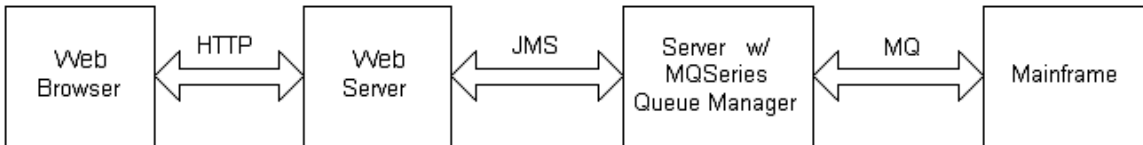
The typical configurations of MQSeries RPC are shown in the following figure.

MQSeries RPC configurations

Java Client



Browser Client

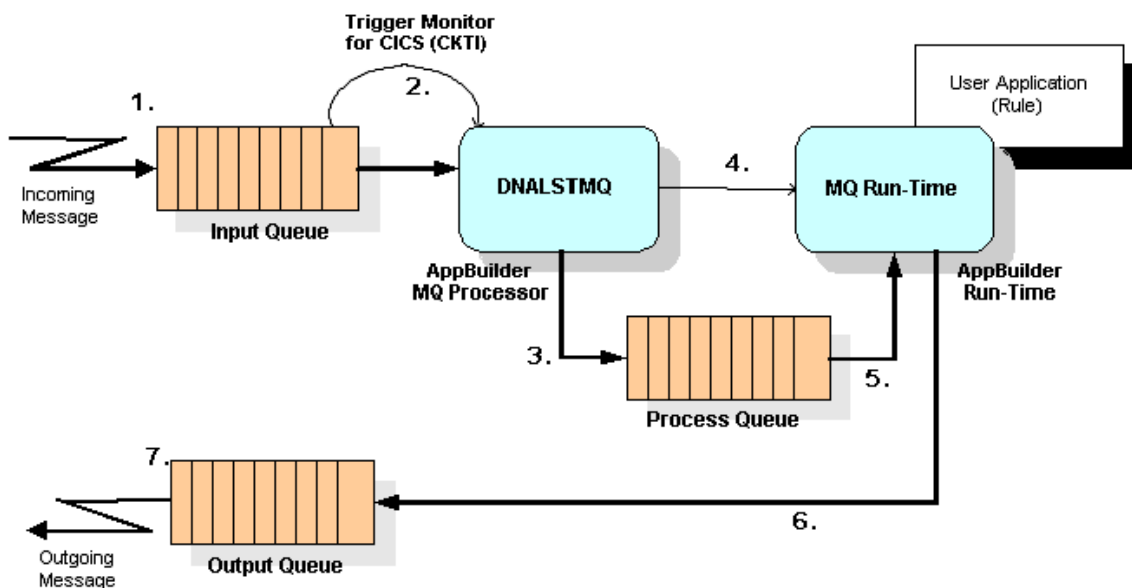


Understanding MQSeries RPC Operation

The information from the Java rule (views and fields) becomes a message that JMS puts in a queue. This message is then transmitted to the mainframe AppBuilder input queue managed by the queue manager connected to CICS. The incoming message triggers the AppBuilder MQ Processor that moves the message from the AppBuilder input queue to the AppBuilder process queue. The appropriate transaction (rule) is then executed.

The rule reads the data from the AppBuilder process queue and writes data to an output queue. The name of the output queue and queue manager are provided in the incoming message in the ReplyTo Queue and ReplyToQueueManager fields as part of the header.

Queue management process



The following list describes the numbered steps shown in [Queue management process](#):

The JMS client writes a message to a local or remote queue. The message is transmitted to the input queue managed by the queue manager connected to CICS.

1. The incoming message causes the IBM-supplied Trigger Monitor for CICS (Transaction ID CKTI) to invoke the AppBuilder MQ Processor (DNALSTMQ). In an effort to increase performance, this should be defined as TRIGGER=FIRST, as the AppBuilder MQ Processor waits for a configurable time-out for the next message.
2. The AppBuilder MQ Processor moves the message to a process queue within a Logical Unit-of-Work (LUW) to ensure it is copied and deleted. The message is browsed, and the relevant user information (user ID, password, transaction-ID, etc.) is extracted.
3. Based on the extracted information, the AppBuilder MQ Processor starts the user application (the rule), passing it the necessary information so it knows which message (messageID) from which queue (process queue) to process.
4. When requested, the AppBuilder Runtime (MQ Runtime) reads the data in the process queue.
5. The output of the application is written to the output queue. The details (queue manager name, queue name, etc.) are provided in the incoming message.
6. The output queue may be a local or remote queue. The message is transmitted to the response queue requested by the client program. Typically this is the same queue manager it is connected to, from which the request originated.



Remember

- Move all messages that the AppBuilder MQ Processor cannot process to a (configurable) dead-letter queue.
- The moving of the messages from the input queue to the separate process queue ensures that each message gets processed once and only once.
- Messages remaining in the process queue for too long probably indicates a problem.
- The level of security, whether a verification based on user ID and password sent by the client is done, can be set using the setting LISTEN_SECURITY in the configuration file dna.ini. (Refer to [Customizing the Mainframe INI File](#).)

Meeting Prerequisites for MQSeries RPC

Before you can use the MQSeries RPC, you must have the following software on these machines:

On Mainframe (CICS)

In the mainframe's CICS region, you must have the IBM MQSeries CICS Adapter with an initiation queue and CKTI running. For more information about the IBM MQSeries CICS adapter, refer to *Part 3. MQSeries and CICS* in the [MQSeries for OS/390 System Management Guide](#) by IBM. That guide contains details for setting up, customizing, and operating the CICS Adapter. It is available in electronic form on the IBM Web site.

On MQ Server

On the MQ Server, you must have IBM MQSeries V5R2 with MQSeries Support Pac MA88 (which includes MQSeries classes for Java that add JMS support to MQSeries). For more information on this software, refer to [MQSeries classes for Java and MQSeries classes for Java Message Service](#). This is available in electronic form on the IBM Web site.

On Java Workstation

The following JAR files, which are part of the MA88 Support Pac installed on the server, must be in the classpath environment variable:

- com.ibm.mqjms.jar
- com.ibm.mq.jar
- jms.jar

Configuring MQSeries RPC

The configuration tasks includes:

- [Configuring the AppBuilder Server](#)
- [Configuring the MQ Server](#)
- [Configuring the Mainframe MQ Series](#)
- [Customizing the Mainframe INI File](#)

Configuring the AppBuilder Server

The method for invoking remote rules is controlled in the appbuildercom.ini file. For MQSeries RPCs, this file can be configured to contain the information needed by JMS to make the RPC call using MQ or to point to a JNDI that then contains this information. The values in appbuildercom.ini that can be configured are summarized in the following table. Examples are given in the appbuildercom.ini example below.

AppBuilder server settings

Value	Description
MQ_QUEUE_MANAGER	Name of server MQSeries queue manager
MQ_CHANNEL	Channel defined on the server used by JMS
MQ_SEND_QUEUE	Queue on server used to send messages to the mainframe
MQ_RECV_QUEUE	Queue on server where messages from the mainframe arrive
MQ_HOST_NAME	MQSeries server name
MQ_PORT	Port that the MQSeries server is listening on. Default MQ Series value is 1414.
MQ_CODEPAGE	Code page on the mainframe
MQ_SERVERID	Unique identifier for this server
MQ_PROTOCOL	JMS or PMJMS. PMJMS is the performance marshalling feature that marshals the view data on the client
MQ_TIMEOUT	Time in seconds that a rule waits for a response from the mainframe

Example appbuildercom.ini Settings

The following example shows a portion of an appbuildercom.ini without using JNDI.

```
[SERVER.JMS]
TYPE=JMS
MQ_QUEUE_MANAGER=QM1      (I)
MQ_CHANNEL=JMS.CHANNEL  (O)
MQ_SEND_QUEUE=AB.SENDQ   (K)
MQ_RECV_QUEUE=AB.RECVQ   (M)
MQ_HOST_NAME=<hostname>  (H)
MQ_PORT=1414
MQ_CODEPAGE=IBM037
MQ_SERVERID=JMSRPC
MQ_PROTOCOL=JMS
MQ_TIMEOUT=0
```

The following example shows a portion of an appbuildercom.ini that uses JNDI to store the information needed by JMS.

```
[SERVER.JMS]
TYPE=JMS
JMS_INITIAL_CONTEXT_FACTORY=com.sun.jndi.fscontext.ReffSContextFactory
JMS_PROVIDER_URL=[file:/C:/JNDI-Directory]
MQ_CODEPAGE=IBM037
MQ_SERVERID=JMSRPC
MQ_PROTOCOL=JMS
MQ_TIMEOUT=0
```

If JNDI is being used, then the MQ JMS Administration tool must be used to define a MQQueueConnectionFactory with the name 'AppbQCF' and two MQQueues named 'AppbSendQ' and 'AppbRecvQ'. The MQ JMS Administration tool is supplied with MQSeries and is documented in the [MQSeries Using Java manual \(SC34-5456-02\)](#) available at IBM's Web site. A sample of the definitions required are:

```
def qcf(AppbQCF) qmanager(QM1) transport(client) hostname(hostname) channel(JMS.CHANNEL) port(1414)
def q(AppbSendQ) qmgr(QM1) queue(AB.SENDQ)
```

The routing section can be updated to use the new server.

```
[ROUTES]
; $ANY specifies the default server entry for all remote rules
$ANY=JMS
```

Configuring the MQ Server

To configure the MQ Server, you must set several definitions. The following definitions can be used; however, customize the values for your environment. The letters in parentheses correspond to the values you should have obtained when you initially collected setup information.

```
DEFINE QLOCAL('AB.SENDQ.XMIT') + (J)
  DESCR('AppBuilder transmit queue for MQSeries RPCs') +
  USAGE(XMITQ) +
  REPLACE

DEFINE QREMOTE('AB.SENDQ') + (K)
  DESCR('Remote AppBuilder queue for MQSeries RPCs') +
  RNAME('CICS02.ABMQ.INPUTQ') + (D)
  RQMNAME(CSQ) + (B)
  XMITQ('AB.SENDQ.XMIT') + (K)
  REPLACE

DEFINE CHL('QM1.CSQ.TCP') + (L)
  CHLTYPE(SDR) +
  TRPTYPE(TCP) +
  CONNAME(TREX) + (A)
  XMITQ(AB.SENDQ.XMIT) + (J)
  REPLACE

DEFINE QLOCAL(AB.RECVQ) + (M)
  DESCR('AppBuilder queue to receive from Mainframe') +
  REPLACE

DEFINE CHL(CSQ.QM1.TCP) + (N)
  CHLTYPE(rcvr) +
  TRPTYPE(TCP) +
  REPLACE

DEFINE CHL('JMS.CHANNEL') + (O)
  CHLTYPE(SVRCONN) +
  TRPTYPE(TCP) +
  MCAUSER(' ') +
  DESCR('Channel for AppBuilder JMS Messages')
  REPLACE
```

Configuring the Mainframe MQ Series

To configure the mainframe MQ Series, you need to set several definitions. The following definitions can be used; customize the values for your environment. The letters in parentheses correspond to the values you should have obtained when you initially collected setup information.

```

DEFINE QLOCAL(CICS02.ABMQ.DEADLETTERQ) +          (G)
DESCR('AppBuilder Dead-Letter Queue') +
REPLACE

DEFINE QLOCAL(CICS02.ABMQ.REPLYQ) +              (F)
DESCR('AppBuilder Reply Queue - Local') +
REPLACE

DEFINE QLOCAL(CICS02.ABMQ.PROCESSQ) +            (E)
DESCR('AppBuilder Process Queue') +
REPLACE

DEFINE QLOCAL(CICS02.ABMQ.INPUTQ) +              (D)
DESCR('AppBuilder Input Queue') +
PROCESS(DNALSTMQ) +
INITQ(CICS02.INITQ) +                            (C)
TRIGGER +
    TRIGTYPE(FIRST) +
    TRIGDPH(1) +
    TRIGMPRI(0) +
REPLACE

DEFINE CHL(QM1.CSQ.TCP) +                        (L)
CHLTYPE(rcvr) +
TRPTYPE(TCP) +
REPLACE

DEFINE QLOCAL(QM1) +                             (I)
DESCR('Xmit Queue to QM1') +
USAGE(XMITQ) +
REPLACE

DEFINE CHL(CSQ.QM1.TCP) +                        (N)
CHLTYPE(SDR) +
TRPTYPE(TCP) +
CONNNAME(HOSTNAME) +                            (H)
XMITQ(QM1) +                                     (I)
REPLACE

DEFINE PROCESS(DNALSTMQ) +
DESCR('AppBuilder MQ Processor') +
APPLTYPE(CICS) +
APPLICID(NETQ) +
REPLACE

```

With regard to the definition for the AppBuilder MQ Processor, consider the following:

- The process name 'DNALSTMQ' must match the process name specified in the definition for the AppBuilder Input Queue (CICS02.ABMQ.INPUTQ).
- The APLICID, in our case 'NETQ', is the actual CICS transaction ID that should point to the supplied AppBuilder program 'DNALSTMQ'.
- Both the program ('DNALSTMQ') and transaction ID ('NETQ') must be defined to CICS. See the following sample definitions.

Here are some sample definitions for CICS.

```

DEFINE PROGRAM(DNALSTMQ) GROUP(gggggggg)
    DESCRIPTION('AppBuilder MQ Processor')
    LANGUAGE(C)
    RELOAD(No)
    RESIDENT(Yes)
    USAGE(Normal)
    DATALOCATION(Any)
    EXECKEY(User)

DEFINE TRANSACTION(NETQ) GROUP(gggggggg)
    DESCRIPTION('AppBuilder MQ Processor')
    PROGRAM(DNALSTMQ)
    TWASIZE(00000)
    PROFILE(DFHICICST)
    STATUS(Enabled)
    TASKDATALOC(Any)
    TASKDATAKEY(User)

```

Customizing the Mainframe INI File

You can customize AppBuilder MQ for performance and configuration considerations. You may use the standard AppBuilder mechanisms and tools to maintain the DNA INI settings. The AppBuilder MQ settings are in the MQMSG section of the mainframe DNA INI. The following entries must be added to the mainframe DNA INI file using the NEAD transaction:

```

MQMSG  DEADLETTER_QUEUENAME    (G)
MQMSG  DEBUGLVL                0
MQMSG  DLQ_FORWARD            DEFAULT
MQMSG  LISTEN_SECURITY        YES
MQMSG  LISTEN_TIMEOUT         30
MQMSG  MAX_MESSAGE_LENGTH     80000
MQMSG  PROCESS_QUEUE_EXPIRY
MQMSG  PROCESS_QUEUEENAME     (E)
MQMSG  REPLY_QUEUENAME        (M)

```

The following provides a detailed description of the possible settings for AppBuilder MQ:

- [DEADLETTER_QUEUENAME](#)
- [DLQ_FORWARD](#)
- [LISTEN_SECURITY](#)
- [LISTEN_TIMEOUT](#)
- [MAX_MESSAGE_LENGTH](#)
- [PROCESS_QUEUE_EXPIRY](#)
- [PROCESS_QUEUEENAME](#)
- [REPLY_QUEUENAME](#)
- [DEBUGLVL](#)

DEADLETTER_QUEUENAME

Possible Values	Valid (existing) queue name
Default Value	ABMQ.DEADLETTERQ

This specifies the name of the queue to which unprocessable messages are forwarded. Messages that cannot be processed can be forwarded to this queue for investigation. By default, if a reply message was successfully generated to notify the client, messages are not forwarded. (They are deleted.) You can force or suppress the forwarding of messages to this dead-letter queue. See [DLQ_FORWARD](#) for details.

DLQ_FORWARD

Possible Values	YES, NO, or blank
Default Value	blank



This means delete if the reply was successfully generated: otherwise, forward to the dead-letter queue.

This either forces or suppresses forwarding unprocessable messages to a dead-letter queue. This forwarding occurs when sending the error reply message to the client is unsuccessful. If successful, the unprocessable message is deleted. This default behavior can be customized through this parameter. If an error message cannot be returned to the client, the message is forwarded to the dead-letter queue unless it is set to NO. Specify NO to suppress and YES to force the message to be forwarded to the dead-letter queue, as specified in the [DEADLETTER_QUEUEENAME](#) parameter.

LISTEN_SECURITY

Possible Values	YES, NO, or PART
Default Value	YES

Indicates the required level of user verification. YES implies that the incoming user ID and password are verified. By specifying PART (partial), no verification is done, but the transaction is started with the incoming user ID. NO causes the user identifier to be ignored and not used.

LISTEN_TIMEOUT

Possible Values	Positive integer (seconds)
Default Value	30

Specifies timeout for the AppBuilder MQ Listener to wait for a new message to arrive on the input queue. To avoid the overhead involved with starting the AppBuilder MQ Listener for every message, use this setting so that the Listener waits the specified number of seconds for a new message. If no new messages arrive, the Listener terminates.

MAX_MESSAGE_LENGTH

Possible Values	Positive integer (bytes)
Default Value	Queue definition/100,000

Specifies the size of buffers to allocate for messages. MQSeries messages can be up to 4MB. However, most application's message size is much lower. To reduce unnecessary storage, use this parameter to specify the size of buffer to allocate. If not specified, the maximum-message-size value specified for the (input) queue is used. If inquiring of the queue's attributes are unsuccessful, a default of 100KB is used.

PROCESS_QUEUE_EXPIRY

Possible Values	Integer (seconds) or blank
Default Value	blank (delete messages)

The AppBuilder MQ Listener forwards messages from the input queue to the process queue for processing by the rule's transaction. These operations are committed, causing the message to be deleted from the input queue. If errors occur starting the transaction, these messages potentially remain in the process queue forever. By default the AppBuilder MQ Listener deletes the message from the process queue if a reply error message is successfully generated to notify the client. Specifying a negative number or zero implies unlimited expiration (that is, it does not expire); whereas, any positive number is used as an expiration period-in seconds-for messages in the process queue. An empty or blank value results in the default behavior, and the message is deleted.

PROCESS_QUEUEENAME

Possible Values	Valid (existing) queue name
Default Value	ABMQ.PROCESSQ

The queue used for forwarding messages to be processed by the rule's transaction.

REPLY_QUEUEENAME

Possible Values	Valid (existing) queue name
------------------------	-----------------------------

Default Value	ABMQ.REPLYQ
----------------------	-------------

Generally, the incoming message (in the header) contains a value in the ReplyToQueue and ReplyToQueueManager fields, specifying the appropriate values to be used for the reply messages. If the ReplyToQueue was not specified in the incoming message, the value specified in this parameter is used.

DEBUGLVL

Possible Values	ERRORS or 0, ERRORS_AND_LENGTHS or 1, ERRORS_AND_DATA or 2, ERRORS_AND_TRACES or 3
Default Value	ERRORS or 0

This sets the debug and trace level to assist troubleshooting AppBuilder MQ Listener related problems. This is used in conjunction with the global setting in the TRACING section. The values specified are compared and, if the AppBuilder MQ Listener's level is not lower, debug trace messages are output.

The following table summarizes when the trace messages are output based on the two debug level settings: this one in the MQMSG section and one in the TRACING section. This allows the AppBuilder MQ Listener trace to be forced or suppressed separately from (though configured in conjunction with) the global setting.

Trace messages output cases

DNA INI Section	[TRACING] DEBUGLVL Values			
[MQMSG] DEBUGLVL Values	0	1	2	3
0	No	No	No	Yes
1	No	No	No	Yes
2	No	No	No	Yes
3	Yes	Yes	Yes	Yes ^a

a. This is the only case that produces the "AppBuilder MQ Started" trace message.

Handling Errors in MQSeries

When certain errors are detected while processing the incoming message, the AppBuilder MQ Listener attempts to rollback (syncpoint rollback) all operations, including MQSeries. After this, the Listener terminates.

The most common errors to cause a rollback are those detected when forwarding a message to the dead-letter queue. If this is the only message in the input (trigger) queue, such rollbacks cause MQSeries to consider this as a new message. As such, the AppBuilder MQ Listener is restarted. It encounters the same error, rollback and terminate, and is restarted. This happens continuously until another message is put on the queue or the Listener is stopped. One way to minimize this is to ensure that the dead-letter queue name and option are correctly set.

There are DNA INI settings that affect the AppBuilder MQ Listener error handling and dead-letter queue configuration. Refer to [Customizing the Mainframe INI File](#).

Configuring Communications in UNIX

The AppBuilder installation creates default servers; however, you are not required to use these servers. You can delete the default servers by deleting their directories: nete, netg, and ne20. Then create and name new servers by running the mkabserv script located in the -bps/appbuilder/config directory.



After running these scripts, run .profile again.

Server Utilities

The utilities described in Server Utilities apply to all of the AppBuilder servers. You can run other utilities for specific servers described in the following sections:

- [Departmental Server](#)
- [Preparation and Test Server](#)
- [Gateway Server](#)

startserv

This script resides in an individual server's directory, and is used to start the server. If the server starts successfully, the script reports the start with the following message:

startserv script

```
appbuilder_servers/ne20$ startserv
Sending nohup output to nohup.out.
Server started seccessfully
appbuilder_servers/ne20$
```

To run OpenCOBOL rules against an Oracle database, edit the startserv script for that server and uncomment the following section:

```
# if [[ "$DBMS" = "ORACLE" ]]
# then
# `which procob32` >/dev/null 2>/dev/null
# if [[ $? = 0 ]]
# then
#   export LD_PRELOAD=$ORACLE_HOME/lib32/libcIntsh.sl
# else
#   export LD_PRELOAD=$ORACLE_HOME/lib/libcIntsh.sl
# fi
# fi
```



Any server using this section cannot be used to prepare rules.

stopserv

This script, used to stop a server, resides in an individual server's directory. If the server is stopped successfully, the script reports the stop with the following message:

stopserv script

```
appbuilder_servers/ne20$ stopserv
Shutdown requested
Shutdown seccessful
appbuilder_servers/ne20$
```

mkabserv

This script takes four parameters and prompts you if you do not specify them.

For example, when you create a Departmental Server, the program prompts you to enter the name of the database the server will use. When creating a departmental server, the script displays as follows:


```
mkabserv R <server_name> <service_name> <database_name>
```

The four parameters are explained below:

- Server type - P (Preparation and Test Server) or R (Departmental Server). Use an uppercase P or R.
- Server name - The directory that is created.
- Service name or port - The name of the server; matches the directory created for the server's files. Either a numerical port or an entry in `/etc/services`.
- Database name - Because this is an Oracle instance, use N/A.

mkabuser

This script registers you as an AppBuilder user. To run this script, you must be a member of the bps group. The script creates an `appbuilder_apps` directory under your home directory.

Departmental Server

During a Departmental Server installation, the default nete server is configured to use a service named "nete." If nete is not in your `/etc/services`, you can either add it (the standard port is 3090) or change the `dna.ini` file to use either a numeric port or a different service name.

Preparation and Test Server

Two servers are created for Preparation and Testing:

- ne20 – Preparation Server
- nete – Test Server.

Nete is configured exactly as it would be for a departmental server. During a Preparation and Test Server installation, the server is configured to use a service named "ne20." If ne20 does not exist in your `/etc/services`, you can either add it (the standard port is 3088) or change the `dna.ini` file to use either a numeric port or a different service name. If you change the port/service, you must match this change to your workbench settings.

Gateway Server

During a Gateway Server installation, the default netg server is configured to use a service named "netg". If netg does not exist in your `/etc/services`, you can either add it (the standard port is 3087) or change the `dna.ini` file to use either a numeric port or a different service name.

Servers created by other users (not bps) are in the `appbuilder_servers` directory of their home directories.

mkabgate

Use this script to create a new gateway in its own directory under your `appbuilder_servers` directory. You must be a member of the bps group to create an AppBuilder gateway.

This script takes four parameters on the command line. If you do not provide their values, the script prompts you for them. These parameters are

- Gateway name - The name of the gateway; matches the directory that is created for its files
- Service name or port - The numeric port or entry in `/etc/services` through which the gateway receives incoming requests
- Target machine - The gateway's destination host name
- Target port - The target port/service on the destination workstation

Starting a Server

As a bps user, you can run an entire AppBuilder system. You can configure additional users to run AppBuilder servers or to serve as the target for AppBuilder preparation steps.

Each server is located in the `appbuilder_servers` directory. The Preparation and Test Server comes with the following default servers:

- nete – Departmental Server
- ne20 – Preparation and Test Server

To start a server, run the `startserv` script that is located in the server's directory (See [startserv](#)). By default, this script is configured to start a forking server in the background. (For gateways, the default configuration is a gateway.) You do not need to run this script with the "&" character. When you use the "nohup" command to run the server in the background, any error messages on startup will reside in the `nohup.out` file.

Each server resides in its own directory, which contains files unique to the server. Only one copy of a server can run at a time. The server outputs its PID to a file in the server directory (`server.pid`). The `startserv` script uses this file to determine if the server is already running. When starting a server, the `startserv` script waits for 2 seconds after the command and then checks to see if the PID is active. If so, it issues a message confirming that the server started successfully. If not, a message prompts you to check the `nohup.out` file for errors.

The following line in the startserv script shows the actual conn_serv parameters. You can modify the line to change the server type:

- Forking (default)

```
cmd="$ABHOME/bin/conn_serv -f -ptcp"
```

- Banking

```
cmd="$ABHOME/bin/conn_serv -b -ptcp"
```

- Gateway (default for gateways created using mkabgate)

```
cmd="$ABHOME/bin/conn_serv -f -g -ptcp"
```

Stopping a Server

You can stop a server by running the stopserv script located in the server's directory (See [stopserv](#)). This script issues a *kill* command against the PID located in the server.pid file. The stopserv script checks to see if the PID is still in the server.pid file. If not, it displays a message confirming that the shutdown was successful. If the script finds the PID, it delays 2 seconds and repeats the test and delay up to 5 times. If the PID is still active, the script displays a warning suggesting that you check the status manually.

Preparing Remotely to HP-UX

Once the Preparation and Test Server is installed and configured, you can use remote preparation to build the application on an HP-UX workstation remotely from a Windows workstation running the Construction Workbench. To begin remote preparation, follow these steps:



Departmental servers must be shut down on the HP-UX workstation before remote preparation begins. The Departmental server caches the AppBuilder rules. If an AppBuilder rule is cached, it will not prepare.

1. Start the Preparation and Test Server on the HP-UX side.

The script for starting the server is in `~bps/appbuilder_servers/ne20` (by default). If you do not start the correct script, the remote preparation does not work properly.

The compiled code for the rules is stored in the specified user's directory. For database table creation, the DBMS on the HP-UX workstation must have been configured correctly.

By default, when you start the Construction Workbench, the communication agent is started. If it is stopped, you can start it with the Management Console.

2. Set the remote preparation options in the Construction Workbench.

On the Windows workstation, in the Construction Workbench, select **Tools > Workbench Options**. On the **Remote Preparation** tab, add the HP-UX workstation and set the appropriate options. Refer to the *Development Tools Reference Guide* for more information about setting the Workbench Options.

3. Configure the application for remote preparation.

On the **Configuration** tab of the Hierarchy diagram in the Construction Workbench window, make sure the application has a configuration that includes the partition and workstation for HP-UX. For more information about specifying partitions and workstations, refer to the *Deploying Applications Guide*.

If you cannot see the **Configuration** tab, Select **File > New Project**. Creating a new project sets up a **Configuration** tabbed page in the display.

Configuring Communications on the Mainframe

This section describes how to use the AppBuilder communications administration utility (admin utility) for mainframes running mainframe CICS. The admin utility provides an easy way to view and edit AppBuilder runtime files, control the MQSeries queue manager, and perform

miscellaneous maintenance operations.

This section also describes how to set up AppBuilder communications clients for LU2 and LU6.2 connections to mainframe servers and how to configure mainframe servers for use in messaging and eventing applications.

In addition, this topic describes the following tasks:

- [Accessing Mainframe Configuration Files](#)
- [Performing Maintenance Operations](#)
- [Configuring LU2 Clients](#)
- [Configuring LU6.2 Clients](#)
- [Configuring LU2 Terminal Emulator](#)
- [Configuring CICS for TCP/IP Listener](#)
- [Configuring CICS HTTP Support](#)
- [Configuring AppBuilder INI Settings for HTTP Support](#)
- [Configuring C HTTP Client](#)
- [Usage Notes](#)
- [DNAINI Settings](#)
- [Understanding Performance Marshalling](#)

Accessing Mainframe Configuration Files

The mainframe admin utility is a 3270 application for mainframe CICS. The following figure shows the main panel displayed when you run the NEAD transaction.

There is now a new ABIN transaction that provides only INI maintenance options. This means basically that the only displayed screen is the INI panel, with all features available.

Regardless whether using NEAD or ABIN, from the INI panel the user can press PF10 to recache the INI settings. This replaces the same feature available via the 1.6 option from the main NEAD panel. The following sections describe the associated menu options.

Admin utility main menu

```
Blue Phoenix Solutions, Inc.
AppBuilder Administration
** Main Menu **

1. Operations
2. Initialization File
3. Route Table
4. Subcell Table
5. Trigger Table
6. Event Table
7. Reserved
8. Reserved
9. Reserved

Selection:

PF3
EXIT

Enter Valid Selection Between 1 - 6. Press Enter.
```

Options 2 through 6 on the main menu invoke a scrollable display of the data in each runtime file. You can view and edit DNA.INI (the initialization file), the route table, the subcell table, and the trigger and event tables.

For each file, a record is limited to a single line on the display. Due to space limitations on the display screen, only selected fields are displayed, and some fields might be truncated. You can select any record to view in full, modify, or delete. You can also add new records to the file.

For example, the following figure shows a sample display of the Initialization File screen:

Initialization file screen

```

Blue Phoenix Solutions, Inc.
AppBuilder Administration
** Initialization File Menu **      Page : 1

Section          Variable          Value
CWS_SERVER      CWS_VIEW_IN_MEMORY    NO
CWS_SERVER      TSQ_MAX_ITEMLEN       8000
CWS_SERVER      TSQ_PREFIX            ABCWSTSQ
DNA             CLIENT_DNARPC_TIMEOUT 0
DNA             CLIENT_POLLING_INTER   250
DNA             CLT_MAX_SERVERS       10
DNA             DNA_CLOSE_SEMANTICS   COMMIT
DNA             DYNAMIC_DATA_OPTIMIZ  N
DNA             FILE_XFER_LOCATION    DNAFXFER
DNA             LOG_UNIT_OF_WORK      52LOCAL
DNA             STATIC_DATA_OPTIMIZE  Y
DNA             VERSION               3.0
DNA             WORKSTATION_ID
DNA_EXITS      DATABASE_EXIT

PF2  PF3  PF4  PF5  PF6  PF7  PF8  PF10
VIEW END  ADD  DELETE  MODIFY  PREV  NEXT  CACHE-INI

Initialization File Cache Successful.

```

To VIEW, DELETE, or MODIFY a record, first select it by placing an uppercase or lowercase "s" (S or s) in the area to the record's left. The selected record is then displayed discretely on a panel, which shows all fields in their entirety.

Select the MODIFY option to change the data as needed. Before any data is actually modified or deleted, however, you must confirm the request by pressing the appropriate PF key.

For example, the following figure shows the panel that would be displayed if you selected the fourth record on the panel shown in [Initialization file screen](#) and pressed the PF6 key to MODIFY:

Modifying an initialization value

```

Blue Phoenix Solutions, Inc.
AppBuilder Administration
** Initialization File - MODIFY **

Section :  NLS
Variable :  LOCAL_CODEPAGE
Value   :  IBM037

PF6  PF3
MODIFY  EXIT

Enter New Value.

```

To use the ADD option, it is not necessary to select a record first. If you do not select a record, a blank panel displays enabling you to fill in the required data. If you select a record first, the panel's fields are initialized with the data from the selected record.

For all files except the initialization file, new records are added immediately after the selected record or at the end of the file if no record was selected. For the initialization file, records are stored alphabetically based on the section and variable name.

The admin utility makes no attempt to validate data. It is the administrator's responsibility to ensure that the data is correct. This is not unlike other AppBuilder communications platforms where the runtime files are plain text files that the administrator can view and change with any text editor.

Performing Maintenance Operations

Option 1 on the main menu invokes the Operations menu shown in the following figure.

Admin utility operations screen

```
Blue Phoenix Solutions, Inc.
AppBuilder Administration
** Operations Menu **

1. Start Messaging Queue Manager
2. Stop Messaging Queue Manager
3. Dump Messaging Shared Memory
4. Reset Global Eventing Shared Memory
5. Dump Global Eventing Shared Memory
6. Cache Initialization File
7. Reserved
8. Reserved
9. Reserved

Selection:

PF3
EXIT

Enter Selection Between 1 - 6.
```

Caching the Initialization File

Option 6 on the Operations menu caches the initialization file (DNA INI). Pressing PF10 in the INI list display will also cache the INI settings. Changes made to the DNA.INI file do not take affect until they are cached.

Configuring LU2 Clients

If a client machine requests services from the mainframe (CICS or IMS) over LU2, the runtime interpretation of a script file constitutes the connection between the client process and the mainframe. This section describes the delivered logon and logoff script files, which must be customized, and the AppBuilder Communications script language.

Initializing the Scripts

Specify in the [LU2] SCRIPT_DIR variable in DNA.INI on the client machine the path to the directory that contains the LU2 logon and logoff scripts. Use the variables [LU2] LOGON_SCRIPT and [LU2] LOGOFF_SCRIPT in the client DNA.INI to identify the files that contain the scripts. The system supplies default logon and logoff scripts named MFLOGON.SCR and MFLOGOFF.SCR, respectively.

Configuring LU6.2 Clients

If a client requests services from mainframe over LU6.2, specify an LU6.2 Mode name in the [LU6.2] MODE_NAME variable in DNA.INI on the client machine.

If you use Microsoft SNA Server on Windows, you must use the default LU. If you use IBM Communications Server, you must set the APPCLLU variable to the local LU alias in the environment settings on the client machine. The value is case-sensitive.

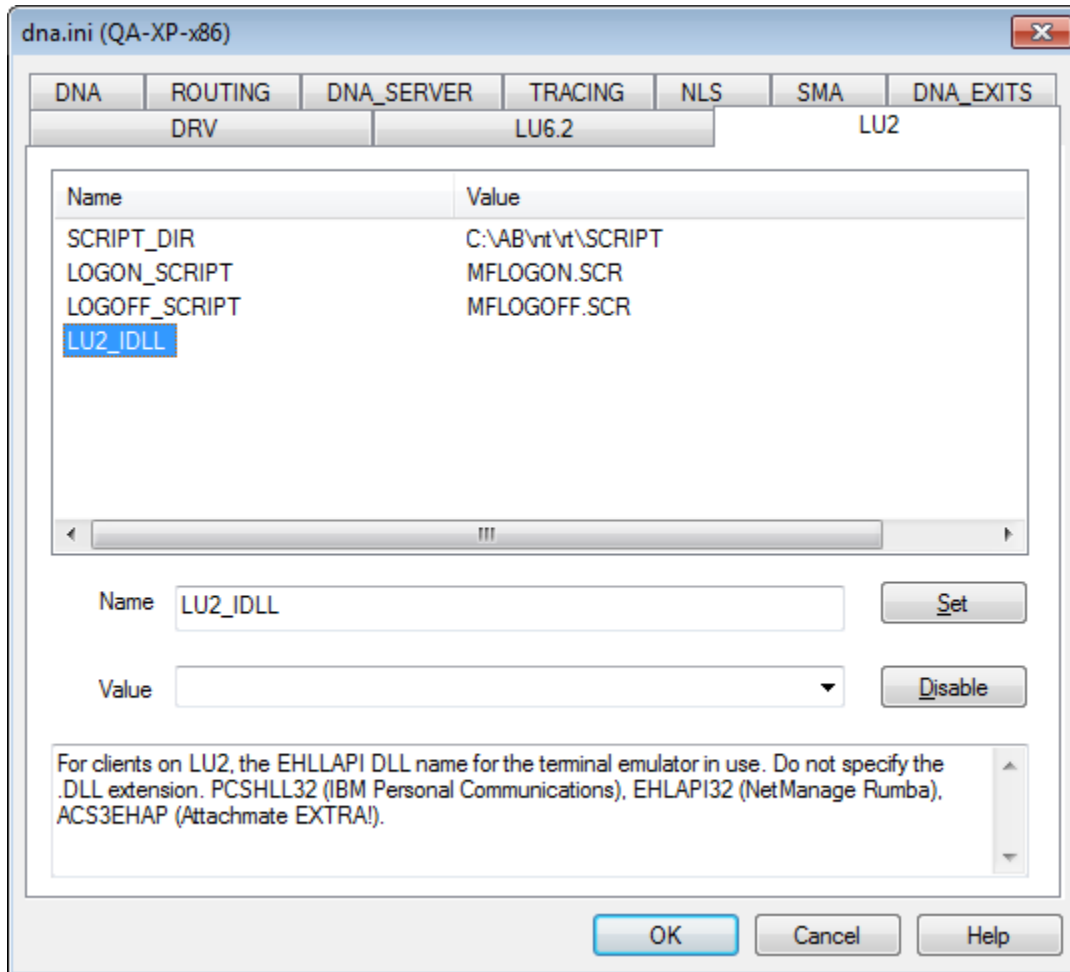
Configuring LU2 Terminal Emulator

To configure the terminal emulator for mainframe communications over LU2, from the Management Console display, click the icon for the

Windows client and select *Edit INI* from the pop-up menu. This opens the dna.ini file. Refer to *INI Settings Reference Guide* for thorough information on the settings in the dna.ini file.

Click the *LU2* tab and select the *LU2_IDLL* value, as shown in the following figure. In the Value field, select the drop-down menu and select the value. *PCSHLL32* is for IBM Personal Communications (PComm). *EHLAPI32* is for NetManage Rumba (Rumba). *ACS3EHAP* is for Attachmate EXTRA! (Extra). Click *Set* to save the setting and then *OK* to close the INI editor when done.

Selecting the terminal emulator for LU2



Configuring CICS for TCP-IP Listener

Perform the following steps after the installation is complete:

STEP 1 - Define the AppBuilder Communications resources to CICS

The samples provided in this section contain the necessary program, transaction, files, and definitions for CICS.

- Example 1: Define the program DNALSTNR to the CICS region:

```
DEFINE PROGRAM(DNALSTNR) GROUP(nnnnnnnn)
  DESCRIPTION(APPBUILDER TCP LISTENER)
  LANGUAGE(C) RELOAD(NO) RESIDENT(NO) USAGE(NORMAL)
  USELPACOPY(NO) STATUS(ENABLED) CEDF(YES)
  DATALOCATION(ANY) EXECKEY(CICS) EXECUTIONSET(FULLAPI)
```

- Example 2: Define a Listener transaction with any valid transaction ID:

```
DEFINE TRANSACTION(tranid) GROUP(nnnnnnnn)
  DESCRIPTION(APPBUILDER TCP/IP LISTENER)
  PROGRAM(DNALSTNR) TWASIZE(0) PROFILE(DFHCICST)
  STATUS(ENABLED) TASKDATALOC(ANY) TASKDATAKEY(USER)
  DYNAMIC(NO) PRIORITY(200) TCLASS(NO) DTIMOUT(NO)
  INDOUBT(BACKOUT) RESTART(NO) SPURGE(NO) TPURGE(NO)
  DUMP(YES) TRACE(YES) RESSEC(NO) CMDSEC(NO)
```

The TCP/IP Listener program and transaction ID must be defined with EXECKEY(CICS). If they are not and storage-protection is enabled, the transaction will fail with an ASRA abend code indicating an incorrect attempt to overwrite the CDSA by EZACIC01. In addition to storage-protection being disabled, if the EZA* resources are defined with EXECKey=USER, DNALSTNR can also fail.

You can rename the TCP/IP Listener transaction to conform to your environment standards. The listener's transaction should, however, specify TASKDATAKEY(USER), as the listener's storage need not be protected. Remember to create a EZAC configuration record for the new transaction ID.



If the last (fourth) character of the Listener's transaction ID is 'L' when the Listener is started with the EZAO transaction, it starts and executes under a separate TCB.

STEP 2 - Define the Listener transactions to the CICS Socket interface

Use the IBM-supplied EZAC transaction to create a configuration record for each Listener transaction. If you use multiple TCP/IP Listeners for IBM CICS applications, assign each a unique transaction ID and port number. Define the transaction IDs to the DNALSTNR program and configure the port numbers for the CICS region. For details about the EZAC transaction, refer to *CICS TCP/IP Socket Interface Guide and Reference*.

NetEssential messages

NetEssential CICS TCP/IP Listener displays a message at startup and termination, providing the requested diagnostic information.

The NetEssential IP Listener displays a message right after it becomes active. When debug trace is enabled, the following messages appear at startup, along with some of the listener's settings:

```
DNA 936 INFO      IP Listener is active on port <port#>, max connections <limit>, backlog <backlog>.
DNA 937 INFO      IP Listener Parameters:
DNA --- DEBUG    ACCEPT timeout.....: <nnn> GIVESOCKET timeout...: <nnn> RECV timeout.....: <nnn>
DNA --- DEBUG    Peek buffer size....: <nnn> CLOSEWAIT.....: <Y|N> Linger time.....: <nnn>
DNA --- DEBUG    Security.....: <secopt>          Enhanced RC: <Y|N> Allow expired rule..
<ruleid>
DNA --- DEBUG    Routing Exit program: <rtxpgm>      Data length: <nnn> Abend Handler.....:
<uahpgm>
```

If debug tracing is not enabled, only the first message is displayed.
The termination message is:

```
DNA 942 INFO      IP Listener on port <port#> is terminating.
```

where:

- *port#* - port number on which the IP listener is waiting for connection requests;
- *<nnn>* - numeric value;
- *<Y/N>* - YES or NO, indicating if the option is enabled/disabled;
- *<secopt>* - the listener security authentication option;
- *<ruleid>* - name of the rule program;
- *<rtxpgm>* - name of the Routing and Transaction switching exit;

- `<uahpgm>` - name of user abend handler program.

After applying (i.e. linking the load module), the NetEssential error message VSAM dataset must be initialized or reinitialized with the new message text, accomplished via REPRO.

Configuration Record Fields

The Listener uses the following EZAC configuration record fields:

APPLID and **TRANID** are used to identify (are the keys to) the Listener's record. In the APPLID field, specify the name of the CICS region. In the TRANID field, specify the Listener's transaction ID.

NUMSOCK is the maximum number of connections used in the INIAPI() call (including those that are pending to be taken and those that are to be read) that the Listener will handle. When reached, the Listener will not process any new connection requests until one of the existing sessions terminates. Messages are displayed when the **MAXSOCK** condition is reached and when MAXSOCK is no longer valid.

The current limitation for a CICS C program, such as the Listener, is 255. It can also be limited by the **SOMAXCONN** value from the TCP/IP configuration file `<profile.tcpip>` for the system. The value used is 255, unless SOMAXCONN is less than this value.



This is not the maximum number of started rule transactions, it is only the maximum number of concurrent sockets the Listener can handle. Once they are taken by the rule transaction and closed by the Listener, they are no longer charged to the Listener.

The number of concurrent CICS transactions using the Sockets' interface is specified in the NTASKS parameter in the EZAC CICS record. NTASKS specifies the number of reusable MVS subtasks to be allocated. This should be the highest number of concurrent CICS transactions that will use Sockets' interface, excluding the Listener(s).

ACCTIME specifies the timeout used in the SELECT() API call when the Listener waits for socket activity. Activity can be either a new connection request, where data is available on an already accepted or started connection, or the rule transaction has taken the socket. During this time, the Listener is in a wait state and cannot check for shutdown requests or process any other time-outs, such as **GIVESOCKET**.

A higher value reduces overhead but may increase the time it takes for the Listener to detect a shutdown request because the call is not interrupted for such requests. The default value is 60 seconds.

GIVTIME specifies the GIVESOCKET timeout. The Listener attempts to pass the socket to the started rule transaction by issuing a GIVESOCKET. If the GIVTIME expires before the rule takes the socket (TAKESOCKET), the Listener issues an error message and closes the socket. Note that this process is not exact because the Listener does not set GIVESOCKET timers, but during each SELECT cycle, as defined in ACCTIME, expired GIVESOCKET time-outs are checked.

WLM groups are used to register the service with Workload Manager (WLM) and are passed as is to EZACIC12 (the WLM De-/Registration module).

Other fields in the EZAC Configuration record are used by the Listener, such as:

PORT specifies the port number the Listener will use.

IMMEDIATE indicates that the Listener is to be automatically started when the CICS Sockets interface is started when YES is specified.

BACKLOG specifies the depth of the backlog queue or number of connection requests used in the LISTEN() call that will queue for the Listener to accept. Specify the value depending on whether it is more desirable to have the client wait for the Listener or for the connection request to fail and retry. It works in conjunction with the NUMSOCK parameter.



After creating new records, the CICS Sockets interface must be restarted to pick up the new definition. Use the EZAO transaction to stop and restart the sockets interface.

STEP 3 - Configure the AppBuilder Communications runtime options

AppBuilder Communications is highly configurable. Some parameters are specified with the CICS Sockets EZAC configuration record, as described in [Configuration Record Fields](#). Most configuration parameters are specified with DNAINI settings. Use the NEAD transaction to update the AppBuilder Communications initialization file (DNAINI). Select option 2 "Initialization File" from the main menu to add or modify the settings.

In most cases, the Listener reads the configuration settings once, at startup. Therefore, the Listener should be restarted whenever there have been modifications to the settings. Restarting the listener is accomplished via the EZAO transaction. To stop the listener, issue 'EZAO STOP,LIST(tran)'. Wait to verify the listener transaction has terminated as it may take up to the interval specified in the EZAC ACCTIME parameter. Restart the listener by issuing 'EZAO START,LIST(tran)'. Use Option 6 (Cache Initialization File) on the Operations menu (option 1

"Operations" from the main menu) or PF10 from the DNA INI Settings menu to activate (re-cache) the changes to the initialization file.

We recommend that you specify keywords and their values even when using default values because the DNAINI settings are cached to improve performance. However, when an entry is not in the cache, AppBuilder Communications issues a physical I/O request in an attempt to read it from the file. For example, every rule transaction must check for an OpenCOBOL environment, so it will always attempt to access the OPEN-COBOL /ENABLED setting. See [DNAINI Settings](#) for a list of the settings.

STEP 4 - Starting the CICS Socket Interface

Use the EZAO transaction to restart the CICS TCP/IP Socket Interface, as described in *TCP/IP V3R2 for MVS: CICS TCP/IP Socket Interface Guide and Reference* published by IBM.

For example, to stop the CICS Sockets interface issue the following command:

```
EZAO, STOP, CICS
```

Use the following command to start the CICS Sockets interface:

```
EZAO, START, CICS
```

STEP 5 - Starting the Listener

All the TCP/IP Listeners configured to start automatically (IMMEDIATE=YES. See [STEP 2 - Define the Listener transactions to the CICS Socket interface](#)) will be started as a result of successfully starting the CICS Sockets interface (EZAO,START,CICS). All other Listeners must be manually started using the EZAO transaction. For example:

```
EZAO, START, LIST(NETL)
```

where NETL is the Listener's transaction ID. Use the TSO command NETSTAT to verify the start of the Listener.

STEP 6 - Routing service requests to the CICS region

Use the Configurator on the client workstation to route service requests to the CICS region over TCP/IP.



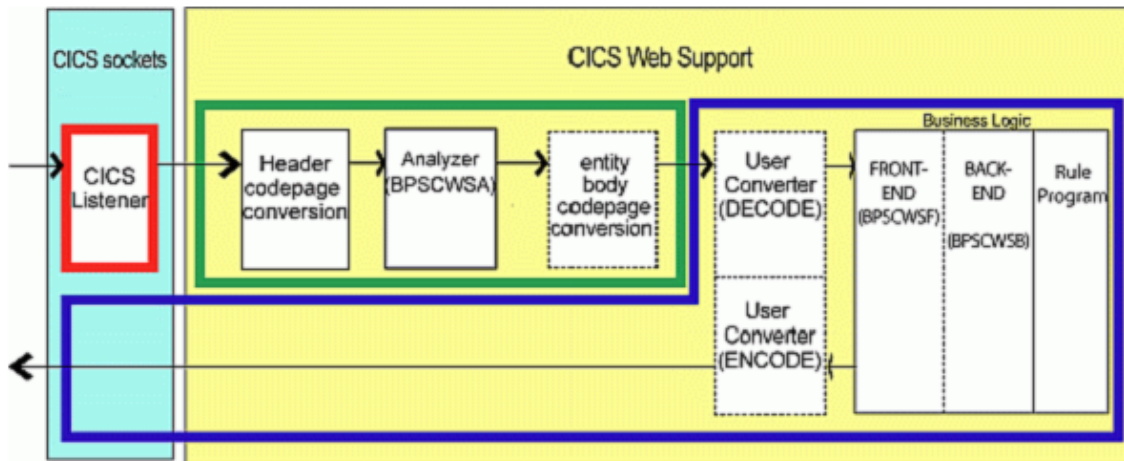
You must specify the CICS transaction ID of the AppBuilder Communications execution server, not the Listener, in the Server ID field. An example of the AppBuilder Communications execution server might be ne20.

Configuring CICS HTTP Support

The AppBuilder HTTP support is implemented in CICS, using the built-in CICS Web Support (CWS). This allows AppBuilder client rule programs to communicate with CICS server rule programs using the HTTP protocol over TCP/IP.

CICS Web Support HTTP Request Processing

CICS Web Support HTTP Request Processing



The three tasks are:

- CICS Socket Listener
- CICS Web Attach task (CWXN)
- Alias transaction (e.g. PCIO)

1. The CICS listener accepts the TCP/IP connection.
2. The HTTP headers are code-page converted.
3. The Analyzer (BPSCWSA) analyzes the inbound request.
4. The entity body is code-page converted (optional).
5. User converter program construct the target (rule) program's input (decode).
6. Business logic (BPSCWSF/BPSCWSB front/back-end) set up to execute the (rule) program.
7. User converter program convert the (rule) program's output for HTTP (encode).
8. The HTTP response is code-page converted and sent.

AppBuilder HTTP support is implemented using CICS Web Support (CWS). CICS drives the process, invoking user-supplied programs at various stages, such as an Analyzer program, a Converter program, and a Business Logic Interface (BLI or BL). The AppBuilder BLI implementation will, for MRO/CICSplex reasons, consist of a front-end Business Logic stub and a back-end Business Logic stub. The front-end Business Logic program will make use of the EXEC CICS WEB API to handle the data communications. It will link to the back-end Business Logic program that interfaces with the (frontier) rule.

The AppBuilder implementation of CICS Web Support comprises of the following components:

1. Analyzer (BPSCWSA) - Validates the inbound data, performs transaction switching and optionally security authentication and sets the converter program name.
2. Business Logic Interface (BPSCWSF/BPSCWSB) - Sets up and calls the frontier rule program, and returns the response to the client. This component is split into two programs, a front- and back-end, to support MRO/CICSplex environments.
3. BLOB Handler (BPSBLOB) - Handles BLOB (get/put) requests.

The Analyzer Program

The analyzer is a user-replaceable program that analyzes each request received from a Web browser. It determines whether the request can be processed by CICS or if it should be rejected. If the analyzer determines that a request is to be processed it also determines which CICS resources are needed to service the request. For example, the analyzer can specify the name of the CICS application program that is to process the request, the transaction ID, the user ID, and conversion-related information, such as the code-page and/or name of the converter program.

The name of an analyzer program is specified in the URM field of each TCPIP SERVICE definition used with CICS Web support. The AppBuilder supplied analyzer is BPSCWSA.

The analyzer runs in the same CICS region as the TCPIP SERVICE which receives the request.

The rule program name is specified via URL parameters. To determine the transaction ID under which the alias (Business Logic) transaction will execute, the Routing and Transaction Switching exit (DNARTX) is called. This is the existing NetEssential exit, using HPSTBLE1/HPSTBLE2 to lookup the rule's transaction ID.

Based on user-configurable options, the analyzer may also perform security verification. This is very similar to the TCP/IP listener implementation; either one issues EXEC CICS VERIFY or invokes the user authentication or authorization exits. This implies that the client will be able to provide the user credentials as part of the standard HTTP Authorization header. The basic form is used, implying that the user-ID and password will be base-64 encoded. However, when security is important, use SSL.

The AppBuilder HTTP client support is responsible for handling code-page conversions. This implies that the data is expected to be performance-marshaled; thus, no additional conversions should be necessary. Any additional and/or special code-page conversion requirements are to be handled in the CWS user-replaceable converter program. The converter name is user-configurable and supplied by the AppBuilder client. (See [URL Format](#) for details.)

If the converter program name is not supplied (or set to default), the analyzer will set it based on the content-type provided in the HTTP header. The analyzer will attempt to locate the appropriate DNAINI setting and if it is specified, will use the provided value. See [Configuring AppBuilder INI Settings for HTTP Support](#) for details.

The Converter Programs

Converter programs are optional; their invocation based on user configuration options. The user is responsible for creating converter programs that follow the CWS guidelines. AppBuilder does not provide a converter program. If encryption/compression is used, the user must implement a converter to decrypt/decompress the input data and encrypt/compress the outbound response.

If used, the user converter program should not change the target application program name. This is the business logic interface and should refer to the AppBuilder-provided program BPSCWSF.

The converter is a user-replaceable program with two functions:

Decode

Using the HTTP request received from the client, and output from the Analyzer program, the decode function can construct the communication area (COMMAREA) passed to the application program. It can also reject the request as well as determining the application program name that is to service the request.

Encode

Using the communication area returned by the application program, the encode function can construct the HTTP response; specify the HTTP status code to be returned as well as selecting the subsequent processing stages.

For a given request, the same converter program is called for both the decode and encode functions.

The converter must run in the same CICS region as the TCPIP SERVICE which receives the request.

The user is responsible for creating converter programs that follow the CWS guidelines. These programs will also perform all encryption and/or compression functionality. As stated, if security is important, it is strongly recommended to use SSL.

It is possible to specify different converter program names for different types of data. This can be set using the DNAINI setting: CWS_CONVERTER <CONVERTER-TYPE>.



When receiving encrypted/compressed data, the Business Logic programs will specify the same value in the response's Content-Type HTTP header. This does not mean that the data is actually encrypted/compressed. Rather, that is the responsibility of the converter program (as part of the encode function) to ensure that the data gets encrypted/compressed and matches the Content-Type header.

The Business Logic Programs

CICS imposes certain limitations which need to be circumvented. We need to be able to pass more than 32K to the rule, as well as executing the rule in a separate CICS region. As is widely known, CICS limits the size of the communication area to 32K. CWS requires the EXEC CICS WEB requests to be issued in the same region as the TCPIP SERVICE which receives the request. In view of these, we will require two stubs to invoke the rule; a front-end and a back-end program.

The front-end stub must run in the same CICS region as the TCPIP SERVICE which receives the request, as it needs to issue EXEC CICS WEB requests to receive or send data when larger than 32K. If the user-data along with the stub's internal header information exceeds 32K, the data will be written to a shared temporary storage queue (TSQ). The TSQ name, as well as an indicator, will be passed to the back-end. The same method will be used to pass back the response data.

The shared TSQ requirement is due to MRO environments where the back end program may execute in a separate region. Single region environments may be configured to allow the front-end to pass large (>32K) view data via addresses in the COMMAREA instead of TSQ.



See the descriptions below and in the *INI Settings Reference Guide* for:
CWS_SERVER TSQ_PREFIX
CWS_SERVER CWS_VIEW_IN_MEMORY

After it retrieves the data from the COMMAREA and if necessary the TSQ, the back-end invokes the rule. This will naturally be different for Classic COBOL and OpenCOBOL rules.

The AppBuilder-supplied Business Logic Interface programs are BPSCWSF (front-end) and BPSCWSB (back-end).

Invoking Classic COBOL Rule Programs

Classic COBOL rules are invoked using the CAPI interface (HPECAP). This implies and imposes that the frontier rule and all called rules and components will execute in the same current CICS region.

Invoking OpenCOBOL Rule Programs

OpenCOBOL rules are invoked via dynamic COBOL calls, in the same manner as they are currently invoked through NetEssential.

Inbound Data Requirements

This section details the requirements for the inbound data, the URL, the HTTP headers and the data.

URL Format

This section includes both information on the standard URL and the URL query-string parameters.

Standard URL

The standard URL is interpreted as follows:

```
name.or.ip[:port]/converter/alias/program/ignored/?token-querystring
```

where:

name.or.ip specifies either the server name (e.g. magicsoftware.com, etc.) or an IP address (e.g. 68.145.209.73 or 192.168.1.1)

port (optional) specifies the port number. The default is 80 for http (non-SSL) and 443 for https (SSL).

converter specifies the name of the converter program to be used for the request. For CWS, this is 1-8 characters. A special value of 'CICS' indicates that no converter is to be used. The workstation can for example, specify the name of the encryption/compression program to be used. This program will be invoked via CWS and thus should follow the CWS guidelines.

alias specifies the transaction ID of the alias transaction (analyzer) for the request. For CWS, it can be up to 4 characters and the default value is ABWS.

program specifies the program name that is to service the request. A value of 'AB' indicates to use the standard AppBuilder front end program (BPSCWSF).

ignored this part is ignored by (the CICS supplied default analyzer (DFHWBADX) and) the AppBuilder analyzer, but may be used by the converter or application.

token-querystring specifies the token data passed from the client. Unless the first part of the query string is one of the supported query-string parameters, the first 8 bytes specify the user token passed to the (user-supplied) converter. The analyzer(s) ignores the rest, but may be used by the converter or application.

For example:

```
test.bphnx.com/CICS/ABWS/AB?RULE=MYRULE&OLEN=120
```

URL Query-String Parameters

The query-string is the tail part of the URL; that is, it is the portion that follows the first '?' character. It typically contains keyword parameters and values, separated by an '&'. For example:

```
test.bphx.com/CICS/ABWS/AB?KEY=value&WORD=parm
```

The AppBuilder HTTP implementation currently supports and generally requires the following query-string parameters:

RULE=<short-name>

The RULE= keyword specifies the program name of the frontier rule to be executed.

VIEWTYPE=<n>

The VIEWTYPE= keyword specifies the type of view(s) the (frontier) rule expects. The possible values are:

Viewtype values

0	No view
1	IN view only
2	OUT view only
3	INOUT view (single)
4	IN & OUT views (separate)

OLEN=<n>

The OLEN= keyword specifies the length, in bytes, of the (frontier rule's) output view. The value of n must consist of numeric characters (0-9), and must match the frontier rule's output view length. A 0 value is used to indicate there is no output view.

HTTP Headers

The AppBuilder HTTP implementation requires the following HTTP Headers to be present:

Content-Length - describes the length of the data.

Content-Type -describes the data format. Currently supported values are:

Content-type values

application/bpoc	OpenCobol view data
application/bpcc	ClassicCobol view data
application/bperr	Server Side error message
application/bp-text	Text data
application/bp-binary	Binary data
application/bp-textid	Text data ID
application/bp-binaryid	Binary data ID

Encrypted/compressed data will have the same content-types, but suffixed with '-compressed'. For example: encrypted OpenCOBOL view data is associated with the content-type value of 'application/bpoc-compressed, while "normal" (un-encrypted) OpenCOBOL view data is associated with 'application/bpoc'.

Text/Binary Blob Support

When the client sends a view that contains either a TEXT or BINARY field, these fields will be transmitted separately before the actual request to invoke the server rule. This is accomplished by specifying the content-type accordingly.

The BLOB related values for the HTTP Content-Type header are:

BLOB values

application/bp-text	Request to store text data
application/bp-binary	Request to store binary data
application/bp-textid	Request to retrieve text data
application/bp-binaryid	Request to retrieve binary data

AppBuilder provides BPSBLOB, a BLOB handler compatible with the existing BLOB interface. Customers may create their own BLOB handler program to meet any additional requirements, such as TS, DB2 or VSAM. In addition to the BLOB handler, customers may also need to create runtime components to provide BLOB access to their rules. The following section details the BLOB handler interface.

The name of the BLOB handler programs (BLOB exit) are configured via DNAINI settings. See [Configuring AppBuilder INI Settings for HTTP Support](#) for details.

Based on the content-type, the front-end Business Logic program will set up (i.e. allocate storage, etc.) to call the BLOB handler. For BLOB retrieve requests, the exit will first be called to provide the length of the BLOB. The back-end Business Logic program is invoked; this in turn links to the BLOB handler to perform the requested BLOB function - retrieve (get) or store (put).

The BLOB handler, after storing the BLOB data, must return a uniquely identifiable token. This token (ID or BLOB-ID) is passed into the rule's view for subsequent retrieval. The BLOB handler must also return the actual length of the returned token.

The process is similar for TEXT/BINARY fields in the output view. Only the token is returned in the output view data. When the client detects this, it initiates another HTTP request to retrieve it. This too follows the above-mentioned configuration. The BLOB handler in this case is responsible for retrieving the BLOB data corresponding to the token and returning the data to the client.

BLOB Content-Type Values

The BLOB related values for the HTTP Content-Type header are:

BLOB Content-type values

application/bp-text	Request to store text data. The request contains the text data and the response data will be the token with a content-type of application/bp-textid.
application/bp-binary	Request to store binary data. The request contains the binary data and the response data will be the token with a content-type of application/bp-binaryid.
application/bp-textid	Request to retrieve text data. The request contains the text token and the response data will be the text data with a content-type of application/bp-textid.
application/bp-binaryid	Request to retrieve binary data. The request contains the binary token ID and the response data will be the binary data with a content-type of application/bp-binary.



Compressed/encrypted data will have the same content-type values, but suffixed with '-compressed'. See [HTTP Headers](#) for details.

BLOB Handler COMMAREA

The following C source sample describes the layout of the COMMAREA passed to the BLOB Handler.

```
typedef struct
{
    char Rule_name[ 8 ]; /* Service rule name */
    char BlobType; /* Blob-Type: */
    char Function; /* Requested function: */
    char DataLoc; /* Blob data location */
    char FFU1_1; /* Filler */
    int Length; /* Length of Blob data */
    char* BlobPtr; /* Blob data pointer */
    char BlobID[ BLOB_ID_MAXLEN ]; /* Blob ID (token) */
    int rc; /* Return-code */
    char BlobMsg[ BLOB_MSG_LEN ]; /* Blob exit (error) message */
    char Blob[ BLOB_FLD_LEN ]; /* Blob data (disregard size) */
} ABBLOB;
```



This structure is defined in BLOBX.H. It defines the values for BLOB_ID_MAXLEN, BLOB_MSG_LEN, and BLOB_FLD_LEN.

Rule_name contains the name of the intended (frontier) rule program that has the BLOB in either its input and/or output view. The purpose of the *Rule_name* field is to allow the BLOB Handler to further qualify the BLOB request based on the name.

BlobType indicates the type of BLOB. Currently supported values are:

BLOB type values

Value	Symbolic Name	Type
'B'	'BLOB_TYPE_BINARY'	Binary (same as Image)
'B'	'BLOB_TYPE_IMAGE'	Image (same as Binary)

'T	'BLOB_TYPE_TEXT	Text
----	-----------------	------

Function indicates the type of BLOB function requested. Currently supported values are:

BLOB function values

Value	Symbolic Name	Function
'L	'BLOB_FUNC_LEN	Retrieve length of BLOB data
'P	'BLOB_FUNC_PUT	Put (store) BLOB data (return BLOB-ID)
'G	'BLOB_FUNC_GET	Get (retrieve) BLOB data

DataLoc indicates the location of the BLOB Data, either to be stored from, or retrieved into. Currently supported values are:

Data location values

Value	Symbolic Name	Data Location
'C	'BLOB_LOC_CA	Blob data is in the Blob field
'P	'BLOB_LOC_PTR	Blob data is pointed to by the BlobPtr field

Length specifies the length of the BLOB data. For BLOB_FUNC_LEN requests, the BLOB Handler should return in this field the length of the BLOB specified in the BlobID field. For BLOB_FUNC_PUT requests, the BLOB Handler should return in this field the length of the BLOB token specified (also returned) in the BlobID field. For BLOB_FUNC_GET requests, the BLOB Handler should return the length of the BLOB specified (also returned) in the BlobID field.

BlobID field contains the BLOB token (ID). This is an input field for BLOB_FUNC_LEN and BLOB_FUNC_GET requests. Successful BLOB_FUNC_PUT requests will provide in this field the newly created BLOB token (ID), as well as specifying the BLOB token's length in the Length field. The size of the field is indicated by the symbolic name BLOB_ID_LEN which currently has a value of 256. As this may change, it is recommended to use the symbolic name.

rc contains the BLOB Handler's return code. Currently supported values are:

BLOB Handler return codes

Value	Symbolic Name	Return Code Description
0	BLOB_RC_OK	Blob function was successful
1	BLOB_RC_ERROR	Blob error (generic)
2	BLOB_RC_PARMERR	Invalid parameter
3	BLOB_RC_BLOBIDERR	Invalid or missing Blob token (ID) value
4	BLOB_RC_IOERR	I/O error

Any other non-zero value is considered an error.

BlobMsg field contains an error description message provided by the BLOB handler. For non-zero return codes from the BLOB Handler, the contents of BlobMsg will be output along with an error message. The size of the field is indicated by the symbolic name BLOB_MSG_LEN which currently has a value of 80. As this may change, it is recommended to use the symbolic name.

Blob will contain the BLOB data itself only when the value of DataLoc indicates so. Although it is declared with a pre-defined length (currently 200, symbolic name BLOB_FLD_LEN), the actual length will vary and may exceed this. The actual size is indicated in the Length field.

Installation Checklist

The following checklist describes the necessary steps to install the AppBuilder HTTP support feature. These steps are detailed below.

1. If not performed when installing the tape, link the provided objects to create the required load modules.
 - BPSCWSA - Analyzer
 - BPSCWSB - Business logic; Back-end
 - BPSCWSF - Business logic; Front-end
 - BPSBLOB - BLOB handler
 - DNARTX - Routing & transaction switching exit

These should be accessible via the CICS DFHRPL concatenation.

In multiple region environments, the analyzer (BPSCWSA) and Business Logic front-end (BPSCWSF) must execute in the same CICS region as the TCPIP SERVICE receiving the request. The Business Logic back-end (BPSCWSB) must execute in the same CICS region as the frontier rule (including HPECAPI for Classic COBOL environments).

2. Prepare CICS to support HTTP. This would include SIT parameters and other resource definitions, for general HTTP support in CICS as well as the AppBuilder specific definitions. If using SSL, additional steps are required.
 - a. CICS HTTP Support:
 - SIT parameters
 - Resource definitions: Transactions, programs, Transient Data queue, and TSMODEL
 - Define and install TCPIP SERVICE(s)
 - b. Install the provided AppBuilder sample definitions, making any required changes. These include the transaction, program, and TCPIP SERVICE definitions.
 - Resource definitions: Transactions, programs, and TSMODEL.



If using SSL, additional steps are required. These are not described here as they are beyond the scope of this manual.

3. Check and set DNAINI settings.
4. If required, create a converter program.

System Initialization parameters for CICS Web support

The CICS System initialization parameters relating to CICS Web Support are:

DOCCODEPAGE: specifies the default host code page to be used by the document domain.

TCPIP: specifies whether CICS TCPIP Services are to be activated at CICS startup. You must specify TCPIP=YES to use CICS Web support.

DFHSIT PARAMETERS

The following SIT parameters relate to SSL security:

ENCRYPYTION: specifies the level of encryption, with possible values of WEAK, NORMAL, and STRONG (40/56/128 bit encryption, respectively). Note that as the STRONG setting is only available in the U.S.A. and Canada and requires additional software, this option was not tested.

KEYRING: species the default KEYRING to be used. It is 1-47 characters, no spaces and mixed case OK. Note that if value is incorrect, the CICS region may not start.

SSLTCBS: Limit of simultaneous SSL threads. Default value is 8.



The TCBS used by SSL can consume considerable storage (most above, but some below). 1264K above/each SSL TCB for SSL enclave, plus 13K-60K below for TCBS/other blocks.

SSDELAY: Timeout value (seconds) for SSL session-id caching. Default value is 600.

CICS HTTP SUPPORT

It is recommended to add the CICS supplied resource definition group DFHWEB which contains:

1. Transactions Required by CICS Web Support (for example, CWBA And CWXN)
2. Programs Supplied with the CICS Web support
3. The CICS Web Support transient data queue for messages, CWBO
4. A Temporary storage model, DFHWEB

Group DFH\$SOT contains sample CICS Web TCPIP SERVICE definitions. DFH\$WBSN contains the resource definitions for the security sample programs.

TCPIP SERVICE Definitions for CICS Web support

You must define and install a TCPIP SERVICE for each port that you use for CICS Web support.

If you want to use more than one analyzer program, or more than one transaction definition associated with the Web Attach task, you must install more than one TCPIP SERVICE definition. Each TCPIP SERVICE must have unique port number.

If you use SSL, you must use separate ports for SSL And non-SSL requests. Each port requires its own TCPIPSERVICE definition.

Before a TCPIPSERVICE can be used, it must be opened. It is recommended to specify in the definition to open the service when it is installed. It can also be done manually using the CEMT Transaction or the SET TCPIPSERVICE System programming command.

TCPIPSERVICE Parameters

URM=BPSCWSA: Specifies that this TCP/IP service uses the BPSCWSA analyzer program.

TRANSACTION=CWXN: Specifies the transaction the HTTP listener runs under. The specified transaction - recommended value CWXN - must point to the IBM supplied program DFHWBXN. The analyzer will perform transaction switching and set the transaction ID under which the business logic and rule program will execute.

SSL={NO|YES|CLIENTAUTH}: Specifies whether SSL is to be used.

A value of NO indicates SSL is not used.

Specifying YES implies CICS is to send server certificate and data is encrypted. No client certificate is required.

CLIENTAUTH is the same as YES, plus client certificate must be sent.

AUTHENTICATION={NO|BASIC|CERTIFICATE|AUTOREGISTER|AUTOMATIC}: Specifies the required level of security authentication:

A value of NO indicates that client certificate is not required, but will be used if supplied.

Specify BASIC for CICS to perform the basic HTTP authentication whereby the user ID and password are transmitted over the network, encrypted using the Base64 encoding scheme. CERTIFICATE implies that the client must send certificate. This requires the SSL parameter to be set to CLIENTAUTH.

AUTOREGISTER - This option requires the client provide a certificate, but does not require that the certificate be associated with a user ID on the CICS host. If the certificate is not already associated with a user ID, then the user will be prompted for a user ID to which the certificate should be associated and the password for that user ID to authenticate the "auto-registration" request.

The AUTOMATIC option combines AUTOREGISTER and BASIC. If a valid SSL certificate is supplied by the client and this is registered with a valid RACF user ID, the associated RACF user ID is used. If the certificate is not registered with RACF, HTTP Basic Authentication is used to obtain a valid RACF user ID and password; when this is received and verified, the certificate is registered with RACF as being associated with this user ID. If no valid SSL client certificate is supplied by the client, HTTP Basic Authentication is used.

CERTIFICATE: This contains the name of the server certificate, when SSL is set to YES or CLIENTAUTH. If blank, the default certificate for the KEYRING specified in the DFHSIT will be used.

For example:

```
DEFINE TCPIPSERVICE(ABCWS) GROUP(ABCWS)
      DESCRIPTION(APPBUILDER HTTP LISTENER)
      URM(BPSCWSA) PORTNUMBER(nnnn) STATUS(OPEN) SSL(NO)
      AUTHENTICATE(BASIC) TRANSACTION(CWXN) BACKLOG(1) SOCKETCLOSE(NO)
```

Defining AppBuilder supplied programs to CICS

```

DEFINE PROGRAM(BPSCWSA) GROUP(ABCWS)
  DESCRIPTION(APPBUILDER HTTP/CWS ANALYZER)
  LANGUAGE(C) RELOAD(NO) RESIDENT(NO) USAGE(NORMAL)
  USELPACOPY(NO) STATUS(ENABLED) CEDF(YES) DATALOCATION(ANY)
  EXECKEY(CICS) CONCURRENCY(QUASIRENT) DYNAMIC(NO)
  EXECUTIONSET(FULLAPI) JVM(NO) HOTPOOL(NO)

DEFINE PROGRAM(BPSCWSB) GROUP(ABCWS)
  DESCRIPTION(APPBUILDER HTTP/CWS BLI BACK-END)
  LANGUAGE(C) RELOAD(NO) RESIDENT(NO) USAGE(NORMAL)
  USELPACOPY(NO) STATUS(ENABLED) CEDF(YES) DATALOCATION(ANY)
  EXECKEY(USER) CONCURRENCY(QUASIRENT) DYNAMIC(NO)
  EXECUTIONSET(FULLAPI) JVM(NO) HOTPOOL(NO)

DEFINE PROGRAM(BPSCWSF) GROUP(ABCWS)
  DESCRIPTION(APPBUILDER HTTP/CWS BLI FRONT-END)
  LANGUAGE(C) RELOAD(NO) RESIDENT(NO) USAGE(NORMAL)
  USELPACOPY(NO) STATUS(ENABLED) CEDF(YES) DATALOCATION(ANY)
  EXECKEY(USER) CONCURRENCY(QUASIRENT) DYNAMIC(NO)
  EXECUTIONSET(FULLAPI) JVM(NO) HOTPOOL(NO)

DEFINE PROGRAM(DNARTX) GROUP(ABCWS)
  DESCRIPTION(NETE ROUTING & TRAN SWITCHING EXIT (KEY=CICS FOR CWS))
  LANGUAGE(C) RELOAD(NO) RESIDENT(NO) USAGE(NORMAL)
  USELPACOPY(NO) STATUS(ENABLED) CEDF(YES) DATALOCATION(ANY)
  EXECKEY(CICS) CONCURRENCY(QUASIRENT) DYNAMIC(NO)
  EXECUTIONSET(FULLAPI) JVM(NO)
  HOTPOOL(NO)

```

Defining AppBuilder supplied transactions to CICS

```

DEFINE TRANSACTION(ABWS) GROUP(ABCWS)
  DESCRIPTION(APPBUILDER HTTP/CWS SUPPORT)
  PROGRAM(BPSCWSF) TWASIZE(0) PROFILE(DFHCICST) STATUS(ENABLED)
  TASKDATALOC(ANY) TASKDATAKEY(USER) STORAGECLEAR(NO)
  RUNAWAY(SYSTEM) SHUTDOWN(DISABLED) ISOLATE(YES) DYNAMIC(NO)
  ROUTABLE(NO) PRIORITY(1) TRANCLASS(DFHTCL00) DTIMOUT(NO)
  RESTART(NO) SPURGE(NO) TPURGE(NO) DUMP(YES) TRACE(YES)
  CONFDATA(NO) ACTION(BACKOUT) WAIT(YES) WAITTIME(0,0,0)
  RESSEC(NO) CMDSEC(NO)

```

Defining TSMODEL (Optional, MRO/CICSplex only)

In MRO environments, passing views larger than 32K is accomplished via a Temporary Storage queue (TSQ). The characteristics of the TSQ are typically defined in a TSMODEL definition, specifying a TSQ prefix. Provide the same prefix value in the INI setting CWS_SERVER TSQ_PREFIX <TSQ-Prefix>.

The default value is 'ABCWSTSQ'.

Configuring AppBuilder INI Settings for HTTP Support

The AppBuilder CICS HTTP configuration is implemented via new settings in the (existing) DNAINI file. The following describes those new DNAINI settings. These are maintained through the existing NEAD or the newly introduced ABIN transaction utility.

CWS_CONVERTER <CONVERTER-TYPE> <Converter-Program>

where <Converter-Program> is the name of the converter program to be used for the associated Content-Type. Please refer to [The Converter](#)

[Programs](#) for details about the converter program.

The supported options are:

OPENCOBOL - Converter for OpenCOBOL view data, associated with application/bpoc

OPENCOBOL-COMPRESSED - Converter for encrypted/compressed OpenCOBOL view data, associated with application/bpoc

CLASSICCOBOL - Converter for ClassicCOBOL view data, associated with application/bpcc

CLASSICCOBOL-COMPRESSED - Converter for ClassicCOBOL view data, associated with application/bpcc

ERRORMESSAGE - Converter for (server-side) error messages, associated with application/bperr.

ERRORMESSAGE-COMPRESSED - Converter for encrypted/compressed (server-side) error messages, associated with application/bperr-compressed.

TEXT - Converter for text BLOB data, associated with application/bp-text.

TEXT-COMPRESSED - Converter for encrypted/compressed text BLOB data, associated with application/bp-text-compressed.

TEXTID - Converter for text BLOB ID, associated with application/bp-textid.

TEXTID-COMPRESSED - Converter for encrypted/compressed text BLOB ID, associated with application/bp-textid-compressed.

BINARY - Converter for binary BLOB data, associated with application/bp-binary.

BINARY-COMPRESSED - Converter for encrypted/compressed binary BLOB data, associated with application/bp-binary.

BINARYID - Converter for binary BLOB ID, associated with application/bp-binaryid.

BINARYID-COMPRESSED - Converter for encrypted/compressed binary BLOB ID, associated with application/bp-binaryid-compressed

BLOB Handler Values

CWS_SERVER	BLOBEXIT_TEXT	<BLOB-Exit-Program>
CWS_SERVER	BLOBEXIT_BIN	<BLOB-Exit-Program>

where *<BLOB-Exit-Program>* is the name of the BLOB handler program to be used for the associated BLOB type - TEXT or BINARY. The default value is "BPSBLOB". Please refer to [Text/Binary Blob Support](#) for details about the BLOB handler programs.

CWS_SERVER TSQ_PREFIX <TSQ-Prefix>

where *<TSQ-Prefix>* is used to prefix the temporary storage queue names used by the Business Logic to pass large view data. Typically there will be a corresponding TSMODEL definition. The default value is "ABCWSTSQ".

CWS_SERVER TSQ_MAX_ITEMLEN <Max-Item-Length>

where *<Max-Item-Length>* specifies the maximum length of a single TSQ item ("record"). The default is 8,000 bytes.

CWS_SERVER WS_VIEW_IN_MEMORY {YES | NO}

This setting is useful in single region environments as specifying YES causes the front-end to pass large (>32K) view data via addresses in the commarea. Any other value will result in data passed via Temporary Storage queues. Please refer to [The Business Logic Programs](#) section for details about the Business Logic programs.

Existing DNAINI Settings

For convenience, the following existing DNAINI settings are also used by the AppBuilder CWS implementation. Please refer to the *INI Settings Reference Guide* manual for details regarding these INI settings.

DNA_EXITS SRV_RTX_EXIT <Routing-TranSwitch-Exit>

Specifies the name of the Routing and Transaction-Switching Exit. The default is the supplied DNARTX.

DNA_SERVER RTX_EXITDATALEN <R-TS-Exit-buffer-size>

Specifies the size of the data buffer provided to the Routing and Transaction-Switching Exit. The default is 120 bytes.



Due to the nature of CWS, the contents of the buffer will NOT be maintained across calls to the Routing and Transaction-Switching Exit. This does not pose any problems or restrictions on the supplied DNARTX. Existing user exits (for the TCP Listener) may rely on the buffer contents and thus would need to be modified for CWS usage.

MVS_LISTENER LISTENER_SECURITY <Security-Option>

For details on Listener Security, see [CICS TCP/IP Listener Security Settings](#).

DNA_SERVER ALLOW_EXPIRED_RULE <Rule-Name>

where *<Rule-Name>* is the name of the rule allowed to proceed despite an expired-password return code.

Configuring C HTTP Client

To enable the client to send via HTTP, in the HPS.INI, set [AE Runtime] COMMS_HANDLER to httpint. This enables HTTP communication from the client and not allows LU2, LU6.2 or TCP/IP from the client.

[HTTP]

This section contains general settings for the client. There can be only one.

DESTINATION

This setting describes the target server. It is either the full name of a SERVER section or the name of a ROUTE section. For example, DESTINATION=ROUTE.table1

MSGLEVEL

This setting controls the messages written to the \$HPSINI\http.log file. There are 3 settings. MSGLEVEL=ERROR will write error messages only. MSGLEVEL=WARNING will write error messages and warnings. A warning may be an exit that cannot be loaded. MSGLEVEL=INFO will write errors and warnings, as well as a sequence of information messages showing the current request process.

DATATRACE

This setting controls how much request data is written to the log file.

DATATRACE=N(ONE) is the default and writes no data to the log

DATATRACE=I(P) writes network buffers to the http.log file

DATATRACE=H(TTP) writes http request and response data to the log

DATATRACE=A(ALL) writes http request/response data, network buffers as well as the C runtime input and output buffers to the log.



The dumping of sensitive information such as userids and passwords are under the control of the AUTHENT_EXIT if present. The exit is asked via an API call whether dumping of sensitive data is permitted or not. If permission is not granted DATATRACE=IP will not work at all and DATATRACE=HTTP will replace the contents of an 'Authorization' header with *****.

TEMP

This setting identifies a directory for the http client to store files. This is currently used when a BLOB object (TEXT or BINARY) is retrieved from the server. The blob is given a UUID as a name and stored in the TEMP directory. Users are responsible for managing the contents of the directory. We will not delete any files.

[SERVER.xxx]

This section defines a particular server. There may be multiple server sections. The section name uses 2 level qualifiers. The first, 'SERVER' indicates that this is a server section. The 'XXX' after the dot is the actual name of this server, i.e. a server named trex would have a server section [SERVER.trex] and would be identified by DESTINATION=SERVER.trex.

URL

This setting defines the target URL for the server process. Only 2 schemes are supported, http and https.

The standard URL is interpreted as follows:

```
name.or.ip[:port]/converter/alias/program/ignored/
```

where:

name.or.ip specifies either the server name (e.g. magic.com, TREX, etc.) or an IP address (e.g. 68.145.209.73 or 192.168.1.1)

port (optional) specifies the port number. The default is 80 for http (non-SSL) and 443 for https (SSL).

converter specifies the name of the converter program to be used for the request. For CWS, this is 1-8 characters. A special value of CICS indicates that no converter is to be used. The workstation can for example, specify the name of the encryption/compression program to be used. This program will be invoked via CWS and thus should follow the CWS guidelines.

alias specifies the transaction ID of the alias transaction (analyzer) for the request. For CWS, It can be up to 4 characters and the default value is ABWS.

program specifies the program name that is to service the request. A value of 'AB' indicates to use the standard AppBuilder front end program - BPSCWSF.

ignored this part is ignored by the CICS supplied default analyzer (DFHWBADX) and the AppBuilder analyzer, but may be used by an optional converter program.

For example:

```
URL=http://trex:3074/CICS/ABWS/AB
```

CONVERTER

This setting identifies the converter associated with this server.

AUTHENT_EXIT

This setting defines an authorization exit. This exit provides a userid and password for sites requiring them. It also provides permission for the dumping of sensitive data.

The exit is defined by the full path to the exit dll.

SSL_EXIT

This setting defines an SSL exit. This is only required if the user is replacing the supplied ssl exit with a custom one. It is the full path to the exit dll.

COMPRESS_EXIT

This setting defines a compression exit. It is the full path to the exit dll.

TIMEOUT

This setting is a numerical value in seconds. It defines how long the client will wait for a response from this particular server.

DATATRACE

This setting overrides the DATATRACE setting in the [HTTP] section for this server only.

[ROUTE.xxx]

This section defines a route. A route is a static or dynamic method of selecting the target server based on the name of the called rule. There may be multiple ROUTE sections. The section name uses 2 level qualifiers. The first, 'ROUTE' indicates that this is a routing section. The 'XXX' after the dot is the actual name of this route entry. That is, a route named 'table1' would have a route section [ROUTE.table1] and would be identified by DESTINATION=ROUTE.table1.

TYPE

This setting defines the type of the ROUTE. There are 2 values. ROUTE=STATIC or ROUTE=EXIT. A static route selects servers based on a lookup table whereas an exit route uses a user supplied exit to select the server.

URL

This setting is the location of the routing table for a TYPE=STATIC entry or the path to the routing exit dll for a TYPE=EXIT entry. For example:

```
URL=file://c:/ab300/httpint/test/httproue.tbl
```

or

```
URL=file://c:/ab300/httpint/test/httpptest.dld
```

[CONVERTER.XXX]

This section defines a language/codepage converter. The section name uses 2 level qualifiers. The first, 'CONVERTER' indicates that this is a converter section. The 'XXX' after the dot is the actual name of this converter entry, i.e. a converter named 'OPENCOBOL' would have a section [CONVERTER.OPENCODOL] and would be identified by CONVERTER=OPENCODOL in the server section.

MODULE

This setting is the name of the converter dll. Currently we supply 2 converters, one for OpenCOBOL (bpoc.dll) and one for Classic COBOL (bpcc.dll). This entry is the full path to one of these dll's. For example:

```
MODULE=c:\appbuilder\nt\sys\bin\bpoc.dll
```

TARGET_CP

This setting defines the codepage of the target system as understood by ICU. For example:

```
TARGET_CP=ibm037
```

Usage Notes

CICS

When the CICS Listener is used in an MRO/Sysplex environment, TCP/IP should be installed in the front-end running the Listener transaction and in any back-end region running AppBuilder transactions.



A CICS Sockets restriction requires all regions in question to be in the same LPAR.

When a remote procedure call (RPC) is received, the front-end CICS looks at the message sent by the client and passes the socket opened by the client request to the back-end CICS. The back-end CICS receives the message, calls the rule requested, and sends the reply. The BACKLOG parameter for the port should be adjusted depending on the transaction load, using the EZAC transaction.



If this value is too low, communication errors might occur with tcpip protocol error code 10060.

Authentication and Authorization Exits

Configuration

To configure the Authentication and Authorization User Exits, specify the Exits through the following DNAINI settings:

- DNA_EXITS SRV_AUTHENT_EXIT = STATIC_LINK
- DNA_EXITS SRV_AUTHOR_EXIT = STATIC_LINK

The User Exit code must have entry points with the following names:

- SrvAuthentLibMain - Entry point for Authentication Exit
- SrvAuthorLibMain - Entry point for Authorization Exit

When pre-linking DNALSTNR, or HPSDNA00, or both, you must either specifically include the (AUTHENTX) object and remove DNACEXIT, or

name the user's object DNACEXIT and ensure it appears first in the OBJECT DD card concatenation.



Security Cookies are not available to the Exits. The Listener cannot pass security cookie contents. Existing exits that require this information cannot use this feature.

Double invocation of exits

The Authentication and Authorization Exits can be called from both the Listener (DNALSTNR) and the rule transaction (HPSDNA00). You might be able to differentiate the two based on the CICS program name or on the transaction ID. To obtain the program name, the User Exit can issue an EXEC CICS ASSIGN PROGRAM(). To obtain the ID, examine the EIBTRNID field in the EIB.

In an MRO environment, the Listener in the TOR can be configured separately from the HPSDNA00 in the AOR so long as they do not share the DNAINI file.

Alternatively, you can use different user exits, one linked with the DNALSTNR and one with HPSDNA00. In this case, the "dummy" exit only needs to return a successful return code:

```
SRV_CHKUSER_SUCCESS / SRV_CHKSERV_SUCCESS
```

DNAINI Settings

The following is a list of the available DNAINI settings used to configure the NetEssential TCP/IP parameters. There are additional required generic settings.

- [TCP/IP Port Number](#)
- [Security Authentication](#)
- [Session Termination Semantics](#)
- [TCP/IP Listener Abend Handler](#)
- [DoS Prevention](#)
- [AppBuilder Communications Tracing](#)
- [IP Listener Dynamic Buffer Size](#)
- [OPEN-COBOL](#)
- [USE_PREAMBLE_RULEID](#)
- [Connection Termination Semantics](#)
- [SERVER_RECV_TIMEOUT](#)
- [TCPIP_ABENDPROGRAM](#)
- [CICS DNAINI settings for the HTTP Listener](#)

TCP/IP Port Number

We strongly recommend maintaining an EZAC configuration record; however, if none exists, the Listener might attempt to use the value specified for the TCP_PORT_NUMBER variable in the DNA_SERVER section. You can add it with a value of *nnnn*, where *nnnn* is the port number assigned to the CICS region. For example:

```
DNA_SERVER      TCPIP_PORT_NUMBER      nnnn
```

Security Authentication

Add the variable LISTENER_SECURITY to the new MVS_LISTENER section. This variable determines how the Listener authenticates the provided user credentials. The Security settings are described in detail in [CICS TCP/IP Listener Security Settings](#).

Session Termination Semantics

There are numerous enhancements in AppBuilder Communications regarding session termination semantics. These enhancements provide control over session termination processing and are implemented with two DNAINI settings:

```
DNA_SERVER      TCPIP_CLOSEWAIT      nnn
DNA_SERVER      TCPIP_LINGERTIME      nnn
```

where *nnn* is the timeout value, specified in seconds. The values for these variables can be different.

After sending the last response to the client, the CICS service closes the connection. This can be done in any of the following ways:

- Issue a CLOSE() for the socket.

- Issue a SELECT with a timeout as specified for TCPIP_CLOSEWAIT, and after either receiving data (the close indication) from the client or when the timeout expires, issue a CLOSE() for the socket.
- Issue a blocking RECV() and after receiving data (the close indication) from the client, issue a CLOSE() for the socket.

This behavior is determined primarily according to the value of TCPIP_CLOSEWAIT:

- If the TCPIP_CLOSEWAIT timeout is not specified or is empty: Do nothing (option 1).
- If the TCPIP_CLOSEWAIT timeout is a positive number: Issue a SELECT (option 2).
- Else (TCPIP_CLOSEWAIT timeout is invalid, negative, or zero): Issue a blocking RECV (option 3).

The TCP Listener's session termination semantics have been enhanced to allow the workstation to initiate the close(). If the existing DNAINI setting?TCPIP_CLOSEWAIT in the DNA_SERVER section - is specified with a positive number, the listener will not issue a close() for the socket. Instead, the listener will add the socket to the EXCEPTION mask, waiting for the workstation to initiate the close. During this waiting time, the listener is free to process other workstation requests.

The socket is also subject to the GIVESOCKET() timeout as specified in the GIVTIME parameter in the listener's EZAC configuration record.

Regardless of the SELECT/RECV, you can also optionally specify a LINGER time.

- If the TCPIP_LINGERTIME is a positive number: Set SO_LINGER to TCPIP_LINGERTIME.
- Else only in the event of SELECT/RECV errors: Set SO_LINGER to default (5 seconds).

If the SELECT timeout or setting blocking mode (IOCTL) fails, SO_LINGER will always be set. If a valid value is not specified, SO_LINGER reverts to its default value (5 seconds).



In the event of SELECT/RECV errors, SO_LINGER will not be set.

To disable the Linger option, specify a LingerTime of 0 (zero); SO_LINGER will not be set.

TCP/IP Listener Abend Handler

You might need to perform some clean-up operation, such as Workload Manager (WLM) de-registration, in the unexpected case of a Listener Abend. This would need to be done in an external abend handler program. A sample program, DNAABEND, is provided in both source (C) and object format that performs WLM de-registration.

To activate the Abend Handler for the TCP/IP Listener, specify DNAABEND as the value for the TCPIP_ABENDPROGRAM variable in the DNA_SERVER section. For example:

```
DNA_SERVER      TCPIP_ABENDPROGRAM  DNAABEND
```

The Listener checks the value and issues a "HANDLE ABEND PROGRAM(DNAABEND)" if it is not empty.

Setting Up the Sample

The source is standard C and does not have any special compiler requirements. It contains EXEC CICS statements and needs to be translated by the CICS C translator. When linking (binding), the CICS TCP/IP module must be included. The Linkage-Editor cards should resemble the following entries:

```
INCLUDE OBJLIB(DNAABEND)
INCLUDE SYSLIB(EZACIC07)
INCLUDE SYSLIB(DFHELII)
INCLUDE SYSLIB(DFHCPCLC)
INCLUDE SYSLIB(DFHCPPLR)
NAME DNAABEND(R)
```

The program must be defined to CICS as a C program.



The Abend Handler (DNAABEND) program's definition must match that of the Listener. For example, if the Listener transaction is defined as TASKDATAKEY(CICS), the Abend Handler program's definition must specify EXECKEY(CICS); otherwise, an "AEZD" abend occurs.

DoS Prevention

In order to defend against denial-of-service (DoS) attacks, the AppBuilder Communications CICS TCP/IP Listener incorporates an "emergency brake." It consists of a configurable timeout that causes the Listener to issue an error message and close the socket if it expires before data is received.



During the timeout period, the Listener will probably consume CPU cycles, as there is always data waiting to be processed. However, other connections as well as existing and new requests will be processed.

The desired timeout value in seconds is specified by a DNAINI setting:

```
DNA_SERVER      SERVER_RECV_TIMEOUT nnn
```



Use caution when specifying the timeout value. Ensure that it takes into account the slowest network connection, yet keep it as low as possible to avoid unnecessary CPU cycles.

AppBuilder Communications Tracing

AppBuilder Communications has a tracing feature that can be used in a test environment or to troubleshoot problems. The level of trace output is specified with the DEBUGLVL variable in the TRACING section. In the same section, HPSCATS specifies the CICS file (DD) name that points to the VSAM error message text file. The name does not have to be DNAERROR; any valid CICS name is acceptable.

```
TRACING      DEBUGLVL      n
TRACING      HPSCATS       0 : DNAERROR
```

where *n* can be one of the following values, either numeric or text (as-is):

- 0 - ERRORS = report only ERROR + status view
- 1 - ERRORS_AND_LENGTHS = report level 0 + additional messages + data lengths
- 2 - ERRORS_AND_DATA = report level 1 + contents of input/output data
- 3 - ERRORS_AND_TRACES = report level 2 + internal tracing

The tracing settings are generic, applicable to all communications protocols - LU2, LU62, HTTP, etc., as well as TCP/IP.

IP Listener Dynamic Buffer Size

The TCP/IP Listener for CICS now supports a dynamic user-configurable sized buffer to be passed to dna_TcpMVSDecode. This value is specified using a DNAINI setting:

```
DNA_SERVER      TCP/IP_PEEKBUFSIZE = 144
```

The value specified indicates the number of bytes that the Listener will pass to the dna_TcpMVSDecode function. The default value is 144, which is also the minimum. If a lower value is specified, a value of 144 is used.

OPEN-COBOL

The OPEN-COBOL settings are generic, and are used with all other communications protocols - LU2, LU62, HTTP, etc., as well as TCP/IP. Example:

```
OPEN-COBOL      ENABLED      Y
OPEN-COBOL      DATE_FORMAT   %Y-%0m-%0d
OPEN-COBOL      TIME_FORMAT   %0t.%0m.%0s.%0f
OPEN-COBOL      EIB_COMMAREA  N
```

To determine whether it is a standard AppBuilder or OpenCOBOL runtime environment, AppBuilder Communications checks the ENABLED setting in the OPEN-COBOL section. If set to Y[ES], the DATE_FORMAT and TIME_FORMAT settings are used to determine the format of OpenCOBOL date and time fields.



Currently the only DATE format supported is "ISO" or "%Y-%0m-%0d", resulting in yyyy-mm-dd. The only TIME format supported is "%0t.%0m.%0s.%0f" which implies hh.mm.ss.mls. Leading zeros must be present. You may however, change either the DATE or TIME delimiters.

Legend:

yyyy - Year, including the century.
 mm - Month
 dd - Date
 hh - Hours in a 24-hour clock
 mm - Minutes
 ss - Seconds
 mls - Milliseconds

USE_PREAMBLE_RULEID

Example:

```
DNA_SERVER      USE_PREAMBLE_RULEID Y
```

For the CICS TCP/IP Listener (DNALSTNR) to support the presence of security cookies, the Listener needs to extract the correct rule-id to perform the required transaction switching. A new DNAINI setting in the DNA_SERVER section is checked. If the value of the keyword USE_PREAMBLE_RULEID is Y (or y), the (looked-up) rule name is taken from the preamble.

There are circumstances in which the rule ID in the preamble is in ASCII code. Therefore, the first character in the rule ID is checked. If in the range x'41' through x'5A' (ASCII A-Z), the name is assumed to be in ASCII and is converted to EBCDIC. This conversion includes characters (lower and upper case), digits, and the blank character:

ABCDEFGHJKLMNOPQRSTUVWXYZ, abcdefghijklmnopqrstuvwxyz, 0123456789 and ' ' (blank).



The security cookie is used by the authentication exit. To implement this, HPSDNA00 must be re-linked with the customized authentication exit (DNACEXIT) and the exit must also be enabled using DNAINI settings:

```
DNA_EXITS      SRV_AUTHENT_EXIT      STATIC_LINK
```

Connection Termination Semantics

TCPIP_CLOSEWAIT

Example:

```
DNA_SERVER      TCPIP_CLOSEWAIT      <seconds>
```

TCPIP_LINGERTIME

Example:

```
DNA_SERVER      TCPIP_LINGERTIME      <seconds>
```

The AppBuilder Communications TCP/IP implementation now provides greater flexibility to implement the various connection closing scenarios, as follows:

There are two DNAINI settings in the DNA_SERVER section: TCPIP_CLOSEWAIT and TCPIP_LINGERTIME. Both settings indicate a time-out value, specified in seconds.

The logic is as follows:

- If the TCPIP_CLOSEWAIT timeout is not specified or is empty: Do nothing.
- If the TCPIP_CLOSEWAIT is a positive number: Issue a SELECT.
- Else (TCPIP_CLOSEWAIT timeout is invalid, negative, or zero): Issue a Blocking RECV.

Regardless of the SELECT/RECV, you can optionally specify a LINGER time. If LINGERTIME is a positive number?Set SO_LINGER to TCPIP_LINGERTIME. Otherwise, SO_LINGER is not set.



In the event of SELECT/RECV errors, SO_LINGER will not be set (it will probably fail). If the SELECT timeout or setting blocking mode (IOCTL) fails, SO_LINGER will always be set. If a valid value is not specified, SO_LINGER is set to the default value (5 seconds). To disable the Linger option, specify a TCPIP_LINGERTIME of 0 (zero), which results in SO_LINGER not being set.

SERVER_RECV_TIMEOUT

Example:

```
DNA_SERVER      SERVER_RECV_TIMEOUT 3
```

In this example, the Listener was vulnerable to DoS attacks by a rogue program connecting and sending only a few bytes of data. The Listener is notified that data is available, and it attempts to receive (peek) a minimum amount of expected header data. The Listener loops attempting to read the data (as it is not available) and therefore enters a wait (for data) state. It is then awoken due to the already pending data.

This scenario is highly unlikely unless someone is maliciously trying to sabotage the Listener. However, to defend against such an attack, the Listener was enhanced and an "emergency brake" was implemented. This consists of a configurable timeout, that, if expires before data is received, causes the Listener to issue an error message and close the socket.



During the timeout period, the Listener will probably consume CPU cycles because there always is data waiting to be processed. Other connections, as well as existing and new requests, will be processed.

TCPIP_ABENDPROGRAM

```
DNA_SERVER      TCPIP_ABENDPROGRAM <program>
```

The Listener checks the value, and, if not empty, it issues a HANDLE ABEND PROGRAM(< program >). No error handling is done. A sample program, DNAABEND, is also provided in source (C) and object format. As provided, it handles the WLM de-registration. However, you can add functionality to suit your needs. For example, restarting the Listener.

How to Set Up the Sample:

The source is standard C and does not have any special compiler requirements. It contains EXEC CICS statements and needs to be translated by the CICS C translator.

When linking (binding), the CICS TCP/IP module must be included. The Linkage-Editor cards would resemble the following:

```
INCLUDE OBJLIB(DNAABEND)
INCLUDE SYSLIB(EZACIC07)
INCLUDE SYSLIB(DFHELII)
INCLUDE SYSLIB(DFHCPLC)
INCLUDE SYSLIB(DFHCPLRR)
NAME DNAABEND(R)
```

The program must be defined to CICS as a C program.



The abend handler (DNAABEND) program's definition must match that of the Listener. This means that if the Listener transaction is defined as TASKDATAKEY(CICS), the abend handler program's definition must specify EXECKEY(CICS). (An 'AEZD' abend will occur if not.)

CICS DNAINI settings for the HTTP Listener

The AppBuilder CICS HTTP configuration is implemented using new settings in the (existing) DNAINI file. The following describes the new DNAINI settings. These are maintained through the existing NEAD transaction utility.

```
CWS_SERVER      TSQ_PREFIX      <TSQ-Prefix>
```

Where *<TSQ-Prefix>* is used to prefix the temporary storage queue names used by the Business Logic to pass large view data. Typically there will be a corresponding TSMODEL definition. The default value is "ABCWSTSQ".

```
CWS_SERVER      TSQ_MAX_ITEMLEN  <Max-Item-Length>
```

Where *<Max-Item-Length>* specifies the maximum length of a single TSQ item ("record"). The default is 8,000 bytes.

```
CWS_SERVER      CWS_VIEW_IN_MEMORY { YES | NO }
```

This setting is useful in single region environments as specifying YES causes the front-end to pass large (>32K) view data via addresses in the commarea. Any other value will result in data passed via Temporary Storage queues. Please refer to The Business Logic Programs section for details about the Business Logic programs.

```
DNA_SERVER      ALLOW_EXPIRED_RULE <Rule-Name>
```

Where *<Rule-Name>* is the name of the rule allowed to proceed despite an expired-password return code.

 See the additional DNAINI settings in *INI Settings Reference Guide*.

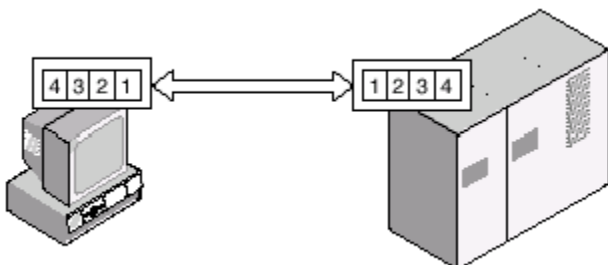
Understanding Performance Marshalling

AppBuilder uses performance marshalling to optimize the sending of data. To use performance marshalling, the value of DYNAMIC_DATA_OPTIMIZE must be set to Y (the default value) in the communications configuration file, dna.ini. If this value is N, performance marshalling fails. To edit the dna.ini file, use the Management Console (see [Managing Computers](#)). In addition, there are two formats for performance marshalling. When the value PERFORMANCE_MARSHALLING_VERSION is set to 2, it is possible to send data larger than 32K. If PERFORMANCE_MARSHALLING_VERSION is set to 1, that data being marshalled must be less than 32 K. It is recommended that PERFORMANCE_MARSHALLING_VERSION be set to the value of 2.

In general, marshalling is the process by which local data is converted into data for another environment in a distributed system and packaged for transmission to that environment. Thus, marshalling involves the conversion of data types and codepages between environments. In AppBuilder, this typically means converting between workstation ASCII data and mainframe EBCDIC data for transmission of data between these environments.

Applications distributed over heterogeneous networks often have unique data-handling requirements because the architectures or operating systems on which they run manage memory and data in different ways. Intel-based and RISC-based systems represent bytes in inverted order (little-endian versus big-endian, as shown in the following figure).

Byte-order data conversion



Some systems require alignment bytes; some do not. Converting incompatible data types in program code is inconvenient and error-prone. In the typical case, you must anticipate the possibility that platform-specific code might make programs non-portable to other systems.

The AppBuilder marshalling feature handles data conversion for you. It resolves incompatibilities between data definitions across systems by converting a representation specific to the sending platform to a platform-independent AppBuilder representation and then to a representation specific to the receiving platform. Where data translation is inappropriate, it supports the transmission of opaque, or untranslated, data. You can also use the feature to create custom data types in cases where data need to be manipulated between platforms for security or other reasons.

AppBuilder marshalling handles byte-order and ASCII-to-EBCDIC conversions and accounts for alignment bytes as necessary. Conventional support is provided for code-page conversions and single-byte and double-byte character sets. Code-page conversion is typically performed on the server, but administrators can use the routing mechanism to specify that conversion takes place on clients. AppBuilder data types are based on the External Data Representation (XDR) for network representation of standard data types, enhanced for self-describing data. AppBuilder supports transmission of binary large objects (BLOBs) as native data types.

Performance marshalling is not supported through an AppBuilder gateway. The AppBuilder gateway redirects data to the appropriate server or to another gateway. The marshalled data sent from the client is not marshalled for the server.

Performance Marshalling for CICS

AppBuilder handles full marshalling at the workstation so that the resource usage on the mainframe (CICS only) is reduced. Performance marshalling is valid only when communicating with CICS on the mainframe. It is configured on the client side by specifying any of the protocols summarized in [Performance marshalling protocols](#).

The advantage of performance marshalling is the reduction of mainframe processing time. The trade-off is that an increase in the amount of data being sent from the client can occur. For example, variable character (VarChar) fields are fully expanded during marshalling. If network bandwidth is more of an issue than CICS processing time, and if there are a large number of VarChar fields used in the views, performance marshalling could actually increase the amount of network traffic.

The client also must be configured to use the mainframe codepage. The Java client supports TCP/IP only. This enables Java applications to communicate directly to CICS via TCP/IP.

Performance marshalling protocols

Protocol name	Description
PMLU2	Performance marshalling protocol for LU2 messages from a C client to a COBOL host in CICS on the mainframe. It is used to call CICS rules prepared with AppBuilder.
PMLU62	Performance marshalling protocol for LU6.2 messages from a C client to a COBOL host in CICS on the mainframe. It is used to call CICS rules prepared with AppBuilder.
PMTCP/IP	Performance marshalling protocol for TCP/IP messages from a Java or C client to a COBOL host in CICS on the mainframe. It is used to call CICS rules prepared with AppBuilder.
POTCP/IP	Performance marshalling protocol for TCP/IP messages from a Java client to a COBOL host in CICS on the mainframe for OpenCOBOL. It is used to call CICS rules using OpenCOBOL. Performance marshalling for OpenCOBOL is not available for C clients.

OpenCOBOL rules and ClassicCOBOL rules must be marshalled and invoked differently. Because of these differences, a particular CICS region should have only one type of generated COBOL. For OpenCOBOL you must set the time and date delimiters in the dna.ini file.

Extending Communications

An exit is a program external call to an application that performs a task specific to a security subsystem, directory services mechanism, DBMS, or operating system. When AppBuilder communications detects an exit, it leaves the application (exits) to execute the external program. The interruption in the processing flow of the application allows system-specific processing to take place.

AppBuilder communications provides the necessary open interfaces to external products. You can perform many tasks that are usually specific to a given platform?security, directory services, database access, and data marshalling?without compromising the overall platform-independent nature of your application. Although the interfaces are packaged in different formats, each essentially provides a convenient way to perform system-dependent coding outside the application.

This topic describes how to extend AppBuilder communications to accomplish these tasks in the Java and C application environments:

- [Communication Exits in Java](#)
- [C Client and Server Exits](#) - how to use external security
- [Using Custom Data Types](#) - how to handle customized data types
- [Directory Services Exits in C](#) - how to use directory services
- [Database Exits in C](#) - how to access unsupported database management systems
- [CICS TCP/IP Listener Security Settings](#) - how to setup and use the TCP/IP listener

- [Security Authentication](#) - specifying the settings

In C, communications exits are implemented in two formats:

1. The authentication, authorization, tracing, and password encryption exits are implemented in dynamic link libraries (DLLs), whose presence AppBuilder communications detects at run time. Templates and makefiles for the exits are provided with the product. You can customize the templates and compile them for the client or server platforms on which they execute.
2. The data encryption, directory services, and open database interconnection exits are made available in shared libraries that are already linked to AppBuilder communications. You customize the shared library for the task you want to perform, rebuild the library, and substitute it for the library supplied with the product.

Regardless of the format, each exit has an entry point whose name is recognized at run time. For example, the entry point for the client authentication exit is called `AuthentLibMain`.

This topic assumes knowledge of the C programming language. References to "dna" in the product (typically in file names and command prefixes) are to the abbreviation of its former name, Dynamic Network Architecture.

Communication Exits in Java

AppBuilder has several communications exits that can be used in Java clients. There is a Java class in the `<AppBuilder>/JAVA/RT` directory. It is a sample Java class to be used for implementing communication exits for Java clients. A listing is given in [Sample Java Class for Exits](#).

To implement a communications exit for a Java client, you must modify a special Java class, compile it with the Java compiler, and place the resulting Java class file in the Java classpath. In addition, you must modify the `appbuildercom.ini` file to indicate the name of the Java class which contains the exit logic. These steps are explained in detail in [Implementing Communications Exits](#).

The Java class that provides the exit logic must extend the `NeteClientExit` class. The class below is a sample Java class that does just this. This file contains the definition of the `Login` class, which provides logic for the following communications exits in the indicated methods:

- Authentication – `getLoginInfo()`
- Authorization – `isAuthorised()`
- Encryption (password only) – `encryptPasswd()`
- Decryption (password only) – `decryptPasswd()`
- RPC End – `RpcEndStatus()`
- Encoding/Compressing Data – `encodeData()`
- Decoding Data – `decodeData()`

You can rename this file. However, if you rename it, you must also modify the name of the Java class it contains. For example, if you change the name of the file to "UserExitExample.java", you must change the name of the Java class to `UserExitExample`.

For more information about authentication exits, refer to:

- [Authentication Overview](#)
- [Authentication Settings](#)

Implementing Communications Exits

Assuming that you have renamed the file and the Java class as indicated above, here are the steps required to implement communications exits for Java clients:

1. Modify one or more of the methods in the `UserExitExample` class (listed above) to provide the appropriate exit logic.
2. Compile this file using the Java compiler (`javac`). The easiest way is to open a command window, switch to the directory that contains this file, and run the following:

```
javac UserExitExample.java
```

If there are no errors in the logic you have provided, the Java compiler generates the file `UserExitExample.class` and places it in the current directory. When you execute the `javac` command at a DOS prompt, if there are no errors, it returns to a command prompt with no indication that anything has happened even though the file was compiled.

3. Place the resulting `UserExitExample.class` in a directory that is in the Java classpath. (If you have left `UserExitExample.java` in the `<AppBuilder>/JAVA/RT` directory, it is already in the classpath, since that directory is automatically added to the classpath by the AppBuilder installer.)
4. Finally, modify the GENERAL section of `appbuildercom.ini` (located by default in `<AppBuilder>/JAVA/RT`) to specify the Java class names for the exits. In our example you would specify:

[GENERAL]
USER_EXITS=UserExitExample

Authentication Overview

AppBuilder currently allows user authentication with the following:

QUERY_AUTHENTICATION_ON_STARTUP – a setting in the NC section of APPBUILDER.INI. This setting, when set to TRUE, displays a dialog at startup requiring the user to enter an identifier and password.

QueryUserAuthentication – an ObjectSpeak method on the Rule that displays a logon dialog for the user identifier and password.

SetUserAuthentication – an ObjectSpeak method on the Rule to set the user identifier and password programmatically.

NetEssential Exit – An external exit program is invoked during the remote rule call, and this program can return a vector of triplets (target system, user identifier, and password). If a remote rule call is made to a workstation defined in this vector, the given triplet is used. If the target workstation is not found in the vector, AppBuilder uses a user identifier and password with a null workstation from the vector if given. If no such triplet is defined, it uses the user identifier and password given through one of the above methods. When there is no valid triplet and no input from the Rule, then a null user identifier and password is sent to the remote rule.

Default Authentication

With default authentication, use the ObjectSpeak methods *setUserAuthentication* and *queryUserAuthentication*. The *setUserAuthentication* exit applies to the entire Java platform for a particular session, the entire session, with a system level call. The *queryUserAuthentication* exit brings up a dialog, so it is only supported on the Java client. The values can be null (blank). This results in an absence of authentication, and thus no authentication information (user ID and password) is sent to the remote server. Refer to the section on the Rule objects in *ObjectSpeak Reference Guide* for more information about these methods.

Authentication Exit

Call an authentication exit to get a vector of LoginInfo objects, each containing a ServerName (the server name), UserID (the user identifier), and Password (a password). Call a Java class and provide a vector with these three. Each time a remote rule is called, AppBuilder calls the user exit specified in the appbuildercom.ini file to retrieve this list. From this list, and the specified default authentication it selects the authentication to use for a specific server. This call applies to all sessions. (There is no way to specify a particular session.) The user exit Java class is specified in the appbuildercom.ini file.

AppBuilder handles authentication by first determining on which server a particular rule is running. Looking in the ROUTE section of the appbuildercom.ini file, it can find out the name of the server. Then it looks for that server name in the SERVER section to see how authentication is set (with user ID and password). It looks for specific authentication in the vector if there is a match of the server name. If not set, it uses the Default Authentication (above). If not in the list, it sets it to blank, or no authentication. The values can be null (blank). This results in an absence of authentication, and thus no security. For the log-in vector, it is possible to have the server name with a user ID and password set to null.

Implementation Differences

There are several configurations possible in Java. These include thick Java clients running with EJBs and thin HTML clients running with servlets. The method that you specify authentication is different for each.

For Thick Client (EJBs)

For Enterprise Java Beans (EJBs) in multi-tier systems, where a client calls a server that calls a higher server, use the communications exit to specify authentication. Specify the servers by name and the user ID and password for each server. Once the system finds authentication for the first server, that becomes the default for that server and it goes to the next level server.

For Thin Clients (HTML)

For thin HTML clients and servlets only, use default authentication because you can set user (session) specific security.

Authentication Settings

The AppBuilder INI setting AUTH_TYPE can take the following values:

- [DIALOG Setting](#)
- [EXIT Setting](#)
- [NONE Setting](#)

There are three mechanisms for setting up authentication in developing Java applications. The first is default authentication using Java client (ObjectSpeak) methods. The second is the authentication exit in the internal communications of AppBuilder. The third is no authentication.

DIALOG Setting

When authentication is set to DIALOG, if a QueryUserAuthentication is invoked, it prompts for a logon dialog and sets a triplet (server identifier, user identifier, and password) in the root context for the application. This will be used in all remote rule calls. The server identifier can be null.

The same triplet also can be set through the SetUserAuthentication.

If both setUserAuthentication and QueryUserAuthentication are not invoked until the remote rule call, instead of invoking the NetEssential Exit, a logon dialog is displayed to enter the same triplet. Moreover, this vector will be cached for reuse on all subsequent remote rule calls. This vector can be refreshed with an explicit QueryUserAuthentication or SetUserAuthentication calls.

DIALOG could be used on the Java client to prompt for the user identifier and password. For thin-clients or gateways, DIALOG is not recommended because you would not want a dialog to appear on the server. However, if it is set to DIALOG in those environments, AppBuilder prompts a dialog to get the user identifier and password.

EXIT Setting

If AUTH_TYPE is set to EXIT, QueryUserAuthentication invokes the NetEssential Exit. The exit can return a vector of triplets that are saved with the Root context for the application, and all the remote rule calls will use the appropriate triplet. The SetUserAuthentication method can be invoked at any time to reset the existing (null, user identifier, password) triplet.

If if the user authentication has not taken place prior to the remote rule call, the NetEssential Exit is invoked to get the vector of triplets. The vector is cached and reused on all subsequent remote rule calls. The vector can be changed with an explicit QueryUserAuthentication or SetUserAuthentication from the Rule. This is the default setting.

If there is a requirement to use a combination of Dialog and Exit, set the AUTH_TYPE to EXIT. Within the exit program, a custom Java dialog class or the following AppBuilder Java classes could be used to get the user identifier and password, as shown in the following example:

Example

```
public class HpsLoginDialog extends JDialog
{
    /** a constructor */
    public HpsLoginDialog(JFrame userFrame);
    /** method displays a logon dialog and returns the user
        identifier and password entered or null if canceled. */
    public LoginInfo showDialog();
}

public class LoginInfo implements Serializable
{
    /** a method to get the login name as string */
    public String getUserId();
    /** a method to get the password as string */
    public String getPassword();
}
```

NONE Setting

You can specify NONE for quick Development/Testing of the application with no authentication or a null triplet (null, null, null). A SetUserAuthentication can still be invoked to set a (null, user identifier, and password) triplet when AUTH_TYPE is set to NONE and this will be the triplet used in all remote rule calls.

Sample Java Class for Exits



Information about sample exits for Java clients can be found in AppBuilder/java/rt/UserExitExample.java.

Here is a listing of the sample Java class.

```
import appbuilder.nete.*;
import java.util.*;

public class UserExitExample extends AbfClientExit
```



```

{
    // Authentication exit
protected Vector getLoginInfo()
    {
        // This authentication set within appbuilder rules using
// the default login dialog or the setAuthentication function
// is the default LoginInfo.
//
// This vector will override this setting on a per server basis

        /*
String serverId, userName, passwd;
Vector m_logins = new Vector(2);

        // serverName is specified in the AppBundleCom.ini :
// [SERVER.<server>] section
// as SERVERID
// a null or blank server name indicates the default server

        serverId = "pcio";

        // refers to SERVERID in the [SERVER.nete] section

        userName = "aUser";
        passwd = "somePassword";

        m_logins.add( new AbfLoginInfo( serverId, userName, passwd ) );

        serverId = null; // for all other systems
userName = "anotherId";
        passwd = "somePassword";

        m_logins.add( new AbfLoginInfo( serverId, userName, passwd ) );
return m_logins;
        */
// return reference to vector containing one or more login
// information structures

        return null;
    }

    // Authorisation exit
protected boolean isAuthorised( AbfAuthoriseArg arg )
    {
        // SAMPLE CODE
// if ( (arg.getStatus().gettargetMachine().compareTo( "nysvr" ) == 0 ) &&
// ( arg.getStatus().gettargetServerId().compareTo( "nete" ) == 0 ) )
// {
//     return true;
// }
// return true if authorized, false otherwise

        return true;
    }

    // Encryption exit
protected String encryptPasswd( String arg )
    {
        // SAMPLE CODE
// String unencryptedPassword = arg
// String encryptedPassword = // do encryption here
// return encryptedPassword;
// return encrypted password

        return arg;
    }

protected String decryptPasswd( String arg )
    {
        // SAMPLE CODE

```

```

// String encryptedPassword = arg;
// String encryptedPassword = // do encryption here
// return decryptedPassword

    return arg;
}

// RPC end exit
protected void RpcEndStatus( AbfRpcEndExitArg arg )
{
    // provide logic here
}

// data encryption exit
protected AbfDataBuffer encodeData( AbfDataBuffer inBuf )
{
    // SAMPLE code - begin
//
// create a new output DataBuffer object;
// DataBuffer is just a wrapper class for the byte buffer
// with start index and the buffer length.
//
// public void AbfDataBuffer( ) - a default constructor
// public void AbfDataBuffer( int len ) - a new byte buffer is allocated for the given length
// public void init( byte[] buf ) - uses the given buffer,
// start=0 and length=buf.length
// public byte[] getBuffer() - returns the current buffer
// public int getStart() - returns the start index
// public int getLength() - returns the buffer length
//
// byte[] encodedBuf = new byte[32760];
// invoke the encode method with inBuf info, start index should be
// used because the buffer might have data
// which should not be encoded.
// int len = encodeMethod( inBuf.getBuffer(), inBuf.getStart(),
// inBuf.getLength(), encodedBuf );
//
// DataBuffer outBuf = new DataBuffer( );
// outBuf.init( encodedBuf, 0, len );
//
// return outBuf;
//
// SAMPLE CODE - end

    return null;
}

// Data decryption exit
protected AbfDataBuffer decodeData( AbfDataBuffer inBuf )
{
    // SAMPLE code - begin
//
// create a new output DataBuffer object; DataBuffer is just a wrapper
// class for the
// byte buffer with start index and the buffer length
// Following are the public methods in DataBuffer...
//
// public void AbfDataBuffer( ) - a default constructor
// public void AbfDataBuffer( int len ) - a new byte buffer is
// allocated for the given length
// public void init( byte[] buf ) - uses the given buffer,
// start=0 and length=buf.length
// public byte[] getBuffer() - returns the current buffer
// public int getStart() - returns the start index
// public int getLength() - returns the buffer length
//
// byte[] decodedBuf = new byte[32760];
// invoke the decode method with inBuf info
// int len = decodeMethod( inBuf.getBuffer(), inBuf.getStart(),
// inBuf.getLength(), decodedBuf );

```

```
//  
// ByteBuffer outBuf = new ByteBuffer( );  
// outBuf.init( decodedBuf, 0, len );  
//  
// return outBuf;  
//  
// SAMPLE CODE - end  
  
    return null;
```

```
}  
}
```

C Client and Server Exits

Secure distributed applications in C Language use three related mechanisms to protect against unauthorized access to data and services:

Authentication establishes the identity of application users. It ensures that users are who they say they are by requiring a password as proof of identity.

Authorization establishes the permissions of users. It ensures that users can run only the application services and controls they are authorized to run.

Encryption protects application data from being accessed outside the program. It insures that even if intruders were to break into the network, they cannot read the data and login information the application transmits.

Each of these mechanisms is widely available in distributed computing. AppBuilder provides an interface to a series of program exits in an intelligent and integrated way so that you can transmit security information to the security subsystems in use at your site. These topics of authentication, authorization, and encryption describe how you can use the communications exits to secure a distributed application.

Authentication

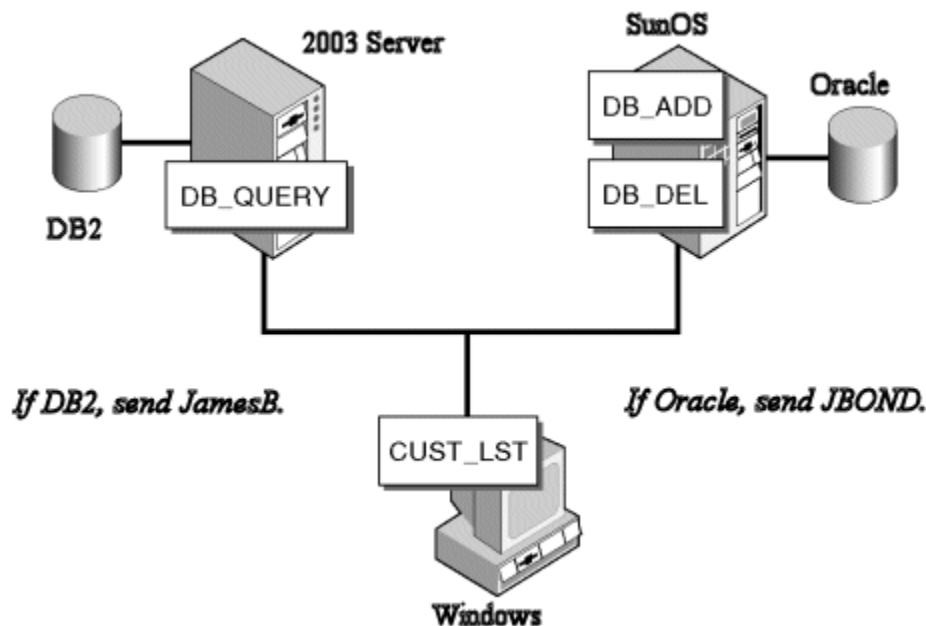
Authentication tests the identities of application users. It answers the question, "Are you who you say you are?" by requiring you to produce a password as proof of the identity you claim. The problem and its solution are familiar ones to computer users. What makes the problem troublesome in distributed computing is the existence of multiple systems for which user identities must be established. The problem is compounded by directory services mechanisms that require the security system to make decisions as to which identity it should use. AppBuilder provides the solution to these problems.

Consider the distributed application shown in the following figure. A Windows client program, CUST_LST, calls the:

- Service DB_QUERY on a Windows 2003 Server machine running a DB2 database
- Services DB_ADD and DB_DEL on a Sun operating system machine running an Oracle database

The user of the program is known to the application and the DB2 database as JAMESB. The user is known to the Oracle database as JBOND.

Handling multiple security identifiers



This configuration is typical of directory services load-balancing schemes and poses no issues for security so long as the user manually signs on to the remote databases. However, manual sign-on can be tedious, and requiring that users remember multiple login IDs and passwords can result in errors and user downtime. This procedure is also expensive because a connection is required for each sign on.

The following example demonstrates how this application could satisfy its security needs more efficiently:

1. Elicit JAMESB's login ID in the client program.
2. Transmit JAMESB's login ID to DB2 for DB_QUERY requests.
3. Transmit JBOND's login ID to Oracle for DB_ADD or DB_DEL requests.

The first problem is how the application obtains JBOND's login ID. The second problem is how the application decides whether to transmit JAMESB or JBOND. A similar problem arises when the user logs on to a gateway as JAMESB and on to the server to which requests are forwarded as JBOND.

Authentication Exit

A communications *authentication exit* on the client side maps user IDs to protocol-specific host destinations. It lists login IDs, passwords, and destination mappings, like these for our sample application:

```
JAMESB 007SPY NT_1
JBOND  GOLDENI SUN_1
```

You typically read the mappings from a file or from another program. Your application continues to elicit JAMESB's login information at startup, to test whether the user should be given access to the application.

An authentication exit is capable of returning multiple user ID and password combinations for use by AppBuilder when communicating with different systems. The first array entry is a default. The 'login_struct' pointer returned to AppBuilder by the exit should point to an array of 'dna_LoginStruct' structures. AppBuilder does not pre-determine how many entries there are in this array. Instead, it requires that the last entry is flagged by each of the three pointers in that entry being set to NULL. For the remainder of the entries, each contains a system name, a user ID, and a password.

When AppBuilder needs to communicate with another system, it scans the entries looking for a match for the system name. (Note, it does a case-sensitive search.) If found, the user ID and password from that entry is used when communicating with that system. If no match for the system name is found, the user ID and password from the first entry is used as a default, regardless of its system name. Hence, for most authentication exit implementations, only two array entries are usually returned: the 'default' entry and the end-flag entry.

AppBuilder also connects to the database on behalf of an AppBuilder application. From the array returned by the authentication exit, it has the user ID and password needed to perform the connection. It uses the value of WORKSTATION_ID from the AE Runtime section of the system initialization file (hps.ini) as the 'system'. In the same way that it looks for a user ID and password for a remote system, if no matching system name for the workstation ID is found, the first entry (the default) is used. For most implementations of authentication exits, this is the user's mainframe user ID and password.

So, if different user IDs are required for mainframe and local databases, the authentication exit should return an array as follows:

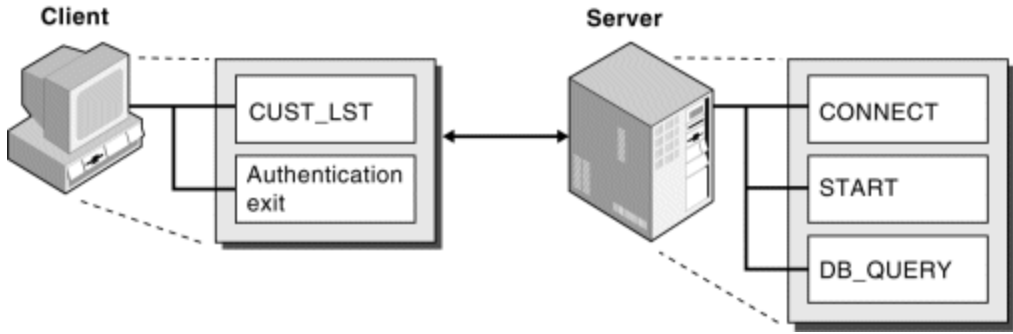
(null-string)	mf-userid	mf-password
workstation id	db-userid	db-password
NULL	NULL	NULL

Because the first entry is used as a default entry, its "system" is irrelevant and is usually specified as a null-string.

At client initialization, AppBuilder detects the presence of the exit and reads the login information/destination mappings into memory. When the client program makes a service request, AppBuilder compares the destination of the request with the destinations received from the exit and determines the correct user identity to transmit. When the server connection is opened, AppBuilder automatically passes the login information to:

- A server-side DBMS, via the database connection exit described in [Customized Data Types in C](#)
- A third-party security subsystem, via LU2 or LU6.2
- A communications server authentication exit, if you do further processing on the server side (see [Client- and Server-side Authorization and Encryption](#))

Passing authentication information to a DBMS



A security ticket is a virtual ID and password for an external security system. If you use a security ticket instead of a user name and password to authenticate a user, you can code a client-side authentication exit that retrieves the ticket and passes it to each client and server exit that needs it.

Authorization

Authorization establishes your permissions as the user with the identity given at the time of authentication. It ensures that you can execute only the application services you are authorized to execute. Following the example from the [Authentication](#) section, JAMESB might be permitted to create and query accounts but not delete them, in which case you would give him authorization to execute DB_ADD and DB_QUERY but not DB_DEL.

Authorization Exit

A communications **authorization exit** on the client or server side maps user identities to application services:

```
JAMESB DB_ADD
JAMESB DB_QUERY
JBOND DB_ADD
JBOND DB_QUERY
```

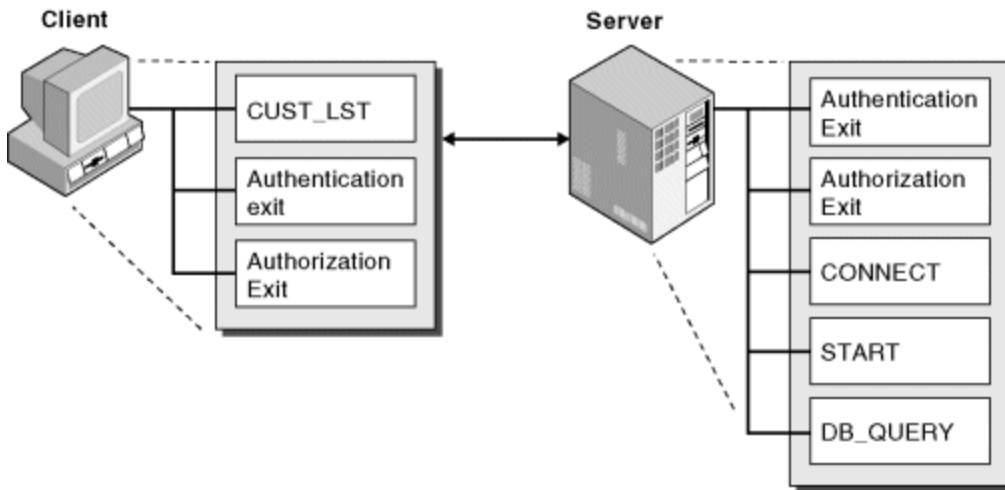
Again, you read the mappings from a file or from another program. When the client program makes a service request, AppBuilder determines the user identity sent with the request based on the information in the client or server authentication exit. The authorization exit receives the user information and compares the names of the authorized services for that user with the name of the requested service to determine whether the request should be fulfilled.

Client- and Server-side Authorization

Authorization checks are typically performed on the server, after a service request is transmitted. You can avoid the expense of opening unnecessary server connections by having authorization performed on the client, before the service request is sent. Client-side authorization is especially useful when you are using a gateway to forward service requests to another server.

The justification for server-side authorization is because client-side authorization is intrinsically less secure. An initialization file variable determines whether AppBuilder detects the client authorization exit in the first place. Unless you protect against users tampering with the initialization file - by putting it on a protected network file server, for example - a user can gain access to every service on the server simply by manipulating the variable.

Client- and server-side authorization



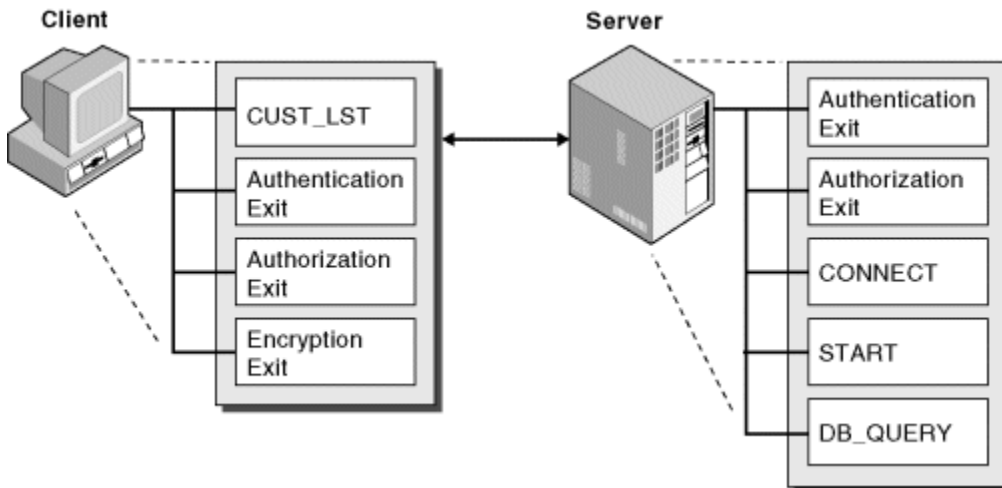
Constraint-based validation is also available. You can make a user's authorization to access a requested service conditional on the input data being passed with the request. You could grant the user authorization to update records only within a specified range. If a request is made to update a record outside that range, access to the service is denied.

Encryption

It is not enough to protect against unauthorized access to data by users of your application if a network intruder outside your application can read the format of the data and the user login information you are transmitting. Encryption ensures that even if intruders were to break into your network, they could not read the data and user information your application transmits.

A communications **encryption exit** on the client side encrypts the user's password; a server-side authentication exit decrypts the password. The following figure shows the encryption exit mechanism.

Encryption exit



You can use the [Open Data Encryption Mechanism](#) to encrypt application data. The data encryption mechanism is open in the sense that you choose the encoding and decoding algorithms that best suit your needs.

Use **trace exists** (also called RPC-end exits) on the client or server side to log service usage or take some other action based on the status of an ending RPC.

Open Data Encryption Mechanism

For data encryption, you build your own encryption library and customize exported entry points into the library. The library is named:

- dnaenc.dll for PC platforms
- dnaenc.sl for UNIX platforms

An empty version of this library ships with AppBuilder. Templates for the data encryption mechanism are provided with the product in the SAMPLES subdirectory. Once you have created your version of the library, replace the empty version with the new version.



You can also use the data encryption mechanism to compress data.

The entry points into the library are as follows:

- [dna_Buflnit](#)
- [dna_BufEncode](#)
- [dna_BufDecode](#)
- [dna_BufClose](#)
- [dna_BufTcpMVSDecode](#) (for TCP/IP on the mainframe only)

Make sure to include the header file dnaenc.h. The following structure in dnaenc.h is used for encoding and decoding data:

```
typedef struct
{
    unsigned char *data;
    unsigned long buffer_length;
} dna_UserBuf_t;
```

Return codes in dnaenc.h are as follows:

```
#define DNAENC_ERROR          -1
#define DNAENC_SUCCESS        0
#define DNAENC_NO_CUSTOMIZE  0
#define DNAENC_CUSTOMIZE     1
```

The data encryption mechanism is not supported on LU2. If you use the mechanism on LU2, only the initialization and close functions are called.

dna_Buflnit

Purpose

Initialize data encryption. Call this function on a client when a connection to a new server is about to be established or on a server when a new connection is being initiated by a client.

Prototype

```
#include <dnaenc.h>
int DNA_ENTRY dna_Buflnit(void** user_context, int flags, dna_Binding_t* binding);
```

Formal Arguments

Formal arguments

Name	Description
user_context	Address of a void pointer to a user-defined structure. The routine is expected to create a context that will be passed to subsequent routines - for example, it can keep the address of memory it allocates to encode or decode data and/or a key used to encrypt or decrypt the data.
flags	Identifies the calling program. The value 1 means that a client is invoking the routine; 0 means that a server is invoking the routine.
binding	For clients, this is a pointer to a structure that contains service binding information. You can use this argument to select an encryption key.

Return Values

An integer equal to DNAENC_CUSTOMIZE if data encoding should take place and DNAENC_NO_CUSTOMIZE if data encoding/decoding should not take place, or DNAENC_ERROR.

dna_BufEncode

Purpose

Encode (encrypt or compress) the transmitted data. You can encode data transmitted from a client to a server or from a server to a client.

Prototype

```
#include <dnaenc.h>
int DNA_ENTRY dna_BufEncode(void* user_context, dna_UserBuf_t* in_buffer, dna_UserBuf_t* enc_buffer);
```

Formal Arguments

Formal arguments

Name	Description
user_context	Pointer to a user-defined structure, as returned to dna_BufInit.
in_buffer	Pointer to the buffer for the input data. The routine encodes the data in this buffer and puts the results in enc_buffer.
enc_buffer	Pointer to the buffer for the encoded data. The routine fills in this structure with a pointer to the encoded data and the encoded data length.



You are responsible for allocating and freeing the buffer for the encoded data. Set enc_buffer->data to the allocated buffer and enc_buffer->buffer_length to the encoded data size.

Return Values

DNAENC_ERROR
DNAENC_SUCCESS

dna_BufDecode

Purpose

Decode (decrypt or decompress) the transmitted data. You can decode data transmitted from a client to a server, or from a server to a client.

Prototype

```
#include <dnaenc.h>
int DNA_ENTRY dna_BufDecode(void* user_context, dna_UserBuf_t* enc_buffer, dna_UserBuf_t* dec_buffer);
```

Formal Arguments

Formal arguments

Name	Description
user_context	Pointer to a user-defined structure, as returned to dna_BufInit.
enc_buffer	Pointer to the buffer for the encoded data. The routine decodes the data in this buffer and puts the results in dec_buffer.
dec_buffer	Pointer to the buffer for the decoded data. The routine fills in this structure with a pointer to the decoded data and the decoded data length.



You are responsible for allocating and freeing the buffer for the decoded data. Set dec_buffer->data to the allocated buffer and dec_buffer->buffer_length to the decoded data size.

Return Values

DNAENC_ERROR
DNAENC_SUCCESS

dna_BufClose

Purpose

Clean up after the connection to a peer is broken down and encoding/decoding is complete.

Prototype

```
#include <dnaenc.h>
int DNA_ENTRY dna_BufClose(void* user_context);
```

Formal Arguments

Formal arguments

Name	Description
user_context	Pointer to a user-defined structure, as returned to dna_BufInit. The user-defined data can reference memory to be freed.

Return Values

DNAENC_ERROR
DNAENC_SUCCESS

dna_BufTcpMVSDecode

Purpose

Decode (decrypt or decompress) part of the transmitted (header) data. This is applicable only to TCP/IP on the mainframe because the NetEssential Listener requires this information.

The size of the encoded buffer passed from the Listener to the exit is determined by a DNAINI setting in the DNA_SERVER section: Set TCP_IP_PEEKBUFSIZE to the desired value. The default, and minimum, value is 144.

Prototype

```
#include <dnaenc.h>
int DNA_ENTRY dna_BufTcpMVSDecode(void* user_context, dna_UserBuf_t *enc_buffer, dna_UserBuf_t*
dec_buffer);
```

Formal Arguments

Formal arguments

Name	Description
user_context	Pointer to a user-defined structure, as returned to dna_BufInit.
enc_buffer	Pointer to the buffer for the encoded data. The routine decodes the data in this buffer and puts the results in dec_buffer.
dec_buffer	Pointer to the buffer for the decoded data. The routine fills in this structure with a pointer to the decoded data and the decoded data length.



You are responsible for allocating and freeing the buffer for the decoded data. Set dec_buffer->data to the allocated buffer and dec_buffer->buffer_length to the decoded data size.

Return Values

DNAENC_ERROR
DNAENC_SUCCESS

Setting Up Security Exits

Templates for the security exits are provided with AppBuilder in the SAMPLES subdirectory. Customize the templates to obtain the authentication and authorization information required by the security setup at your site. The data encryption mechanism requires you to build your own encryption library and customize exported entry points into the library, as described in [Open Data Encryption Mechanism](#).

You can use the security exits in any combination. You typically use the client authentication exit to elicit login information for the remaining exits, but you can also modify login information in the client authorization or server authentication exits. To see a list of the entry points, refer to [Open Data Encryption Mechanism](#).

A typical security and tracing setup looks like this:

1. An authentication routine on the client machine elicits an array of security entries. Each item holds the name of an available remote system and the user login ID and password for that system. The entry point is named AuthentLibMain.

For security systems that use a ticket, the authentication routine transmits the ID and password to the security system. On receipt of the ID and password, the ticket is returned.

2. An authorization routine on the client machine tests the user's authorization to access a remote service. In some cases, the routine uses the security ticket to test the user's authorization. The entry point is named AuthorLibMain.
3. An encryption routine on the client machine encrypts the user password for network transmission. The entry point is named EncryptLibMain.
4. An authentication routine on the server machine decrypts the user password for the security subsystem on that machine and checks the login ID and password for validity. In some cases, the routine uses the security ticket to test the user's authenticity. The entry point is named SrvAuthentLibMain.
5. An authorization routine on the server machine tests the user's authorization to access a service. In some cases, the routine uses the security ticket to test the user's authorization. The entry point is named SrvAuthorLibMain.
6. A trace routine on the client or server machine logs service usage. The entry point is named RpcEndLibMain.

Write the routines in C and compile and link them in a load module for mainframe, a DLL otherwise for the operating systems in use at your site. Make sure to specify the linked module's path and filename in the following variables in the DNA_EXITS section of the dna.ini file on the client or server machine. An empty variable (the default) means that the referenced exit is not invoked.

- DNA_AUTHENT_EXIT points to the client authentication exit
- DNA_AUTHOR_EXIT points to the client authorization exit
- DNA_ENCRYPT_EXIT points to the encryption exit
- DNA_RPC_END_EXIT points to the trace exit
- SRV_AUTHENT_EXIT points to the server authentication exit
- SRV_AUTHOR_EXIT points to the server authorization exit

You can use the custom configuration tools on PC and UNIX hosts to point to the locations of the exits. For more information on configuring the security exits, see [CICS TCP/IP Listener Security Settings](#).

Linking Security Exits on the Mainframe

In the mainframe runtime for C applications (C/C++), the exits must be statically linked into the HPSDNA00. Instead of specifying the path of the exits in the security exit variables in dna.ini file, you must specify the value STATIC_LINK.

In the mainframe runtime (Language Environment (LE)), the exits are dynamically linked. Specify the paths of the exits as described above. The CICS TCP/IP Listener, DNALSTNR, can optionally also invoke the Authentication Exit. If this is required, DNALSTNR must be statically linked with the exit.

DNAINI Settings - In addition to setting SRV_AUTHENT_EXIT=STATIC_LINK in the DNA_EXITS section, set the LISTENER_SECURITY= in the MVS_LISTENER section to either EXIT or EXITONLY.

Messaging and Eventing in Security-enabled Environments

If the parent of a node is a security-enabled mainframe listening over LU6.2, the messaging and eventing agent at the node must point to a client-side authentication exit. The authentication exit passes the user name and password to LU6.2. There is no need to code a server-side exit.

The custom configurators for the agent prompt you to type the full pathname of the exit. If you use the child node for RPCs as well as messaging and eventing, type the pathname you specified for the RPC client.

Security Exits Reference for C

Security entry points include:

- [AuthentLibMain](#)
- [AuthorLibMain](#)
- [EncryptLibMain](#)
- [SrvAuthentLibMain](#)
- [SrvAuthorLibMain](#)
- [RpcEndLibMain](#)

AuthentLibMain

Purpose


Elicit an array of security entries for network transmission. Each item holds the name of an available remote system and the user login ID and password for that system. For details on authentication, see [Authentication](#).

Prototype

```
#include <dna.h>
int AuthentLibMain(Authent_LibFuncs_t func_type, void* authent_args_ptr);
```

Formal Arguments

Formal arguments

Name	Description
func_type	<p>Structure of type <code>Authent_LibFuncs_t</code> that identifies the functions called from the entry point:</p> <pre>typedef enum { GET_LOGIN_CALL, FREE_LOGIN } Authent_LibFuncs_t;</pre>
authent_args_ptr	<p>Pointer to a structure of type <code>dna_LoginArgs_t</code>:</p> <pre>typedef struct { dna_LoginStruct** login_struct; dna_ClnId_ptr_t client_id; dna_SecCookie_t** sec_ticket; } dna_LoginArgs_t;</pre> <p>In <code>dna_loginArgs_t</code>:</p> <ul style="list-style-type: none"> • <code>dna_LoginStruct</code> holds security data: the target system (which has a maximum size of 32 bytes) and the user ID and password for the target system (each of which has a maximum size of 15 bytes). • <code>dna_ClnId_ptr_t</code> identifies the client ID passed to <code>dna_InitClient</code>. • For security systems that use a ticket (see the topic box), <code>dna_SecCookie_t</code> holds the ticket and its data description. <div style="border: 1px solid black; background-color: #ffffcc; padding: 5px; margin-top: 10px;"> <p> Allocate one more structure than is required to hold security data and set the pointers in the additional structure to NULL.</p> </div>

Return Values

LOGON_SUCCESS
LOGON_FAILURE

AuthorLibMain

Purpose


Test the user's authorization to access a remote service. For details on authorization, see [Authorization](#).

Prototype

```
#include <dna.h>
int AuthorLibMain(Author_LibFuncs_t func_type, void* author_args_ptr);
```

Formal Arguments

Formal arguments

Name	Description
func_type	<p>Structure of type Author_LibFuncs_t that identifies the function called from the entry point:</p> <pre>typedef enum { CHECK_AUTHOR_CALL } Author_LibFuncs_t;</pre>
author_args_ptr	<p>Pointer to a structure of type dna_ChkAuthorArgs_t:</p> <pre>typedef struct { dna_LoginStruct* login_struct; dna_StatusView* status_view; void* input_view; dna_ClnId_ptr_t client_id; dna_SecCookie_t* sec_ticket; Data_Info_t* input_ctrl; } dna_ChkAuthorArgs_t;</pre> <p>In dna_ChkAuthorArgs_t:</p> <ul style="list-style-type: none">• dna_LoginStruct holds security data elicited by the client authentication exit (see AuthentLibMain)• dna_StatusView holds the status view• input_view is a void pointer to the input view of a remote service request• dna_ClnId_ptr_t identifies the client ID passed to dna_InitClient• For security systems that use a ticket (see the topic box), dna_SecCookie_t holds the ticket and data description elicited by the client authentication exit• Data_Info_t holds the input data description <div style="border: 1px solid red; padding: 5px; margin-top: 10px;"> Do not change the status view or input data. You can change the security data, but this action is not typical.</div>

Return Values

AUTHOR_SUCCESS
AUTHOR_FAILURE

EncryptLibMain

Purpose

Encrypt the user password for network transmission. The routine should encrypt in place and encrypt to valid characters only (not binary). For more information, see [Encryption](#).

Prototype

```
#include <dna.h>
int EncryptLibMain(Encrypt_LibFuncs_t func_type, void* encrypt_args_ptr);
```

Formal Arguments

Formal arguments

Name	Description
func_type	Structure of type Encrypt_LibFuncs_t that identifies the function called from the entry point: <div style="border: 1px dashed blue; padding: 10px; margin: 10px 0;"> <pre>typedef enum { ENC_PASSWD_CALL } Encrypt_LibFuncs_t;</pre> </div>
encrypt_args_ptr	Pointer to a structure of type dna_EncPasswdArgs_t that points, in turn, to the address of an area that holds the password elicited by the client authentication exit.

Return Values

ENCPASSWD_SUCCESS
ENCPASSWD_FAILURE

SrvAuthentLibMain

Purpose

Decrypt the user password for the security subsystem on the server machine and check the user login ID and password for validity. For more information, see [Authentication](#).


Prototype

```
#include <dna.h>
int SrvAuthentLibMain(srv_Authent_Funcs_t func_type, void* authent_args_ptr);
```

Formal Arguments

Formal arguments

Name	Description
func_type	Structure of type srv_Authent_Funcs_t that identifies the function called from the entry point: <div style="border: 1px dashed blue; padding: 10px; margin: 10px 0;"> <pre>typedef enum { SRV_CHK_USER_CALL } srv_Authent_Funcs_t;</pre> </div>

authnt_args_ptr	<p>Pointer to a structure of type <code>srv_ChkUserArgs_t</code>:</p> <pre style="border: 1px dashed blue; padding: 10px;">typedef struct { srv_ChkUserStruct* userinfo; } srv_ChkUserArgs_t;</pre> <p><code>srv_ChkUserStruct</code> holds:</p> <ul style="list-style-type: none"> • The user ID and password elicited by the client authentication exit • <code>dna_SecCookie_t</code>. For security systems that use a ticket (see the topic box), <code>dna_SecCookie_t</code> holds the ticket and data description elicited by the client authentication exit. <div style="border: 1px solid yellow; padding: 5px; margin-top: 10px;">  AppBuilder retains changes that the routine makes to the password but not to the user ID. If the server is a gateway or calls other remote services, the changed password will be used. </div>
-----------------	---

Return Values

SRV_CHKUSER_SUCCESS
SRV_CHKUSER_FAILURE

SrvAuthorLibMain

Purpose

Test the user's authorization to access a service. For more information, see [Authorization](#).

Prototype

```
#include <dna.h>
int SrvAuthorLibMain(srv_Author_Funcs_t func_type, void* author_args_ptr);
```

Formal Arguments

Formal arguments

Name	Description
func_type	<p>Structure of type <code>srv_Author_Funcs_t</code> that identifies the function called from the entry point:</p> <pre style="border: 1px dashed blue; padding: 10px;">typedef enum { SRV_CHK_SERV_CALL } srv_Author_Funcs_t;</pre>

authent_args_ptr	<p>Pointer to a structure of type <code>srv_ChkServArgs_t</code>:</p> <div style="border: 1px dashed blue; padding: 10px; margin: 10px 0;"> <pre>typedef struct { srv_ChkServStruct* servinfo; void* input_view; void* hps_cntxt; Data_Info_t* input_ctrl; } srv_ChkServArgs_t;</pre> </div> <p>In <code>srv_ChkServArgs_t</code>: <code>srv_ChkServStruct</code> holds:</p> <ul style="list-style-type: none"> • The user ID and password elicited by the client authentication exit. • The name of the requested service. • The client ID passed to <code>dna_InitClient</code>. • <code>dna_SecCookie_t</code>. For security systems that use a ticket, <code>dna_SecCookie_t</code> holds the ticket and data description elicited by the client authentication exit. • <code>input_view</code> is a void pointer to the input view of a remote service request. • <code>hps_cntxt</code> is a void pointer to the mainframe context. Irrelevant. • <code>Data_Info_t</code> holds the input data description. <div style="border: 1px solid red; background-color: #ffe6e6; padding: 5px; margin-top: 10px; display: flex; align-items: center;"> Do not change the security data or input data. </div>
-------------------------	---

Return Values

SRV_CHKSERV_SUCCESS
SRV_CHKSERV_FAILURE

RpcEndLibMain

Purpose

Log service usage or take some other action based on the status of an ending service request.

Prototype

```
#include <dna.h>
int RpcEndLibMain(RpcEnd_LibFuncs_t func_type, void* rpcend_args_ptr);
```

Formal Arguments

Formal arguments

Name	Description
<code>func_type</code>	<p>Structure of type <code>RpcEnd_LibFuncs_t</code> that identifies the function called from the entry point:</p> <div style="border: 1px dashed blue; padding: 10px; margin: 10px 0;"> <pre>typedef enum { END_EXIT_CALL } RpcEnd_LibFuncs_t;</pre> </div>

rpcend_args_ptr	Pointer to a structure of type dna_EndExitArgs_t: <pre style="border: 1px dashed blue; padding: 10px; margin: 10px 0;">typedef struct { dna_StatusView* status_view; } dna_EndExitArgs_t;</pre> dna_StatusView holds the status view.
-----------------	---

Return Values

DNA_RPC_SUCCESS
DNA_RPC_FAILURE

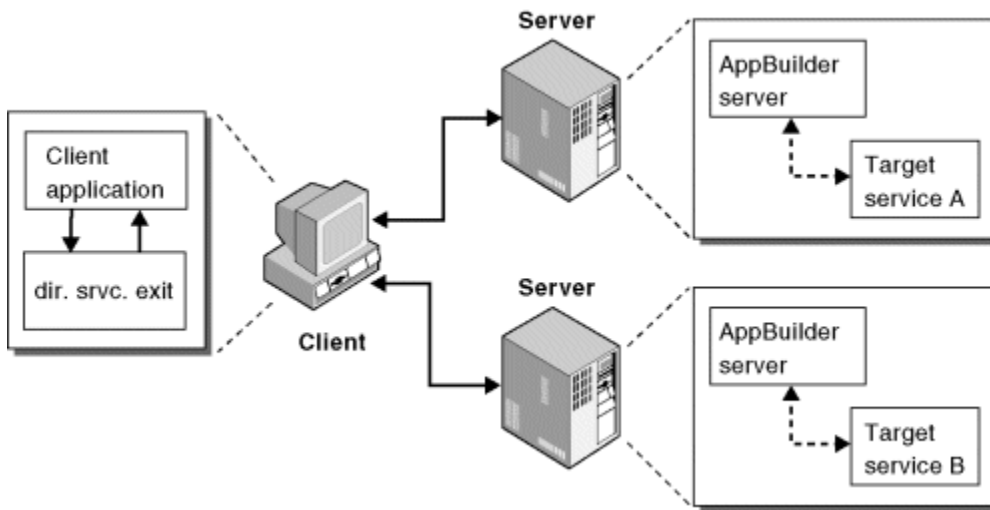
Directory Services Exits in C

In AppBuilder communications, remote procedure calls (RPCs), messaging, global eventing, and implicit eventing all use directory services to determine the target of a communications call:

- RPCs use a route table to identify a requested service
- Messaging uses a route table to identify local and remote queues and a subcell table to identify the routes to the children of the local host
- Global eventing uses a subcell table to identify the routes to the children of the local host
- Implicit eventing uses a subcell table to identify the routes to the children of the local host, a trigger table to identify the triggering service, and an event table to identify the triggered service, message, or event

This topic looks at how you use communications exits to adapt the directory services interface to external directory service schemes. The following figure shows how the directory services exits fit into the processing flow of your application.

Directory services exits



The exits are made available in the form of export functions in the external directory services shared library. You customize the shared library, rebuild it, and substitute it for the library supplied with the product. The library is named libextds, and is already linked to AppBuilder communications when you receive the product. See [Example of Customized Marshalling](#) for information about compilation.

The entry points into the library are as follows:

- [dna_ExtRoutes](#)
- [dna_ExtSubcells](#)
- [dna_ExtTriggers](#)
- [dna_ExtEvents](#)

Make sure to include the header files marshall.h and extds.h. The following structure in extds.h is used for service request routing:

```

typedef struct {
    char service[TBL_SERVICE_LEN];
    char datalen[TBL_RDLEN_LEN];
    char datastart[TBL_RDOFFS_LEN];
    char datatype[TBL_DTYPENAME_LEN];
    char datavalstart[TBL_RANGELEN_LEN];
    char datavalend[TBL_RANGELEN_LEN];
    char host[TBL_HOSTNAME_LEN];
    char protocol[TBL_PROTOCOL_LEN];
    char serverid[TBL_SERVERID_LEN];
    char codepage[TBL_CODEPAGE_LEN];
    char priority[TBL_PRIORITY_LEN];
} route_binding_t;

```

The following structure in extds.h is used for subcell table routing:

```

typedef struct {
    char generic_hostname[TBL_HOSTNAME_LEN]; /* protocol-independent */
    char hostname[TBL_HOSTNAME_LEN]; /* protocol-specific */
    char protocol[TBL_PROTOCOL_LEN];
    char serverid[TBL_SERVERID_LEN];
    char codepage[TBL_CODEPAGE_LEN];
} subcell_binding_t;

```

The following structure in extds.h is used for trigger information:

```

typedef struct event_struct {
    long event_id;
    char event_name[IMPLEVT_LEN];
    char data_len[DATA_DEP_LEN];
    char data_offs[DATA_DEP_OFF];
    char data_type[DATA_DEP_TYP];
    char data_low_limit[DATA_DEP_RANGE];
    char data_high_limit[DATA_DEP_RANGE];
} evtmEvent_t;

```

The following structure in extds.h is used for action information:

```

typedef struct evtmAction_t {
    long event_id;
    char event_name[IMPLEVT_LEN];
    char event_type[IMPLEVT_LEN];
    char event_attribute[IMPLEVT_LEN];
    char subsys[IMPLEVT_LEN];
    char action_name[IMPLEVT_LEN];
    char action_host[IMPLEVT_LEN];
    char locality[VIEW_ATTR_LEN];
    char view_choice[VIEW_ATTR_LEN];
    char view_replace[VIEW_ATTR_LEN];
} evtmAction_t;

```

Return codes in extds.h are as follows:

```

#define EXTDS_SUCCESS 0
#define EXTDS_FAILURE 1

```

If you use external directory services, make sure to set up the configuration file dna.ini as follows:

```

[ROUTING]
NAME_SERVICE = EXTERNAL

[IMPLICIT_EVENTS]
EVENT_SERVICE = EXTERNAL

[SMA]
SMA_SUBCELL_TABLE = EXTERNAL

```



If you use external directory services for any of the features on a host, you must use it for all of them. You cannot use external directory services for subcell table routing, for example, while using communications directory services for RPC routing.

dna_ExtRoutes

Purpose

Identify the route to a service or message queue.

Prototype

```

#include <marshall.h>
#include <extds.h>
int dna_ExtRoutes(char* servicename,
                 Data_Info_t* indit,
                 void* indata,
                 route_binding_t*
                 bind_list,
                 long numroutes,
                 long* num_routes);

```

Formal Arguments

Formal arguments

Name	Description
servicename	String that identifies the name of the service or message queue.
indit	Pointer to the control-structure array for the service request input view. Used for data-dependent routing. Irrelevant for messaging.
indata	Pointer to the service request input view. Used for data-dependent routing. Irrelevant for messaging.
bind_list	Pointer to a preallocated, empty array of size num_routes that you must modify to hold binding information.
numroutes	Number of parallel or alternative routes allowed by AppBuilder communications.
num_routes	Pointer to the number of bindings the exit provides in bind_list.

Return Values

EXTDS_SUCCESS
EXTDS_FAILURE

Example with dna_ExtRoutes

```

int dna_ExtRoutes(char* servicename,
                 Data_Info_t* indit,
                 void* indata,
                 route_binding_t* bind_list,
                 long numroutes,
                 long* num_routes)
{
    route_binding_t* b = bind_list;
    int i;

    printf("\nGet into externalized route service.\n");
    printf("servicename: %s. servicename\n");

    printf("route1: knuth, tcpip, dna_single_tst.\n");
    /* construct one route for "servicename" */
    strcpy(b->service, servicename);
    strcpy(b->datalen, "2");
    strcpy(b->datastart, "input.parm1");
    strcpy(b->datatype, "short");
    strcpy(b->datavalstart, "100");
    strcpy(b->datavalend, "1000");
    strcpy(b->host, "knuth");
    strcpy(b->protocol, "tcpip");
    strcpy(b->serverid, "dna_single_tst");

    b->codepage[0] = '\0';
    strcpy(b->priority, "1");
    b++;

    printf("route2: nasdaq, tcpip, dna_single_tst.\n");
    /* construct 2nd route for "servicename" */
    strcpy(b->service, servicename);
    strcpy(b->datalen, "-");
    strcpy(b->datastart, "-");
    strcpy(b->datatype, "-");
    strcpy(b->datavalstart, "-");
    strcpy(b->datavalend, "-");
    strcpy(b->host, "nasdaq");
    strcpy(b->protocol, "tcpip");
    strcpy(b->serverid, "dna_single_tst");
    b->codepage[0] = '\0';
    strcpy(b->priority, "2");

    *num_routes = 2;

    printf("\nExit sample externalized directory service.\n");

    return EXTDS_SUCCESS;
}

```

dna_ExtSubcells

Purpose

Identify the routes to the children of the local host.

Prototype

```

#include <marshall.h>
#include <extds.h>
int dna_ExtSubcells(subcell_binding_t* binding,
                   long max_bindings,
                   long descend_flag,
                   long* num_subcells);

```

Formal Arguments

Formal arguments

Name	Description
binding	Pointer to a preallocated, empty array of size max_bindings that you must modify to hold binding information.
max_bindings	Maximum number of bindings that can be stored in the binding table.
descend_flag	Identifies whether to route to immediate children only. The value 1 means route to all descendants; 0 means route to immediate children only.
num_subcells	Pointer to the number of bindings the exit provides in the binding argument.

Return Values

EXTDS_SUCCESS
EXTDS_FAILURE

Example with dna_ExtSubcells

```

int dna_ExtSubcells(subcell_binding_t* binding,
                   long* max_bindings,
                   long descend_flag,
                   long* num_subcells)
{
    /* normally get all subordinates binding */
    /* if descend_flag == 1 (messaging), also return descendent bindings */
    return 0; /* num of entries in subcell table */
}

```

dna_ExtTriggers

Purpose

Identify the triggering service.

Prototype

```

#include <marshall.h>
#include <extds.h>
int dna_ExtTriggers(char* service_name,
                   char* trigger_type,
                   evtmEvent_t* event_list,
                   long maxevents,
                   long* num_events);

```

Formal Arguments

H6. Formal arguments

Name	Description
service_name	String that identifies the name of the triggering service.
trigger_type	String that identifies the trigger type. START_RULE means that initiation of the triggering service is postponed until the associated action is complete; END_RULE means that initiation of the action is postponed until the triggering service is complete.
event_list	Pointer to a preallocated, empty array of size max_events that you must modify to hold a list of all events that will be triggered by execution of the triggering service.
max_events	Maximum number of events that can be stored in event_list.
num_events	Pointer to the number of triggered events the exit provides in event_list.

Return Values

EXTDS_SUCCESS
EXTDS_FAILURE

Example with dna_ExtTriggers

```
int dna_ExtTriggers(char* service_name,
                  char* trigger_type,
                  evtmEvent_t* event_list,
                  long maxevents,
                  long* num_events)
{
    printf("\nGet into sample trigger directory service.\n");

    if ((!strcmp(service_name, "ADD")) && (!strcmp(trigger_type, "end_rule")))
    {
        evtmEvent_t* e = event_list;
        printf("trigger EVENT_2\n");
        e->event_id = 2;

        strcpy(e->event_name, "EVENT_2");
        strcpy(e->data_len, "-");
        strcpy(e->data_offs, "-");
        strcpy(e->data_type, "-");
        strcpy(e->data_low_limit, "-");
        strcpy(e->data_high_limit, "-");

        *num_events = 1;
    }
    else
        *num_events = 0;

    printf("\nExit externalized trigger directory service.\n");
    return EXTDS_SUCCESS;
}
```

dna_ExtEvents

Purpose

Identify the triggered service, message, or event. This exit is called each time an event is triggered.

Prototype

```

#include <marshall.h>
#include <extds.h>
int dna_ExtEvents(evtmEvent_t* a_event,
                 evtmAction_t* a_action,
                 long* num_actions);

```

Formal arguments

Formal arguments

Name	Description
a_event	Pointer to the trigger information returned by dna_ExtTriggers . Use this information to identify the event to be triggered.
a_action	Pointer to the action information for the identified event.
num_actions	Address that identifies whether the action information can be used to post an event. The value 1 means yes; 0 means no.

Return values

EXTDS_SUCCESS
EXTDS_FAILURE

Example with dna_ExtEvents

```

int dna_ExtEvents(evtmEvent_t* a_event,
                 evtmAction_t* a_action,
                 long* num_actions)
{
    evtmAction_t* a = a_action;
    printf("\nGet into externalized event directory service.\n");
    a->event_id = 2;

    strcpy(a->event_name, "EVENT_2");
    strcpy(a->event_type, "-");
    strcpy(a->event_attribute, "nontran_sync");
    strcpy(a->subsys, "-");
    strcpy(a->action_name, "IMP_IO_0");
    strcpy(a->action_host, "-");
    strcpy(a->locality, "-");
    strcpy(a->view_choice, "io_0");
    strcpy(a->view_replace, "y");

    *num_actions = 1;

    printf("\nGet into externalized event directory service.\n");
    return EXTDS_SUCCESS;
}

```

Compiling and Linking

This topic looks at how you compile and link directory services exits, database exits, and data marshalling routines for the supported execution platforms with AppBuilder communications. It also describes important usage differences between the command line and graphical user interfaces for the Visual C++ compiler. For security exit compilation in general, see the code samples provided with the product.

- [Using the Visual C++ Compiler](#)
- [Example Makefile for Directory Services Exits](#)
- [Example Makefile for Database Exits](#)

Using the Visual C++ Compiler

The procedures described in the code samples for compiling and linking Windows applications show the required options for the Visual C++

command line compiler and linker. These procedures are written according to Visual C++ 6.0. If you use the graphical user interface for the Visual C++ compiler and linker, you must ensure that the compiler and linker options match exactly those given for command line compiling and linking. You can verify the current option settings by checking them in the IDE. Visual C++ shows the current settings in the IDE whenever a build option is selected.

The following table shows options you should *not* use to compile and link applications in Visual C++.

Visual C++ options to disable

Feature	Default in IDE	Command Line Option
Function-level linking	disabled	/Gy
Minimal rebuild	enabled	/Gm
Incremental compilation	disabled	/Gi
Incremental link	enabled	/incremental:yes

The following table shows options you *should* use to compile and link applications in Visual C++.

Required Visual C++ options

Feature	Default in IDE	Required Option
Runtime DLL	/ML	/MD
Calling convention	[cdecl]	/Gz [stdcall]
Structure packing	/Zp8	/Zp1

To edit the settings in the IDE, follow these steps:

1. Make sure to add the module definition (.DEF) file for each exit to the project with the **/Insert/Files Into Project** command. A sample module definition file appears in the code samples.
2. In the **Project** menu, select the **Settings** menu item.
3. In the **C/C++** tab, select the category **Customize**.
4. In the **Link** tab, select the category **General**.

Example Makefile for Directory Services Exits

This example shows a makefile for the directory services shared library. There are statements for the IBM AIX platform, the Sun Solaris platform, and the Hewlett-Packard HP-UX platform. Adapt the example to the platform on which you are executing the exits.


```

# extds.h, marshall.h, and extds.exp are in nete release
# include directory

AIX_CC=/bin/cc
AIX_LD=/bin/ld
AIX_CFLAG= -qlanglvl=ansi -DRS6000 -DUNIX -DAIX
AIX_IFLAG=-I./ -I../include
AIX_LIB=-lc
AIX_LDFLAG=-bE:extds.exp -bM:SRE
AIX_LIBEXTDS=extds.so
AIX_TGT=libextds_shar

SOLARIS_CC=/opt/SUNWspro/bin/cc
SOLARIS_LD=/opt/SUNWspro/bin/cc
SOLARIS_CFLAG= -Xa -Kpic -DSOLARIS -DSUN -DUNIX -DSVR4
SOLARIS_IFLAG=-I./ -I../include
SOLARIS_LIB=-ldl
SOLARIS_LDFLAG=-G
SOLARIS_LIBEXTDS=libextds.so
SOLARIS_TGT=libextds

HP-UX_CC=/bin/cc
HP-UX_LD=/bin/ld
HP-UX_CFLAG= \-Ac \-Ae \+Z \-DHP_UX \-DUNIX \-DSVR4
HP-UX_IFLAG=-I./ \-I../include
HP-UX_LIB=-lc
HP-UX_LDFLAG=-b
HP-UX_LIBEXTDS=libextds.sl
HP-UX_TGT=libextds

all:
    make -f extds.mk "OS=$(OS)" "CC=$( $(OS)_CC )" "LD=$( $(OS)_LD )"
        "CFLAG=$( $(OS)_CFLAG )" "IFLAG=$( $(OS)_IFLAG )"
        "LIBEXTDS=$( $(OS)_LIBEXTDS )" "LIBS=$( $(OS)_LIB )"
        "LDFLAG=$( $(OS)_LDFLAG )"
        $( $(OS)_TGT)

libextds: extds.o
    $(LD) -o $(LIBEXTDS) extds.o $(LDFLAG) $(LIBS)

extds.o: extds.c extds.h marshall.h
    $(CC) \-c $(CFLAG) $(IFLAG) extds.c

libextds_shar: libextds /bin/ar vr libextds.a $(LIBEXTDS)

install:
clean:

```

Example Makefile for Database Exits

This example shows a makefile for database exits. Adapt the example to the platform on which you are executing the exits. The example DLLs are listed in EXECUTABLES - you can assign them different names, as required. The entry points must be specified exactly as shown.

```

DNADIR = /u/nes/nes30
PROC = proc
PROFLAGS = PARSE=NONE MODE=ANSI IRECLEN=200 ORECLEN=200
SERVICE_CFLAGS = -DUNIX -DTSERVER -D\__SERV__ -D\__CONNECTION__ \
-I$(DNADIR)/include -I.

CFLAGS = -DORACLE $(SERVICE_CFLAGS)
DFFLAGS =
IFLAGS = -I./ -I$(DNADIR)/include
HDRS =
LIBS = -lc
CC = _FULLY_QUALIFIED_COMPILER_NAME $(CFLAGS) $(DFFLAGS) $(IFLAGS)_
LD = _FULLY_QUALIFIED_LINKER_NAME_
EXECUTABLES = apendbg.dll

all: ${EXECUTABLES} clean

sdbi_emp.o: sdbi_emp.c
$(CC) -c sdbi_emp.c

sdbi_ora.c: sdbi_ora.pc
$(PROC) $(PROFLAGS) INAME=sdbi_ora.pc ONAME=sdbi_ora.c

sdbi_ora.o: sdbi_ora.c
$(CC) -c sdbi_ora.c

libsdbi_emp.a: sdbi_emp.o
$(LD) -o libsdbi.a sdbi_emp.o -bE:sdbi.exp -bM:SRE -lc
mv libsdbi.a libsdbi_emp.a

libsdbi_ora.a: sdbi_ora.o
$(LD) -o libsdbi.a sdbi_ora.o -bE:sdbi.exp -bM:SRE \
-L$( _ORACLE_HOME_ )/lib
-lsql $( _ORACLE_HOME_ )/lib/osntab.o -lsqlnet -lora \
-lpls -lnlsrtl3 -lc3v6 -lcore3 \
-lm -lld -lm -lc
mv libsdbi.a libsdbi_ora.a

opendbq.dll: opendbq.pc
$(PROC) $(PROFLAGS) INAME=opendbq.pc ONAME=opendbq.c
$(CC) $(CFLAGS) -c -DRULE_HEADER="VOPENDBQ.h" \
opendbq.c $(DNADIR)/misc/srv_gen.c
$(LD) -o opendbq.dll opendbq.o srv_gen.o $(SERVICE_LFLAGS) \
-bI:opendbq.imp -L. -lc

clean:

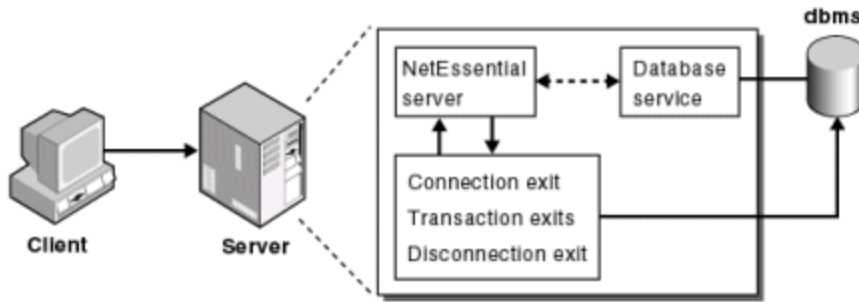
```

Database Exits in C

Database connects, starts, commits, aborts, and disconnects are by far the most DBMS-dependent operations a database program uses. For supported DBMSs, AppBuilder communications exits in C manage these operations for you. If you are not using a supported DBMS, you must customize the exits with the DBMS-specific code for these operations.

The following figure shows how the database exits fit into the processing flow of your application. Refer to *Third-Party Support Matrix* (available on the Customer Support Web site) for details as to which databases and levels are supported.

Database exits



The exits are made available in the form of export functions in the SDBI (database interface) library. You customize the shared library, rebuild it, and substitute it for the library supplied with the product. The library is named `libsdbi` and is already linked to AppBuilder communications when you receive the product. The exits are available on the server side only. See [Example of Customized Marshalling](#) for information about compilation.

The entry points into the library are as follows:

- [sdbi_Connect](#)
- [sdbi_StartTxn](#)
- [sdbi_CommitTxn](#)
- [sdbi_AbortTxn](#)
- [sdbi_Disconnect](#)

✓ Make sure to include the header files `stdio.h`, `stdlib.h`, and `sdbi.h`.

The connection exit returns connection information to AppBuilder. It uses the following data structure:

```
typedef struct {
    HWND sdbi_handle;
    char* sdbi_dbname;
    char* sdbi_userid;
    char* sdbi_passwd;
    sdbi_Output_t sdbi_output;
} sdbi_Connect_t;
```

The remaining exits return success or failure to AppBuilder. They use the following data structure:

```
typedef struct {
    long sdbi_sqlcode;
} sdbi_Output_t;
```

✓ Be sure to include all the exits. The SDBI library supplied with the product contains exits for Informix isolation-mode processing in addition to the routines described above. Because AppBuilder invokes all the exits, it is important to include all nine routines in your customized shared library. The unused exits can be empty? a single statement that returns control works.

sdbi_Connect

Purpose

Open a database connection.

Use the security exits to pass login information to your DBMS. For details, see [C Client and Server Exits](#).

Prototype

```
#include <stdio.h>
#include <stdlib.h>
#include <sdbi.h>
short sdbi_Connect(sdbi_Connect_t* sdbiconn);
```

Formal Arguments

Formal arguments

Name	Description
sdbiconn	Pointer to connection information.

Example with sdbi_Connect

```
short sdbi_Connect(sdbi_Connect_t* sdbiconn)
{
    printf("declaring myself connected\n");
    sdbiconn->sdbi_output.sdbi_sqlcode = 0;
    return 0;
}
```

sdbi_StartTxn

Purpose

Start a database transaction.

Prototype

```
#include <stdio.h>
#include <stdlib.h>
#include <sdbi.h>
short sdbi_StartTxn(sdbi_Output_t* sdbiout);
```

Formal Arguments

Formal arguments

Name	Description
sdbiout	Pointer to the DBMS error code for a start transaction operation.

Example with sdbi_StartTxn

```
short sdbi_StartTxn(sdbi_Output_t* sdbiout)
{
    printf("declaring a begin transaction\n");
    sdbiout->sdbi_sqlcode = 0;
    return 0;
}
```

sdbi_CommitTxn

Purpose

Commit a database transaction.

Prototype

```
#include <stdio.h>
#include <stdlib.h>
#include <sdbi.h>
short sdbi_CommitTxn(sdbi_Output_t* sdbiout);
```

Formal Arguments

Formal arguments

Name	Description
sdbiout	Pointer to the DBMS error code for a commit transaction operation.

Example with sdbi_CommitTxn

```
short sdbi_CommitTxn(sdbi_Output_t* sdbiout)
{
    printf("declaring a commit transaction\n");
    sdbiout->sdbi_sqlcode = 0;
    return 0;
}
```

sdbi_AbortTxn

Purpose

Abort a database transaction.

Prototype

```
#include <stdio.h>
#include <stdlib.h>
#include <sdbi.h>
short sdbi_AbortTxn(sdbi_Output_t* sdbiout);
```

Formal Arguments

Formal arguments

Name	Description
sdbiout	Pointer to the DBMS error code for an abort transaction operation.

Example with sdbi_AbortTxn

```

short sdbi_AbortTxn(sdbi_Output_t* sdbiout)
{
    printf("declaring a abort transaction\n");
    sdbiout->sdbi_sqlcode = 0;
    return 0;
}

```

sdbi_Disconnect

Purpose

Disconnect from a database.

Prototype

```

#include <stdio.h>
#include <stdlib.h>
#include <sdbi.h>
short sdbi_Disconnect(sdbi_Output_t* sdbiout);

```

Formal Arguments

Formal arguments

Name	Description
sdbiout	Pointer to a the DBMS error code for a disconnect operation.

Example with sdbi_Disconnect

```

short sdbi_Disconnect(sdbi_Output_t* sdbiout)
{
    printf("declaring myself disconnected\n");
    sdbiout->sdbi_sqlcode = 0;
    return 0;
}

```

Sample Database Exit Routines

The following sample routines show how you might code the exits for Oracle, a supported DBMS. Follow the examples for the DBMS you are using.

```

#include <stdio.h>
#include <stdlib.h>
#include <sdbi.h>

EXEC SQL INCLUDE SQLCA;

short sdbi_Connect(sdbi_Connect_t* sdbiconn)
{
    EXEC SQL BEGIN DECLARE SECTION;

```

```

char connect_id[5];

EXEC SQL END DECLARE SECTION;

printf("connecting\n");
strcpy(connect_id, "/");

EXEC SQL CONNECT :connect_id;

sdbiconn->sdbi_output.sdbi_sqlcode = sqlca.sqlcode;
return (sqlca.sqlcode == 0 ? 0 : -1);
}

short sdbi_Disconnect(sdbi_Output_t* sdbiout)
{
    printf("disconnecting\n");

    EXEC SQL COMMIT WORK RELEASE;

    sdbiout->sdbi_sqlcode = sqlca.sqlcode;
    return (sqlca.sqlcode == 0 ? 0 : -1);
}

int sdbi_SetIsolationMode_CS(sdbi_Output_t* sdbiout)
{
    printf("declaring isolation to be set to cursor stability\n");
    sdbiout->sdbi_sqlcode = 0;
    return 0;
}

int sdbi_SetIsolationMode_RR(sdbi_Output_t* sdbiout)
{
    printf("declaring isolation to be set to repeatable read\n");
    sdbiout->sdbi_sqlcode = 0;
    return 0;
}

int sdbi_SetIsolationMode_DR(sdbi_Output_t* sdbiout)
{
    printf("declaring isolation to be set to dirty read\n");
    sdbiout->sdbi_sqlcode = 0;
    return 0;
}

int sdbi_SetIsolationMode_CR(sdbi_Output_t *sdbiout)
{
    printf("declaring isolation to be set to committed read\n");
    sdbiout->sdbi_sqlcode = 0;
    return 0;
}

short sdbi_StartTxn(sdbi_Output_t *sdbiout)
{
    printf("beginning transaction\n");
    printf("Note: Oracle requires no start transaction exit");
    sdbiout->sdbi_sqlcode = 0;
    return 0;
}

short sdbi_CommitTxn(sdbi_Output_t* sdbiout)
{
    printf("committing transaction\n");

    EXEC SQL COMMIT WORK;

    sdbiout->sdbi_sqlcode = sqlca.sqlcode;
    return (sqlca.sqlcode == 0 ? 0 : -1);
}

short sdbi_AbortTxn(sdbi_Output_t* sdbiout)

```

```
{  
    printf("rolling back transaction\n");  
  
    EXEC SQL ROLLBACK WORK;  
  
    sdbiout->sdbi_sqlcode = sqlca.sqlcode;
```



```
} return (sqlca.sqlcode == 0 ? 0 : -1);
```

See also [Database Exits in C](#).

CICS TCP-IP Listener Security Settings

This setting has two aspects. The first relates to security authentication/verification and the second relates to whether to pass on the supplied user ID.

Security Authentication

The NO and PART settings imply no security verification.

Specifying YES (or CICS) implies that an EXEC CICS VERIFY command will be issued. Errors result in message 934 (and a code of 406 returned to the client).

EXIT or EXITONLY causes the Listener to invoke the user's authentication exit. A non-zero return code results in message 301 (and a code of 406 returned to the client).

User-ID

There are two types of EXEC CICS START commands; one with the user ID and one without.

When specifying YES (or CICS), PART or EXIT, the user-ID is used.

With the NO and EXITONLY settings, no user-ID is used (and the started transaction "inherits" the user-ID associated with the Listener task.)

Whether or not the user-ID is used, any non-zero return code from the EXEC CICS START command results in error message 414. This is a generic EXEC CICS xxx error message, not specific to failed EXEC CICS START commands.

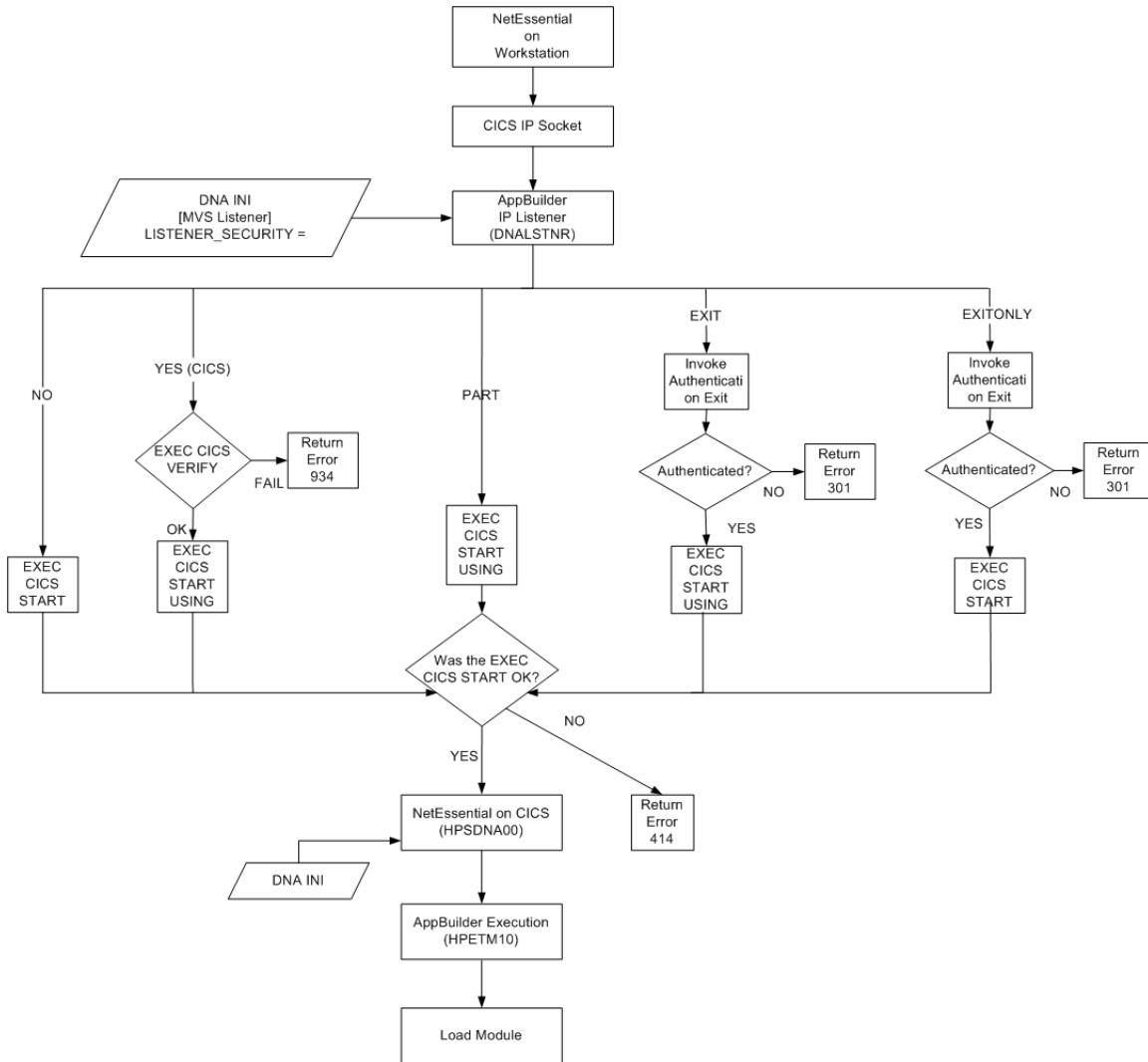
Therefore it would be more accurate to have the USING only on the EXEC CICS START associated with YES (CICS), PART and EXIT. We recommend that the flow for all converge after the EXEC CICS START with the return code checked.

The Listener Security Section is the MVS_Listener section for the DNA.INI. It can be set to one of the following values:

- No
- Yes (CICS)
- Part
- Exit
- Exit Only
- Authentx
- Authentxonly
- Authorx
- Authorxonly

The following figure shows the effect of this setting:

LISTENER_Security settings



For more information see:

- [Setting Up Security Exits](#)
- [Open Data Encryption Mechanism](#)
- [Sample Database Exit Routines](#)

Customized Data Types in C

The communications marshalling primitives convert data representations specific to the sending platform to platform-independent AppBuilder communications representations. Unmarshalling primitives converts these representations to representations specific to the receiving platform. This topic examines how you use the system primitives to marshal and unmarshal data types other than the AppBuilder communications standard data types. You customize the shared library for the marshalling functions, rebuild it, and substitute it for the library supplied with the product. In most situations, you should:

- Identify the new data type to a system table
- Write customized encoding and decoding routines that call the system primitives
- Compile the new code and link it into the system network library

AppBuilder communications data types are based on the External Data Representation (XDR) for network representation of standard data types.

- For a discussion of customer data types, see [Using Custom Data Types](#).
- For identifying the new data type, see [Identifying the Data Type](#).
- For writing customized routines, see [Writing the Marshalling Functions](#).
- For compilation instructions, see [Example of Customized Marshalling](#).
- For more information on marshalling, refer to [Understanding Performance Marshalling](#).

Using Custom Data Types

If the customized data type is an aggregate of existing standard types, the encoding and decoding functions contain only calls to primitive functions. If you create one 64-bit timestamp, for instance, interpreted as composed of two long integers (as opposed to the 96-bit timestamp), the marshalling function on the sending side issues two calls to the primitive function that marshals a long integer. On the receiving side, the unmarshalling function issues two calls to the primitive function that unmarshals a long integer.

If the customized data type requires additional manipulation—for instance, the conversion of a mainframe-based eight-byte character array so that the array appears in inverted order on the workstation—the encoding and decoding functions must do the required manipulation, and the order of events is crucial:

- If manipulation is at the sending side, the customized marshalling function manipulates the data and invokes the primitive marshalling functions to send the data
- If manipulation is at the receiving side, the customized unmarshalling function invokes the primitive unmarshalling functions to accept the data as transmitted and performs the manipulation

To call either primitive or customized functions, AppBuilder reads a table that associates each data type with a pair of pointers: one to a marshalling function, one to an unmarshalling function. Access to the table is by an integer key, which represents a data type as expressed in the control-structure array. Specify a new key when you add a new data type, as described in [Identifying the Data Type](#). Platform-specific versions of the customized functions must be installed on every platform that handles the customized data type.

Identifying the Data Type

Identify a customized data type by adding an entry for it in the system marshalling table and incrementing the value of the #define statement `MARSH_TYPES_COUNT` in the table. The value must reflect the new number of entries.

The source for the system marshalling table is in a platform-specific location, as shown in the following table:

System marshalling table location

Platform	Table location
Mainframe	data set SAMP255, member DNA@MRSH
UNIX	\$DNADIR/misc, member dna_mrsh.c
Windows	\dna\make, member dna_mrsh.c

Each entry in the table is of the type:

```
typedef struct {
    short Data_Type;
    char Data_Type_Desc[32];
    marsh_func_ptr Marsh_Func_Ptr;
    marsh_func_ptr Unmarsh_Func_Ptr;
} MarshFuncStruct;
```

The items in `MarshFuncStruct` (the marshalling function structure) are summarized in the following table:

Marshalling function structure items

Item	Description
<code>Data_Type</code>	#define value for the data type, as shown in <code>marshall.h</code> . Customized data types may have a value between <code>Data_Type_N</code> and <code>Data_Type_Z</code> , inclusive
<code>Data_Type_Desc</code>	String that identifies the type description, in quotes
<code>Marsh_Func_Ptr</code>	Pointer to the marshalling function for the data type
<code>Unmarsh_Func_Ptr</code>	Pointer to the unmarshalling function for the data type

The definition for the type `marsh_func_ptr` follows:

```
typedef int (*marsh_func_ptr)(void *, void *, unsigned short);
```

Writing the Marshalling Functions

Include the prototypes for customized marshalling and unmarshalling functions in `dna_mrsh.c` or a header file. The prototypes for the functions, whether primitive or customized, are as follows:

```
int dna_Marshall_< lowercase-description-string >
(void* handle, void* data, unsigned short data_len);
int dna_UnMarshall_< lowercase-description-string >
(void* handle, void* data, unsigned short data_len);
```

For example:

```
int dna_Marshall_short(void* handle, void* data, unsigned short data_len);
int dna_UnMarshall_long(void* handle, void* data, unsigned short data_len);
```

The formal arguments are summarized in the following table:

Formal arguments for marshalling and unmarshalling functions

Argument	Description
handle	Pointer to a system-assigned value, to be passed when making calls to other marshalling functions
data	Pointer to an area that contains data in a platform-specific representation In the marshalling function, the parameter points to an area that contains the data to be converted to a system representation. In the unmarshalling function, the parameter points to an area that contains the data that the function converted from a system representation.
data_len	Number of bytes of data

The customized routines should return 0 for failure, 1 for success, as do the primitives they call.

If you include multiple primitive functions in the customized marshalling and unmarshalling routines, make sure the data area pointer and data length passed to the primitive functions are appropriate for the segment of the data they handle, as shown in the example below.

Example of Customized Marshalling

This is an example of a customized marshalling routine.

```
#include <my_mrsh.h>
int dna_Marshall_mytype(void* handle, void* data, unsigned short data_len)
{
    char* data_ptr;
    unsigned short tempdata_len;
    data_ptr = (char *)data;
    tempdata_len = 50;
    if (1 != dna_Marshall_chararr(handle, data, tempdata_len));
        return 0;
    data_ptr = data_ptr + 50;
    tempdata_len = 4;
    if (1 != dna_Marshall_long(handle, data_ptr, tempdata_len));
        return 0;
    data_ptr = data_ptr + 4;
    tempdata_len = 2;
    if (1 != dna_Marshall_short(handle, data_ptr, tempdata_len));
        return 0;
    data_ptr = data_ptr + 2;
    if (1 != dna_Marshall_short(handle, data_ptr, tempdata_len));
        return 0;
    return 1;
}
```

To protect against a change in system architecture, the system provides the intermediate functions:

```
dna_GetMarshFunction(short DataType);  
dna_GetUnMarshFunction(short DataType);
```

each of which returns either a pointer to a primitive function or, on failure, NULL. The sole argument in each is the data type of interest, following the data types listed in `marshall.h`.

Supported CodePages

You can use AppBuilder to off-load codepage conversions to clients or servers as appropriate. At runtime, it compares two codepage references on the local machine:

- The LOCAL_CODEPAGE variable in the NLS section of DNA.INI
- The Codepage field in the route table, or the corresponding variable in ROUTING section of dna.ini, SMA section of dna.ini, or the call to the functions `dna_BoundServiceRequest` or `dna_BoundDynamicRequest`.

If the values are different, it converts the data to be transmitted to the route table or corresponding value. If the value is a hyphen (-) in the route table, blank in `dna.ini`, or the null character in the function call, no conversion occurs.

The system then compares two codepage references on the remote machine:

- The LOCAL_CODEPAGE variable in the NLS section of dna.ini
- A codepage identifier in the transmitted data

If the values are different, AppBuilder converts the transmitted data to the codepage specified in `dna.ini`.

It is usually more efficient to perform codepage conversion on the client.

AppBuilder sends text files in ASCII format. If the destination is a mainframe, it converts the text to EBCDIC.

The first column in the following table displays the valid codepage names for the AppBuilder communications software, specified in the `dna.ini` file.

Supported Codepages

AppBuilder Names	Description
cp437	US English
cp819	iso-8859-1
cp850	Western English
cp860	Portuguese
cp863	French
cp865	Nordic
cp874	Thai
cp932	Shift_JIS
cp949	Korean
cp1004	Windows Latin1
cp1252	Windows Latin1
cp1253	Greek with Euro
ibm-037	US English - EBCDIC
ibm-273	German - EBCDIC
ibm-277	Danish - EBCDIC
ibm-278	Swedish - EBCDIC
ibm-280	Italian - EBCDIC

ibm-284	Spanish - EBCDIC
ibm-285	English Ireland - EBCDIC
ibm-297	French EBCDIC
ibm-500	International Latin1 EBCDIC
ibm-930	Japanese EBCDIC
ibm-933	Korean EBCDIC
ibm-939	Kanji EBCDIC
ibm-1047	Open Systems Latin1 - EBCDIC

For the codepage setting in appbuildercom.ini, the supported codepages are given in the table below

Supported Codepages

AppBuilder Name for codepage	Java name	Description
ibm-437	Cp437	US English
ibm-819	Cp819	iso-8859-1
ibm-850	Cp850	Western English
ibm-860	Cp860	Portuguese
ibm-863	Cp863	French
ibm-865	Cp865	Nordic
ibm-874	Cp874	Thai
ibm-932	Cp932	Shift_JIS
ibm-949	Cp949	Korean
ibm-1004	Cp1004	Windows Latin1
ibm-1252	Cp1252	Windows Latin1
ibm-1253	Cp1253	Greek with Euro
ibm-037	Cp037	US English - EBCDIC
ibm-273	Cp273	German - EBCDIC
ibm-277	Cp277	Danish - EBCDIC
ibm-278	Cp278	Swedish - EBCDIC
ibm-280	Cp280	Italian - EBCDIC
ibm-284	Cp284	Spanish - EBCDIC
ibm-285	Cp285	English Ireland - EBCDIC
ibm-297	Cp297	French EBCDIC
ibm-500	Cp500	International Latin1 EBCDIC
ibm-930	Cp930	Japanese EBCDIC
ibm-933	Cp933	Korean EBCDIC
ibm-939	Cp939	Kanji EBCDIC
ibm-1047	Cp1047	Open Systems Latin1 - EBCDIC

Mainframe Logon Script Reference

Use the LU2 scripting language described in this section to configure the mainframe logon. Change the MFLOGON.SCR file that comes with AppBuilder standard installation to reflect site-specific logon procedures. If the script-mediated logon is unsuccessful, try adding WAIT commands or increasing the WAIT and WAITSTRING values.

Every non-blank line in a logon or logoff script must begin with a comment marker (//) or a verb. The verbs in the AppBuilder logon script language are:

COMPARE	SEND
CONNECTPS	SESSION_LIST
DISCONNECTPS	SET
DOWNLOAD_FILE	SETCURSOR
FINDSTRING	SETNETNAME
GETCURSOR	SETTERMID
GOTO	WAIT
IF	WAITSTRING
INC	WRITE
RETURN	

Return values are not available to the script. When a verb returns a non-zero value, AppBuilder quits interpreting the script and writes an error message to the file identified in the TRCFILE variable in the TRACING section of the communications configuration file dna.ini. An error code is set for use by the client application.

In the following descriptions, a parameter value can be a literal. Character or string literals must be within double quotation marks.

COMPARE

Compares a string with the data that begins at the current cursor position on the host display. The value is 0 if identical, non-zero otherwise.
`COMPARE string;`

For example:
`COMPARE "000"`

CONNECTPS

Establishes a connection between the client application and one of the session IDs supplied by SESSION_LIST. The value is 0 if successful, non-zero otherwise.
`CONNECTPS;`

DISCONNECTPS

Drops the connection between the client application and the terminal session. The value is 0 if successful, non-zero otherwise.
`DISCONNECTPS;`

DOWNLOAD_FILE

Downloads a file from the host.
`DOWNLOAD_FILE file;`

For example:
`DOWNLOAD_FILE "HS01";`

FINDSTRING

Searches the host display for a string. The value is 0 if successful, non-zero otherwise.

```
FINDSTRING string;
```

For example:

```
FINDSTRING "READY";
```

GETCURSOR

Returns the current cursor position on the host display into variables *x* and *y*. The value is 0 if successful, non-zero otherwise.

```
GETCURSOR x y;
```

For example:

```
GETCURSOR SYSVAR1 SYSVAR7;
```

GOTO

Transfers flow of control to the line that contains the specified label.

```
GOTO label;
```

For example:

```
GOTO SUCCESS;
```

IF

Performs a logical operation on two simple expressions and executes the specified command when the result of the logical operation is true. The following operands are valid: EQ, LT, GT, NE, LE, GE.

```
IF expression operand expression command;
```

For example:

```
IF STATUSLEVEL EQ 0 GOTO SUCCESS;
```

INC

Increments a system variable by an integer value or the contents of another system variable.

```
INC x integer or y;
```

For example:

```
INC SYSVAR2 20;
```

RETURN

Ends execution of the script file. The value is 0 if successful, non-zero otherwise.

```
RETURN 0;
```

SEND

Sends one or more keystrokes to the connected terminal session: LEFT_TAB, RIGHT_TAB, CLEAR, ENTER, CURSOR_LEFT, CURSOR_RIGHT, CURSOR_UP, CURSOR_DOWN, NEW_LINE, SPACE, RESET, HOME, PA1 through PA3, PF1 through PF24, PLUS, END, and SYSREQ. Use the variable HOSTNAME as shown in the example below if you want to send the value specified in the route table Host field.

The value is 0 if successful, non-zero otherwise.

```
SEND one_or_more_keystrokes;
```

For example:

```
SEND HOSTNAME ENTER
```

Make sure to put spaces around keywords (except at the end of a line). The following example shows the correct syntax:

```
SEND "LOGON APPLID(" HOSTNAME ")" ENTER;
```

As the example shows, the keyword HOSTNAME must be surrounded by spaces. You do not need to put a space after ENTER because it comes at the end of the line.

SESSION_LIST

Lists the sessions available for CONNECTPS to use.

```
SESSION_LIST list_of_session_IDs;
```

For example:

```
SESSION_LIST "A" "B" "C" "D";
```

SET

Initializes a system variable with an integer value or the contents of another system variable. SYSVAR1 through SYSVAR9 are available for use as system variables.

```
SET x integer or y;
```

For example:

```
SET SYSVAR2 SYSVAR1;
```

SETCURSOR

Sets the cursor position.

```
SETCURSOR row_number column_number;
```

For example:

```
SETCURSOR 4 20;
```

SETNETNAME

Sets the network name.

```
SETNETNAME;
```

SETTERMID

Sets the terminal identifier.

```
SETTERMID;
```

WAIT

Waits for the number of seconds specified.

```
WAIT number_of_seconds;
```

For example:

```
WAIT 2;
```

WAITSTRING

Waits for the specified number of seconds until the specified string appears on the host screen. Follow this command with a WAIT 2. The value is 0 if the string appears within the time specified, non-zero otherwise.

```
WAITSTRING string number_of_seconds;
```

For example:

```
WAITSTRING "LOGONID" 20;
```

```
WAIT 2;
```

WRITE

Copies text to the host screen at the current cursor location. The value is 0 if successful, non-zero otherwise.

```
WRITE one_or_more_strings;
```

For example:

```
WRITE "CACF" USERID PASSWORD NEWPASSWORD "1" "0" "0";
```