QAnywhere™

Published: October 2006

Copyright and trademarks

Contents

About This Manual

Subject

This manual describes QAnywhere, which is a messaging platform for mobile and wireless clients as well as traditional desktop and laptop clients.

Audience

This manual is for users of SQL Anywhere and other relational database systems who want to add messaging to their mobile applications, or who want to build new mobile application-to-application messaging solutions.

SQL Anywhere documentation

This book is part of the SQL Anywhere documentation set. This section describes the books in the documentation set and how you can use them.

The SQL Anywhere documentation

The complete SQL Anywhere documentation is available in two forms: an online form that combines all books, and as separate PDF files for each book. Both forms of the documentation contain identical information and consist of the following books:

- **SQL Anywhere 10 - Introduction** This book introduces SQL Anywhere 10—a comprehensive package that provides data management and data exchange, enabling the rapid development of database-powered applications for server, desktop, mobile, and remote office environments.
- **SQL Anywhere 10 - Changes and Upgrading** This book describes new features in SQL Anywhere 10 and in previous versions of the software.
- **SQL Anywhere Server - Database Administration** This book covers material related to running, managing, and configuring SQL Anywhere databases. It describes database connections, the database server, database files, security, backup procedures, security, and replication with Replication Server, as well as administration utilities and options.
- **SQL Anywhere Server - SQL Usage** This book describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.
- **SQL Anywhere Server - SQL Reference** This book provides a complete reference for the SQL language used by SQL Anywhere. It also describes the SQL Anywhere system views and procedures.
- **SQL Anywhere Server - Programming** This book describes how to build and deploy database applications using the C, C++, and Java programming languages, as well as Visual Studio .NET. Users of tools such as Visual Basic and PowerBuilder can use the programming interfaces provided by those tools.
- **SQL Anywhere 10 - Error Messages** This book provides a complete listing of SQL Anywhere error messages together with diagnostic information.
- **MobiLink - Getting Started** This manual introduces MobiLink, a session-based relational-database synchronization system. MobiLink technology allows two-way replication and is well suited to mobile computing environments.
- **MobiLink - Server Administration** This manual describes how to set up and administer MobiLink applications.
- **MobiLink - Client Administration** This manual describes how to set up, configure, and synchronize MobiLink clients. MobiLink clients can be SQL Anywhere or UltraLite databases.
- **MobiLink - Server-Initiated Synchronization** This manual describes MobiLink server-initiated synchronization, a feature of MobiLink that allows you to initiate synchronization or other remote actions from the consolidated database.

- **QAnywhere** This manual describes QAnywhere, which defines a messaging platform for mobile and wireless clients as well as traditional desktop and laptop clients.
- **SQL Remote** This book describes the SQL Remote data replication system for mobile computing, which enables sharing of data between a SQL Anywhere consolidated database and many SQL Anywhere remote databases using an indirect link such as email or file transfer.
- **SQL Anywhere 10 - Context-Sensitive Help** This manual provides context-sensitive help for the Connect dialog, the Query Editor, the MobiLink Monitor, the SQL Anywhere Console utility, the Index Consultant, and Interactive SQL.
- **UltraLite - Database Management and Reference** This manual introduces the UltraLite database system for small devices.
- **UltraLite - AppForge Programming** This manual describes UltraLite for AppForge. With UltraLite for AppForge you can develop and deploy database applications to handheld, mobile, or embedded devices, running Palm OS, Symbian OS, or Windows CE.
- **UltraLite - .NET Programming** This manual describes UltraLite.NET. With UltraLite.NET you can develop and deploy database applications to computers, or handheld, mobile, or embedded devices.
- **UltraLite - M-Business Anywhere Programming** This manual describes UltraLite for M-Business Anywhere. With UltraLite for M-Business Anywhere you can develop and deploy web-based database applications to handheld, mobile, or embedded devices, running Palm OS, Windows CE, or Windows XP.
- **UltraLite - C and C++ Programming** This manual describes UltraLite C and C++ programming interfaces. With UltraLite you can develop and deploy database applications to handheld, mobile, or embedded devices.

Documentation formats

SQL Anywhere provides documentation in the following formats:

- **Online documentation** The online documentation contains the complete SQL Anywhere documentation, including the books and the context-sensitive help for SQL Anywhere tools. The online documentation is updated with each maintenance release of the product, and is the most complete and up-to-date source of documentation.

  To access the online documentation on Windows operating systems, choose Start ► Programs ► SQL Anywhere 10 ► Online Books. You can navigate the online documentation using the HTML Help table of contents, index, and search facility in the left pane, as well as using the links and menus in the right pane.

  To access the online documentation on Unix operating systems, see the HTML documentation under your SQL Anywhere installation or on your installation CD.

- **PDF files** The complete set of SQL Anywhere books is provided as a set of Adobe Portable Document Format (pdf) files, viewable with Adobe Reader.

  On Windows, the PDF books are accessible from the online books via the PDF link at the top of each page, or from the Windows Start menu (Start ► Programs ► SQL Anywhere 10 ► Online Books - PDF Format).

On Unix, the PDF books are accessible on your installation CD.

Documentation conventions

This section lists the typographic and graphical conventions used in this documentation.

Syntax conventions

The following conventions are used in the SQL syntax descriptions:

- **Keywords** All SQL keywords appear in uppercase, like the words ALTER TABLE in the following example:

  ALTER TABLE [*owner*.]*table-name*

- **Placeholders** Items that must be replaced with appropriate identifiers or expressions are shown like the words *owner* and *table-name* in the following example:

  ALTER TABLE [*owner*.]*table-name*

- **Repeating items** Lists of repeating items are shown with an element of the list followed by an ellipsis (three dots), like *column-constraint* in the following example:

  ADD *column-definition* [*column-constraint*, …]

  One or more list elements are allowed. In this example, if more than one is specified, they must be separated by commas.

- **Optional portions** Optional portions of a statement are enclosed by square brackets.

  RELEASE SAVEPOINT [*savepoint-name*]

  These square brackets indicate that the *savepoint-name* is optional. The square brackets should not be typed.

- **Options** When none or only one of a list of items can be chosen, vertical bars separate the items and the list is enclosed in square brackets.

  [**ASC** | **DESC**]

  For example, you can choose one of ASC, DESC, or neither. The square brackets should not be typed.

- **Alternatives** When precisely one of the options must be chosen, the alternatives are enclosed in curly braces and a bar is used to separate the options.

  [**QUOTES** { **ON** | **OFF** }]

  If the QUOTES option is used, one of ON or OFF must be provided. The brackets and braces should not be typed.

File name conventions

The documentation generally adopts Windows conventions when describing operating-system dependent tasks and features such as paths and file names. In most cases, there is a simple transformation to the syntax used on other operating systems.

- **Directories and path names** The documentation typically lists directory paths using Windows conventions, including colons for drives and backslashes as a directory separator. For example,

  ```
  MobiLink\redirector
  ```

  On Unix, Linux, and Mac OS X, you should use forward slashes instead. For example,

  ```
  MobiLink/redirector
  ```

- **Executable files** The documentation shows executable file names using Windows conventions, with the suffix *.exe*. On Unix, Linux, and Mac OS X, executable file names have no suffix. On NetWare, executable file names use the suffix *.nlm*.

  For example, on Windows, the network database server is *dbsrv10.exe*. On Unix, Linux, and Mac OS X, it is *dbsrv10*. On NetWare, it is *dbsrv10.nlm*.

- **install-dir** The installation process allows you to choose where to install SQL Anywhere, and the documentation refers to this location using the convention *install-dir*.

  After installation is complete, the environment variable SQLANY10 specifies the location of the installation directory containing the SQL Anywhere components (*install-dir*). SQLANYSH10 specifies the location of the directory containing components shared by SQL Anywhere with other Sybase applications.

  For more information on the default location of *install-dir*, by operating system, see "File Locations and Installation Settings" [*SQL Anywhere Server - Database Administration*].

- **samples-dir** The installation process allows you to choose where to install the samples that are included with SQL Anywhere, and the documentation refers to this location using the convention *samples-dir*.

  After installation is complete, the environment variable SQLANYSAMP10 specifies the location of the directory containing the samples (*samples-dir*). From the Windows Start menu, choosing Programs ▶ SQL Anywhere 10 ▶ Sample Applications and Projects opens a Windows Explorer window in this directory.

  For more information on the default location of *samples-dir*, by operating system, see "The samples directory" [*SQL Anywhere Server - Database Administration*].

- **Environment variables** The documentation refers to setting environment variables. On Windows, environment variables are referred to using the syntax *%envvar%*. On Unix, Linux, and Mac OS X, environment variables are referred to using the syntax *$envvar* or *${envvar}*.

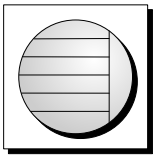
Unix, Linux, and Mac OS X environment variables are stored in shell and login startup files, such as *.cshrc* or *.tcshrc*.

Graphic icons

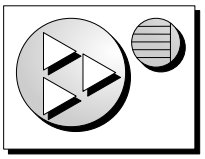
The following icons are used in this documentation.

- A client application.

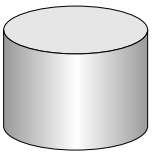
- A database server, such as SQL Anywhere.



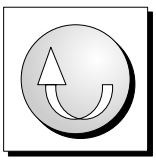
- An UltraLite application.



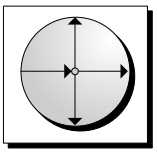
- A database. In some high-level diagrams, the icon may be used to represent both the database and the database server that manages it.



- Replication or synchronization middleware. These assist in sharing data among databases. Examples are the MobiLink server and the SQL Remote Message Agent.



- A Sybase Replication Server



- A programming interface.

Finding out more and providing feedback

Finding out more

Additional information and resources, including a code exchange, are available at the iAnywhere Developer Network at http://www.ianywhere.com/developer/.

If you have questions or need help, you can post messages to the iAnywhere Solutions newsgroups listed below.

When you write to one of these newsgroups, always provide detailed information about your problem, including the build number of your version of SQL Anywhere. You can find this information by entering **dbeng10 -v** at a command prompt.

The newsgroups are located on the *forums.sybase.com* news server. The newsgroups include the following:

- sybase.public.sqlanywhere.general
- sybase.public.sqlanywhere.linux
- sybase.public.sqlanywhere.mobilink
- sybase.public.sqlanywhere.product_futures_discussion
- sybase.public.sqlanywhere.replication
- sybase.public.sqlanywhere.ultralite
- ianywhere.public.sqlanywhere.qanywhere

Newsgroup disclaimer
iAnywhere Solutions has no obligation to provide solutions, information, or ideas on its newsgroups, nor is iAnywhere Solutions obliged to provide anything other than a systems operator to monitor the service and ensure its operation and availability.
iAnywhere Solutions Technical Advisors as well as other staff assist on the newsgroup service when they have time available. They offer their help on a volunteer basis and may not be available on a regular basis to provide solutions and information. Their ability to help is based on their workload.

Feedback

We would like to receive your opinions, suggestions, and feedback on this documentation.

You can email comments and suggestions to the SQL Anywhere documentation team at iasdoc@ianywhere.com. Although we do not reply to emails sent to that address, we read all suggestions with interest.

In addition, you can provide feedback on the documentation and the software through the newsgroups listed above.

Part I. Creating QAnywhere Applications

This part shows you how to set up QAnywhere and write client applications.

CHAPTER 1

Introduction to QAnywhere

Contents

About this chapter

QAnywhere is a comprehensive application-to-application messaging system for mobile users. It provides the infrastructure for you to write applications that exchange messages with remote applications located on a variety of devices running on Windows or Windows CE operating systems.

QAnywhere application-to-application messaging

Application-to-application messaging permits communication between custom programs running on mobile or wireless devices and a centrally located server application. QAnywhere messaging is a useful application-to-application communication mechanism in a variety of situations:

- It provides messaging between mobile devices and between mobile devices and the enterprise.
- It provides communication in occasionally-connected environments.

  The store-and-forward nature of messaging means that messages can be constructed even when the destination application is not reachable over the network; the message is delivered when the network becomes available.

  QAnywhere messages are exchanged via a central server, so that the sender and recipient of a message never have to be connected to the network at the same time.
- It provides network-independent communication.

  QAnywhere messages can be transported over TCP/IP, HTTP, or HTTPS protocols. They can also be delivered from a Windows CE handheld device by ActiveSync. The message itself is independent of the network protocol, and can be received by an application that communicates over a different network.

QAnywhere handles the challenges of wireless networks, such as slow speed, spotty geographic coverage, and dropped network connections. It can protect proprietary or sensitive information by encrypting all messages sent over public networks. You can customize the delivery of messages using transmission rules so that, for example, messages are transmitted during off-peak hours.

QAnywhere compresses and, optionally, encrypts data sent between mobile applications and enterprise servers. Furthermore, it implements a store-and-forward messaging paradigm that guarantees message delivery.

QAnywhere is designed for messaging solutions on a variety of handheld devices. This system provides a QAnywhere C++, Java, .NET, and SQL API to provide solutions to developers with different skill sets.

QAnywhere permits seamless communication with other messaging systems that have a JMS interface. This allows integration with J2EE applications.

What QAnywhere does

QAnywhere provides the following application-to-application features and components.

- **QAnywhere API** The object-oriented QAnywhere API provides the infrastructure to build messaging applications for Windows desktop and Windows CE devices. The QAnywhere API is available in Java, C++, .NET, and SQL.
- **Store-and-forward** QAnywhere applications store messages locally until a connection between the client and the server is available for data transmission.
- **Complements data synchronization** QAnywhere applications use relational databases as a temporary message store. The relational database ensures that the message store has security, transaction-based computing, and the other benefits of relational databases.

  The use of SQL Anywhere relational databases as message stores makes it easy to use QAnywhere together with a data synchronization solution. Both use MobiLink synchronization as the underlying mechanism for exchanging information between client and server.
- **Integration with external messaging systems** In addition to exchanging messages among QAnywhere applications, you can integrate QAnywhere clients into external messaging systems that support a JMS interface.
- **Encryption** Messages can be sent encrypted using transport-layer security. In addition, messages stores can be encrypted using simple encryption or any FIPS-approved AES algorithm.
- **Compression** Message content can be stored compressed using the popular ZLIB compression library.
- **Authentication** You can authenticate QAnywhere clients using a built-in facility or through custom authentication scripts (including existing authentication services used in your organization).
- **Multiple networks** QAnywhere works over any wired or wireless network that supports TCP/IP or HTTP.
- **Failover** You can run multiple MobiLink servers so that there are alternate servers in case one fails.
- **Administration** A QAnywhere application can browse and manipulate messages on the client and server side.
- **Multiple queues** Support for multiple arbitrarily-named queues on client devices permits multiple client applications to coexist on a single device. Applications can send and receive on any number of queues. Messages can be sent between applications that are coexisting on the same device and between applications on different devices.
- **Server-initiated send and receive** QAnywhere can push messages to client devices, allowing client applications to implement message-driven logic.
- **Transmission rules** You can create rules that specify when message transmission should occur.
- **Resumable downloads** Large messages or groups of messages are sent to QAnywhere clients in piecemeal fashion to minimize the retransmission of data during network failures.

- **Guaranteed delivery** QAnywhere guarantees the delivery of messages.
- **Mobile web services** Mobile web services facilitate the transport of web service requests and responses over QAnywhere.

QAnywhere architecture

This section explains the architecture of QAnywhere messaging applications. The discussion begins with a simple messaging scenario and then progresses to more advanced scenarios.

Client applications send and receive messages using the QAnywhere API. Messages are queued in the client message store. Message transmission is the exchange of messages between client message stores through a central QAnywhere server message store.

The following typical messaging scenarios are supported by QAnywhere:

- **Simple messaging** For exchanging messages among QAnywhere clients. Client applications control when to transmit messages between the client and server message stores.

  ☞ See "Simple messaging scenario" on page 7.

- **Messaging with push notifications** For exchanging messages among QAnywhere clients. In this scenario, the MobiLink server can initiate message transmission between clients. This is done by exchanging messages between client and server message stores.

  ☞ See "Scenario for messaging with push notifications" on page 9.

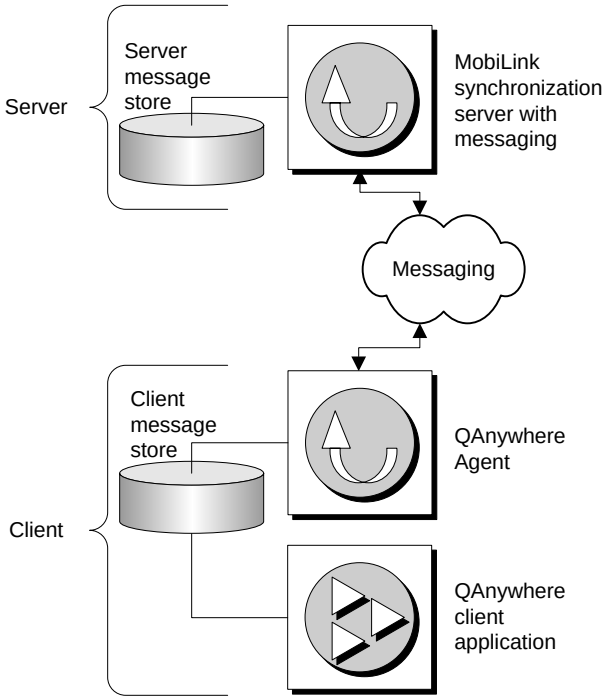
- **Messaging with external messaging systems** For exchanging messages among QAnywhere clients or an external system that supplies a JMS provider, such as BEA WebLogic or Sybase EAServer.

  ☞ See "Scenario for messaging with external messaging systems" on page 11.

Push notifications and external messaging systems can be used together, providing the most general solution.

Simple messaging scenario

A simple QAnywhere messaging setup is illustrated in the following diagram. For simplicity, only a single client is shown. However, a typical scenario has multiple clients with the server message store existing to transmit messages between them.



This setup includes the following components:

- **Server message store** At the server, the messages are stored in a relational database. The database must be set up as a MobiLink consolidated database, and may be any supported consolidated database (SQL Anywhere, Adaptive Server Enterprise, Microsoft SQL Server, DB2, or Oracle).
- **Client message store** The messages at each client are also stored in a relational database. The database used is SQL Anywhere.
- **MobiLink server with messaging** MobiLink synchronization provides the transport for transmitting and tracking messages between QAnywhere clients and the server. MobiLink provides security, authentication, encryption, and flexibility. It also allows messaging to be combined with data synchronization.

  To manage QAnywhere message transmission, the MobiLink server must be started with messaging enabled. You do this by supplying the MobiLink server -m command line option.

  ☞ For more information, see "Starting the MobiLink server for QAnywhere messaging" on page 31.
- **QAnywhere Agent** The QAnywhere Agent manages the transmission of messages on the client side. This process is independent of QAnywhere client applications.

  ☞ For more information, see "Running the QAnywhere Agent" on page 35.
- **QAnywhere client application** An application written using the QAnywhere C++, Java, or .NET API makes method calls to send and receive messages. The basic object used by the client application is the QAManager.

☞ For information about writing applications using the QAnywhere API, see "Writing QAnywhere Client Applications" on page 45.

Messages are sent and received by QAnywhere clients. Messages at the server will not be picked up until the client initiates a message transmission. QAnywhere clients use **policies** to determine when to carry out a message transmission. Policies include on-demand, automatic, scheduled, and custom. The on-demand policy permits the user to control message transmission. The automatic policy initiates a message transmission whenever a message to or from the client is ready for delivery.

☞ For more information, see "Determining when message transmission should occur on the client" on page 36.

Scenario for messaging with push notifications

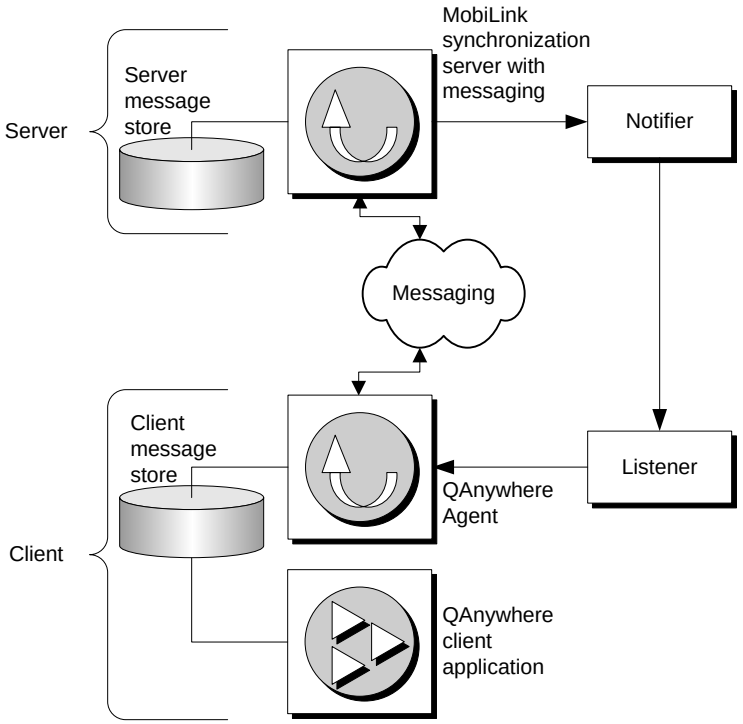
A push notification is a special message delivered from the server to a QAnywhere client. The push notification occurs when a message arrives at the server message store. The messaging server automatically notifies the recipient client Listener of the push request. The client initiates message transmission to receive messages waiting at the server or takes a custom action.

☞ For more information about the client's response to a push notification, see "Determining when message transmission should occur on the client" on page 36.

Push notifications introduce two extra components to the QAnywhere architecture. At the server, a QAnywhere Notifier sends push notifications. At the client, a QAnywhere Listener receives these push notifications and passes them on to the QAnywhere Agent.

If you do not use push notifications, messages are still transmitted from the server message store to the client message store, but the transmission must be initiated at the client, such as by using a scheduled transmission policy.

The architecture for messaging with push notifications is an extension of that described in "Simple messaging scenario" on page 7. The following diagram shows this architecture:



The following components are added to the "Simple messaging scenario" on page 7 enable push notification:

- **QAnywhere Notifier** The Notifier is the component of the MobiLink server that is used to deliver push notifications.

  The QAnywhere Notifier is a specially configured instance of the Notifier that sends push notifications when a message is ready for delivery.

- **QAnywhere Listener** The QAnywhere Listener is a separate process that runs at the client. It receives push notifications and passes them on to the QAnywhere Agent. QAnywhere Agent policies determine if push notifications automatically cause message transmission.

  ☞ For more information, see "Determining when message transmission should occur on the client" on page 36.

See also

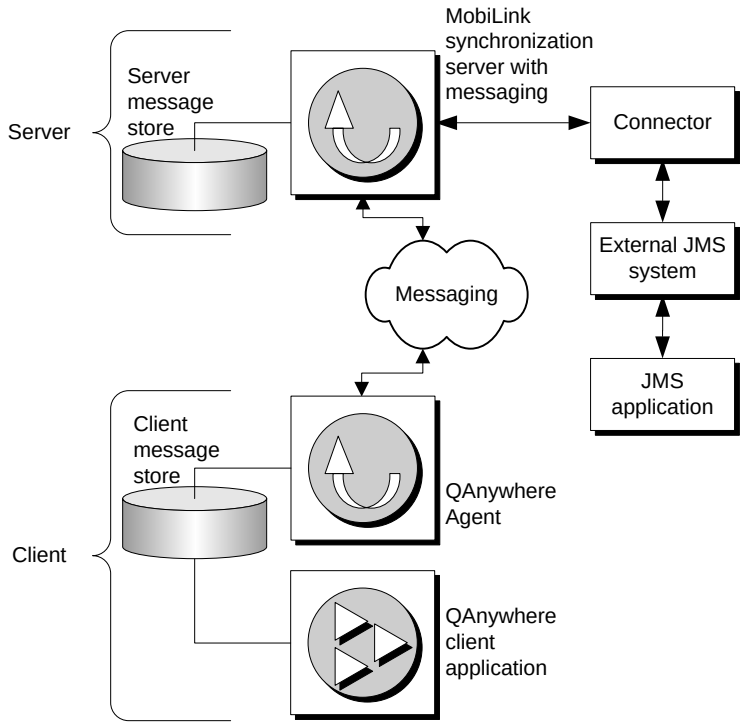
- "Using push notifications" on page 41
- "Receiving messages asynchronously" on page 76
- "Introducing Server-Initiated Synchronization" [*MobiLink - Server-Initiated Synchronization*]

Scenario for messaging with external messaging systems

In addition to exchanging messages among QAnywhere applications, you can exchange messages with systems that have a JMS interface using a specially configured client known as a connector. JMS is the Java Message Service API for adding messaging capabilities to Java applications.

The external messaging system is set up to act like a special client. It has its own address and configuration.

The architecture for messaging with external messaging systems is an extension of the architecture described in "Simple messaging scenario" on page 7. The following diagram shows this architecture:



The component that is added to "Simple messaging scenario" on page 7 in order to enable messaging with an external messaging system is as follows:

- **QAnywhere JMS Connector** The JMS Connector provides an interface between QAnywhere and the external messaging system.

  The JMS Connector is a special QAnywhere client that moves messages between QAnywhere and the external JMS system.

See also

- "JMS Connectors" on page 127
- "Tutorial: Using JMS connectors" on page 141

QAnywhere message delivery

Messages are sent from a client message store to a server message store, and then on to another client message store. QAnywhere does this via queues: a message is put on a queue in the client message store; when it is received by the server message store, it is put on a queue for delivery to one or more client message stores; and when it is received by a client message store, it is put on a queue for pickup.

Once a message is sent, it will be delivered unless one of the following occurs:

- The message expires (only if an expiration is specified).
- The message is cancelled via Sybase Central.
- The device from which the message is sent is lost unrecoverably before it can synchronize with the server message store (or for some other reason, synchronization is impossible).

A message will not be delivered more than once. If an application successfully acknowledges or commits the receipt of a message, then the same message will not be delivered again. There is a possible exception with JMS servers: in the event of the MobiLink server or JMS server crashing, there is a possibility that a message will be delivered twice.

QAnywhere plug-in

The Sybase Central QAnywhere plug-in helps you create and administer your QAnywhere application. With the plug-in, you can:

- Create client and server message stores.
- Create and maintain configuration files for the QAnywhere Agent.
- Browse QAnywhere Agent log files.
- Create or modify destination aliases.
- Create JMS connectors and web service connectors.
- Create and maintain transmission rules files.
- Browse message stores remotely.
- Track messages.

♦ To start the QAnywhere plug-in

1. Start Sybase Central:

   Choose Start ► Programs ► SQL Anywhere 10 ► Sybase Central.
2. From Connections, choose Connect with QAnywhere 10.
3. Specify an ODBC data source name or file, and a user ID and password if required.
4. Click OK.

Quick start

The following steps provide an overview of the tasks required to set up and run QAnywhere messaging.

♦ **To set up and run QAnywhere messaging**

1. Set up a server message store or use an existing MobiLink consolidated database.

   ☞ See "Setting up the server message store" on page 30.

2. Start the MobiLink server with the -m option and a connection to the server message store.

   ☞ See "Starting the MobiLink server for QAnywhere messaging" on page 31.

3. Set up client message stores. These are SQL Anywhere databases that are used to temporarily store messages.

   ☞ See "Setting up the client message store" on page 33.

4. For each client, write a messaging application.

   ☞ See "Writing QAnywhere Client Applications" on page 45.

5. If you want to integrate with an external JMS messaging system, set up JMS messaging for QAnywhere.

   ☞ See "JMS Connectors" on page 127.

6. For each client, start the QAnywhere Agent with a connection to the local client message store.

   ☞ See "Running the QAnywhere Agent" on page 35.

☞ For information about setting up mobile web services, see "Mobile Web Services" on page 181.

Other resources for getting started

- "Tutorial: Exploring TestMessage" on page 15
- "Tutorial: Using JMS connectors" on page 141
- Sample applications are installed to *samples-dir\QAnywhere*. For information about *samples-dir*, see "The samples directory" [*SQL Anywhere Server - Database Administration*].

CHAPTER 2

Tutorial: Exploring TestMessage

Contents

About this chapter

This tutorial explores the capabilities of QAnywhere through a sample client application named TestMessage. QAnywhere applications can run on many devices, such as PDAs, laptops, and tablets, extending application-to-application messaging to these devices. However, for demonstration purposes, this tutorial runs the client on a Windows computer.

About the tutorial

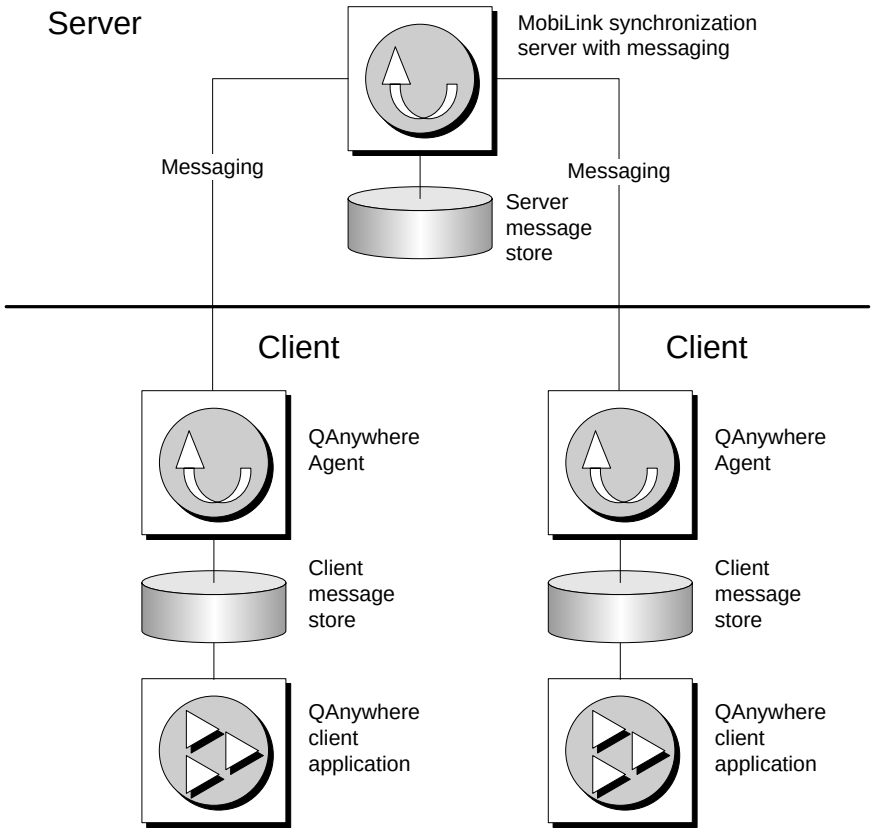
TestMessage is a sample QAnywhere client application. This application demonstrates how you can use QAnywhere to create your own messaging client applications. TestMessage provides a single client-to-client interface to send, receive, and display messages. Being human-readable, text messages provide a useful demonstration of QAnywhere messaging, but QAnywhere provides much more than text messaging. It is a general purpose application-to-application messaging system that provides message-based communication among many clients.

The tutorial is written for a Windows desktop system. While these platforms are convenient for demonstration purposes, you can also use the QAnywhere API to write applications that run on Windows CE devices. Source code is provided for Windows CE for C++, Visual Basic .NET, C#, and Java. There is also a C# version written on the .NET Compact Framework.

Lesson 1: Start MobiLink with messaging

Background

QAnywhere uses MobiLink synchronization to send and receive messages. All messages from one client to another are delivered through a central MobiLink server. The architecture of a typical system, with only two QAnywhere clients, is shown in the following diagram.



The server message store is a database configured for use as a MobiLink consolidated database. The TestMessage sample uses a SQL Anywhere consolidated database as its server message store.

The only tables needed in the server message store are the MobiLink system tables that are included in any supported database that is set up as a MobiLink consolidated database.

The system tables are maintained by MobiLink. A relational database provides a secure, high performance message store. It enables you to easily integrate messaging into an existing data management and synchronization system.

QAnywhere messaging is usually carried out over separate computers, but in this tutorial all components are running on a single computer. It is important to keep track of which activities are client activities and which are server activities.

In this lesson, you carry out actions at the server.

Activity

The MobiLink server can be started with messaging by supplying the -m option, as well as specifying a connection string to the server message store. The TestMessage sample uses a QAnywhere sample database for the server message store. For the TestMessage sample, you can start the MobiLink server for messaging using the command line options, using a sample shortcut in your SQL Anywhere installation, or with the QAnywhere plug-in to Sybase Central.

♦ Start the messaging server

1. From the Windows Start menu, choose Programs ► SQL Anywhere 10 ► MobiLink ► MobiLink with Messaging Sample.

   Alternatively, at a command prompt, navigate to *samples-dir\QAnywhere\server* and type the following command:

   ```
   mlsrv10 -m -c "dsn=QAnywhere 10 Demo" -vcrs -zu+
   ```

   This command line uses the following mlsrv10 options:

Option	Description
-m	The -m option enables messaging. ☞ See "-m option" [*MobiLink - Server Administration*].
-c	The -c option specifies the connection string to the server message store, in this case using the QAnywhere 10.0 Demo ODBC data source. ☞ See "-c option" [*MobiLink - Server Administration*].
-vcrs	The -vcrs option provides verbose logging of server activities, which is useful during development. ☞ See "-v option" [*MobiLink - Server Administration*].
-zu+	The -zu+ option automatically adds user names to the system; this is convenient for tutorial or development purposes but is not normally used in a production environment. ☞ See "-zu option" [*MobiLink - Server Administration*].

2. Move the MobiLink server window to the center of your screen, which represents the server in this tutorial.

Once the MobiLink server is started, you can move on to the next lesson.

Further reading

- "Starting the MobiLink server for QAnywhere messaging" on page 31
- "-m option" [*MobiLink - Server Administration*]
- "Quick start" on page 14

- “Simple messaging scenario” on page 7

Lesson 2: Run the TestMessage application

Background

TestMessage is a simple application that uses QAnywhere to send and receive text messages. Text messaging is used in this tutorial because it provides a simple and accessible demonstration of messaging. QAnywhere is, however, not just a text messaging system; it provides general purpose application-to-application messaging.

In this lesson, you are carrying out activities at a client. Typically, clients run on separate computers from the server.

In this lesson, you start the client message store that is part of the TestMessage sample. In Lesson 3, you will use this message store to send a message to another client message store.

Activity

♦ **To start the QAnywhere Agent with the TestMessage client message store**

1. From the Start menu, choose Programs ► SQL Anywhere 10 ► QAnywhere ► Agent for Client1 Sample.

   This starts an instance of the QAnywhere Agent. This Agent connects to the first TestMessage sample client message store and manages message transmission to and from this message store.

2. Move the first QAnywhere Agent window to the right side of your screen, which represents the first client in this tutorial.

3. From the Start menu, choose Programs ► SQL Anywhere 10 ► QAnywhere ► Agent for Client2 Sample.

   This starts another instance of the QAnywhere Agent. This Agent connects to the second TestMessage sample client message store and manages message transmission to and from this message store.

4. Move the second QAnywhere Agent window to the left side of your screen, which represents the second client in this tutorial.

5. Each of the QAnywhere Agent windows displays a client message store ID, called client1 and client2.

♦ **To start TestMessage**

1. From the Windows Start menu, choose Programs ► SQL Anywhere 10 ► QAnywhere ► TestMessage for Client1 Sample.

   The TestMessage window is displayed. The application is connected to the first TestMessage client message store that you started in the above procedure.

2. Move the TestMessage window to the right side of your screen, together with the first QAnywhere Agent. Both these components belong on the first client.

3. Check the message queue.

From the TestMessage - client 1 Tools menu, choose Options. You will see that the queue name **testmessage** is specified. This is the queue that the TestMessage application is listening on for incoming messages. Do not change this name.

4. From the Windows Start menu, choose Programs ▶ SQL Anywhere 10 ▶ QAnywhere ▶ TestMessage for Client2 Sample.

   The TestMessage window is displayed. The application is connected to the second TestMessage client message store that you started in the above procedure.

5. Move the TestMessage window to the left side of your screen, together with the second QAnywhere Agent. Both these components belong on the second client.

6. Check the message queue.

   From the TestMessage - client 2 Tools menu, choose Options. You will see that the queue name **testmessage** is specified. This is the queue that the TestMessage application is listening on for incoming messages. Do not change this name.

Discussion

You can configure the way that the QAnywhere Agent monitors messages by setting a message transmission policy. This sample starts the QAnywhere Agent using the automatic policy.

- **scheduled** This policy setting instructs the QAnywhere Agent to transmit messages periodically. If you don't specify an interval, the default is 15 minutes.
- **automatic** This default policy setting causes the QAnywhere Agent to transmit messages whenever a message to or from the client message store is ready for delivery.
- **ondemand** This policy setting causes the QAnywhere Agent to transmit messages only when instructed to by an application.
- **custom** In this mode, you provide a set of rules to specify more complicated transmission behavior.

QAnywhere messages are delivered to a QAnywhere address, which consists of a client message store ID and a queue name. The default ID is the computer name on which the QAnywhere Agent is running. Each message store requires its own QAnywhere Agent. Each application can listen to multiple queues, but each queue should be specific to a single application.

Further reading

- “Running the QAnywhere Agent” on page 35
- “Determining when message transmission should occur on the client” on page 36
- “qaagent syntax” on page 145
- “QAnywhere Transmission and Delete Rules” on page 225
- “Writing QAnywhere Client Applications” on page 45
- QAnywhere samples, which are installed to *samples-dir\QAnywhere*. (For more information about *samples-dir*, see “The samples directory” [*SQL Anywhere Server - Database Administration*].)

Lesson 3: Send a message

Background

The TestMessage sample includes two client message stores, which you started in Lesson 1. In this lesson you will send a message from the TestMessage client1 application to the TestMessage client2 application.

Activity

- **To send a message from TestMessage**

1. From the TestMessage - client1 Message menu, choose New. The New Message window appears.
2. In the Destination ID field, enter client2. (Leave testmessage in the Destination Queue field.)
3. Fill out the Subject and Message fields with sample text, and click Send.

   When testing messaging, it is often useful to use the current time as a subject line to make it easy to track individual messages.

   An Alert appears.
4. Read the message.

   Switch to the TestMessage client2 window. Select the message to display its contents in the bottom pane of the window.

Discussion

Like other QAnywhere applications, TestMessage uses the QAnywhere API to manage messages. The QAnywhere API is supplied as a C++ API, a Java API, a Microsoft .NET API, and a SQL API.

Further reading

- "QAnywhere message addresses" on page 50
- "Sending QAnywhere messages" on page 66
- "Message delete rules" on page 239

Lesson 4: Explore the TestMessage client source code

Background

This section of the tutorial takes you on a brief tour of the source code behind the TestMessage client application.

A good deal of the code implements the Windows interface, through which you can send, receive, and view the messages. This portion of the tutorial, however, focuses on the portions of the code given to QAnywhere.

You can find the TestMessage source code in the *samples-dir\QAnywhere*.

Several versions of the TestMessage source code are provided. The following versions are provided for Windows 2000 and Windows XP:

- A C++ version built using the Microsoft Foundation Classes is provided as *Samples\QAnywhere\Desktop\MFC\TestMessage\TestMessage.sln*.
- A Visual Basic .NET version built on the .NET Framework is provided as *Samples\QAnywhere\Desktop\.NET\VB\TestMessage\TestMessage.sln*.
- A C# version built on the .NET Framework is provided as *Samples\QAnywhere\Desktop\.NET\CS\TestMessage\TestMessage.sln*.
- A Java version is provided as *Samples\QAnywhere\Java\TestMessage\TestMessage.java*.

The following version is provided for Pocket PC:

- A C# version built on the .NET Compact Framework is provided as *Samples\QAnywhere\PocketPC\.NET\CS\TestMessage\TestMessage.sln*.

Required software

Visual Studio .NET 2003 or later is required to open the solution files and build the .NET Framework projects and the .NET Compact Framework project.

Exploring the C# source

This section takes you through the C# source code. The two versions are structured in a very similar manner.

Rather than look at each line in the application, this lesson highlights particular lines that are useful for understanding QAnywhere applications. It uses the C# version to illustrate these lines.

1. Open the version of the TestMessage project that you are interested in.

   Double-click the solution file to open the project in Visual Studio .NET. For example, *Samples\QAnywhere\Desktop\.NET\CS\TestMessage\TestMessage.sln* is a solution file. There are several solution files for different environments.

2. Ensure the Solution Explorer is open.

   You can open the Solution Explorer from the View menu.

3. Inspect the Source Files folder.

There are two files of particular importance. The *MessageList* file (*MessageList.cs*) receives messages and lets you view them. The *NewMessage* file (*NewMessage.cs*) allows you to construct and send messages.

4. From the Solution Explorer, open the *MessageList* file.
5. Inspect the included namespaces.

   Every QAnywhere application requires the iAnywhere.QAnywhere.Client namespace. The assembly that defines this namespace is supplied as the DLL *iAnywhere.QAnywhere.Client.dll*. The locations for this file are (relative to your SQL Anywhere installation directory):

   - .NET Framework 1.1: *\Assembly\v1*
   - .NET Framework 2.0: *\Assembly\v2*
   - .NET Compact Framework 1.0: *ce\Assembly\v1*
   - .NET Compact Framework 2.0: *ce\Assembly\v2*

   For your own projects, you must include a reference to this DLL when compiling. The namespace is included using the following line at the top of each file:

   ```
   using iAnywhere.QAnywhere.Client;
   ```

6. Inspect the startReceiver method.

   This method performs initialization tasks that are common to QAnywhere applications:

   - Create a QAManager object.

     ```
     _qaManager =
     QAManagerFactory.Instance.CreateQAManager( null );
     ```

     QAnywhere provides a QAManagerFactory object to create QAManager objects. The QAManager object handles QAnywhere messaging operations: in particular, receiving messages (getting messages from a queue) and sending messages (putting messages on a queue).

     QAnywhere provides two types of manager: QAManager and QATransactionalManager. When using QATransactionalManager, all send and receive operations occur within a transaction, so that either all messages are sent (or received) or none are.

   - Write a method to handle messages.

     The onMessage() method is called by QAnywhere to handle regular non-system messages. The message it receives is encoded as a QAMessage object. The QAMessage class and its children, QATextMessage and QABinaryMessage, provide properties and methods that hold all the information QAnywhere applications need about a message.

     ```
     private void onMessage( QAMessage msg ) {
                 Invoke( new onMessageDelegate( onMessageReceived ),
           new Object [] { msg } );
     }
     ```

     This code uses the Invoke method of the Form to cause the event to be processed on the thread that runs the underlying window so that the user interface can be updated to display the message. This

is also the thread that created the QAManager. With some exceptions, the QAManager can only be accessed from the thread that created it.

- Declare a MessageListener.

```
_receiveListener = new
    QAManager.MessageListener( onMessage );
```

The OnMessage() method is called whenever a message is received by the QAnywhere Agent and placed in the queue that the application listens to.

> **Message listeners and notification listeners**
> Message listeners are different from the Listener component described in "Scenario for messaging with push notifications" on page 9. The Listener component receives notifications, while message listener objects retrieve messages from the queue.

When you set a message listener for the queue, the QAnywhere Manager passes messages that arrive on that queue to that listener. Only one listener can be set for a given queue. Setting with a null listener clears out any listener for that queue.

The MessageListener implementation receives messages asynchronously. You can also receive messages synchronously; that is, the application explicitly goes and looks for messages on the queue, perhaps in response to a user action such as clicking a Refresh button, rather than being notified when messages appear.

Other initialization tasks include:

- Open and start the QAManager object.

```
_qaManager.Open(
        AcknowledgementMode.EXPLICIT_ACKNOWLEDGEMENT );
_qaManager.Start();
```

The AcknowledgementMode enumeration constants determine how the receipt of messages is acknowledged to the sender. The EXPLICIT_ACKNOWLEDGEMENT constant indicates that messages are not acknowledged until a call to one of the QAManager acknowledge methods is made.

- Load any messages that are waiting in the queue.

```
loadMessages();
```

- Assign a listener to a queue for future messages.

The listener was declared in the MessageList_Load() method.

```
_qaManager.SetMessageListener(
  _options.ReceiveQueueName,
  _receiveListener );
```

The Options ReceiveQueueName property contains the string **testmessage**, which is the TestMessage queue as set in the TestMessage Options dialog.

7. Inspect the addMessage() method in the same file.

   This method is called whenever the application receives a message. It gets properties of the message such as its reply-to address, preferred name, and the time it was sent (Timestamp), and displays the information in the TestMessage user interface. The following lines cast the incoming message into a QATextMessage object and get the reply-to address of the message:

   ```
   text_msg = ( QATextMessage )msg;
   from = text_msg.ReplyToAddress;
   ```

   This completes a brief look at some of the major tasks carried out in the *MessageList* file.

8. From the Solution Explorer, open the *NewMessage* file.

9. Inspect the sendMessage() method.

   This method takes the information entered in the New Message dialog and constructs a QATextMessage object. The QAManager then puts the message in the queue to be sent.

   Here are the lines that create a QATextMessage object and set its ReplyToAddress property:

   ```
   qa_manager = MessageList.GetQAManager();
   msg = qa_manager.CreateTextMessage();
   msg.ReplyToAddress = MessageList.getOptions().ReceiveQueueName;
   ```

   Here are the lines that put the message in the queue to be sent. The variable `to` is the destination address, supplied as an argument to the function.

   ```
   qa_manager.PutMessage( dest, msg );
   ```

Further reading

- "QAnywhere C++ API Reference" on page 391
- "iAnywhere.QAnywhere.Client namespace (.NET)" on page 244
- "Writing QAnywhere Client Applications" on page 45
- The TestMessage sample, which is installed to *samples-dir**QAnywhere*. (For information about *samples-dir*, see "The samples directory" [*SQL Anywhere Server - Database Administration*].)

Tutorial cleanup

Shut down all instances of TestMessage, the QAnywhere Agent, and the MobiLink server.

CHAPTER 3

Setting Up QAnywhere Messaging

Contents

About this chapter

This chapter describes how to set up and run QAnywhere messaging.

QAnywhere uses MobiLink synchronization to transport messages. This chapter describes how to set up and run the MobiLink server with messaging.

Setting up server-side components

♦ **Overview of setting up QAnywhere server-side components**

1. Set up a server message store and start it. This can be any MobiLink consolidated database.

   ☞ See "Setting up the server message store" on page 30.

2. Start mlsrv10 with the -m option and a connection to the server message store.

   ☞ See "Starting the MobiLink server for QAnywhere messaging" on page 31.

3. Add client user names to the server message store.

   ☞ See "Registering client user names" on page 31.

Setting up the server message store

The server message store is a relational database on the server that temporarily stores messages until they are transmitted to a client message store, web service or JMS system. Messages are exchanged between clients via the server message store.

A server message store is a MobiLink consolidated database, and so can be any RDBMS that MobiLink supports (SQL Anywhere, Adaptive Server Enterprise, Microsoft SQL Server, Oracle, or DB2). You can create a new database for this purpose, or use an existing database.

To set up a database to use as a MobiLink consolidated database (and hence a server message store), you must run a setup script.

☞ See "Setting up a consolidated database" [*MobiLink - Server Administration*].

☞ For information about creating SQL Anywhere databases, see "The Initialization utility" [*SQL Anywhere Server - Database Administration*].

If you are using a SQL Anywhere database that was created before version 9.0.2, it must be upgraded.

☞ For information on upgrading your database, see "Upgrading to SQL Anywhere 10" [*SQL Anywhere 10 - Changes and Upgrading*].

Example

To create a SQL Anywhere database called *qanytest.db*, type the following at a command prompt:

```
dbinit -n -s qanytest.db
```

Run the MobiLink setup script on the database:

```
%SQLANY10%\MobiLink\setup\syncsa.sql
```

This database is ready to use as a server message store.

Starting the MobiLink server for QAnywhere messaging

QAnywhere uses MobiLink synchronization to transport messages. To use QAnywhere messaging, you must start the MobiLink server (mlsrv10) with the following options:

- **-c *connection-string*** Specifies the connection string to connect to the server message store.

  ☞ See "-c option" [*MobiLink - Server Administration*].

- **-m** Enables QAnywhere messaging.

  ☞ See "-m option" [*MobiLink - Server Administration*].

☞ You can also use other MobiLink server options to customize your operations. For more information, see "mlsrv10 syntax" [*MobiLink - Server Administration*].

Notes

- If you are integrating with a JMS messaging system, there are other options you must specify when you start the MobiLink server.

  See "Starting the MobiLink server for JMS integration" on page 131.

Example

To start QAnywhere messaging when you are using a server message store called *qanyserv.db*, navigate to *samples-dir\QAnywhere\server* and type the following at a command prompt (all on one line):

```
mlsrv10 -m
   -c "dsn=QAnywhere 10 Demo"
```

For information about *samples-dir*, see "The samples directory" [*SQL Anywhere Server - Database Administration*].

Registering client user names

Each client message store has a unique ID that identifies it. In addition, the client message store has a MobiLink user name that you can optionally use to authenticate your client message store with the MobiLink server. You can specify a MobiLink user name with the qaagent -mu option, or if you do not, one is created with the same name as your client message store ID.

You must register the MobiLink user name with the server message store. There are several methods for doing this:

- Use the mluser utility.

  For more information, see "MobiLink user authentication utility [mluser]" [*MobiLink - Server Administration*].

- Use MobiLink Admin mode in Sybase Central.
- Specify the -zu+ option with mlsrv10. In this case, any existing MobiLink users that have not been added to the consolidated database are added when they first synchronize. This is useful during development, but is not recommended for production environments.

  For more information, see "-zu option" [*MobiLink - Server Administration*].

☞ For more information about MobiLink user names, see "About MobiLink users" [*MobiLink - Client Administration*].

☞ For more information about client message store IDs, see "-id option" on page 152.

Setting properties for clients on the server

As a convenience, you can use the QAnywhere plug-in to set properties for clients on the server. When you do this, you need to add the client to the server. The first time you synchronize to the client, the properties will be downloaded.

♦ To add a client user name using Sybase Central

1. Start Sybase Central:
   - Choose Start ► Programs ► SQL Anywhere 10 ► Sybase Central.
   - From Connections, choose Connect with QAnywhere 10.
   - Specify an ODBC data source name or file, and a user ID and password if required. Click OK.
2. Choose File ► New ► Client.
3. Type the name of the client.
4. Click OK.

See also

- "Registering client user names" on page 31

Setting up client-side components

♦ **Overview of setting up client-side components**

1. Create a SQL Anywhere database and initialize it as a client message store.

   ☞ See "Setting up the client message store" on page 33.

2. Write client applications.

   ☞ See "Writing QAnywhere Client Applications" on page 45.

3. Start the QAnywhere Agent.

   ☞ See "Running the QAnywhere Agent" on page 35.

Setting up the client message store

The client message store is a SQL Anywhere database on the remote device. The application connects to this message store using the QAnywhere API.

Using a relational database as a message store provides a secure and high-performance store.

☞ See also: "Creating a secure client message store" on page 176.

♦ **To create a client message store**

1. Create a SQL Anywhere database.

   Note: QAnywhere does not use or manage a transaction log. You can create a database with a transaction log if you want to use one for backups or other uses, but it may grow very large. In most cases, it is recommended that your client message store not have a transaction log.

   ☞ See "Creating a database" [*SQL Anywhere Server - SQL Usage*].

2. Initialize each client message store by running the QAnywhere Agent (qaagent) with the following options:

   ♦ **-c option** to specify a connection string to the database you just created.

   ☞ See "-c option" on page 148.

   ♦ **-si option** to initialize the database. The -si option creates a default database user and password. The QAnywhere agent shuts down after initializing the database.

   ☞ See "-si option" on page 169.

   ♦ **-id option** optionally, if you want to pre-assign a client message store ID.

☞ See "Creating client message store IDs" on page 34 and "-id option" on page 152.

- **-mu option** optionally, if you want to create a user name to use for authentication with the MobiLink server. If you do not use -mu at this point, you can specify it any time you start the QAnywhere Agent and the name will be created if it does not already exist.

3. If you used the -mu option to create a user name, you need to add the name to the server message store. This can be done automatically using the mlsrv10 -zu+ option, or can be done in other ways.

   ☞ See "Registering client user names" on page 31.

4. Change the default passwords and take other steps to ensure that the client message store is secure.

   ☞ See "Creating a secure client message store" on page 176.

You can also upgrade a client message store that was created in a previous version of QAnywhere.

☞ See "-su option" on page 170 and "-sur option" on page 171.

Creating client message store IDs

Client message store IDs can be set in various ways:

- You can specify the ID with the qaagent -id option when you use the qaagent -si option to initialize the client message store.
- You can specify the ID with the -id option the first time you run qaagent after you initialize the client message store.
- If you do not specify an ID in either of the previous ways, then the first time you run qaagent after you run qaagent with -si, the device name is assigned as the client message store ID. The ID appears in the QAnywhere Agent window.

☞ For more information, see "QAnywhere Agent" on page 143.

Example of creating a client message store

The following command creates a SQL Anywhere database called qanyclient.db. (The dbinit -i and -s options are not required, but are good practice on small devices.)

```
dbinit -i -n -s qanyclient.db
```

The following command connects to *qanyclient.db* and initializes it as a QAnywhere client database:

```
qaagent -si -c "DBF=qanyclient.db"
```

☞ For more information about dbinit, see "Initialization utility (dbinit)" [*SQL Anywhere Server - Database Administration*].

Running the QAnywhere Agent

The QAnywhere Agent (qaagent) is a separate process running on the client device. It monitors the client message store and determines when message transmission should occur.

The QAnywhere Agent transmits messages between the server message store and the client message store. You can run multiple instances of the QAnywhere Agent on the same device, but each instance must be connected to its own message store. Each message store must have a unique message store ID.

You can run the Agent on the command line using command line options. At a minimum, you need to start the Agent with the following options:

- **Connection parameters** to connect to the client message store.

  In the Agent command file properties dialog, this is the information on the Message Store tab.

  In the qaagent command line, this is specified with the -c option.

  ☞ See "-c option" on page 148.

- **Client message store ID** to identify the client message store. The first time you run qaagent after you have initialized a client message store, you can optionally use this option to name the message store; if you do not, the device name is used by default. After that, you must use the -id option every time you start qaagent to specify a unique client message store ID.

  In the Agent Command file properties dialog, this is specified on the General tab.

  In the qaagent command line, this is specified with the -id option.

  ☞ See "-id option" on page 152.

- **Network protocol and protocol options** to connect to the MobiLink server. This is required unless the MobiLink server is running on the same device as the QAnywhere agent and default communication parameters are used.

  In the Agent command file properties dialog, this is the server information on the Server tab.

  In the qaagent command line, this is the -x option.

  ☞ See "-x option" on page 173.

☞ For a complete list of all QAnywhere Agent options, see "qaagent syntax" on page 145.

Starting qaagent on Windows CE

On Windows CE, you might want to start the QAnywhere Agent in quiet mode by specifying the -qi option.

☞ See "-qi option" on page 168.

Running multiple instances of QAnywhere Agent

You can run multiple instances of qaagent on a device. However, when you start a second instance:

- The second instance of QAnywhere Agent must be started with a different database file.
- You must specify a unique message store ID using the -id option.

  See "-id option" on page 152.

Stopping QAnywhere Agent

To stop the QAnywhere Agent, click Shutdown on the console.

When you start the QAnywhere Agent in quiet mode, you can only stop it by running **qastop**.

☞ See "-qi option" on page 168.

Processes started by QAnywhere Agent

The QAnywhere Agent starts other processes to handle various messaging tasks. Each of these processes is managed by the QAnywhere Agent, and does not need to be managed separately. When you start the QAnywhere Agent, it spawns the following processes:

- **dbmlsync** The dbmlsync executable is the MobiLink synchronization client. MobiLink synchronization is used to send and receive messages, so the dbmlsync executable is required.
- **dblsn** The dblsn executable is the Listener utility. It receives push notifications. If you are not using push notifications, you do not need to supply the dblsn executable when you deploy your application, and you must run qaagent with -push **none**.

  ☞ See "-push option" on page 165.
- **database server** The client message store is a SQL Anywhere database. QAnywhere Agent requires the SQL Anywhere database server to run the database. For Windows CE, the database server is *dbsrv10.exe*. For Windows, the database server is the personal database server *dbeng10.exe*.

  The QAnywhere Agent can spawn a database server or connect to a running server, depending on the communication parameters that you specify in the qaagent -c option.

  ☞ See "-c option" on page 148.

Deploying QAnywhere Agent

☞ For deployment information, see "Deploying QAnywhere applications" on page 89.

Determining when message transmission should occur on the client

On the client side, you determine when message transmission should occur by specifying **policies**. A policy tells the QAnywhere Agent when a message should be moved from the client message store to the server message store. If you do not specify a policy, transmission occurs automatically when a message is queued for delivery to the server by default. There are three pre-defined policies: scheduled, automatic, and ondemand, as well as a custom policy.

You can speciy policies in two ways:

- Using the QAnywhere plug-in in Sybase Central, choose the task Create an Agent Command File. Policies are specified on the General tab of the command file Properties dialog.

  To specify custom properties, you must also choose the task Create an Agent Rule File. This task creates a file with a *.qar* extension; this extension is a Sybase Central convention.

- Run qaagent on the command line using the -policy option. For custom policies, create a rules file and specify it.

Scheduled policy

The scheduled policy instructs the Agent to perform a transmission at a specified time interval.

To invoke a schedule, choose **scheduled** in the command file Properties dialog or specify the keyword when you start the QAnywhere Agent:

qaagent –policy scheduled [*interval*] ...

where *interval* is in seconds.

The default is 900 seconds (15 minutes).

When a schedule is specified, every *n* seconds the Agent performs message transmission if any of the following conditions are met:

- New messages were placed in the client message store since the previous time interval elapsed.
- A message status change occurred since the previous time interval elapsed. This typically occurs when a message is acknowledged by the application.

  For more information about acknowledgement, see "AcknowledgementMode enumeration" on page 244 for .NET clients, "AcknowledgementMode class" on page 392 for C++ clients and "Interface AcknowledgementMode" on page 504 for Java clients.
- A push notification was received since the previous time interval elapsed.
- A network status change notification was received since the previous time interval elapsed.
- Push notifications are disabled.

You can call the trigger send receive method to override the time interval. It forces message transmission to occur before the time interval elapses.

For more information about trigger send receive, see:

- .NET: "TriggerSendReceive method" on page 309
- C++: "triggerSendReceive function" on page 461
- Java: "setStringStoreProperty method" on page 564
- SQL: "ml_qa_triggersendreceive" on page 672

Automatic policy

The automatic policy attempts to keep the client and server message stores as up-to-date as possible.

When using the automatic policy, message transmission is performed when any of the following conditions occurs:

- PutMessage() is called.

  For more information about PutMessage see:

  - .NET: "PutMessage method" on page 298
  - C++: "putMessage function" on page 452
  - Java: "getStringStoreProperty method" on page 557
  - SQL: "ml_qa_putmessage" on page 672

- A message status changes has occurred. This typically occurs when a received message is acknowledged by the application.

  For more information about acknowledgement, see:

  - .NET: "AcknowledgementMode enumeration" on page 244
  - C++: "AcknowledgementMode class" on page 392
  - Java: "Interface AcknowledgementMode" on page 504
  - SQL: all messaging using the SQL API is transactional

- A Push Notification is received.

  For more information about push notifications, see "Using push notifications" on page 41.

- A Network Status Change Notification is received.

  For more information, see "Notifications of push notification" on page 53.

- TriggerSendReceive() is called.

  For more information about TriggerSendReceive, see:

  - .NET: "TriggerSendReceive method" on page 309
  - C++: "triggerSendReceive function" on page 461
  - Java: "setStringStoreProperty method" on page 564
  - SQL: "ml_qa_triggersendreceive" on page 672

Ondemand policy

The ondemand policy causes message transmission to occur only when instructed to do so by an application.

An application forces a message transmission to occur by calling TriggerSendReceive().

When the agent receives a Push Notification or a Network Status Change Notification, a corresponding message is sent to the system queue. This allows an application to detect these events and force a message transmission by calling TriggerSendReceive().

☞ For more information about TriggerSendReceive, see:

- .NET: "TriggerSendReceive method" on page 309
- C++: "triggerSendReceive function" on page 461
- Java: "setStringStoreProperty method" on page 564
- SQL: "ml_qa_triggersendreceive" on page 672

☞ For more information about handling push notifications and network status changes, see "System queue" on page 51.

Custom policy

A custom policy allows you to define when message transmission occurs and which messages to send in the message transmission. The custom policy is defined by a set of transmission rules.

Each rule is of the following form:

schedule = *condition*

where *schedule* defines when *condition* is evaluated. For more information, see "Rule syntax" on page 226.

All messages satisfying *condition* are transmitted. In particular, if *schedule* is automatic, the condition is evaluated when any of the following conditions occurs:

- PutMessage() is called.

  For more information about PutMessage, see:

  - .NET: "PutMessage method" on page 298
  - C++: "putMessage function" on page 452
  - Java: "getStringStoreProperty method" on page 557
  - SQL: "ml_qa_putmessage" on page 672

- A message status change has occurred. This typically occurs when a message is acknowledged by the application.

  For more information about acknowledgement, see:

  - .NET: "AcknowledgementMode enumeration" on page 244
  - C++: "AcknowledgementMode class" on page 392
  - Java: "Interface AcknowledgementMode" on page 504
  - SQL: all messaging using the SQL API is transactional

- A Push Notification is received.

  For more information about push notifications, see "Using push notifications" on page 41.

- A Network Status Change Notification is received.

- TriggerSendReceive ( ) is called.

  For more information about TriggerSendReceive, see:

  - .NET: "TriggerSendReceive method" on page 309
  - C++: "triggerSendReceive function" on page 461
  - Java: "setStringStoreProperty method" on page 564
  - SQL: "ml_qa_triggersendreceive" on page 672

☞ For more information about transmission rules, see "Message transmission rules" on page 234.

☞ For more information about creating policies, see "-policy option" on page 163.

Using push notifications

A push notification is a special message delivered from the server message store to a QAnywhere client that prompts the client to initiate a message transmission. Push notification is on by default but is optional. Push notifications introduce extra components to the QAnywhere architecture:

- At the server, a QAnywhere Notifier sends push notifications.
- At the client, a QAnywhere Listener receives these push notifications and passes them on to the QAnywhere Agent.
- At the client, a notification of each push notification is sent to the system queue.

If you use the scheduled or automatic QAnywhere Agent policies, push notifications automatically cause clients to initiate message transmission. If you use the ondemand policy, you must handle push requests manually using an event handler.

☞ For more information about manually handling push notifications, see "Notifications of push notification" on page 53.

☞ For more information about QAnywhere Agent policies, see "Determining when message transmission should occur on the client" on page 36.

Push notifications are enabled by default: the qaagent -push option is by default set to connected. In connected mode, push notifications are sent over TCP/IP persistent connection.

If you are using UDP, push notifications are likely to work without any configuration, but due to a limitation in the UDP implementation of ActiveSync, they will not work with ActiveSync.

See also

- "Scenario for messaging with push notifications" on page 9
- "Notifications of push notification" on page 53
- "-push option" on page 165

Setting up a failover mechanism

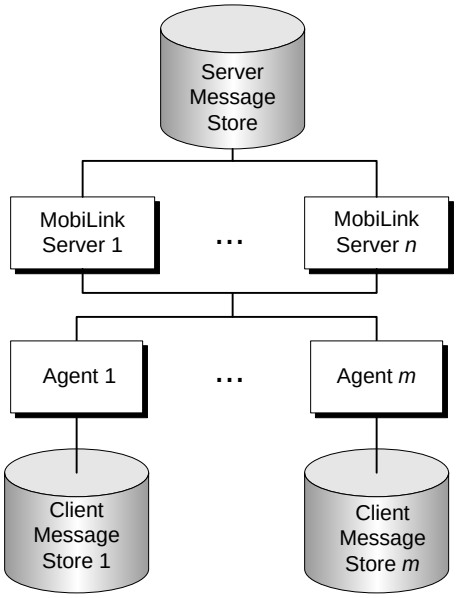
QAnywhere applications can be set up with a failover mechanism so that alternate MobiLink servers can be used if one fails. In order to support failover, each QAnywhere Agent must be started with a list of MobiLink servers. The first MobiLink server specified in the list is the primary server. The remaining servers in the list are alternate servers.

For example, running the following command on the remote device will start the QAnywhere Agent with one primary server and one alternate server:

```
qaagent -x tcpip(host=ml1.ianywhere.com)
        -x tcpip(host=ml2.ianywhere.com)
```

Each QAnywhere Agent can have a different primary server.

The following diagram describes a failover configuration in which you have multiple MobiLink servers and multiple QAnywhere agents. You have multiple client message stores, but all MobiLink servers are connected to the same server-side message store.



This configuration has the following characteristics:

- When a message transmission occurs, all messages in the server message store are delivered to the client message store regardless of the server that the QAnywhere Agent is connected to.
- Push Notifications are sent to a QAnywhere Agent only when the QAnywhere Agent is connected to its primary server.
- There is a single point of failure. If the machine with the server message store is unavailable, no messaging can take place.

By default, when you set up failover MobiLink servers, the QAnywhere Agent always tries an alternate server immediately upon a failure to reach the primary server. If you want to change this default behavior, you can use the QAnywhere Agent -fr option to cause the QAnywhere Agent to try the primary server again before going to the alternate server, and to specify the number of times it should retry. You can use the -fd option to specify the amount of time between retries of the primary server.

The -fr and -fd options apply only to the primary server. If a connection to the primary server cannot be established after the specified number of attempts, the QAnywhere Agent tries to connect to an alternate server. The Agent attempts to connect to each alternate server only once. An error is issued if the Agent cannot establish a connection to an alternate server.

See also

- "-x option" on page 173
- "-fd option" on page 150
- "-fr option" on page 151
- "Running the QAnywhere Agent" on page 35

CHAPTER 4

Writing QAnywhere Client Applications

Contents

About this chapter

This chapter describes how to write QAnywhere client applications.

Introduction to the QAnywhere interfaces

QAnywhere client applications manage the receiving and sending of QAnywhere messages. The applications can be written using one of several QAnywhere APIs:

- QAnywhere .NET API
- QAnywhere C++ API
- QAnywhere Java API
- QAnywhere SQL API

You can use a combination of client types in your QAnywhere system. For example, messages that are generated using QAnywhere SQL can also be received by a client created using the APIs for .NET, C++, or Java. If you have configured a JMS connector on your server, the messages can also be received by JMS clients. Similarly, QAnywhere SQL can be used to receive messages that were generated by QAnywhere .NET, C++, Java, or JMS clients.

QAnywhere .NET API

The QAnywhere .NET API is a programming interface for deployment to Windows computers using the Microsoft .NET Framework and to handheld devices running the Microsoft .NET Compact Framework. The QAnywhere .NET API is provided as the iAnywhere.QAnywhere.Client namespace.

QAnywhere supports Microsoft Visual Studio .NET 2003 and 2005.

Note
In this document, code samples for the .NET API use the C# programming language, but the API can be accessed using any programming language that Microsoft .NET supports.

Versions of the TestMessage sample application are written in Java, C#, and Visual Basic.NET. There is also a .NET compact framework sample.

☞ For more information about the .NET version of the TestMessage sample application, see "Lesson 4: Explore the TestMessage client source code" on page 23.

☞ For more information about the QAnywhere .NET API, see "iAnywhere.QAnywhere.Client namespace (.NET)" on page 244.

QAnywhere C++ API

The QAnywhere C++ API supports Microsoft Visual C++ 6.0, Microsoft Visual Studio .NET 2003, Microsoft eMbedded Visual C++ 3.0, Microsoft eMbedded Visual C++ 4.0, and Visual Studio 2005.

The QAnywhere C++ API consists of the following files:

- A set of header files (the main one being *qa.hpp*) located in the *QAnywhere\h* subdirectory of your SQL Anywhere installation.

- An import library (*qany10.lib*) located in the *\ce\arm.30\lib*, *\ce\arm.50\lib*, *QAnywhere\lib*, and *ce\x86.30\lib* subdirectories of your SQL Anywhere installation.
- A run-time DLL (*qany10.dll*) located in the *win32*, *ce\arm.30*, and *ce\x86.30* subdirectories of your SQL Anywhere installation.

Your source code file must include the header file in order to access the API. The import library is used to link your application to the run-time DLL. The run-time DLL must be deployed with your application.

A version of the TestMessage sample application written in C++ is supplied in *samples-dir\QAnywhere*. (For information about *samples-dir*, see "The samples directory" [*SQL Anywhere Server - Database Administration*].)

☞ For more information about the QAnywhere C++ API, see "QAnywhere C++ API Reference" on page 391.

QAnywhere Java API

The QAnywhere Java API supports JRE 1.4.2 and 1.5.0.

The QAnywhere Java API consists of the following files:

- API reference material, available in this book or in Javadoc format in the *\docs\\javadocs\QAnywhere* subdirectory of your SQL Anywhere 10 installation.
- Runtime DLLs (*qany10jni.dll and qany10.dll*), located in the *win32* subdirectory of your SQL Anywhere 10 installation.
- An archive of the class files (*qaclient.jar*), located in the *java* subdirectory of your SQL Anywhere 10 installation.

The class file archive must be included in your path when you compile your application. The run-time DLLs must be deployed with your application.

A version of the TestMessage sample application written in Java is supplied in *\samples-dir\QAnywhere \Desktop\J2SE*. (For information about *samples-dir*, see "The samples directory" [*SQL Anywhere Server - Database Administration*].)

☞ For more information about the QAnywhere Java API, see "QAnywhere Java API Reference" on page 503.

QAnywhere SQL API

The QAnywhere SQL API is a set of stored procedures that implement a messaging API in SQL. Using the QAnywhere SQL API, you can create messages, set or get message properties and content, send and receive messages, trigger message synchronization, and set and get message store properties.

☞ For more information about QAnywhere SQL, see "QAnywhere SQL API Reference" on page 635.

JMS connector

QAnywhere includes a JMS connector that provides connectivity between QAnywhere and JMS applications.

For more information, see:

- "Scenario for messaging with external messaging systems" on page 11
- "Introduction" on page 128
- "Tutorial: Using JMS connectors" on page 141

Mobile web services connector

QAnywhere includes a mobile web services connector for messaging between QAnywhere and web services.

☞ For more information, see "Mobile Web Services" on page 181.

Overview of writing a client application

♦ **To build a client application**

1. Initialize the appropriate QAnywhere API. See:
   - ♦ "Setting up .NET applications" on page 54
   - ♦ "Setting up C++ applications" on page 56
   - ♦ "Setting up Java applications" on page 58
   - ♦ "Setting up SQL applications" on page 59
2. Set QAnywhere manager configuration properties. See "QAnywhere manager configuration properties" on page 62.
3. Write application code and compile. See the following sections:
   - ♦ "Message headers and message properties" on page 206
   - ♦ "Client message store properties" on page 215
   - ♦ "Sending QAnywhere messages" on page 66
   - ♦ "Receiving QAnywhere messages" on page 75
   - ♦ "Reading very large messages" on page 80
   - ♦ "Implementing transactional messaging" on page 68
   - ♦ "Shutting down QAnywhere" on page 88
4. Deploy the application to the target device.

   ☞ See "Deploying QAnywhere applications" on page 89.

QAnywhere message addresses

A QAnywhere message destination has two parts, the client message store ID and the application queue name:

id\queue-name

The queue name is specified inside the application, and must be known to instances of the sending application on other devices. For information about client message store IDs, see "Setting up the client message store" on page 33.

When constructing addresses as strings in an application, be sure to escape the backslash character if necessary. Follow the string escaping rules for the programming language you are using. If your JMS destination contains a backslash, you must escape it with another backslash.

System queue

Notifications and network status changes are both sent to QAnywhere applications as **system messages**. System messages are the same as other messages, but are received in a separate queue named **system**.

☞ See "System queue" on page 51.

Sending a message to a JMS connector

A QAnywhere-to-JMS destination address has two parts:

- The connector address. This is the value of the ianywhere.connector.address property.

  For more information, see "JMS connector properties" on page 132.

- The JMS queue name. This is a queue that you create using your JMS administration tools.

The form of the destination address is:

connector-address\JMS-queue-name

For more information about addressing messages in a JMS application, see:

- "Addressing QAnywhere messages meant for JMS" on page 137
- "Addressing JMS messages meant for QAnywhere" on page 139
- "JMS Connectors" on page 127

Destination aliases

A **destination alias** is a list of message addresses and other destination aliases. When a message is sent to a destination alias, it is sent to all members of the list.

A member of a destination alias can have a delivery condition associated with it. Only messages that match the condition are forwarded to the corresponding member.

Example

Define a destination alias called all_clients with members client1 and client2.

Define the following delivery condition for client1:

```
ias_Priority=1
```

Define the following delivery condition for client2:

```
ias_Priority=9
```

Only messages with priority 1 are sent to client1 and those with priority 9 are sent to client2.

Creating destination aliases

You can create and manage a destination alias using the following methods:

- Server management requests

  See "Creating destination aliases using server management requests" on page 117.

- Sybase Central

  See "Creating destination aliases using Sybase Central" on page 51.

Creating destination aliases using Sybase Central

You can use Sybase Central to create or modify a destination alias.

- **To create a destination alias using Sybase Central**

1. Start Sybase Central:
   - Choose Start ► Programs ► SQL Anywhere 10 ► Sybase Central.
   - Choose Connections ► Connect to QAnywhere 10.
   - Specify an ODBC data source name or file, and a user ID and password if required.
   - Click OK.
2. Choose File ► New ► Destination Alias.
3. In the Alias field, type a name for the alias.
4. In the Destinations field, type the name of each destination on its own line.
5. Click OK.

System queue

A special queue called **system** exists to receive QAnywhere system messages. There are two types of message that are sent to the system queue:

- "Network status notifications" on page 52
- "Notifications of push notification" on page 53

Example

The following C# code processes system and normal messages and can be useful if you are using an ondemand policy. It assumes that you have defined the message handling methods onMessage() and onSystemMessage() that implement the application logic for processing the messages.

```
// Declare the message listener and system listener.
private QAManager.MessageListener _receiveListener;
private QAManager.MessageListener _systemListener;
...

// Create a MessageListener that uses the appropriate message handlers.
_receiveListener = new QAManager.MessageListener( onMessage );
_systemListener = new QAManager.MessageListener( onSystemMessage );
...

// Register the message handler.
mgr.SetMessageListener( queue-name, _receiveListener );
mgr.SetMessageListener( "system", _systemListener );
```

The system message handler may query the message properties to identify what information it contains. The message type property indicates if the message holds a network status notification. For example, for a message `msg`, you could perform the following processing:

```
msg_type = (MessageType)msg.GetIntProperty( MessageProperties.MSG_TYPE );
if( msg_type == MessageType.NETWORK_STATUS_NOTIFICATION ) {
  // Process a network status change.
  mgr.TriggerSendReceive( );
} else if ( msg_type == MessageType.PUSH_NOTIFICATION ) {
  // Process a push notification.
  mgr.TriggerSendReceive( );
} else if ( msg_type == MessageType.REGULAR ) {
  // This message type should not be received on the
  // system queue. Take appropriate action here.
}
```

Network status notifications

When there is a change in network status, a message of type NETWORK_STATUS_NOTIFICATION is sent to the system queue. It has an expiry of one minute. This expiry time cannot be changed.

When a device goes into network coverage or out of network coverage, a message is sent to the system queue that contains the following information:

- **ias_Adapters** String. A list of network adapters that can be used to connect to the MobiLink server. The list is delimited by a vertical bar. This property can be read but should not be set.

  For more information, see:

  - .NET: "ADAPTER field" on page 246
  - C++: "ADAPTER variable" on page 395

- Java: “ADAPTERS variable” on page 507

- **ias_RASNames** String. A list of network names that can be used to connect to the MobiLink server. The list is delimited by a vertical bar.

  For more information, see:

  - .NET: “RASNAMES field” on page 251
  - Java: “RASNAMES variable” on page 509
  - C++: “RASNAMES variable” on page 399

- **ias_NetworkStatus** Int. The state of the network connection. The value is 1 if connected, 0 otherwise.

  For more information, see:

  - .NET: “NETWORK_STATUS field” on page 249
  - Java: “NETWORK_STATUS variable” on page 508
  - C++: “NETWORK_STATUS variable” on page 397

Monitoring network availability

You can use network status notifications to monitor network availability and take action when a device comes into coverage. For example, use the ondemand policy and call QAManagerBase triggerSendReceive when a system queue message is received of type NETWORK_STATUS_NOTIFICATION with ias_NetworkStatus=1.

See also

- ias_MessageType in “Pre-defined message properties” on page 209
- “System queue” on page 50

Notifications of push notification

A message of type PUSH_NOTIFICATION is sent to the system queue when a push notification is received from the server. This message is a notification that messages are queued on the server. It has an expiry of one minute. This expiry time cannot be changed.

This type of system message is useful if you are using the ondemand policy. For example, you can call QAManagerBase triggerSendReceive when a system queue message is received of type PUSH_NOTIFICATION.

See also

- “Scenario for messaging with push notifications” on page 9
- “Using push notifications” on page 41
- “System queue” on page 50
- “Receiving messages asynchronously” on page 76
- ias_MessageType in “Pre-defined message properties” on page 209
- .NET: “MessageProperties class” on page 245
- C++: “MessageProperties class” on page 394
- Java: “Interface MessageProperties” on page 505

Initializing a QAnywhere API

Before you can send or receive messages using QAnywhere, you must complete the following initialization tasks.

Setting up .NET applications

Before you can send or receive messages using QAnywhere .NET clients, you must complete the following initialization tasks.

You must make two changes to your Visual Studio .NET project to be able to use it:

- Add a reference to the QAnywhere .NET DLL. Adding a reference tells Visual Studio.NET which DLL to include to find the code for the QAnywhere .NET API.
- Add a line to your source code to reference the QAnywhere .NET API classes. In order to use the QAnywhere .NET API, you must add a line to your source code to reference the data provider. You must add a different line for C# than for Visual Basic.NET.

In addition, you must initialize the QAnywhere .NET API.

♦ To add a reference to the QAnywhere .NET API in a Visual Studio .NET project

1. Start Visual Studio .NET and open your project.
2. In the Solution Explorer window, right-click the References folder and choose Add Reference from the popup menu.

   The Add Reference dialog appears.
3. On the .NET tab, click Browse to locate iAnywhere.QAnywhere.Client.dll. The default locations are (relative to your SQL Anywhere installation directory):

   - .NET Framework 1.1: *\Assembly\v1*
   - .NET Framework 2.0: *\Assembly\v2*
   - .NET Compact Framework 1.0: *ce\Assembly\v1*
   - .NET Compact Framework 2.0: *ce\Assembly\v2*

   Select the DLL and click Open.
4. You can verify that the DLL is added to your project. Open the Add Reference dialog and then click the .NET tab. iAnywhere.QAnywhere.Client.dll appears in the Selected Components list. Click OK to close the dialog.

Referencing the data provider classes in your source code

♦ To reference the QAnywhere .NET API classes in your code

1. Start Visual Studio .NET and open your project.

2. If you are using C#, add the following line to the list of using directives at the beginning of your file:

```
using iAnywhere.QAnywhere.Client;
```

3. If you are using Visual Basic .NET, add the following line to the list of imports at the beginning of your file:

```
Imports iAnywhere.QAnywhere.Client
```

This line is not strictly required. However, it allows you to use short forms for the QAnywhere classes. Without it, you can still use the fully qualified class name in your code. For example:

```
iAnywhere.QAnywhere.Client.QAManager
mgr =
 new iAnywhere.QAnywhere.Client.QAManagerFactory.Instance.CreateQAManager
(
"qa_manager.props" );
```

instead of

```
QAManager mgr = QAManagerFactory.Instance.CreateQAManager(
 "qa_manager.props" );
```

♦ To initialize the QAnywhere .NET API

1. Include the iAnywhere.QAnywhere.Client namespace, as described in the previous procedure.

```
using iAnywhere.QAnywhere.Client;
```

2. Create a QAManager object.

For example, to create a default QAManager object, invoke CreateQAManager with null as its parameter:

```
QAManager mgr;
mgr = QAManagerFactory.Instance.CreateQAManager( null );
```

> **Tip**
> For maximum concurrency benefits, multi-threaded applications should create a QAManager for each thread. See "Multi-threaded QAManager" on page 61.

☞ For more information about QAManagerFactory, see "QAManagerFactory class" on page 310.

You can alternatively create a QAManager object that is customized using a properties file. The properties file is specified in the CreateQAManager method:

```
mgr = QAManagerFactory.Instance.CreateQAManager(
  "qa_mgr.props" );
```

where *qa_mgr.props* is the name of the properties file that resides on the remote device.

3. Initialize the QAManager object. For example:

```
mgr.Open(
   AcknowledgementMode.EXPLICIT_ACKNOWLEDGEMENT);
```


The argument to the open method is an acknowledgement mode, which indicates how messages are to be acknowledged. It must be one of IMPLICIT_ACKNOWLEDGEMENT or EXPLICIT_ACKNOWLEDGEMENT. With implicit acknowledgement, messages are acknowledged as soon as they are received by the client. With explicit acknowledgement, you must call the acknowledgement method on the QAManager to acknowledge the message.

☞ For more information about acknowledgement modes, see "AcknowledgementMode enumeration" on page 244.

You are now ready to send messages.

Instead of creating a QAManager, you can create a QATransactionalManager. See "Implementing transactional messaging for .NET clients" on page 68.

See also

- "iAnywhere.QAnywhere.Client namespace (.NET)" on page 244

Setting up C++ applications

Before you can send or receive messages using QAnywhere C++ clients, you must complete the following initialization tasks.

♦ To initialize the QAnywhere C++ API

1. Include the QAnywhere header file.

   ```
   #include <qa.hpp>
   ```

   qa.hpp defines the QAnywhere classes.

2. Initialize QAnywhere.

   To do this, initialize a factory for creating QAManager objects.

   ```
   QAManagerFactory * factory;

   factory = QAnywhereFactory_init();
   if( factory == NULL ) {
         // Fatal error.
   }
   ```

   ☞ For more information about QAManagerFactory, see "QAManagerFactory class" on page 462.

3. Create a QAManager instance.

   You can create a default QAManager object as follows:

   ```
   QAManager *  mgr;

   // Create a manager
   mgr = factory->createQAManager( NULL );
   ```



```
if( mgr == NULL ) {
  // fatal error
}
```

☞ For more information about QAManager, see "QAManager class" on page 427.

Tip
For maximum concurrency benefits, multi-threaded applications should create a QAManager for each thread. See "Multi-threaded QAManager" on page 61.

You can customize a QAManager object programmatically or using a properties file.

- To customize QAManager programmatically, use setProperty().

  For more information, see "Setting QAnywhere manager configuration properties programmatically" on page 64.

- To use a properties file, specify the properties file in createQAManager():

  ```
  mgr = factory->createQAManager( "qa_mgr.props" );
  ```

  where *qa_mgr.props* is the name of the properties file on the remote device.

  For more information, see "Setting QAnywhere manager configuration properties in a file" on page 62.

4. Initialize the QAManager object.

   ```
   qa_bool  rc;
   rc=mgr->open(
        AcknowledgementMode::IMPLICIT_ACKNOWLEDGEMENT );
   ```

   The argument to the open method is an acknowledgement mode, which indicates how messages are to be acknowledged. It must be one of **IMPLICIT_ACKNOWLEDGEMENT** or **EXPLICIT_ACKNOWLEDGEMENT**. With implicit acknowledgement, messages are acknowledged as soon as they are received by the client. With explicit acknowledgement, you must call one of the acknowledge methods on the QAManager to acknowledge the message.

   ☞ For more information about acknowledgement modes, see "AcknowledgementMode class" on page 392.

Instead of creating a QAManager, you can create a QATransactionalManager. See "Implementing transactional messaging for C++ clients" on page 70.

You are now ready to send messages.

See also

- "QAnywhere C++ API Reference" on page 391

Setting up Java applications

Before you can send or receive messages using QAnywhere Java clients, you must complete the following initialization tasks.

♦ **To initialize the QAnywhere Java API**

1. Add the location of *qaclient.jar* to your classpath. By default, it is located in the *java* subdirectory of your SQL Anywhere installation.
2. Import the ianywhere.qanywhere.client package.

   ```
   import ianywhere.qanywhere.client.*;
   ```

3. Create a QAManager object.

   ```
   QAManager mgr;
   mgr = QAManagerFactory.getInstance().createQAManager(null);
   ```

   You can also customize a QAManager object by specifying a properties file to the createQAManager method:

   ```
   mgr = QAManagerFactory.getInstance().createQAManager("qa_mgr.props.");
   ```

   Tip
   For maximum concurrency benefits, multi-threaded applications should create a QAManager for each thread. See "Multi-threaded QAManager" on page 61.

4. Initialize the QAManager object.

   ```
   mgr.open(AcknowledgementMode.EXPLICIT_ACKNOWLEDGEMENT);
   ```

   The argument to the open method is an acknowledgement mode, which indicates how messages are to be acknowledged. It must be one of IMPLICIT_ACKNOWLEDGEMENT or EXPLICIT_ACKNOWLEDGEMENT. With implicit acknowledgement, messages are acknowledged as soon as they are received by the client. With explicit acknowledgement, you must call one of the acknowledge methods on the QAManager to acknowledge the message.

   ☞ For more information about acknowledgement modes, see "Interface AcknowledgementMode" on page 504.

Instead of creating a QAManager, you can create a QATransactionalManager. See "Implementing transactional messaging for Java clients" on page 71.

You are now ready to send messages.

See also

♦ "QAnywhere Java API Reference" on page 503

Setting up SQL applications

QAnywhere SQL allows you to perform, in SQL, much of the messaging functionality of the QAnywhere .NET, C++, and Java APIs. This functionality includes creating messages, setting or getting message properties and content, sending and receiving messages, triggering message synchronization, and setting and getting message store properties.

Messages that are generated with QAnywhere SQL can also be received by clients created with the programming APIs. If you have configured a JMS connector on your server, the messages can also be received by JMS clients. Similarly, QAnywhere SQL can be used to receive messages that were generated by QAnywhere .NET, C++, or Java API, or JMS clients.

QAnywhere SQL messaging coexists with user transactions. This means that committing a transaction commits all the QAnywhere operations on that connection.

☞ For more information about QAnywhere applications, see "Writing QAnywhere Client Applications" on page 45.

Permissions

Only users with DBA privilege have automatic permission to execute the QAnywhere stored procedures. To give permission to a user, a user with DBA privilege must call the procedure ml_qa_grant_messaging_permissions.

☞ See "ml_qa_grant_messaging_permissions" on page 670.

Acknowledgement modes

The QAnywhere SQL API does not support IMPLICIT_ACKNOWLEDGEMENT or EXPLICIT_ACKNOWLEDGEMENT modes. All messaging through the SQL API is transactional.

Example

The following example creates a trigger on an inventory table. The trigger sends a message when the inventory for an item falls below a certain threshold. The message is sent after the transaction invoking the trigger is committed. If the transaction is rolled back, the message is not sent.

```
CREATE TRIGGER inventory_trigger AFTER UPDATE ON inventory
REFERENCING old AS oldinv new AS newinv
FOR EACH ROW
begin
   DECLARE msgid VARCHAR(128);
   IF oldinv.quantity > newinv.quantity AND newinv.quantity < 10 THEN
      -- Create the message
      SET msgid = ml_qa_createmessage();
      -- Set the message content
      CALL ml_qa_settextcontent( msgid,
         'Inventory of item ' || newinv.itemname
            || ' has fallen to only ' || newinv.quantity );
      -- Make the message high priority
      CALL ml_qa_setpriority( msgid, 9 );
      -- Set a message subject
      CALL ml_qa_setstringproperty( msgid,
         'tm_Subject', 'Inventory low!' );
      -- Send the message to the inventoryManager queue
      CALL ml_qa_putmessage( msgid,
```



```
            'inventoryManager' );
      end if;
   end
```

See also

- "QAnywhere SQL API Reference" on page 635

Multi-threaded QAManager

Access to a QAManager is serialized. When you have multiple threads accessing a single QAManager, threads will block while one thread performs a method call on the QAManager. Use a different QAManager for each thread in order to maximize concurrency. Only one thread is allowed to access an instance of QAManager at one time. Other threads will block until the QAManager method that was invoked by the first thread returns.

QAnywhere manager configuration properties

You can set QAnywhere manager configuration properties in one of the following ways:

- Create a properties text file to define the QAnywhere manager configuration properties that will be used by one Manager instance.

  See "Setting QAnywhere manager configuration properties in a file" on page 62.

- Set QAnywhere manager configuration properties programmatically.

  See "Setting QAnywhere manager configuration properties programmatically" on page 64.

Following are the QAnywhere manager configuration properties:

- **COMPRESSION_LEVEL=n** Set the compression level.

  n is the compression factor, which is expressed as is an integer between 0 and 9, where 0 indicates no compression and 9 indicates maximum compression.

- **CONNECT_PARAMS=connect-string** Specify a connection string for the QAnywhere manager to use to connect to the message store database. Specify each connection option in the form *keyword=value* with multiple options separated by semi-colons.

  The default is `"eng=qanywhere;uid=ml_qa_user;pwd=qanywhere"`

  ☞ For a list of options, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

  ☞ For information about managing the database user and password, see "Writing Secure Messaging Applications" on page 175.

- **LOG_FILE=filename** Specify the name of a file to use to write logging messages. Specifying this option implicitly enables logging.

- **MAX_IN_MEMORY_MESSAGE_SIZE=n** When reading a message, *n* is the largest message, in bytes, for which a buffer is allocated. A message larger than *n* bytes must be read using streaming operations. The default value is 1MB on Windows and 64KB on Windows CE.

Setting QAnywhere manager configuration properties in a file

The information in a QAnywhere manager properties file is specific to one instance of a QAManager.

When using a properties file, it must be configured for and installed on the remote device with each deployed copy of your application.

For information on specifying the name of the property file, see:

- .NET API: "CreateQAManager method" on page 313

- C++ API: "createQAManager function" on page 462
- Java API: "createQAManager method" on page 568
- SQL API: You cannot set properties in a file using the QAnywhere SQL API. See "Setting QAnywhere manager configuration properties programmatically" on page 64.

If the properties file does not reside in the same directory as your client executable, you must also specify the absolute path. If you want to use the default settings for the properties, use NULL instead of a file name.

Values set in the file permit you to enable or disable some of the QAnywhere features, such as automatic message compression and logging.

Entries in a QAnywhere manager configuration properties file take the form *name=value*. For a list of property names, see "QAnywhere manager configuration properties" on page 62. If *value* has spaces, enclose it in double-quotes. Comment lines start with **#**. For example:

```
# contents of QAnywhere manager configuration properties file
LOG_FILE=.\sender.ini.txt
# A comment
CONNECT_PARAMS=eng=qanywhere;uid=ml_qa_user;pwd=qanywhere
MAX_IN_MEMORY_MESSAGE_SIZE=2048
COMPRESSION_LEVEL=0
```

Referencing the configuration file

Suppose you have a QAnywhere manager configuration properties file called *mymanager.props* with the following content:

```
COMPRESSION_LEVEL=9
CONNECT_PARMS=DBF=mystore.db
```

When you create QAManager, you reference the file by name.

Following is an example using C#:

```
QAManager     mgr;
mgr = QAManagerFactory.Instance.CreateQAManager( "mymanager.props" );
mgr.Open( AcknowledgeMode.EXPLICIT_ACKNOWLEDGEMENT );
```

☞ For more information, see "QAManager interface" on page 271 and "QAManagerFactory class" on page 310 in the QAnywhere .NET API reference.

Following is an example using C++:

```
QAManagerFactory *  qa_factory;
QAManager *    mgr;
qa_factory = QAnywhereFactory_init();
qa_factory->createQAManager( "mymanager.props" );
mgr->open( AcknowledgementMode::EXPLICIT_ACKNOWLEDGEMENT );
```

☞ For more information, see "QAManager class" on page 427 and "QAManagerFactory class" on page 462 in the QAnywhere C++ API reference.

Following is an example using Java:


```
QAManager mgr;
mgr = QAManagerFactory.getInstance().createQAManager("mymanager.props");
mgr.open( AcknowledgementMode.EXPLICIT_ACKNOWLEDGEMENT );
```

☞ For more information, see "Class QAManagerFactory" on page 566 and "Interface QAManager" on page 536 in the QAnywhere Java API reference.

Setting QAnywhere manager configuration properties programmatically

In the QAnywhere APIs, you can use the QAManagerBase set property method to set properties programmatically. Setting QAnywhere manager configuration properties programmatically must be done before calling the open method of a QAManager instance.

☞ For more information about QAManagerProperties, see "QAnywhere manager configuration properties" on page 62.

Example

The following C# example sets properties programmatically. When you create the QAManager, you specify the property settings.

```
QAManager     mgr;
mgr = QAManagerFactory.Instance.CreateQAManager( null );
mgr.SetProperty( "COMPRESSION_LEVEL", "9" );
mgr.SetProperty( "CONNECT_PARAMS", "DBF=mystore.db" );
mgr.Open( AcknowledgeMode.EXPLICIT_ACKNOWLEDGEMENT );
```

☞ For more information, see"QAManager interface" on page 271 and "QAManagerFactory class" on page 310 in the QAnywhere .NET API reference.

The following C++ example sets properties programmatically. When you create the QAManager, you specify the property settings.

```
QAManagerFactory *  qa_factory;
QAManager *    mgr;
qa_factory = QAnywhereFactory_init();
mgr = qa_factory->createQAManager( NULL );
mgr->setProperty( "COMPRESSION_LEVEL", "9" );
mgr->setProperty( "CONNECT_PARAMS", "DBF=mystore.db" );
mgr->open( AcknowledgementMode::EXPLICIT_ACKNOWLEDGEMENT );
```

☞ For more information, see "QAManager class" on page 427 and "QAManagerFactory class" on page 462 in the QAnywhere C++ API reference.

The following Java example sets properties programmatically. When you create the QAManager, you specify the property settings.

```
QAManager mgr;
mgr = QAManagerFactory.getInstance().createQAManager(null);
mgr.setProperty("COMPRESSION_LEVEL", 9);
mgr.setStringProperty("CONNECT_PARMS", "DBF=mystore.db");
mgr.open(AcknowledgementMode.EXPLICIT_ACKNOWLEDGEMENT);
```


☞ For more information, see “Class QAManagerFactory” on page 566 and “Interface QAManager” on page 536 in the QAnywhere Java API reference.

Sending QAnywhere messages

The following procedures describe how to send messages from QAnywhere applications. These procedures assume that you have created and opened a QAManager object.

Sending a message from your application does not ensure it is delivered from your device. It simply places the message on a queue to be delivered. The QAnywhere Agent carries out the task of sending the message to the MobiLink server, which in turn delivers it to its destination.

☞ For more information about when message transmission occurs, see "Determining when message transmission should occur on the client" on page 36.

♦ To send a message (.NET)

1. Create a new message.

   You can create either a text message or a binary message, using CreateTextMessage() or CreateBinaryMessage(), respectively.

   ```
   QATextMessage     msg;
   msg = mgr.CreateTextMessage();
   ```

2. Set message properties.

   Use methods of the QATextMessage or QABinaryMessage class to set properties.

   ☞ For more information, see "Message headers and message properties" on page 206.

3. Put the message on the queue, ready for sending.

   ```
   mgr.PutMessage( "store-id\\queue-name", msg );
   ```

   where *store-id* and *queue-name* are strings that combine to form the destination address.

   ☞ For more information, see "PutMessage method" on page 298 in the .NET API.

   ☞ For more information about determining when a message is transmitted, see "Determining when message transmission should occur on the client" on page 36.

♦ To send a message (C++)

1. Create a new message.

   You can create either a text message or a binary message, using createTextMessage() or createBinaryMessage(), respectively.

   ```
   QATextMessage *   msg;
   msg = mgr->createTextMessage();
   ```

2. Set message properties.

   Use methods of the QATextMessage or QABinaryMessage class to set message properties.

☞ For more information, see "Message headers and message properties" on page 206.

3. Put the message on the queue, ready for sending.

```
if( msg != NULL ) {
  if( !mgr->putMessage( "store-id\\queue-name", msg ) ) {
    // Display error using mgr->getLastErrorMsg().
    }
    mgr->deleteMessage( msg );
}
```

where *store-id* and *queue-name* are strings that combine to form the destination address.

☞ For more information, see "putMessage function" on page 452 in the C++ API.

☞ For more information about determining when a message is transmitted, see "Determining when message transmission should occur on the client" on page 36.

♦ To send a message (Java)

1. Create a new message.

   You can create a text message or a binary message, using QAManagerBase.createTextMessage() or QAManagerBase.createBinaryMessage(), respectively.

```
QATextMessage msg;
msg = mgr.createTextMessage();
```

2. Set message properties.

   Use QATextMessage or QABinaryMessage methods to set message properties.

   ☞ For more information, see "Message headers and message properties" on page 206.

3. Put the message on the queue.

```
mgr.putMessage("store-id\\queue-name", msg);
```

☞ For more information, see "getStringStoreProperty method" on page 557 in the Java API.

☞ For more information about determining when a message is transmitted, see "Determining when message transmission should occur on the client" on page 36.

♦ To send a message (SQL)

1. Declare a variable to hold the message ID.

```
begin
    declare @msgid varchar(128);
```

2. Create a new message.

```
    set @msgid = ml_qa_createmessage();
```


3. Set message properties.

   ☞ For more information, see "Message properties" on page 645.

4. Put the message on the queue.

```
        call ml_qa_putmessage( @msgid, 'someclient\someaddress' );
        commit;
    end
```

☞ For more information, see "ml_qa_putmessage" on page 672.

For more information about determining when a message is transmitted, see "Determining when message transmission should occur on the client" on page 36.

Implementing transactional messaging

Transactional messaging provides the ability to group messages in a way that guarantees that either all messages in the group are delivered, or none are. This is more commonly referred to as a single **transaction**.

When implementing transactional messaging, you create a special QAManagerBase object called QATransactionalManager.

For more information, see:

- .NET clients: "QATransactionalManager interface" on page 337
- C++ clients: "QATransactionalManager class" on page 492
- Java clients: "Interface QATransactionalManager" on page 594
- SQL clients: all messaging is transactional for SQL clients and no transactional manager is required

Implementing transactional messaging for .NET clients

- **To create a transactional manager**

1. Initialize QAnywhere.

   This step is the same as in non-transactional messaging.

```
using iAnywhere.QAnywhere.Client;
```

2. Create a QATransactionalManager object.

   For example, to create a default QATransactionalManager object, invoke CreateQATransactionalManager with null as its parameter:

```
QAManager mgr;
mgr =
  QAManagerFactory.Instance.CreateQATransactionalManager(
  null );
```

   ☞ For more information about QAManagerFactory, see "QAManagerFactory class" on page 310.

You can alternatively create a QATransactionalManager object that is customized using a properties file. The properties file is specified in the CreateQATransactionalManager method:

```
mgr =
  QAManagerFactory.Instance.CreateQATransactionalManager(
  "qa_mgr.props" );
```

where *qa_mgr.props* is the name of the properties file that resides on the remote device.

3. Initialize the QAManager object.

```
mgr.Open();
```

You are now ready to send messages. The following procedure sends two messages in a single transaction.

♦ To send multiple messages in a single transaction

1. Initialize message objects.

```
QATextMessage msg_1;
QATextMessage msg_2;
```

2. Send the messages.

   The following code sends two messages in a single transaction:

```
msg_1 = mgr.CreateTextMessage();
if( msg_1 != null ) {
  msg_2 = mgr.CreateTextMessage();
  if( msg_2 != null ) {
    if( !mgr.PutMessage( "jms_1\\queue_name", msg_1 ) ) {
      // Display message using mgr.GetLastErrorMsg().
    } else {
      if( !mgr.PutMessage( "jms_1\\queue_name", msg_2 ) ) {
        // Display message using mgr.GetLastErrorMsg().
      } else {
        mgr.Commit();
      }
    }
  }
}
```

   The Commit method commits the current transaction and begins a new transaction. This method commits all PutMessage() method and GetMessage() method invocations.

> **Note**
> The first transaction begins with the call to open method.

See also

- "QATransactionalManager interface" on page 337

Implementing transactional messaging for C++ clients

♦ To create a transactional manager

1. Initialize QAnywhere.

   This step is the same as in non-transactional messaging.

   ```
   #include <qa.hpp>
   QAManagerFactory *   factory;

   factory = QAnywhereFactory_init();
   if( factory == NULL ) {
       // Fatal error.
   }
   ```

2. Create a transactional manager.

   ```
   QATransactionalManager *  mgr;
   mgr = factory->createQATransactionalManager( NULL );
   if( mgr == NULL ) {
       // Fatal error.
   }
   ```

   As with non-transactional managers, you can specify a properties file to customize QAnywhere behavior. In this example, no properties file is used.

3. Initialize the manager.

   ```
   if( !mgr->open() ) {
      // Display message using mgr->getLastErrorMsg().
   }
   ```

You are now ready to send messages. The following procedure sends two messages in a single transaction.

♦ To send multiple messages in a single transaction

1. Initialize message objects.

   ```
   QATextMessage *  msg_1;
   QATextMessage *   msg_2;
   ```

2. Send the messages.

   The following code sends two messages in a single transaction:

   ```
   msg_1 = mgr->createTextMessage();
   if( msg_1 != NULL ) {
     msg_2 = mgr->createTextMessage();
     if( msg_2 != NULL ) {
       if( !mgr->putMessage( "jms_1\\queue_name", msg_1 ) ) {
         // Display message using mgr->getLastErrorMsg().
       } else {
         if( !mgr->putMessage( "jms_1\\queue_name", msg_2 ) ) {
           // Display message using mgr->getLastErrorMsg().
         } else {
           mgr->commit();
         }
       }
       mgr->deleteMessage( msg_2 );
   ```



```
    }
    mgr->deleteMessage( msg_1 );
}
```

The commit method commits the current transaction and begins a new transaction. This method commits all putMessage() method and getMessage() method invocations.

> **Note**
> The first transaction begins with the call to open method.

See also

- C++: "QATransactionalManager class" on page 492
- Java: "Interface QATransactionalManager" on page 594

Implementing transactional messaging for Java clients

♦ To create a transactional manager

1. Initialize QAnywhere.

   This step is the same as in non-transactional messaging.

   ```
   import ianywhere.qanywhere.client;
   QAManagerFactory factory = new QAManagerFactory();
   ```

   ☞ For more information about QAManagerFactory, see "QAManagerFactory class" on page 310.

2. Create a QATransactionalManager object.

   For example, to create a default QATransactionalManager object, invoke createQATransactionalManager with null as its parameter:

   ```
   QAManager mgr;
   mgr = factory.createQATransactionalManager( null );
   ```

   You can alternatively create a QATransactionalManager object that is customized using a properties file. The properties file is specified in the createQATransactionalManager method:

   ```
   mgr = factory.createQATransactionalManager( "qa_mgr.props" );
   ```

   where *qa_manager.props* is the name of the properties file that resides on the remote device.

3. Initialize the QAManager object.

   ```
   mgr.open();
   ```

You are now ready to send messages. The following procedure sends two messages in a single transaction.

♦ To send multiple messages in a single transaction

1. Initialize message objects.


```
QATextMessage msg_1;
QATextMessage msg_2;
```

2. Send the messages.

   The following code sends two messages in a single transaction:

```
msg_1 = mgr.createTextMessage();
if( msg_1 != null ) {
  msg_2 = mgr.createTextMessage();
  if( msg_2 != null ) {
    if( !mgr.putMessage( "jms_1\\queue_name", msg_1 ) ) {
      // Display message using mgr.getLastErrorMsg().
    } else {
      if( !mgr.putMessage( "jms_1\\queue_name", msg_2 ) ) {
        // Display message using mgr.getLastErrorMsg().
      } else {
        mgr.commit();
      }
    }
  }
}
```

   The commit method commits the current transaction and begins a new transaction. This method commits all putMessage() method and getMessage() method invocations.

> **Note**
> The first transaction begins with the call to open method.

Cancelling QAnywhere messages

Cancelling a QAnywhere message puts the message into a cancelled state before it is transmitted. With the default delete rules of the QAnywhere Agent, cancelled messages are eventually deleted from the message store. Cancelling a QAnywhere message fails if the message is already in a final state, or if it has been transmitted to the central messaging server.

The following procedures describe how to cancel QAnywhere messages.

Note
You cannot cancel a message using the QAnywhere SQL API.

♦ To cancel a message (.NET)

1. Get the ID of the message to cancel.

```
// msg is a QAMessage instance that has not been
// transmitted.
string msgID = msg.getMessageID();
```

2. Call CancelMessage with the ID of the message to cancel.

```
mgr.CancelMessage(msgID);
```

☞ For more information, see "CancelMessage method" on page 283.

♦ To cancel a message (C++)

1. Get the ID of the message to cancel.

```
// msg is a QAMessage instance that has not been
// transmitted.
qa_string msgID = msg->getMessageID();
```

2. Call cancelMessage with the ID of the message to cancel.

```
bool result = mgr->cancelMessage(msgID);
```

☞ For more information, see "cancelMessage function" on page 438.

♦ To cancel a message (Java)

1. Get the ID of the message to cancel.

```
// msg is a QAMessage instance that has not been
// transmitted.
String msgID = msg.getMessageID();
```

2. Call cancelMessage with the ID of the message to cancel.


```
boolean result = mgr.cancelMessage(msgID);
```

For more information, see “cancelMessage method” on page 544.

Receiving QAnywhere messages

The following topics describe how to receive QAnywhere messages.

Receiving messages synchronously

To receive messages synchronously, your application explicitly polls the queue for messages. It may poll the queue periodically, or when a user initiates a particular action such as clicking a Refresh button.

♦ To receive messages synchronously (.NET)

1. Declare message objects to hold the incoming messages.

```
QAMessage msg;
QATextMessage text_msg;
```

2. Poll the message queue, collecting messages:

```
if(mgr.start())  {
  for(;;) {
    msg = mgr.GetMessageNoWait("queue-name");
    if( msg == null ) {
      break;
    }
    addMessage( msg );
  }
  mgr.stop();
}
```

☞ For more information, see "GetMessageNoWait method" on page 292.

♦ To receive messages synchronously (C++)

1. Declare message objects to hold the incoming messages.

```
QAMessage * msg;
QATextMessage * text_msg;
```

2. Poll the message queue, collecting messages:

```
if( mgr->start() ) {
  for( ;; ) {
    msg = mgr->getMessageNoWait( "queue-name" );
    if( msg == NULL ) {
      break;
    }
    addMessage(msg);
  }
  mgr->stop();
}
```

☞ For more information, see "getMessageNoWait method" on page 552.

♦ **To receive messages synchronously (Java)**

1. Declare message objects to hold the incoming messages.

```
QAMessage msg;
QATextMessage text_message;
```

2. Poll the message queue, collecting messages:

```
if(mgr.start()) {
  for ( ;; ) {
    msg = mgr.getMessageNoWait("queue-name");
    if ( msg == null ) {
      break;
    }
    addMessage(msg);
  }
  mgr.stop();
}
```

☞ For more information, see "getMessageNoWait method" on page 552.

♦ **To receive messages synchronously (SQL)**

1. Declare an object to hold the message ID.

```
begin
    declare @msgid varchar(128);
```

2. Poll the message queue, collecting messages.

```
    loop
        set @msgid = ml_qa_getmessagenowait( 'myaddress' );
        if @msgid is null then leave end if;
       message 'a message with content ' || ml_qa_gettextcontent( @msgid )
|| ' has been received';
    end loop;
    commit;
end
```

For more information, see:

- "ml_qa_getmessagenowait" on page 668
- "ml_qa_getmessagetimeout" on page 669
- "ml_qa_getmessage" on page 666

Receiving messages asynchronously

To receive messages asynchronously using the .NET, C++, and Java APIs, you can write and register a message listener function that is called by QAnywhere when a message appears in the queue. The message listener takes the incoming message as a parameter. The task you perform in your message listener depends on your application. For example, in the TestMessage sample application the message listener adds the message to the list of messages in the main TestMessage window.

♦ To receive messages asynchronously (.NET)

1. Implement a message handler method.

```
private void onMessage(QAMessage msg) {
    // Process message.
}
```

2. Register the message handler.

   To register a message handler, create a QAManager.MessageListener object that has the message handler function as its argument. Then use the QAManager.SetMessageListener function to register the MessageListener with a specific queue. In the following example, *queue-name* is a string and *listener* is the name of the queue the QAManager object listens to.

```
MessageListener listener;
listener = new MessageListener( onMessage );
mgr.SetMessageListener( "queue-name", listener );
```

   ☞ For more information about MessageListener, see "MessageListener delegate" on page 245.

   ☞ For more information about SetMessageListener, see "SetMessageListener method" on page 303.

♦ To receive messages asynchronously (C++)

1. Create a class that implements the QAMessageListener interface.

```
class MyClass: public QAMessageListener {
   public:
   void onMessage( QAMessage * Msg);
};
```

   ☞ For more information, see "QAMessageListener class" on page 487.

2. Implement the onMessage method.

   The QAMessageListener interface contains one method, onMessage. Each time a message arrives in the queue, the QAnywhere library calls this method, passing the new message as the single argument.

```
void MyClass::onMessage(QAMessage * msg) {
     // Process message.
}
```

3. Register the message listener.

```
my_listener = new MyClass();
mgr->setMessageListener( "queue-name", my_listener );
```

   ☞ For more information, see "setMessageListener function" on page 457.

♦ To receive a message asynchronously (Java)

1. Implement a message handler method and an exception handler method.


```
class MyClass implements QAMessageListener {
  public void onMessage(QAMessage message) {
    // Process the message.
  }
  public void onException(
    QAException exception, QAMessage message) {
    // Handle the exception.
  }
}
```

2. Register the message handler.

```
MyClass listener = new MyClass();
mgr.setMessageListener("queue-name", listener);
```

For more information, see:

- "Interface QAMessageListener" on page 588
- "setLongStoreProperty method" on page 561

♦ To receive messages asynchronously (SQL)

- Create a stored procedure with the name **ml_qa_listener_*queue***, where *queue* is the name of a message queue.

  This procedure is called whenever a message is queued on the given queue.

  ☞ See "ml_qa_listener_queue" on page 670.

Development tip for .NET, C++ and Java

It is safer to use QAManagers in mode EXPLICIT_ACKNOWLEDGEMENT to guard against the possibility of an application error occurring part way through the processing of received messages and the message being acknowledged anyway.

If the QAManager is opened in mode EXPLICIT_ACKNOWLEDGEMENT, the message can be acknowledged in the onMessage method only after it has been successfully processed. That way if there was an error processing the message, the message will be received again because it was not acknowledged.

If the QAManager is opened in mode IMPLICIT_ACKNOWLEDGEMENT, the message passed to onMessage is acknowledged implicitly when onMessage returns. If the user application encounters an error while processing the message, the message is acknowledged and never received again.

Receiving messages using a selector

You can use **message selectors** to select messages for receiving. A message selector is a SQL-like expression that specifies a condition to select a subset of messages to consider for receive operations.

The syntax and semantics of message selectors are exactly the same as the condition part of transmission rules.

☞ See "Condition syntax" on page 228.

Example

The following C# example gets the next message from receiveQueue that has a message property called intprop with value 1.

```
msg = receiver.GetMessageBySelectorNoWait(
   receiveQueue, "intprop=1" );
```

The following C++ example gets the next message from receiveQueue that has a message property called intprop with value 1.

```
msg = receiver->getMessageBySelectorNoWait(
  receiveQueue, "intprop=1" );
```

The following Java example gets the next message from receiveQueue that has a message property called intprop with value 1.

```
msg = receiver.getMessageBySelectorNoWait(
  receiveQueue, "intprop=1");
```

See also

- .NET: “GetMessageBySelector method” on page 290 and “GetMessageBySelectorNoWait method” on page 291
- C++: “getMessageBySelector function” on page 446 and “getMessageBySelectorNoWait function” on page 447
- Java: “getMessageBySelector method” on page 550 and “getMessageBySelectorNoWait method” on page 551
- SQL: the SQL API does not support receiving messages using a selector

Reading very large messages

Sometimes messages are so large that they exceed the limit set with the QAManager property MAX_IN_MEMORY_MESSAGE_SIZE or its defaults of 1MB on Windows and 64KB on Windows CE. In this case, the message object cannot contain the full content of the message in memory, so methods that rely on the full content of the message being loaded into memory, such as readInt() and readString(), cannot be used. However, you can read very large messages directly from the message store in pieces. To do this, use QATextMessage.readText() or QABinaryMessage.readBinary() in a loop.

For more information, see:

- .NET: "ReadBinary method" on page 258 and "ReadText method" on page 336
- C++: "readBinary function" on page 406 and "readText function" on page 490
- Java: "readBinary method" on page 517 and "readText method" on page 591
- SQL: the SQL API does not support receiving very large messages

When you do this, you cannot use a QAManager that was opened with IMPLICIT_ACKNOWLEDGEMENT. You must use a QAManager that was opened with EXPLICIT_ACKNOWLEDGEMENT and you must complete all calls to readText() or readBinary() before acknowledging the message.

☞ For more information, see "Acknowledgement modes" on page 59.

Browsing QAnywhere messages

You can browse messages in incoming and outgoing queues. Browse operations do not affect the status of messages.

☞ For more information about message status, see ias_Status in "Pre-defined message properties" on page 209.

The following topics describe how to browse QAnywhere messages.

Browse all messages

You can browse the messages in all queues by calling the appropriate browseMessages() method.

Example

The following .NET example uses the QAManager.BrowseMessages() method to browse all queues:

```
QAMessage msg;
IEnumerator msgs = mgr.BrowseMessages();
while( msgs.MoveNext() ) {
      msg = (QAMessage)msgs.Current;
      // Process message.
}
```

The following C++ example uses the QAManager browseMessages function to browse all queues:

```
QAMessage *msg;
qa_browse_handle bh = mgr->browseMessages();
for (;;) {
  msg = mgr->browseNextMessage( bh );
  if( msg == qa_null ) {
    break;
  }
  // Process message.
  mgr->browseClose( bh );
}
```

The following Java example uses the QAManager.browseMessages method to browse all queues:

```
QAMessage msg;
java.util.Enumeration msgs = mgr.browseMessages();
while( msgs.hasMoreElements() ) {
    msg = (QAMessage)msgs.nextElement();
  // Process message.
}
```

See also

- .NET: "BrowseMessages method" on page 280
- C++: "browseMessages function" on page 434
- Java: "browseMessages method" on page 542
- SQL: the SQL API does not support browsing messages

Browsing messages in a queue

You can browse the messages in a given queue by supplying the queue name to the appropriate browseMessagesByQueue() method.

Example

The following .NET example uses the QAManager.BrowseMessagesByQueue method to browse a queue:

```
QAMessage msg;
IEnumerator msgs = mgr.BrowseMessagesByQueue( "q1" );
while( msgs.MoveNext() ) {
  msg = (QAMessage)msgs.Current;
  // Process message.
}
```

The following C++ example uses the QAManager browseMessagesByQueue function to browse a queue:

```
QAMessage *msg;
qa_browse_handle bh = mgr->browseMessagesByQueue( _T("q1") );
for (;;) {
  msg = mgr->browseNextMessage( bh );
  if( msg == qa_null ) {
    break;
  }
  // Process message.
}
mgr->browseClose( bh );
```

The following Java example uses the QAManager.browseMessagesByQueue method to browse a queue:

```
QAMessage msg;
java.util.Enumeration msgs = mgr.browseMessagesByQueue( "q1" );
while( msgs.hasMoreElements() ) {
  msg = (QAMessage)msgs.nextElement();
  // Process message.
}
```

See also

- .NET: "BrowseMessagesByQueue method" on page 282
- C++: "browseMessagesByQueue function" on page 436
- Java: "browseMessagesByQueue method" on page 543
- SQL: the SQL API does not support browsing messages

Browsing a message by ID

You can browse a particular message by specifying its ID to a browseMessagesbyID() method.

Example

The following .NET example uses the QAManager.BrowseMessageByID method to browse a message:

```
QAMessage msg;
IEnumerator msgs = mgr.BrowseMessagesByID( "ID:123" );
if( msgs.MoveNext() ) {
  msg = (QAMessage)msgs.Current;
  // Process message.
}
```


The following C++ example uses the QAManager browseMessageByID function to browse a message :

```
QAMessage *msg;
qa_browse_handle bh = mgr->browseMessagesByID( _T( "ID:123" ) );
msg = mgr->browseNextMessage( bh );
if( msg != qa_null ) {
  // Process message.
}
mgr->browseClose( bh );
```

The following Java example uses the QAManager.browseMessageByID method to browse a message:

```
QAMessage msg;
java.util.Enumeration msgs = mgr.browseMessagesByID( "ID:123" );
if( msgs.hasMoreElements() ) {
  msg = (QAMessage)msgs.nextElement();
  // Process message.
}
```

See also

- .NET: "BrowseMessagesByID method" on page 281
- C++: "browseMessagesByID function" on page 435
- Java: "browseMessagesByID method" on page 542
- SQL: the SQL API does not support browsing messages

Browsing messages using a selector

You can use **message selectors** to select messages for browsing. A message selector is a SQL-like expression that specifies a condition to select a subset of messages to consider for browse operations.

The syntax and semantics of message selectors are exactly the same as the condition part of transmission rules.

☞ See "Condition syntax" on page 228.

Example

The following .NET example browses all messages in the message store that have a property called intprop with value 1.

```
QAMessage msg;
IEnumerator msgs = mgr.BrowseMessagesBySelector( "intprop = 1" );
while( msgs.MoveNext() ) {
  msg = (QAMessage)msgs.Current;
  // Process message.
}
```

The following C++ example browses all messages in the message store that have a property called intprop with value 1.

```
QAMessage *msg;
qa_browse_handle bh = mgr->browseMessagesBySelector( _T("intprop = 1") );
for (;;) {
  msg = mgr->browseNextMessage( bh );
  if( msg == qa_null ) {
    break;
```



```
  }
  // Process message.
}
mgr->browseClose( bh );
```

The following Java example browses all messages in the message store that have a property called intprop with value 1.

```
QAMessage msg;
java.util.Enumeration msgs = mgr.browseMessagesBySelector( "intprop = 1" );
while( msgs.hasMoreElements() ) {
  msg = (QAMessage)msgs.nextElement();
  // Process message.
}
```

See also

- .NET: "BrowseMessagesBySelector method" on page 282
- C++: "browseMessagesBySelector function" on page 436
- Java: "browseMessagesBySelector method" on page 544
- SQL: the SQL API does not support browsing messages

Handling QAnywhere exceptions

The QAnywhere C++, Java, and .NET APIs include special objects and properties for exception handling.

.NET exceptions

The QAException class encapsulates QAnywhere client application exceptions. After you catch a QAnywhere exception, you can use the QAException ErrorCode and Message properties to determine the error code and error message.

Note that if a QAException is thrown inside a message listener delegate and it is not caught in the message listener, then it will be logged to the QAManager log file. Since uncaught QAExceptions are only logged, it is recommended that all exceptions be handled within message listener delegates so that they can be dealt with appropriately.

☞ For more information about the log file, see "QAnywhere manager configuration properties" on page 62.

When a QAException is thrown the current transaction is rolled back. When this happens in a message listener with a QATransactionalManager, the message that was being processed when the QAException was thrown is put back in the receive queue and so that it will be re-received. You can use the message store property ias_MaxDeliveryAttempts to prevent an infinite loop.

When the property ias_MaxDeliveryAttempts is set to a positive integer *n* by a QAnywhere application, as in `mgr.SetIntStoreProperty( "ias_MaxDeliveryAttempts", 5 )`, the QAnywhere client will attempt to receive an unacknowledged message up to *n* times before setting the status of the message to unreceivable. If the property ias_MaxDeliveryAttempts is not set or is negative, the QAnywhere client will attempt to receive messages an unlimited number of times.

For more information, see:

- "QAException class" on page 269
- "ErrorCode property" on page 271

The following table lists error code values for the LastError property:

LastError value	Description
0	No error.
1000	Initialization error.
1001	Termination error.
1002	Unable to access the client properties file.
1003	No destination.
1004	The function is not implemented.
1005	You cannot write to a message as it is in read-only mode.

LastError value	Description
1006	Error storing a message in the client message store.
1007	Error retrieving a message from the client message store.
1008	Error initializing the background thread.
1009	Error opening a connection to the message store.
1010	There is an invalid property in the client properties file.
1011	Error opening the log file.
1012	Unexpected end of message reached.
1013	The message store is too large relative to the free disk space on the device.
1014	The message store has not been initialized for messaging.
1015	Error getting queue depth.
1016	Cannot use QAManagerBase.getQueueDepth when the message store ID has not been set.
1017	Cannot use QAManagerBase.getQueueDepth on a given destination when filter is ALL.
1018	Error cancelling message.
1019	Error cancelling message. Cannot cancel a message that has already been sent.
1020	Error acknowledging the message.
1021	The QAManager is not open.
1022	The QAManager is already open.
1023	The given selector has a syntax error.
1024	The timestamp is outside of the acceptable range.

C++ exceptions

For C++, the QAError class encapsulates QAnywhere client application exceptions. You can use the QAManagerBase::getLastError() method or QAManagerFactory::getLastError() method to determine the error code associated with the last executed method. You can use the corresponding getLastErrorMessage () method to obtain the error text.

☞ For a list of error codes and more information, see "QAError class" on page 418.

For more information about getLastError and getLastErrorMessage, see:

- QAManagerBase "getLastError function" on page 444 and "getLastErrorMsg function" on page 444.

- QAManagerFactory "getLastError function" on page 464 and "getLastErrorMsg function" on page 464.

Java exceptions

For Java, the QAException class encapsulates QAnywhere client application exceptions. You can catch the exception to determine the error code and error text associated with the last executed method.

☞ For a list of error codes and more information, see "Class QAException" on page 530.

Shutting down QAnywhere

After you have completed sending and receiving messages, you can shut down the QAnywhere messaging system by completing one of the following procedures.

♦ To shut down QAnywhere (.NET)

- Stop and close the QAnywhere manager.

```
mgr.Stop();
mgr.Close();
```

♦ To shut down QAnywhere (C++)

1. Close the QAnywhere manager.

```
mgr->stop();
mgr->close();
```

2. Terminate the factory.

```
QAnywhereFactory_term();
```

   This step shuts down the messaging part of your application.

♦ To shut down QAnywhere (Java)

- Stop and close the QAnywhere manager.

```
mgr.stop();
mgr.close();
```

See also

- .NET: “Stop method” on page 309
- C++: “stop function” on page 460
- Java: “setStoreProperty method” on page 564
- SQL: the SQL API does not support shutting down QAnywhere

Deploying QAnywhere applications

☞ For information about the files needed to deploy QAnywhere applications, see "Deploying QAnywhere applications" [*MobiLink - Server Administration*].

CHAPTER 5

Server management requests

Contents

About this chapter

This chapter describes how to use server management requests to perform administrative tasks and to monitor QAnywhere messaging.

About server management requests

A QAnywhere client application can send special messages to the server called **server management requests**. These messages contain content that is formatted as XML and are addressed to the QAnywhere system queue. They require a special authentication string. Server management requests can perform a variety of functions, such as:

- Starting and stopping connectors and web services.

  See "Opening connectors" on page 106 and "Closing connectors" on page 106.

- Monitoring connector status.

  See "Monitoring connectors" on page 107.

- Setting and refreshing client transmission rules.

  See "Specifying transmission rules with a server management request" on page 116.

- Monitoring message status.

  See "Monitoring QAnywhere" on page 120.

- Setting, updating, deleting, and querying client message store properties on the server.

  See "Setting server properties with a server management request" on page 114.

- Cancelling messages.

  See "Cancelling messages" on page 101.

- Querying for active clients, message store properties, and messages.

Addressing server management requests

By default, server management requests must be addressed to **ianywhere.server\system.** To change the client ID portion of this address, set the ianywhere.qa.server.id property and restart the server. For example, if the ianywhere.qa.server.id property is set to myServer, server management requests are addressed to myServer\system.

☞ For more information about setting the ianywhere.qa.server.id property, see "Server properties" on page 222.

☞ For more information about addressing QAnywhere messages, see "Sending QAnywhere messages" on page 66.

☞ For more information about the system queue, see "System queue" on page 51.

Authenticating server management requests

The message string property ias_ServerPassword specifies the server password. The server password is set using the ianywhere.qa.server.password.e property. If this property is not set, the password is QAnywhere.

The server password is transmitted as text. Use an encrypted communication stream to send server management requests that require a server password.

For more information about the ianywhere.qa.server.password.e property, see "Server properties" on page 222.

Examples

The following is a sample message details request. It generates a single report that displays the message ID, status, and target address of all messages with priority 9 currently on the server.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
   <MessageDetailsRequest>
      <request>
         <requestId>testRequest</requestId>
         <condition>
            <priority>9</priority>
         </condition>
         <status/>
         <address/>
      </request>
   </MessageDetailsRequest>
</actions>
```

The following example is in C#. It sets a server-side transmission rule for a client such that messages from the server are only transmitted to the client called someClient if the priority is greater than 4.

```
QAManager mgr = …;    // Initialize the QAManager
QAMessage msg = mgr.CreateTextMessage();
msg.SetStringProperty( "ias_ServerPassword", "QAnywhere" );

// Indenting and newlines are just for readability
msg.Text = "<?xml version="1.0" encoding="UTF-8"?>\n"
+ "<actions>\n"
+ "    <SetProperty>\n"
+ "        <prop>\n"
+ "            <client>someClient</client>\n"
+ "            <name>ianywhere.qa.server.rules</name>\n"
+ "            <value>ias_Priority &gt; 4</value>\n"
+ "        </prop>\n"
+ "    </SetProperty>\n"
+ "    <RestartRules>\n"
+ "        <client>someClient</client>\n"
+ "    </RestartRules>\n"
+ "</actions>\n";

mgr.PutMessage( @"ianywhere.server\system", msg );
```

Writing server management requests

Server management requests contain content that is formatted as XML.

Note
You cannot use symbols such as **>** or **<** in the content of server management requests. Instead, use **>** and **<**.

Each server management request starts with an <actions> tag.

Each type of server management request includes it own XML tags. For example, to close a connector you use the <CloseConnector> tag.

request tag

In addition, most server management requests can include a <request> tag that describes the request. Within a <request> tag, you can use the following subtags:

<request> subtags	Description
<condition>	Groups conditions for including a message in the report. Only used in the <request> tag, which is a subtag of <MessageDetailsRequest> and <CancelMessageRequest>.
<onEvent>	Specifies the events upon which the server should generate reports. Only used with <ClientStatusRequest>. You can include one or more <onEvent> tags, with one event type per tag. If these tags are omitted, the Client Status Request produces a one-time request. Otherwise, the Client Status Request registers event listeners for the specified events.
<persistent>	Specifies that the results of the request should be made persistent in the server database (so that the report is sent even if the server is restarted). Only used with schedules.
<report>	Specifies that a report should be sent each time the request is activated. Only used in the <request> tag, which is a subtag of <CancelMessageRequest>.
<requestId>	Specifies a unique identifier for the request that is included in each report generated as a result of this request. Only used when a server management request generates a response or report. Using different values for this field allows more than one request to be active at the same time. Using the same request ID allows the client to override or delete active requests.
<replyAddr>	Specifies the return address for each report generated as a result of this request. If this tag is omitted, the default return address of reports is the reply address of the originating message. Only used when a server management request generates a response or report.
<schedule>	Specifies that the report should be generated on a schedule. Only used when a server management request generates a response or report. See “Scheduling server management requests” on page 98.

Condition tag

Use the following condition subtags to filter the messages to include in the MessageDetailsRequest. You can specify as many of these tags as you want in the <condition> tag. If you use more than one of the same

tag, then the values given are logically "OR"ed together, whereas if you use two different tags, the values are logically "AND"ed together.

<condition> subtags	Description
<address>	Selects the messages that are addressed to the specified address.
<customRule>	Selects messages based on rules. See "Custom message requests" on page 95.
<kind>	Filters either binary or text messages. For example, **<kind>text</kind>** filters text messages, and **<kind>binary</kind>** filters binary messages.
<messageId>	Selects the message with a particular message ID.
<originator>	Selects messages that originated from the specified client.
<priority>	Selects the messages that currently have the priority specified.
<property>	Selects messages that have the specified message property. To check a property name and value, use the syntax **<property>***property-name=property-value***</property>**. To check the existence of a property, use the format **<property>***property-name***</property>**.
<status>	Selects messages that currently have the status specified.

Custom message requests

To construct more complex condition statements, use the <customRule> tag as a subtag to the <condition> tag (and other tags). This tag takes as its data a server rule similar to those used for server transmission rules. You can construct these queries in the same manner as the condition part of a transmission rule.

☞ See "Condition syntax" on page 228.

Example

The following condition selects messages following the search criteria: (priority=4) AND (originator LIKE '%sender%' AND status >= 20)

```
<condition>
    <priority>4</priority>
    <customRule>ias_Originator LIKE '%sender%' AND ias_Status &gt;=
ias_FinalState</customRule>
</condition>
```

Server management request DTD

Following is the complete definition of the server management request XML document type. This DTD is provided as a summary of the server management tags that are described in this chapter.


```
<!-- Set of requests -->

<!ELEMENT actions ((CloseConnector|OpenConnector|RestartRules|SetProperty
    |ClientStatusRequest|MessageDetailsRequest|CancelMessageRequest
    |GetClientList)+)>

<!-- Request for list of all clients -->

<!ELEMENT GetClientList EMPTY>

<!-- Request to close a connector -->

<!ELEMENT CloseConnector (client)>

<!-- Request to open a connector -->

<!ELEMENT OpenConnector (client)>

<!-- Request to restart transmission rules for a client -->

<!ELEMENT RestartRules (client)>

<!-- Request for setting a property -->

<!ELEMENT SetProperty (client,prop)>

<!-- Request for client properties -->

<!ELEMENT GetProperties (client,replyAddr?)>

<!-- Request for the status on a connector -->

<!ELEMENT ClientStatusRequest (request)>

<!-- Request for clients -->

<!ELEMENT MessageDetailsRequest (request)>
<!ELEMENT CancelMessageRequest (request)>

<!ELEMENT request (requestId?,replyAddr?,schedule*,onEvent*,condition?,

persistent?,report?,messageId?,status?,priority?,address?,originator?,kind?,
    statusTime?,contentSize?,customRule?,property*)>

<!ELEMENT client (#PCDATA)>

<!ELEMENT prop (name?,value?)>

<!ELEMENT name (#PCDATA)>

<!ELEMENT value (#PCDATA)>

<!ELEMENT replyAddr (#PCDATA)>

<!ELEMENT requestId (#PCDATA)>

<!ELEMENT persistent EMPTY>

<!ELEMENT report EMPTY>

<!ELEMENT schedule ((starttime|
between)?,everyhour?,everyminute?,everysecond?,
    ondayofweek*,ondayofmonth*)>
```



```
<!ELEMENT between (starttime,endtime)>

<!ELEMENT starttime (#PCDATA)>
<!ELEMENT endtime (#PCDATA)>
<!ELEMENT everyhour (#PCDATA)>
<!ELEMENT everyminute (#PCDATA)>
<!ELEMENT everysecond (#PCDATA)>
<!ELEMENT ondayofweek (#PCDATA)>
<!ELEMENT ondayofmonth (#PCDATA)>

<!ELEMENT onEvent (#PCDATA)>

<!ELEMENT condition ((messageId|status|priority|address|originator|kind|
    customRule|property)+)>

<!ELEMENT messageId (#PCDATA)>
<!ELEMENT status (#PCDATA)>
<!ELEMENT priority (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT originator (#PCDATA)>
<!ELEMENT kind (#PCDATA)>
<!ELEMENT statusTime (#PCDATA)>
<!ELEMENT expires (#PCDATA)>
<!ELEMENT contentSize (#PCDATA)>
<!ELEMENT customRule (#PCDATA)>
<!ELEMENT property (#PCDATA)>

<!-- Reports and response sent back by the server -->

<!-- Report returned as a response to a CancelMessageRequest -->

<!ELEMENT CancelMessageReport (requestId,UTCDatetime,statusDescription,
    messageCount,message*)>

<!-- Report returned as a response to a ClientStatusRequest -->

<!ELEMENT ClientStatusReport (requestId,componentReport)>

<!-- Report returned as a response to a MessageDetailsRequest -->

<!ELEMENT MessageDetailsReport (requestId,UTCDatetime,statusDescription,
    messageCount,message*)>

<!-- Response to a GetPropertiesRequest -->

<!ELEMENT GetPropertiesResponse (client,prop*)>

<!-- Response to a GetClientList -->

<!ELEMENT GetClientListResponse (client*)>

<!ELEMENT UTCDatetime (#PCDATA)>

<!ELEMENT statusDescription (#PCDATA)>

<!ELEMENT messageCount (#PCDATA)>

<!ELEMENT message ((messageId|status|priority|address|originator|kind|
    statusTime|expires|contentSize|property)*)>

<!-- Report on a specific server component (such as a connector) -->

<!ELEMENT componentReport (client,UTCDatetime,statusCode,statusSubcode?,
    statusDescription?,vendorStatusCode?,vendorStatusDescription?)>
```



```
<!ELEMENT statusCode (#PCDATA)>
<!ELEMENT statusSubcode (#PCDATA)>
<!ELEMENT vendorStatusCode (#PCDATA)>
<!ELEMENT vendorStatusDescription (#PCDATA)>
```

Scheduling server management requests

You can optionally set up server management requests to run on a schedule. Use the following <schedule> subtags to define the schedule on which the request runs.

<schedule> subtags	Description
<starttime>	Defines the time of day at which the server begins generating reports. For example: `<starttime>09:00:00</starttime>`

<between>	Contains two subtags, starttime and endtime, which define an interval during which the server generates reports. May not be used in the same schedule as starttime. For example:
	`<between>`
	`  <starttime>Mon Jan 16 09:00:00 EST 2006</starttime>`
	`  <endtime>Mon Jan 17 09:00:00 EST 2006</endtime>`
	`</between>`

<everyhour>	Defines the interval between subsequent reports in hours. May not be used in the same schedule as everyminute or everysecond. For example, the following request generates a report every two hours starting on January 16 at 9 AM:
	`<schedule>`
	`  <starttime>09:00:00</starttime>`
	`  <everyhour>2</everyhour>`
	`</schedule>`

<everyminute>	Defines the interval between subsequent reports in minutes. May not be used in the same schedule as everyhour or everysecond.
	`<schedule>`
	`  <everyminute>10</everyminute>`
	`</schedule>`

<everysecond>	Defines the interval between subsequent reports in seconds. May not be used in the same schedule as everyhour or everyminute.
	`<schedule>`
	`  <everysecond>45</everysecond>`
	`</schedule>`

<ondayofweek>	Each tag contains one day of the week in which the schedule is active. For example, the following schedule runs on Mondays and Tuesdays:
	`<schedule>`
	`  <ondayofweek>Monday</ondayofweek>`
	`  <ondayofweek>Tuesday</ondayofweek>`
	`</schedule>`

<schedule> subtags	Description
<ondayofmonth>	Each tag contains one day of the month on which the schedule is active. For example, the following schedule runs on the fifteenth of the month: <schedule> <ondayofmonth>15</ondayofmonth> </schedule>
<startdate>	The date on which the schedule becomes active. For example: <startdate>Mon Jan 16 2006</startdate>

To modify a schedule, register a new server management request with the same requestId. To delete a schedule, register a server management request with the same requestId, but include the schedule tag <schedule>none</schedule>.

Notes

- Each tag, except for the <ondayofweek> and <ondayofmonth> tags, can only be used once in a schedule.
- The <between> tag and the individual <starttime> tag may not both be used in the same schedule.
- Only one of <everysecond>, <everyminute>, and <everyhour> may be used in the same schedule.

Example

The following example creates a persistent schedule that will report on all the messages on the server, including the ID and status of each message. It will also overwrite any previous persistent requests assigned to the request ID dailyMessageStatus.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
     <MessageDetailsRequest>
         <request>
              <replyAddr>myclient\messageStatusQueue</replyAddr>
              <requestId>dailyMessageStatus</requestId>
              <schedule>
                   <everyhour>24</everyhour>
              </schedule>
              <persistent/>
              <messageId/>
              <status/>
         </request>
     </MessageDetailsRequest>
</actions>
```

Following is an example of what the report might look like. It is sent to the address myclient \messageStatusQueue. It indicates that there are two messages on the server, one with status 60 (received) and one with status 1 (pending).

```
<?xml version="1.0" encoding="UTF-8"?>
<MessageDetailsReport>
      <requestId>dailyMessageStatus</requestId>
      <UTCDatetime>Mon Jan 16 15:03:04 EST 2007</UTCDatetime>
      <statusDescription>Scheduled report</statusDescription>
      <messageCount>2</messageCount>
      <message>
           <messageId>ID:26080b8927f83f9722357eab0a0628eb</messageId>
```



```
            <status>60</status>
        </message>
        <message>
            <messageId>ID:fe857fa8-a7d7-4266-985b-a1818a85d1a2</messageId>
            <status>1</status>
        </message>
    </MessageDetailsReport>
```


Administering the server message store with server management requests

You can use server management requests to administer the server message store.

☞ For an overview of how to use server management requests, including how to authenticate and schedule them, see "About server management requests" on page 92.

Refreshing client transmission rules

When a server-side client transmission rule is changed, the rules for the corresponding client must be refreshed. You can do this in a server management request by setting the property ianywhere.qa.server.rules.

A RestartRules tag contains a single client tag, which specifies the name of the client to refresh.

<RestartRules> subtags	Description
<client>	The name of the client for which to refresh transmission rules.

Example

The server XML needs to specify the new transmission rule property and then restart rule processing using the RestartRules tag. For example, the following XML changes the server-side transmission rule for client myclient to `auto = ias_Priority > 4`. Note the proper encoding of ">" in the XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <SetProperty>
         <prop>
           <client>myclient</client>
           <name>ianywhere.qa.server.rules</name>
           <value>auto = ias_Priority &gt; 4</value>
         </prop>
    </SetProperty>
    <RestartRules>
    <client>myclient</client>
    </RestartRules>
</actions>
```

Cancelling messages

You can create a server management request to cancel messages in the server message store. You can create a one-time cancellation request or you can schedule your cancellation request to happen automatically. You can also optionally generate a report that details the messages that have been cancelled.

Messages can only be cancelled if they are on the server and in a non-final state when the request is activated.

<CancelMessageRequest> subtags	
<request>	Groups information about a particular request. Specifying more than one <request> tag is equivalent to sending multiple separate server managment requests.

<Request> subtags	Description
<condition>	Groups conditions for including a message to be cancelled. See "Condition tag" on page 94.
<persistent>	Specifies that the request should be made persistent in the server database (so that messages can be cancelled even if the server is restarted). Only used with schedules.
<requestId>	Specifies a unique identifier for the request that is included in each report generated as a result of this request. Using different values for this field allows more than one request to be active at the same time. Using the same request id allows the client to override or delete active requests.
<replyAddr>	The return address for each report generated as a result of this request. If this tag is omitted, the default return address of reports is the return address of the originating message.
<report>	Causes a report to be sent each time the request is activated. To cause a report to be sent each time the request is activated, put an empty <report> tag inside the <request> tag.
<schedule>	Specifies that the report should be generated on a schedule. See "Scheduling server management requests" on page 98.

Example

This request cancels messages on the server with the address ianywhere.connector.myConnector\deadqueue:

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <CancelMessageRequest>
        <request>
            <requestId>cancelRequest</client>
            <condition>
                <customRule>ias_Address='ianywhere.connector.myConnector
\deadqueue'</customRule>
            </condition>
        </request>
    </CancelMessageRequest>
</actions>
```

Deleting messages

To specify a clean-up policy on the server, set the property ianywhere.qa.server.deleteRules for the special client ianywhere.server.deleteRules with the rule or rules governing which messages can be deleted from the server.

The following example changes the message clean-up policy to delete expired and cancelled messages:


```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <SetProperty>
    <prop>
        <client>ianywhere.server.deleteRules</client>
        <name>ianywhere.qa.server.deleteRules</name>
       <value>auto = ias_Status in ( ias_ExpiredStatus, ias_CancelledStatus )
and ias_TransmissionStatus = IAS_TRANSMITTED</value>
    </prop>
    </SetProperty>
    <RestartRules>
    <client>ianywhere.server.deleteRules</client>
    </RestartRules>
</actions>
```


Administering connectors

You can use server management requests to create, configure, delete, start, stop, and monitor connectors.

☞ For an overview of how to use server management requests, including how to authenticate and schedule them, see "About server management requests" on page 92.

See also

- "JMS Connectors" on page 127
- "Setting up web service connectors" on page 195

Creating and configuring connectors

To create connectors, use <OpenConnector>.

Example

In the following example, the server management request first sets a number of relevant properties and associates them with the client ianywhere.connector.jboss, which is the client ID of the new connector. JMS-specific properties are set in such a way that a connector to a local JBOSS JMS server are indicated. The connector is then started using the OpenConnector tag. Note that if you have not started the MobiLink server with the relevant jar files of the JMS client, the connector will not be started.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <SetProperty>
        <prop>
          <client>ianywhere.connector.jboss</client>
          <name>ianywhere.connector.nativeConnection</name>
        <value>ianywhere.message.connector.jms.NativeConnectionJMS</value>
      </prop>
        <prop>
          <client>ianywhere.connector.jboss</client>
          <name>ianywhere.connector.address</name>
        <value>ianywhere.connector.jboss</value>
      </prop>
        <prop>
          <client>ianywhere.connector.jboss</client>
          <name>xjms.jndi.factory</name>
        <value>org.jnp.interfaces.NamingContextFactory</value>
      </prop>
        <prop>
          <client>ianywhere.connector.jboss</client>
          <name>xjms.jndi.url</name>
        <value>jnp://0.0.0.0:1099</value>
      </prop>
        <prop>
          <client>ianywhere.connector.jboss</client>
          <name>xjms.topicFactory</name>
        <value>ConnectionFactory</value>
      </prop>
        <prop>
          <client>ianywhere.connector.jboss</client>
          <name>xjms.queueFactory</name>
        <value>ConnectionFactory</value>
      </prop>
```



```
          <prop>
            <client>ianywhere.connector.jboss</client>
            <name>xjms.receiveDestination</name>
          <value>qanywhere_receive</value>
        </prop>
          <prop>
            <client>ianywhere.connector.jboss</client>
            <name>xjms.deadMessageDestination</name>
          <value>qanywhere_deadMessage</value>
        </prop>
    </SetProperty>
    <OpenConnector>
        <client>ianywhere.connector.jboss</client>
    </OpenConnector>
</actions>
```

Modifying connectors

To modify connectors, close the connector, change properties with the <SetProperty> tag, and then open the connector.

Example

In the following example, the logging level of the connector is changed to 4. The connector with the ID ianywhere.connector.jboss is closed; the connector property logLevel is changed to 4, and then the connector is re-opened with the new log level.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <CloseConnector>
        <client>ianywhere.connector.jboss</client>
    </CloseConnector>
    <SetProperty>
        <prop>
          <client>ianywhere.connector.jboss</client>
        <name>ianywhere.connector.logLevel</name>
        <value>4</value>
      </prop>
    </SetProperty>
    <OpenConnector>
        <client>ianywhere.connector.jboss</client>
    </OpenConnector>
</actions>
```

Deleting connectors

To delete connectors, use <SetProperty> with a client name but no other values.

Example

In the following example, the connector with the ID ianywhere.connector.jboss is closed. All of its properties are deleted by the <SetProperty> tag, omitting the name and value tags.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <CloseConnector>
        <client>ianywhere.connector.jboss</client>
    </CloseConnector>
```



```
    <SetProperty>
        <prop>
          <client>ianywhere.connector.jboss</client>
      </prop>
    </SetProperty>
</actions>
```

Opening connectors

To open connectors, use <OpenConnector>.

An OpenConnector tag contains a single client tag that specifies the name of the connector to open.

<OpenConnector> subtag	Description
<client>	The name of the connector to open.

See also

- "JMS Connectors" on page 127
- "Setting up web service connectors" on page 195

Example

The following example opens the simpleGroup connector.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <OpenConnector>
     <client>simpleGroup</client>
    </OpenConnector>
</actions>
```

Closing connectors

To close connectors, use <CloseConnector>. A CloseConnector tag contains a single client tag that specifies the name of the connector to close.

<CloseConnector> subtags	Description
<client>	The name of the connector to close.

See also

- "JMS Connectors" on page 127
- "Setting up web service connectors" on page 195

Example

The following example closes the simpleGroup connector.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <CloseConnector>
```



```
        <client>simpleGroup</client>
      </CloseConnector>
</actions>
```

Monitoring connectors

To obtain information about connectors, write a special kind of server management request called a client status request. It contains a <ClientStatusRequest> tag that uses one or more <request> tags containing the information necessary to register the request.

Your client status request can obtain reports about connectors in several ways:

- Make a one-time request.
- Register a State Change Listener to have a report sent whenever the connector's state changes.
- Register an Error Listener to have a report sent whenever an error occurs on the connector.

In addition, you can schedule a report to be sent at certain times or intervals.

ClientStatusRequest tag

To get information about connectors, use <ClientStatusRequest>.

A client status request is composed of one or more <request> tags containing all the necessary information to register the request.

<ClientStatusRequest> subtag	
<request>	Groups information in requests.

request tag for client status requests

In the <request> tag, use an optional <replyAddr> tag to specify the return address for each report generated as a result of this request. If this tag is omitted, the default return address of reports is the reply address of the originating message.

Use an optional <requestId> to add a label for the request that is included in each report. When you register multiple requests, or when you delete or modify requests, the ID makes it possible to distinguish which reports were generated from a particular request.

To specify a list of connectors for the request, include one or more <client> tags, each with one connector address. In the case of a one-time request, all of the connectors are included in the report. In the case of an event listener request, the server listens to each of these connectors.

To specify that event details should be made persistent during any server downtime, specify the <persistent> tag. This tag does not require any data and can be of the form <persistent/> or <persistent></persistent>.

You can optionally specify a list of events by including one or more <onEvent> tags with one event type per tag. If these tags are omitted, the client status request produces a one-time request. Otherwise, the client status request registers event listeners for the specified events.

<request> subtags for client status requests	Description
<client>	You can include one or more <client> tags, with one connector address per tag. In the case of a one-time request, all of the connectors listed are included in the report. In the case of an event listener request, the server will begin to listen to each of these connectors.
<onEvent>	Specifies the events upon which the server should generate reports. You can include one or more <onEvent> tags, with one event type per tag. If these tags are omitted, the Client Status Request will produce a one-time request. Otherwise, the Client Status Request will be used to register event listeners for the specified events.
<persistent>	Specifies that the details information in this Client Status Request should be made persistent in the server database.
<replyAddr>	Specifies the return address for each report generated as a result of this request. If this tag is omitted, the default return address of reports is the reply address of the originating message.
<requestId>	A label for the report. This value is used as a label for the request and is included in each report generated as a result of this request. This makes it possible to distinguish which reports were generated from a particular request when multiple requests have been registered and to delete or modify outstanding requests.
<schedule>	See "Scheduling server management requests" on page 98.

condition tag

To filter the request, use <condition> subtags. You can use as many of the following subtags as you want in a <condition> tag. If you use more than one of the same tag, the values are logically "OR"ed together, whereas if you use two different tags, the values are logically "AND"ed together.

<condition> subtags	Description
<messageId>	Selects the message with a particular message ID.
<status>	Selects messages that currently have the status specified.
<priority>	Selects the messages that currently have the priority specified.
<address>	Selects the messages that are addressed to the specified address.
<originator>	Selects messages that originated from the specified client.

<condition> subtags	Description
<kind>	Filters either binary or text messages. For example, **<kind>text</kind>** filters text messages, and **<kind>binary</kind>** filters binary messages.
<property>	Selects messages that have the specified message property. To check a property name and value, use the syntax **<property>***property-name=property-value***</property>**. To check the existence of a property, use the format **<property>***property-name***</property>**.
<customRule>	Selects messages based on rules. See "Custom message requests" on page 95.

One-time client status requests

You create a one-time request by omitting <onEvent> tags from the client status request. In this case, a single report is generated that contains the current status information for each connector specified in the client status request.

The following XML message omits the <onEvent> tag and so is an example of a one-time request. It generates a single report containing the current status information for each connector specified in the <ClientStatusRequest> tag.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <ClientStatusRequest>
        <request>
            <replyAddr>ianywhere.connector.beajms\q11</replyAddr>
            <requestId>myOneTimeRequest</requestId>
            <client>ianywhere.server</client>
            <client>ianywhere.connector.beajms</client>
        </request>
    </ClientStatusRequest>
</actions>
```

On-event client status requests

To specify events for which you want the QAnywhere Server to generate status reports, include one or more <onEvent> tags in your client status request. Unlike one-time requests, the server will not immediately respond to the request, but instead will begin listening for events to occur. Each time one of these events is triggered, a report is sent containing information about the connector that caused the event.

The following events are supported for on-event requests:

Event	When it occurs
open	A closed connector is opened.
close	A previously opened or paused connector is closed.
statusChange	The status of the connector is changed from one state to another. Possible states are open and close.

Event	When it occurs
error	An unexpected error is thrown by the connector.
fatalError	An unhandled fatal error is thrown by the connector.
none	This never occurs. This effectively removes all previous event watches from the connector.

In the following example, the connector with address ianywhere.connector.beajms\q11 is sent a status report each time the server connector changes its status or generates an error.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <ClientStatusRequest>
        <request>
            <replyAddr>ianywhere.connector.beajms\q11</replyAddr>
            <requestId>myEventRequest</requestId>
            <client>ianywhere.server</client>
            <onEvent>statusChange</onEvent>
            <onEvent>error</onEvent>
        </request>
    </ClientStatusRequest>
</actions>
```

Multiple simultaneous requests

Each return address can have its own set of event listeners for any number of connectors, including the server connector. Adding an event listener to a connector will not disturb any other event listeners in the server (except possibly one that it is replacing).

Request replacement

If you add an event listener to a connector that already has an event listener registered to it by the same return address, it will replace the old listener with the new one. For example, if a statusChange listener for connector abc is registered to address x/y and you register an error listener for abc to address x/y, abc will no longer respond to statusChange events.

To register more than one event to the same address, you must create a single request with more than one <onEvent> tag.

Removing a request

If an event listener for a connector is registered to an address, you can remove the event listener by providing another client status request from the same address with the "none" event specified.

In the following example, all event listeners are removed for the server connector registered to the address ianywhere.connector.beajms\q11:

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <ClientStatusRequest>
        <request>
            <replyAddr>ianywhere.connector.beajms\q11</replyAddr>
            <client>ianywhere.server</client>
            <onEvent>none</onEvent>
        </request>
```



```
    </ClientStatusRequest>
</actions>
```

Persistent client status requests

To specify that the details of a request are saved into the global properties table on the message store (where they can be automatically reinstated after a server restart), include the <persistent> tag in a client status request. Persistence can be used with scheduled events and event listeners, but not one-time requests. The rules for adding and removing persistent requests are similar to those for regular requests, except that scheduled events and event listeners cannot be added separately. Instead, when adding a persistent request, the client must specify all event listeners and schedules for a particular connector/reply address pair in the same request.

The following example adds the event listener and schedule to ianywhere.connector.myConnector and makes them persistent. It also overwrites any previous persistent requests from this connector/reply address pair. A report will be sent every half hour, as well as any time a connector status change occurs.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <ClientStatusRequest>
        <request>
            <replyAddr>ianywhere.connector.beajms\q11</replyAddr>
            <client>ianywhere.connector.myConnector</client>
            <onEvent>statusChange</onEvent>
            <schedule>
                <everyminute>30</everyminute>
            </schedule>
            <persistent/>
        </request>
    </ClientStatusRequest>
</actions>
```

Client status reports

A client status report is generated by the server each time a report is requested by a connector or a registered event occurs. It is generated as a simple text message which does not contain any message properties.

Depending on what information is available at the time of the event, any of the following values may be included in each component report:

- client (always present)
- UTCDatetime (always present)
- vendorStatusDescription (always present)
- statusCode (always present)
- vendorStatusCode
- statusSubCode
- statusDescription

For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<ClientStatusReport>
    <requestId>myRequest</requestId>
    <componentReport>
        <client>ianywhere.server</client>
```



```
            <UTCDatetime>Tue May 31 13:53:02 EDT 2005</UTCDatetime>
            <statusCode>Running</statusCode>
            <vendorStatusDescription></vendorStatusDescription>
        </componentReport>
        <componentReport>
            <client>ianywhere.connector.beajms</client>
            <UTCDatetime>Tue May 31 13:53:02 EDT 2005</UTCDatetime>
            <statusCode>Not running</statusCode>
            <vendorStatusDescription></vendorStatusDescription>
        </componentReport>
    </ClientStatusReport>
```

Monitoring connectors

Each return address can have its own set of event listeners for any number of connectors, including the server connector. Adding an event listener to a connector will not disturb any other event listeners in the server (except possibly one that it is replacing).

Adding an event listener to a connector that already has an event listener registered to it by the same return address will replace the old listener with the new one. For example, if a statusChange listener for connector abc is registered to address x/y, and then you register an error listener for abc to address x/y, abc will no longer respond to statusChange events. To register more than one event to the same address, you must use a single request by specifying more than one <onEvent> tag.

Removing a request

If an event listener for a connector is registered to an address, you can remove the event listener by providing another client status request from the same address with the "none" event specified.

In the following example, all event listeners for the server connector registered to the address ianywhere.connector.beajms\q11 are removed.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <ClientStatusRequest>
        <request>
            <replyAddr>ianywhere.connector.beajms\q11</replyAddr>
            <client>ianywhere.server</client>
            <onEvent>none</onEvent>
        </request>
    </ClientStatusRequest>
</actions>
```

Event listener persistence

If a connector is closed, any event listeners it has registered to its address will persist in the server until the server is shut down. If the connector is reopened, the stored event listeners will become active again.

Connector states

A connector can be in one of 2 states:

- **running** The connector is accepting and processing incoming and outgoing messages. This state is reflected in the connector property ianywhere.connector.state=1.

- **not running** The connector is not accepting or processing incoming or outgoing messages. When the connector state is changed to "running" the connector will be initialized from scratch. This state is reflected in the connector property ianywhere.connector.state=2.

☞ For information about how to change the connector state, see "Modifying connectors" on page 105.

Setting server properties with a server management request

A <SetProperty> tag contains one or more <prop> tags, each of which specifies a property to set. Each prop tag consists of a <client> tag, a <name> tag, and a <value> tag. To delete a property, omit the <value> tag.

<prop> subtags	Description
<client>	The name of the client for which to set a server property.
<name>	The name of the property to set.
<value>	The value of the property being set. If not included, the property will be deleted.

☞ For an overview of how to use server management requests, including how to authenticate and schedule them, see "About server management requests" on page 92.

Example

The following server management request sets the ianywhere.qa.member.client3 property to Y for the destination alias called simpleGroup, which adds client3 to simpleGroup.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <SetProperty>
     <prop>
        <client>simpleGroup</client>
        <name>ianywhere.qa.member.client3</name>
        <value>Y</value>
     </prop>
    </SetProperty>
</actions>
```

The next example does the following:

- Creates or modifies the value of the client1 property myProp1 to 3.
- Deletes the client1 property myProp2.
- Modifies the value of the client2 property myProp3 to "some value".

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <SetProperty>
    <prop>
        <client>client1</client>
        <name>myProp1</name>
        <value>3</value>
    </prop>
    <prop>
        <client>client1</client>
        <name>myProp2</name>
    </prop>
    <prop>
        <client>client2</client>
        <name>myProp3</name>
```



```
            <value>some value</value>
        </prop>
        </SetProperty>
    </actions>
```


Specifying transmission rules with a server management request

With a server management request, you can specify default server transmission rules that apply to all users, or you can specify transmission rules for each client. You specify server transmission rules using the ianywhere.qa.server.rules property for a client.

To specify default transmission rules, set the ianywhere.qa.server.rules property for the client ianywhere.server.defaultClient.

☞ For an overview of how to use server management requests, including how to authenticate and schedule them, see "About server management requests" on page 92.

Example

The following example creates the default rule that only high priority messages (priority greater than 6) should be sent:

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
  <SetProperty>
    <prop>
      <client>ianywhere.server.defaultClient</client>
      <name>ianywhere.qa.server.rules</name>
      <value>auto = ias_Priority &gt; 6</value>
    </prop>
  </SetProperty>
  <RestartRules>
    <client>ianywhere.server.defaultClient</client>
  </RestartRules>
</actions>
```

The following example creates a rule for client sample_store_id that only messages with a content size greater than 100 should be transmitted during business hours (8 a.m. and 6 p.m.):

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
  <SetProperty>
    <prop>
      <client>ianywhere.server.defaultClient</client>
      <name>ianywhere.qa.server.rules</name>
      <value>auto = ias_ContentSize &lt; 100
        or ias_CurrentTime &lt; '8:00:00'
        or ias_CurrentTime &gt; '18:00:00'</value>
    </prop>
  </SetProperty>
  <RestartRules>
    <client>ianywhere.server.defaultClient</client>
  </RestartRules>
</actions>
```


Creating destination aliases using server management requests

You can use server management requests to crete and modify destination aliases.

☞ For more information about destination aliases, see “Destination aliases” on page 50.

☞ For an overview of how to use server management requests, including how to authenticate and schedule them, see “About server management requests” on page 92.

To create a destination alias, send a server management request in which the client name is the name of the destination alias and the following properties are specified. The group is identified by the group, address, and nativeConnection properties. Members of the group are specified with the member property.

```
<prop>
  <client>simpleGroup</client>
  <name>ianywhere.connector.nativeConnection</name>
  <value>ianywhere.message.connector.group.GroupConnector
  </value>
</prop>
```

Property	Description

ianywhere.qa.group	Set this property to Y to indicate that you are configuring a destination alias. For example:
	
	`<prop>`
	`  <client>simpleGroup</client>`
	`  <name>ianywhere.qa.group</name>`
	`  <value>Y</value>`
	`</prop>`

ianywhere.connector.address	Specify the client ID of the destination alias. For example:
	
	`<prop>`
	`  <client>simpleGroup</client>`
	`  <name>ianywhere.connector.address</name>`
	`  <value>simpleGroup</value>`
	`</prop>`

ianywhere.connector.nativeConnection	Set to ianywhere.message.connector.group.GroupConnector. For example:
	
	`<prop>`
	`  <client>simpleGroup</client>`
	`  <name>ianywhere.connector.nativeConnection</name>`
	
	`  <value>ianywhere.message.connector.group.GroupConnector`
	`  </value>`
	`</prop>`

Property	Description

ianywhere.qa.member.*client-name\queue-name*	Specify Y to add a member or N to remove a member. You can also optionally specify a delivery condition. See "Condition syntax" on page 228. For example, to add client1 to the destination alias simpleGroup, set the property as follows. The queue-name is optional. Repeat this property for every client you want to add:
	
	`<prop>`
	`  <client>simpleGroup</client>`
	`  <name>ianywhere.qa.member.client1\queue1</name>`
	`  <value>Y</value>`
	`</prop>`

☞ For more information about server management requests, see "About server management requests" on page 92.

See also

- "QAnywhere Transmission and Delete Rules" on page 225

Example

The following server management request creates a destination alias called simpleGroup with members called client1 and client2\queue11. This example starts the destination alias so that it immediately begins handling messages.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
  <SetProperty>
    <prop>
      <client>simpleGroup</client>
      <name>ianywhere.qa.group</name>
      <value>Y</value>
    </prop>
    <prop>
      <client>simpleGroup</client>
      <name>ianywhere.connector.address</name>
      <value>simpleGroup</value>
    </prop>
    <prop>
      <client>simpleGroup</client>
      <name>ianywhere.connector.nativeConnection</name>
      <value>ianywhere.message.connector.group.GroupConnector</value>
    </prop>
    <prop>
      <client>simpleGroup</client>
      <name>ianywhere.connector.logLevel</name>
      <value>4</value>
    </prop>
    <prop>
      <client>simpleGroup</client>
      <name>ianywhere.qa.member.client1</name>
      <value>Y</value>
    </prop>
    <prop>
      <client>simpleGroup</client>
      <name>ianywhere.qa.member.client2\q11</name>
      <value>Y</value>
    </prop>
```



```
  </SetProperty>
  <OpenConnector>
    <client>simpleGroup</client>
  </OpenConnector>
</actions>
```

Adding and removing members in a destination alias

To add members to a destination alias, create a server management request that specifies the member in a property. The group must be restarted for the member setting to take effect.

The following example adds the member client3 and restarts the group simpleGroup:

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <SetProperty>
    <prop>
        <client>simpleGroup</client>
        <name>ianywhere.qa.member.client3</name>
        <value>Y</value>
    </prop>
    </SetProperty>
    <CloseConnector>
    <client>simpleGroup</client>
    </CloseConnector>
    <OpenConnector>
    <client>simpleGroup</client>
    </OpenConnector>
</actions>
```

To remove members from a destination alias, create a server management request that contains a property setting indicating that the member must be removed. The group must be restarted for the member removal setting to take effect.

The following example removes the member client3 and restarts the group simpleGroup:

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <SetProperty>
    <prop>
        <client>simpleGroup</client>
        <name>ianywhere.qa.member.client3</name>
    </prop>
    </SetProperty>
    <CloseConnector>
    <client>simpleGroup</client>
    </CloseConnector>
    <OpenConnector>
    <client>simpleGroup</client>
    </OpenConnector>
</actions>
```


Monitoring QAnywhere

You can use a server management request to get information about a set of messages. The server compiles the information and sends it back to the client in a message. You can create a one-time message details request or schedule your message details request to happen automatically. In addition, you can specify that your request should be persistent, so that the message is sent even if the server is restarted.

☞ For an overview of how to use server management requests, including how to authenticate and schedule them, see "About server management requests" on page 92.

Message details requests

To write a server management request for message details, use the <MessageDetailsRequest> tag.

A message details request contains one or more <request> tags containing all the necessary information to register the request. Specifying more than one <request> tag is equivalent to sending multiple separate message details requests.

Use the optional <replyAddr> tag to specify the return address for each report generated as a result of the request. If this tag is omitted, the default return address of reports is the reply address of the originating message.

Use a <requestId> tag to specify a unique identifier for the request that is included in each report generated as a result of this request. Using different values for this field allows more than one request to be active at the same time. Using the same request ID allows the client to override or delete active requests.

Specify a <condition> tag to determine which messages should be included in the report. See "Condition tag" on page 94.

You can also specify a list of details to determine what details of each message should be included in the report. You do this by by including a set of empty detail element tags in the request.

You can use the <persistent> tag to specify that event details should be made persistent during any server downtime. This tag does not require any data and can be of the form <persistent/> or <persistent></persistent>.

You can use <schedule> to include all the necessary details needed to register a scheduled report. See "Scheduling server management requests" on page 98.

<MessageDetailsRequest> sub-tags	**Description**
<request>	Groups information about a particular request. Specifying more than one <request> tag is equivalent to sending multiple separate server managment requests for message information. See below.

Request tag

<Request> subtags	Description
<address>	Displays the address of each message.
<condition>	Groups conditions for including a message in the report. See “Condition tag” on page 94.
<contentSize>	Requests the content size of each message.
<customRule>	See “Custom message requests” on page 95.
<expires>	Requests the expiration time of each message.
<kind>	Requests whether the message is text or binary.
<messageId>	Requests the message ID of each message.
<originator>	Requests the originator or each message.
<persistent>	Including this tag indicates that the results of the request should be made persistent in the server database (so that the report is sent even if the server is restarted).
<priority>	Requests the priority of each message.
<property>	Requests a list of all message properties and values for each message.
<statusTime>	Requests the status time of each message.
<replyAddr>	Specifies the return address for each report generated as a result of this request. If this tag is omitted, the default return address of reports is the reply address of the originating message.
<requestId>	This value is a unique identifier for the request and is included in each report generated as a result of this request. Using different values for this field allows more than one request to be active at the same time. Using the same request id allows the client to override or delete active requests.
<schedule>	Including this tag indicates that the report should be generated on a schedule. Subtags of <schedule> identify the schedule on which the report runs. ☞ See “Scheduling server management requests” on page 98.
<status>	Requests the status of each message.

MessageDetailsReport tag

Each Message Details Report is an XML message containing the <MessageDetailsReport> tag, and is composed of a report header followed by optional <message> tags. The header of each report consists of the following tags:

<MessageDetailsReport> subtags	Description
<message>	The body of the report consists of a list of <message> tags whose subtags display the specific details of each message that satisfied the selection criteria. If no messages were selected, or no detail elements were specified in the original request, then no <message> tags will be included in the report. Otherwise, each message will have its own <message> tag.
<messageCount>	The number of messages that satisfy the selection criteria of the request.
<requestId>	The ID of the request that generated the report.
<statusDescription>	A brief description of the reason why this report was generated.
<UTCDateline>	The time and date that this report was generated.

Examples

Following is an example of a message details report:

```
<?xml version="1.0" encoding="UTF-8"?>
<MessageDetailsReport>
    <requestId>testReport</requestId>
    <UTCDatetime>Mon Jan 16 15:03:04 EST 2006</UTCDatetime>
    <statusDescription>Scheduled report</statusDescription>
    <messageCount>1</messageCount>
    <message>
        <messageId>ID:26080b8927f83f9722357eab0a0628eb</messageId>
        <status>60</status>
        <property>
            <name>myPropName</name>
            <value>myPropVal</value>
        </property>
    </message>
</MessageDetailsReport>
```

The following condition selects messages following the search criteria: (msgId=ID:144... OR msgId=ID225...) AND (status=pending) AND (kind=textmessage) AND (contains the property 'myProp' with value 'myVal')

```
<condition>
    <messageId>ID:144d7e44dc2d7e1d</messageId>
    <messageId>ID:22578sd5dsd99s8e</messageId>
    <status>1</status>
    <kind>text</kind>
    <property>myProp=myVal</property>
</condition>
```

A one-time request is a request that has omitted the <schedule> tag. These requests are used to generate a single report and are deleted as soon as the report has been sent. This request generates a single report that displays the message id, status, and target address of all messages with priority 9 currently on the server.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
```



```
    <MessageDetailsRequest>
        <request>
            <requestId>testRequest</client>
            <condition>
                <priority>9</priority>
            </condition>
            <messageId/>
            <status/>
            <address/>
        </request>
    </MessageDetailsRequest>
</actions>
```

The following sample message details request generates a report that includes the message ID and message status.

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
  <MessageDetailsRequest>
    <!-- ... -->
    <messageId />
    <status />
  </MessageDetailsRequest>
</actions>
```


Monitoring QAnywhere clients

You can use a server management request to obtain a list of clients currently on the server. This list contains clients who are registered on the server, including remote clients, open connectors, and destination aliases.

☞ For an overview of how to use server management requests, including how to authenticate and schedule them, see "About server management requests" on page 92.

To obtain a list of clients, use the <GetClientList> tag in your server management request. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <GetClientList/>   (or <GetClientList></GetClientList> )
</actions>
```

The response that is generated is sent to the reply address of the message containing the request. The response contains a list of <client> tags, each naming one client connected to the server. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<GetClientListResponse>
    <client>ianywhere.server</client>
    <client>ianywhere.connector.myConnector</client>
 <client>myClient</client>
</GetClientListResponse>
```


Monitoring properties

You can use a server management request to see what properties are set for a client. The response lists only the properties that have been set for the client (not defaults).

☞ For an overview of how to use server management requests, including how to authenticate and schedule them, see "About server management requests" on page 92.

To get a list of properties for a client, use the <GetProperties> tag in your server management request. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<actions>
    <GetProperties>
        <client>ianywhere.connector.myConnector</client>
    </GetProperties>
</actions>
```

The response that is generated is sent to the reply address of the message containing the request. The response contains the name of the client and a list of <prop> tags, each containing the details of one property. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<GetPropertiesResponse>
    <client>ianywhere.connector.myConnector</client>
    <prop>
        <name>ianywhere.connector.logLevel</name>
        <value>4</value>
    </prop>
    <prop>
        <name>ianywhere.connector.state</name>
        <value>2</value>
    </prop>
</GetPropertiesResponse>
```


CHAPTER 6

JMS Connectors

Contents

About this chapter

This chapter describes how to write QAnywhere client applications to exchange messages with external messaging systems that support a JMS interface.

Introduction

The Java Message Service (JMS) API provides messaging capabilities to Java applications. In addition to exchanging messages among QAnywhere client applications, you can exchange messages with external messaging systems that support a JMS interface. You do this using a specially configured client known as a connector. In a QAnywhere application, the external messaging system is set up to act like a QAnywhere client. It has its own address and configuration.

☞ For more information about the architecture of this approach, see "Scenario for messaging with external messaging systems" on page 11.

Setting up JMS connectors

The following steps provide an overview of the tasks required to set up QAnywhere with JMS connectors, assuming that you already have QAnywhere set up.

♦ **Overview of integrating a QAnywhere application with an external JMS system**

1. Create JMS queues using the JMS administration tools for your JMS system. The QAnywhere connector listens on a single JMS queue for JMS messages. You must create this queue if it does not already exist.

   ☞ See the documentation of your JMS product for information about how to create queues.

2. Open Sybase Central and connect to your server message store.
3. Choose File ► New Connector.

   The Connector wizard appears.

4. Ensure that JMS is selected and then select the type of web server you are using. Click Next.
5. In the Connector Names page, enter the following values:

   - **Connector name** The connector address that a QAnywhere client should use to address the connector.

     ☞ See "Addressing QAnywhere messages meant for JMS" on page 137.

   - **Receiver destination** The queue name used by the connector to listen for messages from JMS targeted for QAnywhere clients.

6. In the JNDI Settings page, enter the following values:

   - **JNDI factory** The factory name used to access the external JMS JNDI name service.
   - **Name service URL** The URL to access the JMS JNDI name service.
   - **User name** The authentication name to connect to the external JMS JNDI name service.
   - **Password** The authentication password to connect to the external JMS JNDI name service.

7. In the JMS Queue Settings page, enter the following values:

   - **Queue factory** The external JMS provider queue factory name.
   - **User name** The user ID to connect to the external JMS queue connection.
   - **Password** The password to connect to the external JMS queue connection.

8. In the JMS Topic Settings page, enter the following values:

   - **Topic factory** The external JMS provider topic factory name.
   - **User name** The user ID to connect to the external JMS topic connection.

- **Password** The password to connect to the external JMS topic connection.

9. Click Finish.

   You are prompted to add the client JAR files in the mlsrv10 command line.

10. Start the MobiLink server with a connection to the server message store and the -sl java option.

    ☞ See "Starting the MobiLink server for JMS integration" on page 131.

11. To set additional options on your JMS connector, right-click the connector you just created and choose properties; or you can use server management requests.

    ☞ For a list of available properties, see "JMS connector properties" on page 132.

    For information about how to set connector properties with server management requests, see "Administering connectors" on page 104.

To send messages

1. To send a message from an application in your QAnywhere system to the external messaging system, create a QAnywhere message and send it to ***connector-address\JMS-queue-name***.

   ☞ See "Addressing QAnywhere messages meant for JMS" on page 137.

2. To send a message from the external messaging system to an application in your QAnywhere system:

   - Create a JMS message.
   - Set the ias_ToAddress property to the QAnywhere ***id\queue*** (where *id* is the ID of your client message store and *queue* is your application queue name).
   - Put the message in the JMS queue.

   ☞ See "Addressing JMS messages meant for QAnywhere" on page 139.

Other resources for getting started

- QAnywhere JMS samples are installed to *samples-dir\QAnywhere\connectors*. (For information about *samples-dir*, see "The samples directory" [*SQL Anywhere Server - Database Administration*].)

Starting the MobiLink server for JMS integration

To exchange messages with an external messaging system that supports a JMS interface, you must start the MobiLink server (mlsrv10) with the following options:

- **-c *connection-string*** To connect to the server message store.

  ☞ See "-c option" [*MobiLink - Server Administration*].

- **-m** To enable QAnywhere messaging.
- **-sl java (-cp "*jarfile*.jar")** To add the client jar files required to use the external JMS provider.

  ☞ See "-sl java option" [*MobiLink - Server Administration*].

Example

The following example starts a MobiLink server using a JMS client library called *jmsclient.jar* (in the current working directory) and the QAnywhere sample database as a message store. The command should be entered all on one line.

```
mlsrv10 -sl java(-cp "jmsclient.jar")
             -m -c "QAnywhere 10.0 Demo" ...
```

JMS connector properties

You use JMS connector properties to specify connection information with the JMS system. They configure a connector to a third party JMS messaging system such as BEA WebLogic or Sybase EAServer.

You can set and/or view properties in several places:

- Sybase Central Connector Wizard.

  See "Setting up JMS connectors" on page 129.
- Sybase Central Connector Properties dialog.
- Server management requests.

  See "Creating and configuring connectors" on page 104.
- The ml_qa_global_props MobiLink system table.

  See "ml_qa_global_props" [*MobiLink - Server Administration*].

The following properties are used to configure the JMS connector:

- **ianywhere.connector.nativeConnection** The Java class that implements the connector. It is for QAnywhere internal use only, and should not be deleted or modified.
- **ianywhere.connector.id (deprecated)** An identifier that uniquely identifies the connector. The default is the value of the connector property ianywhere.connector.address.
- **ianywhere.connector.address** The connector address that a QAnywhere client should use to address the connector. This address is also used to prefix all logged error, warning, and informational messages appearing in the server console for this connector.

  ☞ For more information, see "Addressing QAnywhere messages meant for JMS" on page 137.

  In Sybase Central, set this property in the Connector wizard, Connector Names page, Connector Name field.
- **ianywhere.connector.incoming.retry.max** The maximum number of times the connector will retry transferring a JMS message to a QAnywhere message store before giving up. After the maximum number of failed attempts, the JMS message is re-addressed to the ianywhere.connector.jms.deadMessageDestination property value. The default is -1, which means that the connector will never give up.
- **ianywhere.connector.outgoing.deadMessageAddress** The address that a message is sent to when it cannot be processed. For example, if a message contains a JMS address that is malformed or unknown, the message is marked as unreceivable and a copy of the message is sent to the dead message address.

  If no dead message address is specified, the message is marked as unreceivable but no copy of the message is sent.

In Sybase Central, you can set this property in the connector properties dialog Properties tab, by clicking New.

- **ianywhere.connector.logLevel** The amount of connector information displayed in the MobiLink server console and log file. Values for the log level are as follows:
  - **1** Log error messages.
  - **2** Log error and warning messages.
  - **3** Log error, warning, and information messages.
  - **4** Log error, warning, information, and debug messages.

  In Sybase Central, set this property on the connector properties dialog, on the General tab, in the Logging Level section.

  You can also set this property for all connectors. To do this in Sybase Central, connect to a server message store and choose the task Change Properties of this Message Store. Open the Server Properties tab.
- **ianywhere.connector.compressionLevel** The default message compression factor of messages received from JMS: an integer between 0 and 9, with 0 indicating no compression and 9 indicating maximum compression.

  In Sybase Central, set this property on the connector properties dialog, on the General tab, in the Compression Level section.

  You can also set this property for all connectors. To do this in Sybase Central, connect to a server message store, choose the task Change Properties of this Message Store, and open the Server Properties tab.
- **ianywhere.connector.jms.deadMessageDestination** The address that a JMS message is sent to when it cannot be converted to a QAnywhere message. This might occur if the JMS message is an instance of an unsupported class, if the JMS message does not specify a QAnywhere address, if an unexpected JMS provider exception occurs, or if an unexpected QAnywhere exception occurs.

  In Sybase Central, set this property on the connector properties dialog, on the JMS tab, in the Other section, in the Dead Message Destination field.
- **ianywhere.connector.outgoing.retry.max** The default number of retries for messages going from QAnywhere to the external messaging system. The default value is 5. Specify 0 to have the connector retry forever.

  In Sybase Central, you can set this property in the connector properties dialog, Properties tab, by clicking New.
- **ianywhere.connector.runtimeError.retry.max** The number of times a connector retries a message that causes a RuntimeException. If a dead message queue is specified, the message is put in that queue. Otherwise, the message is marked as unreceivable and skipped. Specify a value of 0 to have the server never give up.
- **ianywhere.connector.startupType** Startup types can be automatic, manual, or disabled.
- **xjms.jndi.authName** The authentication name to connect to the external JMS JNDI name service.

In Sybase Central, set this property in the Connector wizard, JNDI Settings page, User Name field; or on the connector properties dialog on the JMS tab, JNDI section, User Name field.

- **xjms.jndi.factory** The factory name used to access the external JMS JNDI name service.

  In Sybase Central, set this property in the Connector wizard, JNDI Settings page, Password field; or on the connector properties dialog on the JMS tab, JNDI section, Password field.

- **xjms.jndi.password.e** The authentication password to connect to the external JMS JNDI name service.

  In Sybase Central, set this property in the Connector wizard, JNDI Settings page, Name Service URL field; or on the connector properties dialog on the JMS tab, JNDI section, URL field.

- **xjms.jndi.url** The URL to access the JMS JNDI name service.

  In Sybase Central, set this property in the Connector wizard, JNDI Settings page, Name Service URL field; or on the connector properties dialog on the JMS tab, JNDI section, URL field.

- **xjms.password.e** The authentication password to connect to the external JMS provider.

- **xjms.queueConnectionAuthName** The user ID to connect to the external JMS queue connection.

  In Sybase Central, set this property in the Connector wizard, JMS Queue Settings page, User Name field; or on the connector properties dialog on the JMS tab, Queue section, User Name field.

- **xjms.queueConnectionPassword.e** The password to connect to the external JMS queue connection.

  In Sybase Central, set this property in the Connector wizard, JMS Queue Settings page, Password field; or on the connector properties dialog on the JMS tab, Queue section, Password field.

- **xjms.queueFactory** The external JMS provider queue factory name.

  In Sybase Central, set this property in the Connector wizard, JMS Queue Settings page, Queue Factory field; or on the connector properties dialog on the JMS tab, Queue section, Queue Factory field.

- **xjms.receiveDestination** The queue name used by the connector to listen for messages from JMS targeted for QAnywhere clients.

  In Sybase Central, set this property in the Connector wizard, Connector Names page, Receiver Destination field.

- **xjms.topicFactory** The external JMS provider topic factory name.

  In Sybase Central, set this property in the Connector wizard, JMS Topic Settings page, Topic Factory field; or on the connector properties dialog on the JMS tab, Topic section, Topic Factory field.

- **xjms.topicConnectionAuthName** The user ID to connect to the external JMS topic connection.

  In Sybase Central, set this property in the Connector wizard, JMS Topic Settings page, User Name field; or on the connector properties dialog on the JMS tab, Topic section, User Name field.

- **xjms.topicConnectionPassword.e** The password to connect to the external JMS topic connection.

In Sybase Central, set this property in the Connector wizard, JMS Topic Settings page, Password field; or on the connector properties dialog on the JMS tab, Topic section, Password field.

Configuring multiple connectors

QAnywhere can connect to multiple JMS message systems by defining a JMS connector for each JMS system. The only property value that must be unique among the configured connectors is ianywhere.connector.address.

The ianywhere.connector.address property is the address prefix that QAnywhere clients must specify to address messages meant for the JMS system.

☞ For information about specifying the address of QAnywhere clients, see "Addressing QAnywhere messages meant for JMS" on page 137.

☞ For information about the connector properties, see "JMS connector properties" on page 132.

☞ For information about configuring connectors, see "Creating and configuring connectors" on page 104.

Addressing QAnywhere messages meant for JMS

A QAnywhere client can send a message to a JMS system by setting the address to the following value:

*connector-address**JMS-queue-name*

The *connector-address* is the value of the connector property ianywhere.connector.address, while *JMS-queue-name* is the name used to look up the JMS queue or topic using the Java Naming and Directory Interface.

If your *JMS-queue-name* contains a backslash, you must escape the backslash with another backslash. For example, a queue called qq in the context ss should be specified as `ss\\qq`.

```
// C# example
QAMessage msg;
QAManager mgr;
...
mgr.PutMessage( @"ianywhere.connector.wsmqfs\ss\\qq",msg );

// C++ example
QAManagerBase *mgr;
QATextMessage *msg;
...
mgr->putMessage( "ianywhere.connector.easerver\\ss\\\\qq", msg );
```

Example

For example, if the ianywhere.connector.address is set to ianywhere.connector.easerver and the JMS queue name is myqueue, then the code to set the address would be:

```
// C# example
QAManagerBase mgr;
QAMessage msg;
// Initialize the manager.
...
msg = mgr.CreateTextMessage();
// Set the message content.
...
mgr.PutMessage(@"ianywhere.connector.easerver\myqueue", msg );

 // C++ example
QAManagerBase *mgr;
QATextMessage *msg;
// Initialize the manager.
...
msg = mgr.createTextMessage();
// Set the message content.
...
mgr->putMessage( "ianywhere.connector.easerver\\myqueue", msg );
```

See also

- "QAnywhere message addresses" on page 50
- "JMS connector properties" on page 132

Mapping QAnywhere messages on to JMS messages

QAnywhere messages are mapped naturally on to JMS messages.

QAnywhere message content

QAnywhere	JMS	Remarks
QATextMessage	javax.jms.TextMessage	message text copied as Unicode
QABinaryMessage	javax.jms.BytesMessage	message bytes copied exactly

QAnywhere built-in headers

The following table describes the mapping of built-in headers. In C++ and JMS, these are method names; for example, Address is called getAddress() or setAddress() for QAnywhere, and getJMSDestination() or setJMSDestination() for JMS. In .NET, these are properties with the exact name given below; for example, Address is Address.

QAnywhere	JMS	Remarks
Address	JMSDestination and JMS property ias_ToAddress	If the destination contains a backslash, you must escape it with a second backslash. Only the JMS part of the address is mapped to the Destination. Under rare circumstances, in the case of a message looping back into QAnywhere, there may be an additional QAnywhere address suffix. This is put in ias_ToAddress.
Expiration	JMSExpiration	
InReplyToID	N/A	Not mapped.
MessageID	N/A	Not mapped.
Priority	JMSPriority	
Redelivered	N/A	Not mapped.
ReplyToAddress	JMS property ias_ReplyToAddress	Mapped to JMS property.
Connector's xjms.receiveDestination property value	JMSReplyTo	ReplyTo set to Destination used by connector to receive JMS messages.
Timestamp	N/A	Not mapped.

QAnywhere properties

QAnywhere properties are all mapped naturally to JMS properties, preserving type, with the following exception: if the QAnywhere message has a property called JMSType, then this is mapped to the JMS header property JMSType.

Addressing JMS messages meant for QAnywhere

A JMS client can send a message to a QAnywhere client by setting the JMS message property ias_ToAddress to the QAnywhere address, and then sending the message to the JMS Destination corresponding to the connector property xjms.receiveDestination.

☞ For more information, see "QAnywhere message addresses" on page 50.

Example

For example, to send a message to the QAnywhere address "qaddr" (where the connector setting of xjms.receiveDestination is "qanywhere_receive"):

```
import javax.jms.*;
...
try {
  QueueSession session;
  QueueSender sender;
  TextMessage mgr;
  Queue connectorQueue;
  // Initialize the session.
  ...
  connectorQueue = session.createQueue( "qanywhere_receive" );
  sender = session.createSender( connectorQueue );
  msg = session.createTextMessage();
  msg.setStringProperty( "ias_ToAddress", "qaddr" );
  // Set the message content.
  ...
  sender.send( msg );
} catch( JMSException e ) {
  // Handle the exception
  ...
}
```

Mapping JMS messages on to QAnywhere messages

JMS messages are mapped naturally on to QAnywhere messages.

JMS message content

JMS	QAnywhere	Remarks
javax.jms.TextMessage	QATextMessage	Message text copied as Unicode
javax.jms.BytesMessage	QABinaryMessage	Message bytes copied exactly
javax.jms.StreamMessage	N/A	Not supported

JMS	QAnywhere	Remarks
javax.jms.MapMessage	N/A	Not supported
javax.jms.ObjectMessage	N/A	Not supported

JMS built-in headers

The following table describes the mapping of built-in headers. In C++ and JMS, these are method names; for example, Address is called getAddress() or setAddress() for QAnywhere, and getJMSDestination() or setJMSDestination() for JMS. In .NET, these are properties with the exact name given below; for example, Address is Address.

JMS	QAnywhere	Remarks
JMS Destination	N/A	The JMS destination must be set to the queue specified in the connector property xjms.receiveDestination.
JMS Expiration	Expiration	
JMS CorrelationID	InReplyToID	
JMS MessageID	N/A	Not mapped.
JMS Priority	Priority	
JMS Redelivered	N/A	Not mapped.
JMS ReplyTo and connector's ianywhere.connector.address property value	ReplyToAddress	The connector address is concatenated with the JMS ReplyTo Destination name delimited by '\'.
JMS DeliveryMode	N/A	Not mapped.
JMS Type	QAnywhere message property JMSType	
JMS Timestamp	N/A	Not mapped.

JMS properties

JMS properties are all mapped naturally to QAnywhere properties, preserving type, with a few exceptions. The QAnywhere Address property is set from the value of the JMS message property ias_ToAddress. If the JMS message property ias_ReplyToAddress is set, then the QAnywhere ReplyToAddress is additionally suffixed with this value delimited by a '\'.

Tutorial: Using JMS connectors

A JMS connector provides connectivity between a JMS message system and QAnywhere. This tutorial sends messages between a JMS system and QAnywhere.

About the tutorial

This tutorial starts a JMS connector and sends a message from a JMS client to a QAnywhere client.

Required software

For this tutorial, you need access to a JMS provider and basic knowledge of how to configure it. In addition, you need JDK version 1.3.1 or later and any JAR files required by a JMS client of the JMS provider.

Lesson 1: Start a JMS connector

♦ To prepare your JMS provider

1. Start your JMS server.

   ☞ See the documentation for your JMS server.

2. Create two queues within your JMS server: qa_testmessage and qa_receive. You may need to restart your JMS server after creating the queues.

   ☞ See the documentation for your JMS server.

♦ To start QAnywhere client and server components

1. Create a directory to hold the files you create for this tutorial. For example, *c:\JMSTestMessage*. Navigate to that directory.
2. Create a QAnywhere connector:

   ♦ In Sybase Central, choose File ► New Connector and following the prompts in the Connector wizard.

   For more information, see "Setting up JMS connectors" on page 129.

3. Start the MobiLink server for messaging:

   From the Windows Start menu, choose Programs ► SQL Anywhere 10 ► MobiLink ► MobiLink with Messaging Sample.

   Alternatively, at a command prompt, navigate to *samples-dir\QAnywhere\server* and type the following command:

   ```
   mlsrv10 -m -c "dsn=QAnywhere 10.0 Demo" -sl java(-cp "jarfiles") -vcrs -zu
   +
   ```

4. Start the QAnywhere Agent:

From the Start menu, choose Programs ▶ SQL Anywhere 10 ▶ QAnywhere ▶ Agent for Client1 Sample.

5. Start the TestMessage sample:

   From the Windows Start menu, choose Programs ▶ SQL Anywhere 10 ▶ QAnywhere ▶ TestMessage for Client1 Sample.

♦ To start the Java version of the TestMessage client

1. At a command prompt, navigate to *Samples\QAnywhere\connectors\JMS\TestMessage* and type the following:

   ```
   java -cp .;JMS-client-jar-files ianywhere.message.samples.TestMessage
   ```

   where *JMS-client-jar-files* is a semi-colon delimited list of jar files that are required to access the JMS server. See your JMS server documentation for details.

   For Sybase EAServer, this command would be:

   ```
   java -cp .;path\easclient.jar;path\easj2ee.jar
   ianywhere.message.samples.TestMessage
   ```

   where *path* is the location of the jar files.

   Note
   On Unix, use colons instead of semicolons.

2. Move the JMS TestMessage window to the right side of your screen under the existing TestMessage for Client1 window.

Lesson 2: Send a message from a JMS client to a QAnywhere client

♦ To send a message from a JMS client to a QAnywhere client

1. From the JMS TestMessage Message menu, choose New.

   The New Message window appears.

2. In the To field, enter the client message store ID of client1.
3. Fill out the Subject and Message fields with sample text, and click Send.
4. Within a short time a message box appears, indicating that a message has been received by TestMessage for Client2.

Tutorial cleanup

Shut down TestMessage clients, the QAnywhere Agent, and the MobiLink server.

CHAPTER 7

QAnywhere Agent

Contents

About this chapter

This chapter describes the syntax and options of the QAnywhere Agent, qaagent.

☞ For an overview of how to run the QAnywhere Agent, see "Running the QAnywhere Agent" on page 35.

qaagent syntax

Use the QAnywhere Agent to send and receive messages for all QAnywhere applications on a single client device.

Syntax

qaagent [*option* ...]

Option	Description
@data	Reads options from the specified environment variable or configuration file. See "@data option" on page 147.
-c *connection-string*	Specifies a connection string to the client message store. See "-c option" on page 148.
-id *id*	Specifies the ID of the client message store that the QAnywhere Agent is to connect to. See "-id option" on page 152.
-iu *upload-size*	Specifies the maximum size of an upload to use during a message transmission . See "-iu option" on page 153.
-lp *number*	Specifies the port on which the Listener listens for notifications from the MobiLink server. The default is 5001. See "-lp option" on page 154.
-mp *password*	Specifies a new password for the MobiLink user. See "-mp option" on page 156.
-mp *password*	Specifies the MobiLink password for the ID being synchronized. See "-mp option" on page 156.
-mu *username*	Specifies the MobiLink user. See "-mu option" on page 157.
-o *logfile*	Specifies a file to which to log output messages. See "-o option" on page 158.
-on *size*	Specifies a maximum size for the QAnywhere Agent message log file, after which the file is renamed with the extension .old and a new file is started. See "-on option" on page 159.
-os *size*	Specifies a maximum size for the QAnywhere Agent message log file, after which a new log file with a new name is created and used. See "-os option" on page 160.
-ot *logfile*	Specifies a file to which to log output messages. See "-ot option" on page 161.
-pc{**+**|**-**}	Enables persistent connections for message transmission. See "-pc option" on page 162.
-policy *policy-type*	Specifies the transmission policy used by the QAnywhere Agent. See "-policy option" on page 163.

Option	Description
-push *mode*	Enables or disables push notifications. The default is enabled. See "-push option" on page 165.
-q	Starts the QAnywhere Agent in quiet mode with the window minimized in the system tray. See "-q option" on page 167.
-qi	Starts the QAnywhere Agent in quiet mode with the window completely hidden. See "-qi option" on page 168.
-si	Initializes the database for use as a client message store. See "-si option" on page 169.
-su	Upgrades a client message store to the current version without running dbunload/reload. See "-su option" on page 170.
-sur	Upgrades a client message store to the current version and performs dbunload/reload of the message store. See "-sur option" on page 171
-v [*levels*]	Specifies a level of verbosity. See "-v option" on page 172.
-x { **http**|**tcpip**|**tls**|**https** } [**(***keyword***=***value***;...)**]	Specifies protocol options for communication with the MobiLink server. See "-x option" on page 173.

Usage

For usage information about qaagent, see "Running the QAnywhere Agent" on page 35.

@data option

Function

Reads options from the specified environment variable or configuration file.

Syntax

qaagent @{ *filename* | *environment-variable* } ...

Remarks

With this option, you can put command line options in an environment variable or configuration file. If both exist with the name you specify, the environment variable is used.

☞ For more information about configuration files, see "Using configuration files" [*SQL Anywhere Server - Database Administration*].

If you want to protect passwords or other information in the configuration file, you can use the File Hiding utility to obfuscate the contents of the configuration file.

This option is useful for Windows CE because command lines in shortcuts are limited to 256 characters.

☞ See "File Hiding utility (dbfhide)" [*SQL Anywhere Server - Database Administration*].

Sybase Central equivalent

The QAnywhere plug-in to Sybase Central has a task called Create An Agent Command File. When you choose it, you are prompted to enter a file name and then a Properties dialog appears that helps you enter the command information. The file that is produced has a *.qaa* extension. The *.qaa* file extension is a Sybase Central convention; this file is the same as what you would create for the @data option. You can use the command file created by Sybase Central as your @data configuration file.

-c option

Function

Specify a connection string to the client message store.

Syntax

qaagent -c *connection-string* ...

Defaults

Connection parameter	Default value
uid	**ml_qa_user**
pwd	**qanywhere**

Remarks

The connection string must specify connection parameters in the form *keyword=value*, separated by semicolons, with no spaces between parameters.

DSNs are not typically used on client devices. ODBC is not used by qaagent.

☞ For a complete list of connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

Following are some of the connection parameters you may need to use:

- **dbf=*filename*** Connect to a message store with the specified filename.
- **dbn=*database-name*** If the client message store is already running when the QAnywhere Agent starts, you can connect to it by specifying a database name rather than a database file.
- **eng=*server-name*** If you want to use a database server that is already running, use this option to specify the server name. The default value is the name of the database.
- **uid=*user*** Specify a database user ID to connect to the client message store. This parameter is required if you change the defaults.
- **pwd=*password*** Specify the password for the database user ID. This is required if you change the defaults.
- **dbkey=*key*** If the client message store is encrypted using strong encryption, specify the encryption key required to access the database.

See also

- "Connection parameters" [*SQL Anywhere Server - Database Administration*]
- "Connecting to a Database" [*SQL Anywhere Server - Database Administration*]

Example

```
qaagent -id Device1 -c "DBF=qanyclient.db" -x tcpip(host=hostname) -policy
automatic
```


-fd option

Function

This option, when specified in conjunction with the -fr option, specifies the delay between attempts to connect to the MobiLink server.

Syntax

qaagent -fd *seconds* ...

Default

- If you specify -fr and do not specify -fd, the delay is 0 (no delay between retry attempts).
- If you do not specify -fr, the default is no retry attempts.

Remarks

You must use this option with the -fr option. The -fr option specifies how many times to retry the connection to the primary server, and the -fd option specifies the delay between retry attempts.

This option is typically used when you specify failover MobiLink servers with the -x option. By default when you set up a failover MobiLink server, the QAnywhere Agent tries an alternate server immediately upon a failure to reach the primary server. You can use the -fr option to cause the QAnywhere Agent to try the primary server again before going to the alternate server, and you can use the -fd option to specify the amount of time between retries of the primary server.

It is recommended that you set this option to 10 seconds or less.

See also

- "-fr option" on page 151
- "-x option" on page 173
- "Setting up a failover mechanism" on page 42

-fr option

Function

This option specifies the number of times that the QAnywhere Agent should retry the connection to the primary MobiLink server.

Syntax

qaagent -fr *number-of-retries* ...

Default

0 (the QAnywhere Agent will not attempt to retry the primary MobiLink server)

Remarks

By default, if the QAnywhere Agent is not able to connect to the MobiLink server, there is no error and messages are not sent. This option specifies that the QAnywhere Agent should retry the connection to the MobiLink server, and specifies the number of times that it should retry before trying an alternate server or issuing an error if you have not specified an alternate server.

This option is typically used when you specify failover MobiLink servers with the -x option. By default when you set up a failover MobiLink server, the QAnywhere Agent tries an alternate server immediately upon a failure to reach the primary server. This option causes the QAnywhere Agent to try the primary server again before going to the alternate server.

In addition, you can use the -fd option to specify the amount of time between retries of the primary server.

See also

- "-fd option" on page 150
- "-x option" on page 173
- "Setting up a failover mechanism" on page 42

-id option

Function

Specify the ID of the client message store that the QAnywhere Agent is to connect to.

Syntax

qaagent -id *id* ...

Default

The default value of the ID is the device name on which the Agent is running. In some cases, device names may not be unique, in which case you must use the -id option.

Remarks

Each client message store is represented by a unique sequence of characters called the message store ID. If you do not supply an ID when you first connect to the message store, the default is the device name. On subsequent connections, you must always specify the same message store ID with the -id option.

The message store ID corresponds to the MobiLink remote ID. It is required because in all MobiLink applications, each remote database must have a unique ID.

☞ For more information, see "Creating and registering MobiLink users" [*MobiLink - Client Administration*].

If you are starting a second instance of the qaagent on a device, the -id option must be used to specify a unique message store ID.

You cannot use the following characters in an ID:

- double quotes
- control characters
- double backslashes

The following additional constraints apply:

- You can use a single backslash only if it is used as an escape character.
- If your client message store database has the quoted_identifier database option set to Off (not the default), then your ID can only include alphanumeric characters and underscores, at signs, pounds, and dollar signs.

See also

- "About MobiLink users" [*MobiLink - Client Administration*]
- "Setting up the client message store" on page 33

-iu option

Function

Specifies the increment upload size.

Syntax

qaagent -iu *upload-size* [**K** | **M**] ...

Default

256K.

Remarks

This option specifies the size in bytes of the upload part of a message transmission. Use the suffix k or m to specify units of kilobytes or megabytes, respectively.

When the QAnywhere Agent starts, it assigns the value specified by this option to the ias_MaxUploadSize message store property. This message store property defines an upper bound on the size of an upload. When a transmission is triggered, the Agent tags messages for delivery to the server until the total size of all messages reaches the limit set with this option. When the limit is reached, these messages are sent to the server. As long as the messages arrive at the server and an acknowledgement is successfully sent from the server to the client, these messages are considered to be successfully delivered, even if the download phase of the transmission fails. The Agent continues sending batches of messages to the server until all queued messages have been delivered. Transmission rules are re-executed after each batch of messages is transmitted so that if a high priority messages gets queued during a transmission, it will jump to the front of the queue.

An upload will always contain at least one message (if there are messages queued for delivery) and messages will not be split. Therefore, the incremental upload size is an approximation, and it will be a poor approximation if there is a message to be uploaded that is many times larger than the incremental upload size.

See also

- ias_MaxUploadSize in "Pre-defined client message store properties" on page 215

-lp option

Function

Specifies the Listener port.

Syntax

qaagent -lp *number* ...

Default

5001.

Remarks

The port number on which the Listener listens for UDP notifications from the MobiLink server. Notifications are used to inform the QAnywhere Agent that a message is waiting.

A UDP listener port is only established if the Agent is started with the -push disconnected option.

See also

- "Scenario for messaging with push notifications" on page 9
- "-push option" on page 165

-mn option

Function

Specify a new password for the MobiLink user.

Syntax

qaagent -mp *password* ...

Default

None.

Remarks

Use to change the password.

See also

- "MobiLink Users" [*MobiLink - Client Administration*]
- "-mp option" on page 156
- "-mu option" on page 157

-mp option

Function

Specify the MobiLink password for the MobiLink user.

Syntax

qaagent -mp *password* ...

Default

None.

Remarks

If the MobiLink server requires user authentication, use -mp to supply the MobiLink password.

See also

- "MobiLink Users" [*MobiLink - Client Administration*]
- "-mu option" on page 157

-mu option

Function

Specify the MobiLink user.

Syntax

qaagent -mu *username* ...

Default

The client message store ID.

Remarks

The MobiLink user is used for authentication with the MobiLink server.

If you specify a user name that does not exist, it is created for you.

All MobiLink user names must be registered in the server message store. See "Registering client user names" on page 31.

See also

- "MobiLink Users" [*MobiLink - Client Administration*]
- "-id option" on page 152
- "-mp option" on page 156
- "Remote IDs" [*MobiLink - Client Administration*]

-o option

Function

Sends output to a log file.

Syntax

qaagent -o *logfile* ...

Default

None.

Remarks

The QAnywhere Agent logs output to the file name that you specify. If the file already exists, new log information is appended to the file. The SQL Anywhere synchronization client (dbmlsync) logs output to a file with the same name, but including the suffix *_sync*. The Listener utility (dblsn) logs output to a file with the same name, but including the suffix *_lsn*.

For example, if you specify the log file *c:\tmp\mylog.out*, then qaagent logs to *c:\tmp\mylog.out*, dbmlsync logs to *c:\tmp\mylog_sync.out*, and dblsn logs to *c:\tmp\mylog_lsn.out*.

See also

- "-ot option" on page 161
- "-on option" on page 159
- "-os option" on page 160
- "-v option" on page 172

-on option

Function

Specifies a maximum size for the QAnywhere Agent message log file, after which the file is renamed with the extension *.old* and a new file is started.

Syntax

qaagent –on *size* [**k** | **m**]…

Default

None.

Description

The *size* is the maximum file size for the output log, in bytes. Use the suffix k or m to specify units of kilobytes or megabytes, respectively. The minimum size limit is 10KB.

When the log file reaches the specified size, the QAnywhere Agent renames the output file with the extension *.old*, and starts a new one with the original name.

Notes

If the *.old* file already exists, it is overwritten. To avoid losing old log files, use the -os option instead. This option cannot be used with the -os option.

See also

- “-o option” on page 158
- “-ot option” on page 161
- “-os option” on page 160
- “-v option” on page 172

-os option

Function

Specifies a maximum size for the QAnywhere Agent message log file, after which a new log file with a new name is created and used.

Syntax

qaagent –os *size* [**k** | **m**] …

Default

None.

Description

The *size* is the maximum file size for logging output messages. The default units is bytes. Use the suffix k or m to specify units of kilobytes or megabytes, respectively. The minimum size limit is 10K.

Before the QAnywhere Agent logs output messages to a file, it checks the current file size. If the log message will make the file size exceed the specified size, the QAnywhere Agent renames the message log file to *yymmddxx.mls*. In this instance, *xx* are sequential characters ranging from 00 to 99, and *yymmdd* represents the current year, month, and day.

You can use this option to prune old message log files to free up disk space. The latest output is always appended to the file specified by -o or -ot.

> **Note**
> This option cannot be used with the -on option.

See also

- “-o option” on page 158
- “-ot option” on page 161
- “-on option” on page 159
- “-v option” on page 172

-ot option

Function

Truncates the log file and appends output messages to it.

Syntax

qaagent -ot *logfile* ...

Default

None.

Remarks

The QAnywhere Agent logs output to the file name that you specify. If the file exists, it is first truncated to a size of 0. The SQL Anywhere synchronization client (dbmlsync) logs output to a file with the same name, but including the suffix *_sync*. The Listener utility (dblsn) logs output to a file with the same name, but including the suffix *_lsn*.

For example, if you specify the log file *c:\tmp\mylog.out*, then qaagent logs to *c:\tmp\mylog.out*, dbmlsync logs to *c:\tmp\mylog_sync.out*, and dblsn logs to *c:\tmp\mylog_lsn.out*.

See also

- "-o option" on page 158
- "-on option" on page 159
- "-os option" on page 160
- "-v option" on page 172

-pc option

Function

Maintain a persistent connection to the MobiLink server between synchronizations.

Syntax

qaagent -pc { **+** | **-** } ...

Default

-pc-

Remarks

Enabling persistent connections (-pc+) is useful when network coverage is good and there is heavy message traffic over QAnywhere. In this scenario, you can reduce the network overhead of setting up and taking down a TCP/IP connection every time a message transmission occurs.

Disabling persistent connections (-pc-) is useful in the following scenarios when the client device has a public IP address and is reachable by UDP or SMS:

- The client device is using dial-up networking and connection time charges are an issue.
- There is light message traffic over QAnywhere. Persistent TCP/IP connections consume network server resources, and so could have an impact on scalability.
- The client device network coverage is unreliable. You can use the automatic policy to transmit messages when connection is possible. Trying to maintain persistent connections in this environment is not useful and can waste CPU resources.

See also

- "-push option" on page 165
- "-pc option" [*MobiLink - Client Administration*]

-policy option

Function

Specifies a policy that determines when message transmission occurs.

Syntax

qaagent -policy *policy-type* ...

policy-type: **ondemand** | **scheduled**[*interval-in-seconds*] | **automatic** | *rules-file*

Defaults

- The default policy type is **automatic**.
- The default interval for scheduled policies is 900 seconds (15 minutes).

Remarks

QAnywhere uses a policy to determine when message transmission occurs. The *policy-type* can be one of the following values:

- **ondemand** Only transmit messages when the QAnywhere client application makes the appropriate method call.

  The QAManager PutMessage() method causes messages to be queued locally. These messages are not transmitted to the server until the QAManager TriggerSendReceive() method is called. Similarly, messages waiting on the server are not sent to the client until TriggerSendReceive() is called by the client.

  When using the ondemand policy, the application is responsible for causing a message transmission to occur when it receives a push notification from the server. A push notification causes a system message to be delivered to the QAnywhere client. In your application, you may choose to respond to this system message by calling TriggerSendReceive().

  ☞ For an example, see "System queue" on page 51.

- **scheduled** Transmit messages at a specified interval. The default value is 900 seconds (15 minutes).

  Transmission of messages between the client and the server takes place at a specified time interval.

  The QAManager PutMessage() method causes messages to be queued locally. These messages are not transmitted until the time interval has elapsed. Messages queued on the server for delivery to the client are also transmitted when the time interval has elapsed.

  If push notifications are enabled, messages queued on the server for delivery to the client are transmitted when the next time interval elapses.

  TriggerSendReceive() can override the time interval. It forces a message transmission to occur before the time interval elapses.

  The optional *interval* argument is the number of seconds between send/receive operations. For example, the following command schedules the QAnywhere Agent to send/receive messages every 20 minutes:

  ```
  qaagent.exe -policy scheduled[1200]
  ```


- **automatic** Transmit messages when one of the events described below occurs.

  The QAnywhere agent attempts to keep message queues as current as possible. Any of the following events cause messages queued on the client to be delivered to the server and messages queued on the server to be delivered to the client:

  - Invoking PutMessage().
  - Invoking TriggerSendReceive().
  - A push notification.

    For information about notifications, see "Scenario for messaging with push notifications" on page 9.
  - A message status change on the client. For example, a status change occurs when an application retrieves a message from a local queue which causes the message status to change from pending to received.

- ***rules-file*** Specifies a client transmission rules file. The transmission rules file can indicate a more complicated set of rules to determine when messages are transmitted.

  ☞ See "Client transmission rules" on page 234.

See also

- "Determining when message transmission should occur on the client" on page 36
- "Scenario for messaging with push notifications" on page 9

-push option

Function

Specifies whether push notifications are enabled.

Syntax

qaagent -push *mode* ...

mode : **none** | **connected** | **disconnected**

Default

connected

Options

Mode	Description
none	Push notifications are disabled for this agent. The Listener (dblsn) is not started.
connected	Push notifications are enabled for this agent over TCP/IP with persistent connection. The Listener (dblsn) is started by qaagent and attempts to maintain a persistent connection to the MobiLink server. This mode is useful when the client device does not have a public IP address or when the MobiLink server is behind a firewall that does not allow UDP messages out. This is the default.
disconnected	Push notifications are enabled for this agent over UDP without a persistent connection. The Listener (dblsn) is started by qaagent but does not maintain a persistent connection to the MobiLink server. Instead, a UDP listener receives push notifications from MobiLink. This mode is useful in the following scenarios when the client device has a public IP address and is reachable by UDP or SMS: ♦ The client device is using dial-up networking and connection time charges are an issue. ♦ There is light message traffic over QAnywhere. Persistent TCP/IP connections consume network server resources, and so could have an impact on scalability. ♦ The client device network coverage is unreliable. You can use the automatic policy to transmit messages when connection is possible. Trying to maintain persistent connections in this environment is not useful and can waste CPU resources. See "-lp option" on page 154.

Remarks

If you do not want to use notifications, set this option to none. You then do not have to deploy the *dblsn.exe* executable with your clients.

☞ For a description of QAnywhere without notifications, see "Simple messaging scenario" on page 7.

If you are using UDP, you cannot use push notifications in disconnected mode with ActiveSync due to the limitations of the UDP implementation of ActiveSync.

See also

- “Using push notifications” on page 41
- “-pc option” on page 162
- “Running the QAnywhere Agent” on page 35
- “Notifications of push notification” on page 53

-q option

Function

Starts the QAnywhere Agent in quiet mode with the window minimized in the system tray.

Syntax

qaagent -q ...

Default

None.

Remarks

When you start the QAnywhere Agent in quiet mode with -q, the main window is minimized to the system tray. In addition, the database server for the message store is started with the -qi option.

See also

- "-qi option" on page 168

-qi option

Starts the QAnywhere Agent in quiet mode with the window completely hidden.

Syntax

qaagent -qi ...

Default

None.

Remarks

When you start the QAnywhere Agent in quiet mode, on Windows desktop the main window is minimized to the system tray, and on Windows CE the main window is hidden. In addition, the database server for the message store is started with the -qi option.

Quiet mode is useful for some Windows CE applications because it prevents an application from being closed when Windows CE reaches its limit of 32 concurrent processes. Quiet mode allows the QAnywhere Agent to run like a service.

When in -qi quiet mode, you can only stop the QAnywhere Agent by typing **qastop**.

See also

- "-q option" on page 167

-si option

Function

Initializes the database for use as a client message store.

Syntax

qaagent -c "*connection-string*" -si ...

Default

None. You only use this option once, to initialize the client message store.

Remarks

Before using this option, you must create a SQL Anywhere database. When you use -si, the QAnywhere Agent initializes the database with database objects such as QAnywhere system tables; it then exits immediately.

When you run -si, you must specify a connection string with the -c option that indicates which database to initialize. The connection string specified in the -c option should also specify a user ID with DBA privileges. If you do not specify a user ID and password, the default user DBA with password SQL is used.

The -si option creates a database user named **ml_qa_user** and password **qanywhere** for the client message store. The user called ml_qa_user has permissions suitable for QAnywhere applications only. If you do not change this database user name and password, then you do not need to specify the pwd or uid in the -c option when you start qaagent. If you change either of them, then you must supply the uid and/or pwd in the -c option on the qaagent command line.

> **Note**
> You should change the default passwords. To change them, use the GRANT statement. For more information, see "Changing a password" [*SQL Anywhere Server - Database Administration*].

The -si option does not provide an ID for the client message store. You can assign an ID using the -id option when you run -si or the next time you run qaagent; or, if you do not do that, qaagent will by default assign the device name as the ID.

When a message store is created but is not set up with an ID, QAnywhere applications local to the message store can send and receive messages, but cannot exchange messages with remote QAnywhere applications. Once an ID is assigned, remote messaging may also occur.

See also

- "Setting up the client message store" on page 33
- "Creating a secure client message store" on page 176

Examples

The following command connects to a database called *qaclient.db* and initializes it as a QAnywhere client message store. The QAnywhere Agent immediately exits when the initialization is complete.

```
qaagent -si -c "DBF=qaclient.db"
```


-su option

Function

Upgrades a client message store to the current version. If you are upgrading from a pre-10.0.0 message store, you must first manually unload and reload the message store.

Syntax

qaagent -su -c "*connection-string*" ...

Remarks

This option is useful if you want to perform custom actions after the unload/reload and before the qaagent upgrade. Use the -sur option if you are upgrading from a pre-10.0.0 message store and you want the Agent to automatically perform the unload/reload step for you.

This operation exits when the upgrade is complete.

This operation cannot be undone.

See also

- "-sur option" on page 171

Example

To upgrade from a version 9 database, first, unload and reload the database:

```
dbunload  -q  -c "UID=dba;PWD=sql;DBF=qanywhere.db" -ar
```

Next, run qaagent with the -su option:

```
qaagent -q -su -c "UID=dba;PWD=sql;DBF=qanywhere.db"
```

-sur option

Function

Upgrades a client message store to the current version.

Syntax

qaagent -sur -c "*connection-string*" ...

Remarks

Specify the database to upgrade in the connection string. The -sur option automatically unloads the message store, reloads it, and upgrades it.

The unload/reload is necessary to upgrade from a version 9 message store to a version 10 message store. The unload/reload can be done manually along with the -su option. For example, if you need to perform custom actions after the reload and before the upgrade, use the -su option.

This operation exits when the upgrade is complete.

This operation cannot be undone.

See also

- “-su option” on page 170

Example

The following example unloads and reloads a version 9.0.2 SQL Anywhere database called qanywhere.db, making it useful with QAnywhere version 10.0.0.

```
qaagent -q -sur -c "UID=dba;PWD=sql;DBF=qanywhere.db"
```


-v option

Function

Allows you to specify what information is logged to the message log file and displayed in the QAnywhere Agent console. A high level of verbosity may affect performance and should normally be used in the development phase only.

Syntax

qaagent -v *levels* ...

Default

Minimal verbosity.

Remarks

The -v option affects the log files and console. You only have a message log if you specify -o or -ot on the qaagent command line.

If you specify –v alone, a small amount of information is logged.

The values of *levels* are as follows. You can use one or more of these options at once; for example, -vlm.

- **+** Turn on all logging options.
- **l** Show all MobiLink Listener logging. This causes the MobiLink Listener (dblsn) to start with verbosity level -v3.

  ☞ For more information, see the -v option in the "Listener syntax" [*MobiLink - Server-Initiated Synchronization*].
- **m** Show all dbmlsync logging. This causes the SQL Anywhere synchronization client (dbmlsync) to start with verbosity level -v+.

  ☞ For more information, see the dbmlsync "-v option" [*MobiLink - Client Administration*].
- **n** Show all network status change notifications. the QAnywhere Agent receives these notifications from the Listener utility.
- **p** Show all message push notifications. The QAnywhere Agent receives these notifications from the Listener utility via the MobiLink server, which includes a MobiLink Notifier.
- **q** Show the SQL that is used to represent the transmission rules.
- **s** Show all the message synchronizations that are initialized by QAnywhere Agent.

See also

- "-o option" on page 158
- "-ot option" on page 161
- "-on option" on page 159
- "-os option" on page 160

-x option

Function

Specify the network protocol and the protocol options for communication with the MobiLink server.

Syntax

qaagent -x *protocol* [**(** *protocol-options*;... **)** ...

protocol: **http**, **tcpip**, **https**, **tls**

protocol-options: *keyword*=*value*

Remarks

For a complete list of *protocol-options*, see "MobiLink Client Network Protocol Options" [*MobiLink - Client Administration*].

The -x option is required when the MobiLink server is not on the same device as the QAnywhere Agent.

You can specify -x multiple times. This allows you to set up failover to multiple MobiLink servers. When you set up failover, the QAnywhere Agent attempts to connect to the MobiLink servers in the order in which you enter them on the command line.

The QAnywhere Agent also has a Listener that receives notifications from the MobiLink server that messages are available at the server for transmission to the client. This Listener only uses the first MobiLink server that is specified, and does not fail over to others.

See also

- "MobiLink Client Network Protocol Options" [*MobiLink - Client Administration*]
- "Encrypting the communication stream" on page 178
- "Transport-Layer Security" [*SQL Anywhere Server - Database Administration*]
- "Setting up a failover mechanism" on page 42
- "-fd option" on page 150
- "-fr option" on page 151

CHAPTER 8

Writing Secure Messaging Applications

Contents

About this chapter

This chapter describes techniques for implementing a secure messaging solution.

Creating a secure client message store

To secure your client message store, you can:

- Change the default passwords.

  See "Manage client message store passwords" on page 176.

- Encrypt the contents of the message store.

  See "Encrypting the client message store" on page 177.

Example

First, create a SQL Anywhere database with an encryption key:

```
dbinit mystore.db -n -i -s -ek some_phrase
```

The -i and -s options are optimal for small devices. The -ek option specifies the encryption key for strong encryption. The -n option initializes the database without a transaction log, because a transaction log is not required for or managed by QAnywhere. See "Initialization utility (dbinit)" [*SQL Anywhere Server - Database Administration*].

Next, initialize the database as a client message store:

```
qaagent -id mystore -si -c "dbf=mystore.db;dbkey=some_phrase"
```

Next, create a new remote user with DBA authority, and a password for this user. Revoke the default QAnywhere user and change the password of the default DBA user. Log in as user DBA with password SQL and execute the following SQL statements:

```
GRANT CONNECT TO secure_user IDENTIFIED BY secure_password
GRANT MEMBERSHIP IN GROUP ml_qa_user_group TO secure_user
GRANT REMOTE dba TO secure_user
REVOKE CONNECT FROM ml_qa_user
GRANT CONNECT TO dba IDENTIFIED BY new_dba_password
COMMIT
```

Note
All QAnywhere users must belong to ml_qa_user_group and have remote DBA authority.

Next, start the QAnywhere Agent with the secure DBA user:

```
    qaagent -id mystore -c
"dbf=mystore.db;dbkey=some_phrase;uid=secure_user;pwd=secure_password"
```

Manage client message store passwords

You should change the passwords for the default user IDs that were created for the message store. The default user ID DBA with password SQL is created for every SQL Anywhere database. In addition, the qaagent -si

option creates a default user ID of ml_qa_user, and creates a default password of qanywhere. To change these passwords, use the GRANT statement.

☞ For more information, see "Changing a password" [*SQL Anywhere Server - Database Administration*].

Encrypting the client message store

The following command can be used to encrypt the client message store when you create it.

```
dbinit -n -i -s –ek encryption-key database-file
```

(The -i and -s options are good practice for creating databases on small devices.) When a message store has been initialized with an encryption key, the encryption key is required to start the database server on the encrypted message store.

Use the following command to specify the encryption key to start the QAnywhere Agent with an encrypted message store. The QAnywhere Agent automatically starts the database server on the encrypted message store using the encryption key provided.

```
qaagent –c "DBF=database-file;DBKEY=encryption-key"
```

Any application can now access the encrypted message store through the QAnywhere APIs. Note that, since the database server used to manage the message store is already running, the application does not need to provide the encryption key.

If the QAnywhere Agent is not running and an application needs to access an encrypted message store, the QAnywhere APIs automatically starts the database server using the connection parameters specified in the QAnywhere Manager initialization file. In order to start the database server on an encrypted message store, the encryption key must be specified in the database connection parameters as follows.

```
CONNECT_PARAMS=DBF=database-file;DBKEY=encryption-key
```

See also

- "Encrypting a database" [*SQL Anywhere Server - Database Administration*]
- "Initialization utility (dbinit)" [*SQL Anywhere Server - Database Administration*]
- QAnywhere Agent "-c option" on page 148

Encrypting the communication stream

The qaagent -x option can be used to specify a secure communication stream that the QAnywhere Agent can use to communicate with a MobiLink server. It allows you to implement server authentication using server-side certificates, and it allows you to encrypt the communication stream using strong encryption.

☞ For more information, see "-x option" on page 173.

You must set up transport-layer security for the MobiLink server as well. For information about creating digital certificates and setting up the MobiLink server, see "Encrypting MobiLink client/server communications" [*SQL Anywhere Server - Database Administration*].

Separately licensed component required
ECC encryption and FIPS-approved encryption require a separate license. All strong encryption technologies are subject to export regulations.
See "Separately licensed components" [*SQL Anywhere 10 - Introduction*].

Examples

The following examples show how to establish a secure communication stream between the QAnywhere Agent and the MobiLink server. They use sample certificates that are installed when the SQL Anywhere security option is installed.

Secure TCP/IP using RSA:

```
mlsrv10 -x tls
(tls_type=rsa;certificate=rsaserver.crt;certificate_password=test)
qaagent –x tls(tls_type=rsa;trusted_certificates=rsaroot.crt)
```

Secure TCP/IP using ECC:

```
mlsrv10 -x tls
(tls_type=ecc;certificate=sample.crt;certificate_password=tJ1#m6+W)
qaagent –x tls(tls_type=ecc;trusted_certificates=eccroot.crt)
```

Secure HTTP using HTTPS (only RSA certificates are supported for HTTPS):

```
mlsrv10 -x https(certificate=rsaserver.crt;certificate_password=test)
qaagent –x https(trusted_certificates=rsaroot.crt)
```


Using password authentication with MobiLink

Once you have established a secure communication stream between the remote device and the server, you may also want to authenticate the user of the device to ensure that they are allowed to communicate with the server.

You do this by creating a MobiLink user name for the client message store and registering it on the server message store.

See also

- "-mu option" on page 157
- "-mp option" on page 156
- "MobiLink Users" [*MobiLink - Client Administration*]

CHAPTER 9

Mobile Web Services

Contents

About this chapter

QAnywhere provides functionality that greatly simplifies the development of mobile web service applications. This functionality is provided in the Java and .NET APIs.

This chapter describes how to write QAnywhere client applications that access a web service.

Introducing mobile web services

Web Services have become a popular way to expose application functionality and enable better interoperability between the resources of various enterprises. They broaden the capabilities of mobile applications and simplify the development process.

Implementing web services in a mobile environment can be challenging because connectivity may not be available (or may be interrupted) and because of other limitations of wireless environments and devices. For example, a user working with a mobile application may want to make a request to a web service while offline and obtain the response when they go online, or an IT administrator may want to specify rules that restrict the size of web service responses based on the type of network connectivity the mobile application is using (such as GPRS, 802.11, or cradled).

QAnywhere addresses these challenges with mobile-optimized asynchronous web services that leverage the QAnywhere store-and-forward messaging architecture. By using QAnywhere mobile web services, your mobile applications can make web service requests, even when they are offline, and have those requests queued up for transmission later. The requests are delivered as QAnywhere messages and then a web services connector on the server side makes the request, gets the response from the web service, and returns the response to the client as a message. QAnywhere transmission rules can control which requests and responses are transmitted based on a wide variety of parameters (network being used, size of request/response, location, time of day, and so on). The result is a sophisticated and flexible architecture that allows mobile applications to tap into the vast functionality of web services using proven technology and a simple programming model.

From a development point of view, you can work with web service proxy classes much as you would in a connected environment and QAnywhere handles all of the transmission, authentication, serialization, and so on. A WSDL compiler is provided to take a WSDL document and generate special proxy classes (either .NET or Java) that a mobile application can use to invoke a web service. These classes use the underlying QAnywhere infrastructure to send requests and receive responses. When an object method call is made, a SOAP request is built automatically and delivered as a message to the server where a connector makes the web service request and returns the result as a message.

See also

- "Mobile web services" [*SQL Anywhere 10 - Introduction*]

Setting up mobile web services

The following steps provide an overview of the tasks required to set up mobile web services.

♦ Overview of setting up mobile web services

1. Set up a server message store, if you don't already have one.

   ☞ See "Setting up the server message store" on page 30.

2. Start the MobiLink server with the -m option and a connection to the server message store.

   ☞ See "Starting the MobiLink server for QAnywhere messaging" on page 31.

3. Set up client message stores, if you don't already have them. These are SQL Anywhere databases that are used to temporarily store messages.

   ☞ See "Setting up the client message store" on page 33.

4. Run the QAnywhere WSDL compiler to create classes you can use in your application.

   ☞ See "Running the QAnywhere WSDL compiler" on page 184.

5. For each client, write a web service client application that uses the classes generated by the WSDL compiler.

   ☞ See "Writing mobile web service applications" on page 185.

6. Create a web services connector.

   ☞ See "Setting up web service connectors" on page 195.

7. For each client, start the QAnywhere Agent (qaagent) with a connection to the local client message store.

   ☞ See "Running the QAnywhere Agent" on page 35.

Other resources for getting started

- A simple example using a hypothetical web service is described in "Mobile web service example" on page 198.
- A full-featured mobile web service sample application is installed to *samples-dir**QAnywhere**MobileWebServices*. (For information about *samples-dir*, see "The samples directory" [*SQL Anywhere Server - Database Administration*].) This sample, which is provided in both Java and C#, demonstrates how to use mobile web services to make asynchronous web service requests.

Running the QAnywhere WSDL compiler

Given a WSDL file that describes a web service, the QAnywhere WSDL compiler generates a set of Java or C# proxy classes that you include in your application. These classes expose web service operations as method calls. The classes that are generated are:

- The main service binding class (this class inherits from WSBase in the mobile web services runtime).
- A proxy class for each complex type specified in the WSDL file.

For information about the generated proxy classes, see:

- .NET: "iAnywhere.QAnywhere.WS namespace (.NET)" on page 342
- Java: "ianywhere.qanywhere.ws package" on page 602

The WSDL compiler supports WSDL 1.1 and SOAP 1.1 over HTTP and HTTPS.

Syntax

wsdlc -l *programming-language wsdl-file* [*options*]

Parameters

programming-language: **cs** | **java**

wsdl-file: the name of the WSDL file that describes a web service

Options	Description
-h	Print a help screen.
-v	Print verbose information.
-o *output-directory*	Specify an output directory for generated files.
-d	Print debug information.
-n	For C# output only, specify a namespace.
-p	For Java output only, specify a package name.

Writing mobile web service applications

Your application sends a web service request to QAnywhere, which sends the request to the mobile web service connector in the MobiLink server. The connector sends the request to the web service or queues the request until the web service is available. When QAnywhere receives the response, it notifies your application or queues the response until your application is available.

Setting up .NET mobile web service applications

Before using .NET with QAnywhere, you must make the following changes to your Visual Studio .NET project:

- Add references to the QAnywhere .NET DLL and the mobile web services .NET DLL. This tells Visual Studio.NET which DLL to include to find the code for the QAnywhere .NET API and the mobile web services .NET API.
- Add lines to your source code to reference the QAnywhere .NET API classes and the mobile web services .NET API classes. In order to use the QAnywhere .NET API, you must add a line to your source code to reference the data provider. You must add a different line for C# than for Visual Basic.NET.

Complete instructions follow.

♦ To add references to the QAnywhere .NET API and mobile web services API in a Visual Studio .NET project

1. Start Visual Studio .NET and open your project.
2. In the Solution Explorer window, right-click the References folder and choose Add Reference from the popup menu.

   The Add Reference dialog appears.
3. On the .NET tab, click Browse to locate *iAnywhere.QAnywhere.Client.dll* and *iAnywhere.QAnywhere.WS.dll*. The location of these files is (relative to your SQL Anywhere installation directory):

   - .NET Framework 1.1: *\Assembly\v1*
   - .NET Framework 2.0: *\Assembly\v2*
   - .NET Compact Framework 1.0: *ce\Assembly\v1*
   - .NET Compact Framework 2.0: *ce\Assembly\v2*

   From the appropriate directory for your environment, select each DLL and click Open.
4. To verify that the DLLs are added to your project, open the Add Reference dialog and open the .NET tab. *iAnywhere.QAnywhere.Client.dll* and *iAnywhere.QAnywhere.WS.dll* appear in the Selected Components list. Click OK.

Referencing the data provider classes in your source code

♦ To reference the QAnywhere .NET API and mobile web services API classes in your code

1. Start Visual Studio .NET and open your project.
2. If you are using C#, add the following lines to the list of using directives at the beginning of your file:

```
using iAnywhere.QAnywhere.Client;
using iAnywhere.QAnywhere.WS;
```

3. If you are using Visual Basic .NET, add the following lines to the list of imports at the beginning of your file:

```
Imports iAnywhere.QAnywhere.Client
Imports iAnywhere.QAnywhere.WS
```

   The Imports lines are not strictly required. However, they allow you to use short forms for the QAnywhere and mobile web services classes. Without them, you can still use the fully qualified class name in your code. For example, the following code uses the long form:

```
iAnywhere.QAnywhere.Client.QAManager
mgr =
 new iAnywhere.QAnywhere.Client.QAManagerFactory.Instance.CreateQAManager
(
"qa_manager.props" );
```

   The following code uses the short forms:

```
QAManager mgr = QAManagerFactory.Instance.CreateQAManager(
 "qa_manager.props" );
```

♦ To initialize QAnywhere and mobile web services for .NET

1. Include the iAnywhere.QAnywhere.Client and iAnywhere.QAnywhere.WS namespaces, as described in the previous procedure.

```
using iAnywhere.QAnywhere.Client;
using iAnywhere.QAnywhere.WS;
```

2. Create a QAManager object.

   For example, to create a default QAManager object, invoke CreateQAManager with null as its parameter:

```
QAManager mgr;
mgr = QAManagerFactory.Instance.CreateQAManager( null );
```

> **Tip**
> For maximum concurrency benefits, multi-threaded applications should create a QAManager for each thread. See "Multi-threaded QAManager" on page 61.

☞ For more information about QAManagerFactory, see "QAManagerFactory class" on page 310.

Alternatively, you can create a QAManager object that is customized using a properties file. The properties file is specified in the CreateQAManager method:


```
mgr = QAManagerFactory.Instance.CreateQAManager(
  "qa_mgr.props" );
```

where *qa_mgr.props* is the name of the properties file that resides on the remote device.

3. Initialize the QAManager object. For example:

```
mgr.Open(
   AcknowledgementMode.EXPLICIT_ACKNOWLEDGEMENT);
```

The argument to the open method is an acknowledgement mode, which indicates how messages are to be acknowledged. It must be one of IMPLICIT_ACKNOWLEDGEMENT or EXPLICIT_ACKNOWLEDGEMENT.

QAnywhere messages used by mobile web services are not accessible to the mobile web services application. When using a QAManager in EXPLICIT_ACKNOWLEDGEMENT mode, use the Acknowledge method of WSResult to acknowledge the QAnywhere message that contains the result of a web services request. This method indicates that the application has successfully processed the response.

For more information about acknowledgement modes, see:

- WSBase "SetQAManager method" on page 346
- WSResult "Acknowledge method" on page 358

> Instead of creating a QAManager, you can create a QATransactionalManager. See "Implementing transactional messaging for .NET clients" on page 68.

4. Create an instance of the service binding class.

The mobile web services WSDL compiler generates the service binding class from the WSDL document that defines the web service.

The QAManager is used by the instance of the web service binding class to perform messaging operations in the process of making web service requests. You specify the connector address to use to send web service requests through QAnywhere by setting the property WS_CONNECTOR_ADDRESS of the service binding class. You configure each QAnywhere web service connector with the URL of a web service to connect to, and if an application needs web services located at more than one URL, configure the connector for each URL.

For example:

```
CurrencyConverterSoap service = new CurrencyConverterSoap( )
service.SetQAManager(mgr);
service.setProperty(
   "WS_CONNECTOR_ADDRESS",
   "ianywhere.connector.currencyconvertor\\");
```

Note that the final \\ in the address must be included.

See also

- "iAnywhere.QAnywhere.WS namespace (.NET)" on page 342

- "iAnywhere.QAnywhere.Client namespace (.NET)" on page 244

Example

To initialize mobile web services, you must create a QAManager and create an instance of the service binding class. For example:

```
// QAnywhere initialization
        QAManager mgr = QAManagerFactory.Instance.CreateQAManager( null );
        mgr.SetProperty( "CONNECT_PARAMS",
"eng=qanywhere;dbf=qanywhere.db;uid=ml_qa_user;pwd=qanywhere" );
        mgr.Open( AcknowledgementMode.IMPLICIT_ACKNOWLEDGEMENT );
        mgr.Start();

        // Instantiate the web service proxy
        CurrencyConvertorSoap service = new CurrencyConvertorSoap();
        service.SetQAManager( mgr );
        service.SetProperty( "WS_CONNECTOR_ADDRESS",
"ianywhere.connector.currencyconvertor\\" );
```

Setting up Java mobile web service applications

To create mobile web service applications in Java, you must complete the following initialization tasks.

- **To initialize QAnywhere and mobile web services for Java**

1. Add the location of the following files to your classpath. By default, they are located in *install-dir* *java*:

   - *qaclient.jar*
   - *iawsrt.jar*
   - *jaxrpc.jar*

2. Import the ianywhere.qanywhere.client and ianywhere.qanywhere.ws packages:

   ```
   import ianywhere.qanywhere.client.*;
   import ianywhere.qanywhere.ws.*;
   ```

3. Create a QAManager object.

   ```
   QAManager mgr;
   mgr = QAManagerFactory.getInstance().createQAManager(null);
   ```

   You can also customize a QAManager object by specifying a properties file to the createQAManager method:

   ```
   mgr = QAManagerFactory.getInstance().createQAManager("qa_mgr.props.");
   ```

   Tip
   For maximum concurrency benefits, multi-threaded applications should create a QAManager for each thread. See "Multi-threaded QAManager" on page 61.

4. Initialize the QAManager object.

   ```
   mgr.open(AcknowledgementMode.EXPLICIT_ACKNOWLEDGEMENT);
   ```


The argument to the open method is an acknowledgement mode, which indicates how messages are to be acknowledged. It must be one of IMPLICIT_ACKNOWLEDGEMENT or EXPLICIT_ACKNOWLEDGEMENT.

QAnywhere messages used by mobile web services are not accessible to the mobile web services application. When using a QAManager in EXPLICIT_ACKNOWLEDGEMENT mode, use the Acknowledge method of WSResult to acknowledge the QAnywhere message that contains the result of a web services request. This method indicates that the application has successfully processed the response.

For more information about acknowledgement modes, see:

- WSBase "setQAManager method" on page 605
- WSResult "acknowledge method" on page 611

> Instead of creating a QAManager, you can create a QATransactionalManager. See "Implementing transactional messaging for Java clients" on page 71.

5. Create an instance of the service binding class.

   The mobile web services WSDL compiler generates the service binding class from the WSDL document that defines the web service.

   In the process of making web service requests, the QAManager is used by the instance of the web service binding class to perform messaging operations. You specify the connector address to use to send web service requests through QAnywhere by setting the WS_CONNECTOR_ADDRESS property of the service binding class. Each QAnywhere web service connector is configured with a URL of a web service to connect to. This means that if an application needs web services located at more than one URL, then a QAnywhere connector must be configured for each service URL.

   For example:

   ```
   CurrencyConverterSoap service = new CurrencyConverterSoap( );
   service.setQAManager(mgr);
   service.setProperty("WS_CONNECTOR_ADDRESS",
   "ianywhere.connector.currencyconvertor\\");
   ```

   Note that the final \\ in the address must be included.

See also

- "ianywhere.qanywhere.ws package" on page 602
- "ianywhere.qanywhere.client package" on page 504

Example

To initialize mobile web services, you must create a QAManager and create an instance of the service binding class. For example:

```
// QAnywhere initialization
     Properties props = new Properties();
     props.put( "CONNECT_PARAMS",
"eng=qanywhere;dbf=qanywhere.db;uid=ml_qa_user;pwd=qanywhere" );
```



```
        QAManager mgr = QAManagerFactory.getInstance().createQAManager( props );
        mgr.open( AcknowledgementMode.IMPLICIT_ACKNOWLEDGEMENT );
        mgr.start();

        // Instantiate the web service proxy
        CurrencyConvertorSoap service = new CurrencyConvertorSoap();
        service.setQAManager( mgr );
        service.setProperty( "WS_CONNECTOR_ADDRESS",
    "ianywhere.connector.currencyconvertor\\" );
```

Multiple instances of the service binding class

You should create an instance of the service binding class for each QAManager. If a mobile web services application has more than one instance of a service binding class, it is important that the service ID be set using the SetServiceID method. For example:

```
service1.SetServiceID("1")
service2.SetServiceID("2")
```

The service ID is combined with the service name to form a queue name for receiving web service responses. It is important that each instance of a given service has a unique service ID so that a given instance does not get responses to requests made by another instance of the service. If the service ID is not set, it defaults to "". The service ID is also important for preventing multiple applications that use the same service from conflicting with each other, since queue names persist messages in the message store across applications that are transient.

Compiling and running mobile web service applications

Runtime libraries

The runtime library for Java is *iawsrt.jar*, located in the *java* subdirectory of your SQL Anywhere installation.

The runtime library for C# is *iAnywhere.QAnywhere.WS.dll*, located in the following directories (relative to your SQL Anywhere directory):

- .NET Framework 1.1: *\Assembly\v1*
- .NET Framework 2.0: *\Assembly\v2*
- .NET Compact Framework 1.0: *ce\Assembly\v1*
- .NET Compact Framework 2.0: *ce\Assembly\v2*

The following sections describe the files you need to compile and run mobile web service applications.

Required runtime libraries (Java)

Include the following files, located in the *java* subdirectory of your SQL Anywhere 10 installation, in your classpath:

- *jaxrpc.jar*
- *qaclient.jar*
- *iawsrt.jar*

Required runtime libraries (.NET)

The SQL Anywhere 10 installation automatically includes the following files in your Global Assembly Cache:

- *iAnywhere.QAnywhere.Client.dll*
- *iAnywhere.QAnywhere.WS.dll*

Shutting down mobile web services

A mobile web services application performs orderly shutdown by closing the QAManager. For example:

```
// QAnywhere finalization in C#:
mgr.Stop();
mgr.Close();

// QAnywhere finalization in Java:
mgr.stop();
mgr.close();
```


Making web service requests

There are two basic methods of making web service requests in a mobile web services application:

- **Synchronous** See "Synchronous web service requests" on page 192.
- **Asynchronous** See "Asynchronous web service requests" on page 192.

Synchronous web service requests

Synchronous web service requests are used when the application is connected to a network. With this method, a web service request is made by calling a method on the service binding class, and the result is returned only when the web service response has been received from the server.

Example

The following example makes a request to get the USD-to-CAD exchange rate:

```
//C#
double r = service.ConversionRate( Currency.USD, Currency.CAD );

// Java
double r = service.conversionRate( NET.webserviceX.Currency.USD,
NET.webserviceX.Currency.CAD );
```

Asynchronous web service requests

Asynchronous web service requests are useful when the mobile web service application is only occasionally connected to a network. With this method, a web service request is made by calling a method on the service binding class to place the request in an outgoing queue. The method returns a WSResult, which can be used to query the status of the response at a later time, even after the application has been restarted.

The following example makes an asynchronous request to get the USD-to-CAD exchange rate:

```
// C#
WSResult r = service.AsyncConversionRate( Currency.USD, Currency.CAD );

// Get the request ID.  Save it for later use if necessary.
string reqID = r.GetRequestID();

// Later: get the response for the specified request ID
WSResult r = service.GetResult( reqID );
if( r.GetStatus() == WSStatus.STATUS_RESULT_AVAILABLE ) {
  Console.WriteLine( "The conversion rate is " + r.GetDoubleValue
( "ConversionRateResult" ) );
} else {
  Console.WriteLine( "Response not available" );
}

// Java
WSResult r = service.asyncConversionRate( NET.webserviceX.Currency.USD,
NET.webserviceX.Currency.CAD );

// Get the request ID.  Save it for later use if necessary.
```



```
String reqID = r.getRequestID();

// Later: get the response for the specified request ID
WSResult r = service.getResult( reqID );
if( r.getStatus() == WSStatus.STATUS_RESULT_AVAILABLE ) {
    System.out.println( "The conversion rate is " + r.getDoubleValue
( "ConversionRateResult" ) );
} else {
    System.out.println( "Response not available" );
}
```

It is also possible to use a WSListener to get an asynchronous callback when the response to a web service request is available. For example:

```
// C#
// Make a request to get the USD to CAD exchange rate
WSResult r = service.AsyncConversionRate( Currency.USD, Currency.CAD );

// Register a listener for the result
service.SetListener( r.GetRequestID(), new CurrencyConvertorListener() );

// Java
// Make a request to get the USD to CAD exchange rate
WSResult r = service.asyncConversionRate( NET.webserviceX.Currency.USD,
NET.webserviceX.Currency.CAD );

// Register a listener for the result
service.setListener( r.getRequestID(), new CurrencyConvertorListener() );
```

The WSListener interface defines two methods for handling asynchronous events:

- **OnResult** An OnResult method is implemented to handle a response to a web service request. It is passed a WSResult object that represents the result of the web service request.
- **OnException** An OnException method is implemented to handle errors that occurred during processing of the response to the web service request. It is passed a WSException object and a WSResult object. The WSException object contains information about the error that occurred, and the WSResult object can be used to obtain the request ID that the response corresponds to.

```
// C#
class CurrencyConvertorListener : WSListener
{
   public CurrencyConvertorListener() {
   }

   public void OnResult( WSResult r ) {
    try {
         USDToCAD._statusMessage = "USD to CAD currency exchange rate: " +
r.GetDoubleValue( "ConversionRateResult" );
    } catch( Exception exc ) {
        USDToCAD._statusMessage = "Request " + r.GetRequestID() + " failed: "
+ exc.Message;
    }
   }

   public void OnException( WSException exc, WSResult r ) {
    USDToCAD._statusMessage = "Request " + r.GetRequestID() + " failed: " +
exc.Message;
   }
}
```



```
// Java
private class CurrencyConvertorListener implements WSListener
{
  public CurrencyConvertorListener() {
  }

  public void onResult( WSResult r ) {
   try {
        USDToCAD._statusMessage = "USD to CAD currency exchange rate: " +
r.getDoubleValue( "ConversionRateResult" );
   } catch( Exception exc ) {
        USDToCAD._statusMessage = "Request " + r.getRequestID() + " failed: "
+ exc.getMessage();
   }
   }

   public void onException( WSException exc, WSResult r ) {
        USDToCAD._statusMessage = "Request " + r.getRequestID() + " failed: "
+ exc.getMessage();
   }
}
```

Setting up web service connectors

A web service connector listens for QAnywhere messages sent to a particular address, and makes web service calls when messages arrive. Web service responses are sent back to the originating client as QAnywhere messages. All messages sent to the web services connector should be created using the proxy classes generated by the QAnywhere WSDL compiler.

♦ To create a web service connector

1. Open Sybase Central and connect to your server message store.
2. Choose File ► New Connector.

   The Connector wizard appears.
3. In the Connector Type page, choose Web Services and click Next.
4. In the Connector Name page, enter the Connector Name.

   This is the connector address that a QAnywhere client should use to address the connector. It sets the property ianywhere.connector.address.
5. In the Communication Parameters page, enter the URL.

   This is the URL where the web service is located. (For example, *http://localhost:8080/qanyserv/F2C*.) It sets the property webservice.url.

   You can optionally specify a timeout period in milliseconds, which cancels requests if the web service does not respond in the amount of time your specify. This sets the property webservice.socket.timeout.
6. In the HTTP Parameters page, optionally enter the following values:

   - **HTTP User Name** If the web service requires HTTP authentication, use this property to specify the user name.

     This sets the property webservice.http.authName.
   - **HTTP Password** If the web service requires HTTP authentication, use this property to specify the password.

     This sets the property webservice.http.password.e.
   - **Proxy Host Name** If the web service must be accessed through an HTTP proxy, use this property to specify the host name. If you specify this property, you must specify the webservice.http.proxy.port property.

     This sets the property webservice.http.proxy.host.
   - **Proxy Port** The port to connect to on the proxy server. If you specify this property, you must specify the webservice.http.proxy.host property.

     This sets the property webservice.http.proxy.port.

- **Proxy User Name** The proxy user name to use if the proxy requires authentication. If you specify this property, you must also specify the webservice.http.proxy.password.e property.

  This sets the property webservice.http.proxy.authName.

- **Proxy Password** The proxy password to use if the proxy requires authentication. If you specify this property, you must also specify the webservice.http.proxy.authName property.

  This sets the property webservice.http.proxy.password.e.

7. Click Finish.
8. To set additional options on your web service connector, you can right-click the connector you just created and choose Properties; or you can use server management requests.

   ☞ For a list of available properties, see "Web service connector properties" on page 196.

   ☞ For information about using server management requests, see "Administering connectors" on page 104.

Web service connector properties

Use web service connector properties to specify connection information with the web service. You can set these properties in the Sybase Central Connector wizard.

☞ See "Setting up web service connectors" on page 195.

You can view web service connector properties in the Sybase Central Connector Properties dialog, or in the ml_qa_global_props MobiLink system table.

To open the Connector Properties dialog, right-click the connector in Sybase Central and choose Properties.

☞ For more information about the ml_qa_global_props MobiLink system table, see "ml_qa_global_props" [*MobiLink - Server Administration*].

Web service connector properties

- **ianywhere.connector.nativeConnection** The Java class that implements the connector. It is for QAnywhere internal use only, and should not be deleted or modified.
- **ianywhere.connector.id (deprecated)** An identifier that uniquely identifies the connector. The default is ianywhere.connector.address.
- **ianywhere.connector.address** The connector address that a QAnywhere client should use to address the connector. This address is also used to prefix all logged error, warning, and informational messages appearing in the server console for this connector.

  In Sybase Central, you set this property in the Connector wizard, Connector Name page, Connector Name field.

- **ianywhere.connector.compressionLevel** The default compression factor of messages received from the web service. Compression is an integer between 0 and 9, with 0 indicating no compression and 9 indicating maximum compression.

  In Sybase Central, you set this property on the connector properties dialog, on the General tab, in the Compression Level section.

- **ianywhere.connector.logLevel** The amount of connector information displayed in the MobiLink server console and log file. Values for the log level are as follows:

  - **1** Log error messages.
  - **2** Log error and warning messages.
  - **3** Log error, warning, and information messages.
  - **4** Log error, warning, information, and debug messages.

  In Sybase Central, you set this property on the connector properties dialog, on the General tab, in the Logging Level section.

- **ianywhere.connector.outgoing.retry.max** The default number of retries for messages going from QAnywhere to the external messaging system. The default value is 5. Specify 0 to have the connector retry forever.

  In Sybase Central, you can set this property in the connector properties dialog Properties tab, by clicking New.

- **ianywhere.connector.startupType** Startup types can be automatic, manual, or disabled.

- **webservice.http.authName** If the web service requires HTTP authentication, use this property to specify the user name.

- **webservice.http.password.e** If the web service requires HTTP authentication, use this property to specify the password.

- **webservice.http.proxy.authName** If the proxy requires authentication, use this property to set the proxy user name. If you specify this property, you must also specify the webservice.http.proxy.password.e property.

- **webservice.http.proxy.host** If the web service must be accessed through an HTTP proxy, use this property to specify the host name. If you specify this property, you must specify the webservice.http.proxy.port property.

- **webservice.http.proxy.password.e** If the proxy requires authentication, use this property to set the proxy password. If you specify this property, you must also specify the webservice.http.proxy.authName property.

- **webservice.http.proxy.port** The port to connect to on the proxy server. If you specify this property, you must specify the webservice.http.proxy.host property.

Mobile web service example

This example shows you how to create a mobile web service application. The example uses a non-existent web service and so is designed to be read, not run.

For a more full-featured example, see the sample that is installed to *samples-dir**QAnywhere**MobileWebServices*. (For information about *samples-dir*, see "The samples directory" [*SQL Anywhere Server - Database Administration*].)

Global Weather web service

Suppose there is a web service called Global Weather. The following WSDL file, called *globalweather.wsdl*, describes this web service:

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:s="http://www.w3.org/2001/XMLSchema"
                  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
                  xmlns:tns="http://www.myweather.com"
                  targetNamespace="http://www.myweather.com"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

  <wsdl:types>
    <s:schema targetNamespace="http://www.myweather.com">
      <s:element name="GetWeather">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="CityName"
type="s:string" />
            <s:element minOccurs="0" maxOccurs="1" name="CountryName"
type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="GetWeatherResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="GetWeatherResult"
type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
  </wsdl:types>


  <wsdl:message name="GetWeatherSoapIn">
    <wsdl:part name="parameters" element="tns:GetWeather" />
  </wsdl:message>
  <wsdl:message name="GetWeatherSoapOut">
    <wsdl:part name="parameters" element="tns:GetWeatherResponse" />
  </wsdl:message>

  <wsdl:portType name="GlobalWeatherSoap">
    <wsdl:operation name="GetWeather">
      <wsdl:input message="tns:GetWeatherSoapIn" />
      <wsdl:output message="tns:GetWeatherSoapOut" />
    </wsdl:operation>
  </wsdl:portType>
```

```
  <wsdl:binding name="GlobalWeatherSoap" type="tns:GlobalWeatherSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
style="document" />
    <wsdl:operation name="GetWeather">
      <soap:operation soapAction="http://www.myweather.com/GetWeather"
style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="GlobalWeather">
    <wsdl:port name="GlobalWeatherSoap" binding="tns:GlobalWeatherSoap">
      <soap:address location="http://www.myweather.com/" />
    </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

Generate proxy class

To create a mobile application to access the Global Weather web service, you first run the QAnywhere WSDL compiler. It generates a proxy class that can be used in an application to make requests of the global weather service. In this example, the application is written in Java.

```
wsdlc -l java globalweather.wsdl
```

This command generates a proxy class called *GlobalWeatherSoap.java*, located in the *com\myweather* directory (relative to the current directory). This proxy class is the service binding class for your application. The following is the content of *GlobalWeatherSoap.java*:

```
/*
 * GlobalWeatherSoap.java
 *
 * Generated by the iAnywhere WSDL Compiler
 */

package com.myweather;

import ianywhere.qanywhere.ws.*;
import ianywhere.qanywhere.client.QABinaryMessage;
import ianywhere.qanywhere.client.QAException;

public class GlobalWeatherSoap extends ianywhere.qanywhere.ws.WSBase
{

    public GlobalWeatherSoap(String iniFile) throws WSException
    {
        super(iniFile);
        init();
    }

    public GlobalWeatherSoap() throws WSException
    {
        init();
    }
```



```
    public void init()
    {

        setServiceName("GlobalWeather");
    }



    public java.lang.String getWeather(java.lang.String cityName,
                        java.lang.String countryName)  throws
QAException,WSException,WSFaultException
    {
        StringBuffer    soapRequest = new StringBuffer();
        QABinaryMessage    qaRequestMsg = null;
        String        responsePartName = "GetWeatherResult";
        java.lang.String    returnValue;

        writeSOAPHeader( soapRequest, "GetWeather", "http://
www.myweather.com" );
        soapRequest.append( WSBaseTypeSerializer.serialize
("CityName",cityName,"string","http://www.w3.org/2001/
XMLSchema",true,true) );
        soapRequest.append( WSBaseTypeSerializer.serialize
("CountryName",countryName,"string","http://www.w3.org/2001/
XMLSchema",true,true) );
        writeSOAPFooter( soapRequest, "GetWeather" );

        qaRequestMsg = createQAMessage( soapRequest.toString(), "http://
www.myweather.com/GetWeather", "GetWeatherResponse" );

        WSResult wsResult = invokeWait( qaRequestMsg );

        returnValue = wsResult.getStringValue(responsePartName);

        return returnValue;
    }


    public WSResult asyncGetWeather(java.lang.String cityName,
                        java.lang.String countryName)  throws
QAException,WSException
    {
        StringBuffer    soapRequest = new StringBuffer();
        QABinaryMessage    qaRequestMsg = null;

        writeSOAPHeader( soapRequest, "GetWeather", "http://
www.myweather.com" );
        soapRequest.append( WSBaseTypeSerializer.serialize
("CityName",cityName,"string","http://www.w3.org/2001/
XMLSchema",true,true) );
        soapRequest.append( WSBaseTypeSerializer.serialize
("CountryName",countryName,"string","http://www.w3.org/2001/
XMLSchema",true,true) );
        writeSOAPFooter( soapRequest, "GetWeather" );

        qaRequestMsg = createQAMessage( soapRequest.toString(), "http://
www.myweather.com/GetWeather", "GetWeatherResponse" );

        WSResult wsResult = invoke( qaRequestMsg );

        return wsResult;
    }

}
```


Write mobile web service applications

Next, write applications that use the service binding class to make requests of the web service and process the results. Following are two applications, both of which make web service requests offline and process the results at a later time.

The first application, called RequestWeather, makes a request of the global weather service and displays the ID of the request:

```
import ianywhere.qanywhere.client.*;
import ianywhere.qanywhere.ws.*;
import com.myweather.GlobalWeatherSoap;

class RequestWeather
{
    public static void main( String [] args ) {
       try {
           // QAnywhere initialization
           QAManager mgr = QAManagerFactory.getInstance().createQAManager();
           mgr.open( AcknowledgementMode.EXPLICIT_ACKNOWLEDGEMENT );
           mgr.start();

           // Instantiate the web service proxy
           GlobalWeatherSoap service = new GlobalWeatherSoap();
           service.setQAManager( mgr );
           service.setProperty( "WS_CONNECTOR_ADDRESS",
"ianywhere.connector.globalweather\\" );


           // Make a request to get weather for Beijing
           WSResult r = service.asyncGetWeather( "Beijing", "China" );

           // Display the request ID so that it can be used by ShowWeather
           System.out.println( "Request ID: " + r.getRequestID() );

           // QAnywhere finalization
           mgr.stop();
           mgr.close();

       } catch( Exception exc ) {
           System.out.println( exc.getMessage() );
       }
    }
}
```

The second application, called ShowWeather, shows the weather conditions for a given request ID:

```
import ianywhere.qanywhere.client.*;
import ianywhere.qanywhere.ws.*;
import com.myweather.GlobalWeatherSoap;

class ShowWeather
{
   public static void main( String [] args ) {
      try {
        // QAnywhere initialization
        QAManager mgr = QAManagerFactory.getInstance().createQAManager();
        mgr.open( AcknowledgementMode.EXPLICIT_ACKNOWLEDGEMENT );
        mgr.start();

        // Instantiate the web service proxy
        GlobalWeatherSoap service = new GlobalWeatherSoap();
```



```
        service.setQAManager( mgr );

        // Get the response for the specified request ID
        WSResult r = service.getResult( args[0] );
        if( r.getStatus() == WSStatus.STATUS_RESULT_AVAILABLE ) {
        System.out.println( "The weather is " + r.getStringValue
( "GetWeatherResult" ) );
        r.acknowledge();
        } else {
        System.out.println( "Response not available" );
        }


        // QAnywhere finalization
        mgr.stop();
        mgr.close();

    } catch( Exception exc ) {
        System.out.println( exc.getMessage() );
    }
    }
}
```

Compile the application and the service binding class:

```
javac -classpath ".;%sqlany10%\java\iawsrt.jar;%sqlany10%\java\qaclient.jar"
com\myweather\GlobalWeatherSoap.java   RequestWeather.java
javac -classpath ".;%sqlany10%\java\iawsrt.jar;%sqlany10%\java\qaclient.jar"
com\myweather\GlobalWeatherSoap.java   ShowWeather.java
```

Create QAnywhere message stores and start a QAnywhere Agent

Your mobile web service application requires a client message store on each mobile device. It also requires a server message store, but this example uses the QAnywhere sample server message store.

To create a client message store, create a SQL Anywhere database with the dbinit utility and then run the QAnywhere Agent to set it up as a client message store:

```
dbinit -n -i qanywhere.db
qaagent -q -si -c "dbf=qanywhere.db"
```

Start the QAnywhere Agent to connect to your client message store. The following must all be on one command line:

```
qaagent
  -c "dbf=qanywhere.db;eng=qanywhere;uid=ml_qa_user;pwd=qanywhere"
  -policy automatic
```

Start the MobiLink server. This example uses the QAnywhere sample database as the server message store. The following must all be on one command line:

```
mlsrv10
  -m
  -zu+
  -c "dsn=QAnywhere 10 Demo;uid=ml_server;pwd=sql;start=dbsrv10
       -xs http(port=8080)"
  -v+
  -ot qanyserv.mls
```

For more information about these components, see:

- "Setting up the client message store" on page 33
- "Setting up the server message store" on page 30
- "Running the QAnywhere Agent" on page 35
- "Starting the MobiLink server for QAnywhere messaging" on page 31

Create a web service connector

You must create a web service connector that listens for QAnywhere messages sent to the GetWeather web service, makes web service calls when messages arrive, and sends back responses to the originating client.

Open Sybase Central and connect to your server message store. To create your web service connector, choose File ► New Connector. When the Connector wizard appears, choose Web Services. In the wizard pages, you must set the following properties to match the mobile applications you created earlier in the example:

- In the Connector Name page, enter the Connector Name **ianywhere.connector.globalweather**
- In the Communication Parameters page, enter the URL **http://www.myweather.com/GetWeather**

CHAPTER 10

QAnywhere Properties

Contents

About this chapter

This chapter describes the message properties and client message store properties that you can use in QAnywhere transmission rules and applications.

Message headers and message properties

QAnywhere messages consist of the following parts:

- headers
- properties
- content

Message properties can be referenced in transmission rules and delete rules or in your application.

The following sections describe message headers and properties, and how you can set them in QAnywhere messages.

Notes

- Message headers, message properties, and message content cannot be altered after the message is sent.
- You can read message headers, message properties, and message content after a message is received. If you are using the QAnywhere SQL API, these become unreadable after a commit or rollback occurs.
- The content is unreadable after acknowledgement or commit in all APIs.

Message headers

All QAnywhere messages support the same set of header fields. Header fields contain values that are used by both clients and providers to identify and route messages.

The following message headers are pre-defined. How you use them depends on the type of client application you have.

- **Message ID** Read-only. The message ID of the new message. This header has a value only after the message is sent. See:
  - .NET API: “MessageID property” on page 318
  - C++ API: “getMessageID function” on page 474 and “setMessageID function” on page 484
  - Java API: “getMessageID method” on page 577
  - SQL API: “ml_qa_createmessage” on page 666 and “ml_qa_getmessage” on page 666
- **Message creation timestamp** Read-only. The Timestamp header field contains the time a message was created. It is a coordinated universal time (UTC). It is not the time the message was actually transmitted, because the actual send may occur later due to transactions or other client-side queuing of messages. You can read this header after a message is received and until a rollback or commit occurs; after that you cannot read it. See:
  - .NET API: “Timestamp property” on page 319
  - C++ API: “getTimestamp function” on page 478 and “setTimestamp function” on page 486
  - Java API: “getTimestamp method” on page 581
  - SQL API: “ml_qa_gettimestamp” on page 641

- **Reply-to address** Read-write. The reply address as VARCHAR(128) or NULL if it does not exist. You can read this header after a message is received and until a rollback or commit occurs; after that you cannot read it. See:

  - .NET API: "ReplyToAddress property" on page 319
  - C++ API: "getReplyToAddress function" on page 476 and "setReplyToAddress function" on page 485
  - Java API: "getReplyToAddress method" on page 579 and "setReplyToAddress method" on page 587
  - SQL API:"ml_qa_getreplytoaddress" on page 640 and "ml_qa_setreplytoaddress" on page 645

- **Message address** Read-write. The QAnywhere message address as VARCHAR(128). QAnywhere message addresses take the form *id\queue-name*. You can read this header after a message is received and until a rollback or commit occurs; after that you cannot read it. See:

  - .NET API: "Address property" on page 316
  - C++ API: "getAddress function" on page 469 and "setAddress function" on page 480
  - Java API: "getAddress method" on page 572 and "setAddress method" on page 582
  - SQL API:"ml_qa_getaddress" on page 636 and "ml_qa_getaddress" on page 636

- **Redelivered state of message** Read-only. The redelivered value as BIT. A value of 1 indicates that the message is being redelivered; 0 indicates that it is not being redelivered.

  A message may be redelivered if it was previously received but not acknowledged. For example, the message was received but the application receiving the message did not complete processing the message content before it crashed. In these cases, QAnywhere marks the message as redelivered to alert the receiver that the message might be partly processed.

  For example, assume that the receipt of a message occurs in three steps:

  1. An application using a non-transactional QAnywhere manager receives the message.
  2. The application writes the message content and message ID to a database table called T1, and commits the change.
  3. The application acknowledges the message.

  If the application fails between steps 1 and 2 or between steps 2 and 3, the message is redelivered when the application restarts.

  If the failure occurs between steps 1 and 2, you should process the redelivered message by running steps 2 and 3. If the failure occurs between steps 2 and 3, then the message is already processed and you only need to acknowledge it.

  To determine what happened when the application fails, you can have the application call ml_qa_getredelivered to check if the message has been previously redelivered. Only messages that are redelivered need to be looked up in table T1. This is more efficient than having the application access the received message's message ID to check whether the message is in the table T1, because application failures are rare.

  You can read this header after a message is received and until a rollback or commit occurs; after that you cannot read it.

See:

- .NET API: "Redelivered property" on page 318
- C++ API: "getRedelivered function" on page 475 and "setRedelivered function" on page 484
- Java API: "getRedelivered method" on page 579
- SQL API: "ml_qa_getredelivered" on page 639

- **Expiration of message** Read-only except in the SQL API, where it is read-write. The expiration time as TIMESTAMP. Returns NULL if there is no expiration. A message expires if it is not received by the intended recipient in the specified time. The message may then be deleted using default QAnywhere delete rules. You can read this header after a message is received and until a rollback or commit occurs; after that you cannot read it. See:

  - .NET API: "Expiration property" on page 317
  - C++ API: "getExpiration function" on page 471
  - Java API: "getExpiration method" on page 574
  - SQL API: "ml_qa_getexpiration" on page 637 and "ml_qa_setexpiration" on page 642

- **Priority of message** Read-write. The QAnywhere API defines ten levels of priority value, with 0 as the lowest priority and 9 as the highest. Clients should consider priorities 0-4 as gradations of normal priority and priorities 5-9 as gradations of expedited priority. You can read this header after a message is received and until a rollback or commit occurs; after that you cannot read it. See:

  - .NET API: "Priority property" on page 318
  - C++ API: "getPriority function" on page 474
  - Java API: "getPriority method" on page 577
  - SQL API: "ml_qa_getpriority" on page 638

- **Message ID of a message for which this message is a reply** Read-write. The in-reply-to ID as VARCHAR(128). A client can use the InReplyToID header field to link one message with another. A typical use is to link a response message with its request message. The in-reply-to ID is the ID of the message that this message is replying to. You can read this header after a message is received and until a rollback or commit occurs; after that you cannot read it. See:

  - .NET API: "InReplyToID property" on page 317
  - C++ API: "getInReplyToID function" on page 472
  - Java API: "getInReplyToID method" on page 575
  - SQL API: "ml_qa_getinreplytoid" on page 637

Some message headers can be used in transmission rules. For details, see "Variables defined by the rule engine" on page 231.

See also

- .NET API: "QAMessage members" on page 315
- C++ API: "QAMessage class" on page 466
- Java API: "Interface QAMessage" on page 570
- SQL API: "Message headers" on page 636

Message properties

Each message contains a built-in facility for supporting application-defined property values. These message properties allow you to implement application-defined message filtering.

Message properties are name-value pairs that you can optionally insert into messages to provide structure. For example, in the .NET API the pre-defined message property ias_Originator, identified by the constant MessageProperties.ORIGINATOR, provides the message store ID that sent the message. Message properties can be used in transmission rules to determine the suitability of a message for transmission.

There are two types of message property:

- **Pre-defined message properties** These message properties are always prefixed with ias_ or IAS_.
- **Custom message properties** These are message properties that you defined. You cannot prefix them with ias_ or IAS_.

In either case, you access message store properties using get and set methods and pass the name of the pre-defined or custom property as the first parameter.

☞ For more information, see "Managing message properties" on page 211.

Pre-defined message properties

Some message properties have been pre-defined for your convenience. Pre-defined properties can be read but should not be set. The predefined message properties are:

- **ias_Adapters** For network status notification messages, a list of network adapters that can be used to connect to the MobiLink server. The list is a string and is delimited by a vertical bar.
- **ias_DeliveryCount** Int. The number of attempts that have been made so far to deliver the message.
- **ias_MessageType** Int. Indicates the type of the message. The message types can be:

Value	Message type	Description
0	REGULAR	If a message does not have the ias_MessageType property set, it is a regular message.
13	PUSH_NOTIFICATION	When a push notification is received from the server, a message of type PUSH_NOTIFICATION is sent to the system queue. See "Notifications of push notification" on page 53.
14	NETWORK_STATUS_NOTIFICATION	When there is a change in network status, a message of this type is sent to the system queue. See "Network status notifications" on page 52.

- **ias_RASNames** String. For network status notification messages, a list of RAS entry names that can be used to connect to the MobiLink server. The list is delimited by a vertical bar.

- **ias_NetworkStatus** Int. For network status notification messages, the state of the network connection. The value is 1 if connected, 0 otherwise.
- **ias_Originator** String. The message store ID of the originator of the message.
- **ias_Status** Int. The current status of the message. This property is not supported in the SQL API. The values can be:

Status Code	Description
1	Pending - The message has been sent but not received.
10	Receiving - The message is in the process of being received, or it was received but not acknowledged.
20	Final - The message has achieved a final state.
30	Expired - The message was not received before its expiration time has passed.
40	Cancelled - The message has been cancelled.
50	Unreceivable - The message is either malformed, or there were too many failed attempts to deliver it.
60	Received - The message has been received and acknowledged.

There are constants for the status values. See:

- .NET API: "StatusCodes enumeration" on page 340
- C++ API: "StatusCodes class" on page 498
- Java API: "Interface StatusCodes" on page 598

- **ias_StatusTime** The time at which the message became its current status. It is in units that are natural for the platform. It is a local time. In the C++ API, for Windows and PocketPC platforms, the timestamp is the SYSTEMTIME, converted to a FILETIME, which is copied to a qa_long value. This property is not supported in the SQL API.

API	This property returns...
.NET	DateTime
C++	string
Java	java.util.Date object

Message property constants

The QAnywhere APIs for .NET, C++, and Java provide constants for specifying message properties. See:

- .NET API: "MessageProperties members" on page 245
- C++ API: "MessageProperties class" on page 394
- Java API: "Interface MessageProperties" on page 505

Custom message properties

QAnywhere allows you to define message properties using the C++, Java, or .NET APIs. Custom message properties allow you to create name-value pairs that you associate with an object. For example:

```
msg.SetStringProperty("Product", "widget");
msg.SetFloatProperty("Price",1.00);
msg.SetIntProperty("Quantity",10);
```

Message property names are case insensitive. You can use a sequence of letters, digits and underscores, but the first character must be a letter. The following names are reserved and may not be used as message property names:

- NULL
- TRUE
- FALSE
- NOT
- AND
- OR
- BETWEEN
- LIKE
- IN
- IS
- ESCAPE
- Any name beginning with **ias_**

Managing message properties

The following QAMessage methods can be used to manage message properties.

You can get and set custom properties, but should only get pre-defined properties.

.NET methods to manage message properties

- Object GetProperty(String name)
- void SetProperty(String name, Object value)
- boolean GetBooleanProperty(String name)
- void SetBooleanProperty(String name, boolean value)
- byte GetByteProperty(String name)
- void SetByteProperty(String name, byte value)
- short GetShortProperty(String name)
- void SetShortProperty(String name, short value)
- int GetIntProperty(String name)
- void SetIntProperty(String name, int value)
- long GetLongProperty(String name)
- void SetLongProperty(String name, long value)
- float GetFloatProperty(String name)
- void SetFloatProperty(String name, float value)

- double GetDoubleProperty(String name)
- void SetDoubleProperty(String name, double value)
- String GetStringProperty(String name)
- void SetStringProperty(String name, String value)
- IEnumerator GetPropertyNames()
- void ClearProperties()
- PropertyType GetPropertyType(string propName)
- bool PropertyExists(string propName)

☞ For more information, see "QAMessage interface" on page 314.

C++ methods to manage message properties

- qa_bool getBooleanProperty(qa_const_string name, qa_bool * value)
- qa_bool setBooleanProperty(qa_const_string name, qa_bool value)
- qa_bool getByteProperty(qa_const_string name, qa_byte * value)
- qa_bool setByteProperty(qa_const_string name, qa_byte value)
- qa_bool getShortProperty(qa_const_string name, qa_short * value)
- qa_bool setShortProperty(qa_const_string name, qa_short value)
- qa_bool getIntProperty(qa_const_string name, qa_int * value)
- qa_bool setIntProperty(qa_const_string name, qa_int value)
- qa_bool getLongProperty(qa_const_string name, qa_long * value)
- qa_bool setLongProperty(qa_const_string name, qa_long value)
- qa_bool getFloatProperty(qa_const_string name, qa_float * value)
- qa_bool setFloatProperty(qa_const_string name, qa_float value)
- qa_bool getDoubleProperty(qa_const_string name, qa_double * value)
- qa_bool setDoubleProperty(qa_const_string name, qa_double value)
- qa_int getStringProperty(qa_const_string name, qa_string value, qa_int len)
- qa_bool setStringProperty(qa_const_string name, qa_const_string value)
- void QAMessage::clearProperties()
- qa_short QAMessage::getPropertyType(qa_const_string name)
- qa_bool QAMessage::propertyExists(qa_const_string name)
-

☞ For more information, see "QAMessage class" on page 466.

Java methods to manage message properties

- void clearProperties()
- boolean getBooleanProperty(String name)
- void setBooleanProperty(String name, boolean value)
- byte getByteProperty(String name)
- void setByteProperty(String name, byte value)
- double getDoubleProperty(String name)
- void setDoubleProperty(String name, double value)
- java.util.Date getExpiration() void setFloatProperty(String name, float value)
- float getFloatProperty(String name)
- int getIntProperty(String name)
- void setIntProperty(String name, int value)

- long getLongProperty(String name)
- void setLongProperty(String name, long value)
- Object getProperty(String name)
- void setProperty(String name, Object value)
- java.util.Enumeration getPropertyNames()
- short getPropertyType(String name)
- short getShortProperty(String name)
- void setShortProperty(String name, short value)
- String getStringProperty(String name)
- void setStringProperty(String name, String value)
- boolean propertyExists(String name)

☞ For more information, see "Interface QAMessage" on page 570.

SQL stored procedures to manage message properties

- ml_qa_getbooleanproperty
- ml_qa_getbyteproperty
- ml_qa_getdoubleproperty
- ml_qa_getfloatproperty
- ml_qa_getintproperty
- ml_qa_getlongproperty
- ml_qa_getpropertynames
- ml_qa_getshortproperty
- ml_qa_getstringproperty
- ml_qa_setbooleanproperty
- ml_qa_setbyteproperty
- ml_qa_setdoubleproperty
- ml_qa_setfloatproperty
- ml_qa_setfloatproperty
- ml_qa_setintproperty
- ml_qa_setlongproperty
- ml_qa_setshortproperty
- ml_qa_setstringproperty

☞ For more information, see "Message properties" on page 645.

Example

```
// C++ example.
QAManagerFactory factory;
QAManager * mgr = factory->createQAManager( NULL );
mgr->open(AcknowledgementMode::EXPLICIT_ACKNOWLEDGEMENT);
QAMessage * msg = mgr->createTextMessage();
msg->setStringProperty( "tm_Subject", "Some message subject." );
mgr->putMessage( "myqueue", mgr );

// C# example.
QAManager mgr = QAManagerFactory.Instance.CreateQAManager(null);
mgr.Open(AcknowledgementMode.EXPLICIT_ACKNOWLEDGEMENT);
QAMessage msg = mgr.CreateTextMessage();
msg.SetStringProperty( "tm_Subject", "Some message subject." );
mgr.PutMessage( "myqueue", msg );
```



```
// Java example
QAManager mgr = QAManagerFactory.getInstance().createQAManager(null);
mgr.open(AcknowledgementMode.EXPLICIT_ACKNOWLEDGEMENT);
QAMessage msg = mgr.createTextMessage();
msg.setStringProperty("tm_Subject", "Some message subject.");
mgr.putMessage("myqueue", mgr);

-- SQL example
begin
    DECLARE @msgid VARCHAR(128);
    SET @msgid = ml_qa_createmessage();
    CALL ml_qa_setfloatproperty( @msgid, 'myfloatproperty1', -1.3e-5 );
    CALL ml_qa_setfloatproperty( @msgid, 'myfloatproperty2', 1.3e5 );
    CALL ml_qa_putmessage( @msgid, 'someclient\someaddress' );
    COMMIT;
end
```

Client message store properties

There are two types of client message store property:

- **Pre-defined message store properties** These message store properties are always prefixed with ias_ or IAS_.
- **Custom message store properties** These are message store properties that you defined. You cannot prefix them with ias_ or IAS_.

In either case, you access client message store properties using get and set methods defined in the appropriate class and pass the name of the pre-defined or custom property as the first parameter.

☞ For more information, see "Managing client message store properties" on page 219.

You can also use message store properties in transmission rules, delete rules, and message selectors. See:

- "QAnywhere Transmission and Delete Rules" on page 225

Pre-defined client message store properties

A number of client message store properties have been pre-defined for your convenience. The predefined message store properties are:

- **ias_Adapters** A list of network adapters that can be used to connect to the MobiLink server. The list is a string and is delimited by a vertical bar.
- **ias_MaxDeliveryAttempts** When defined, the maximum number of times that a message can be received without being acknowledged before its status is set to UNRECEIVABLE. By default, this property is not defined and is equivalent to a value of -1, which means that the client library will continue to attempt to deliver an unacknowledged message forever.
- **ias_MaxUploadSize** The upload increment size. By default, QAnywhere uploads messages in increments of 256K, but no matter what the upload size is set to, QAnywhere will send at least one message per increment and will not split messages. This property is identical to the qaagent -iu option. See "-iu option" on page 153.
- **ias_Network** Information about the current network in use. This property can be read but should not be set. ias_Network is a special property. It has a number of built-in attributes that provide information regarding the current network that is being used by the device. The following attributes are automatically set by QAnywhere:
  - **ias_Network.Adapter** The current name of the network card, if any. (The name of the network card that is assigned to the Adapter attribute is displayed in the Agent window when the network connection is established.)
  - **ias_Network.RAS** The current RAS entry name, if any.
  - **ias_Network.IP** The current IP address assigned to the device, if any.

  - **ias_Network.MAC** The current MAC address of the network card being used, if any.

- **ias_RASNames** String. A list of RAS entry names that can be used to connect to the MobiLink server. The list is delimited by a vertical bar.
- **ias_StoreID** The message store ID.
- **ias_StoreInitialized** True if this message stores has successfully been initialized for QAnywhere messaging; otherwise False.

  ☞ See "-si option" on page 169.

- **ias_StoreVersion** The QAnywhere-defined version number of this message store.

For information about managing pre-defined message properties, see:

- C++ API: "MessageStoreProperties class" on page 401
- .NET API: "MessageStoreProperties class" on page 253
- Java API: "Interface MessageStoreProperties" on page 510
- SQL API: "Message store properties" on page 664

Custom client message store properties

QAnywhere allows you to define your own client message store properties using the QAnywhere C++, Java, SQL or .NET APIs. These properties are shared between applications connected to the same message store. They are also synchronized to the server message store so that they are available to server-side transmission rules for this client.

Client message store property names are case insensitive. You can use a sequence of letters, digits, and underscores, but the first character must be a letter. The following names are reserved and may not be used as message store property names:

- NULL
- TRUE
- FALSE
- NOT
- AND
- OR
- BETWEEN
- LIKE
- IN
- IS
- ESCAPE
- Any name beginning with **ias**_

Using custom client message store property attributes

Client message store properties can have attributes that you define. An attribute is defined by appending a dot after the property name followed by the attribute name. The main use of this feature is to be able to use information about your network in your transmission rules.

Example

Following is a simple example of how to set custom client message store property attributes. In this example, the Object property has two attributes: Shape and Color. The value of the Shape attribute is Round and the value of the Color attribute is Blue.

```
// C++ example.
mgr->setStringStoreProperty( “Object.Shape”, “Round” );
mgr->setStringStoreProperty( “Object.Color”, “Blue” );

// C# example.
mgr.SetStoreStringProperty( “Object.Shape”, “Round” );
mgr.SetStringStoreProperty( “Object.Color”, “Blue” );

// Java example
mgr.setStringStoreProperty( "Object.Shape", "Round" );
mgr.setStringStoreProperty( "Object.Color", "Blue" );

-- SQL example
BEGIN
    CALL ml_qa_setstoreproperty( 'Object.Shape', 'Round' );
    CALL ml_qa_setstoreproperty( 'Object.Color', 'Blue' );
    COMMIT;
END
```

All client message store properties have a Type attribute that initially has no value. The value of the Type attribute must be the name of another property. When setting the Type attribute of a property, the property inherits the attributes of the property being assigned to it. In the following example, the Object property inherits the attributes of the Circle property. Therefore, the value of Object.Shape is Round and the value of Object.Color is Blue.

```
// C++ example
QAManager    qa_manager;
qa_manager->setStoreStringProperty( “Circle.Shape”, “Round” );
qa_manager->setStoreStringProperty( “Circle.Color”, “Blue” );
qa_manager->setStoreStringProperty( “Object.Type”, “Circle” );

// C# example
QAManager    qa_manager;
qa_manager.SetStringStoreProperty( “Circle.Shape”, “Round” );
qa_manager.SetStringStoreProperty( “Circle.Color”, “Blue” );
qa_manager.SetStringStoreProperty( “Object.Type”, “Circle” );

// Java example
QAManager    qa_manager;
qa_manager.setStringStoreProperty( "Circle.Shape", "Round" );
qa_manager.setStringStoreProperty( "Circle.Color", "Blue" );
qa_manager.setStringStoreProperty( "Object.Type", "Circle" );

-- SQL example
BEGIN
    CALL ml_qa_setstoreproperty( 'Circle.Shape', 'Round' );
    CALL ml_qa_setstoreproperty( 'Circle.Color', 'Blue' );
    CALL ml_qa_setstoreproperty( 'Object.Type', 'Circle');
```



```
    COMMIT;
END
```

Example

The following C# example shows how you can use message store properties to provide information about your network to your transmission rules.

Assume you have a Windows laptop that has the following network connectivity options: LAN, Wireless LAN, and Wireless WAN. Access to the network via LAN is provided by a network card named My LAN Card. Access to the network via Wireless LAN is provided by a network card named My Wireless LAN Card. Access to the network via Wireless WAN is provided by a network card named My Wireless WAN Card.

Assume you want to develop a messaging application that sends all messages to the server when connected using LAN or Wireless LAN and only high priority messages when connected using Wireless WAN. You define high priority messages as those whose priority is greater than or equal to 7.

First, find the names of your network adapters. The names of network adapters are fixed when the card is plugged in and the driver is installed. To find the name of a particular network card, connect to the network through that adapter, and then run qaagent with the -vn option. The QAnywhere Agent displays the network adapter name, as follows:

"Listener thread received message '[netstat] *network-adapter-name* !...'

Next, define three client message store properties for each of the network types: LAN, WLAN, and WWAN. Each of these properties will be assigned a Cost attribute. The Cost attribute is a value between 1 and 3 and represents the cost incurred when using the network. A value of 1 represents the lowest cost.

```
QAManager    qa_manager;
qa_manager.SetStoreProperty( “LAN.Cost”, “1” );
qa_manager.SetStoreProperty( “WLAN.Cost”, “2” );
qa_manager.SetStoreProperty( “WWAN.Cost”, “3” );
```

Next, define three client message store properties, one for each network card that will be used. The property name must match the network card name. Assign the appropriate network classification to each property by assigning the network type to the Type attribute. Each property will therefore inherit the attributes of the network types assigned to them.

```
QAManager    qa_manager;
qa_manager.SetStoreProperty( “My LAN Card.Type”, “LAN” );
qa_manager.SetStoreProperty( “My Wireless LAN Card.Type”, “WLAN” );
qa_manager.SetStoreProperty( “My Wireless WAN Card.Type”, “WWAN” );
```

When network connectivity is established, QAnywhere automatically defines the Adapter attribute of the ias_Network property to one of My LAN Card, My Wireless LAN Card or My Wireless WAN Card, depending on the network in use. Similarly, it automatically sets the Type attribute of the ias_Network property to one of My LAN Card, My Wireless LAN Card, or My Wireless WAN Card so that the ias_Network property inherits the attributes of the network being used.

Finally, create the following transmission rule.

```
automatic=ias_Network.Cost < 3 or ias_Priority >= 7
```


☞ For more information about transmission rules, see "QAnywhere Transmission and Delete Rules" on page 225.

Enumerating client message store properties

The QAnywhere .NET, C++, and Java APIs can provide an enumeration of predefined and custom client message store properties.

.NET example

☞ For more information, see "GetStorePropertyNames method" on page 297.

```
// qaManager is a QAManager instance.
IEnumerator propertyNames = qaManager.GetStorePropertyNames();
```

C++ example

☞ For more information, see "beginEnumStorePropertyNames function" on page 433.

```
// qaManager is a QAManager instance.
qa_store_property_enum_handle handle = qaManager->beginEnumStorePropertyNames
();
qa_char propertyName[256];
if( handle != qa_null ) {
  while( qaManager->nextStorePropertyName( handle, propertyName, 255 ) !=
-1 ) {
        // Do something with the message store property name.
  }
  // Message store properties cannot be set after
  // the beginEnumStorePropertyNames call
  // and before the endEnumStorePropertyNames call.
  qaManager->endEnumStorePropertyNames(handle);
}
```

Java example

☞ For more information, see "getStorePropertyNames method" on page 556.

```
// qaManager is a QAManager instance.
Enumeration propertyNames = qaManager.getStorePropertyNames();
```

Managing client message store properties

Client message store properties can be set in your client application for each client message store.

☞ See "Managing client message store properties in your application" on page 220.

Client message store properties can be used in transmission rules to filter messages to the client or used in delete rules to determine messages to add.

☞ See "QAnywhere Transmission and Delete Rules" on page 225.

Client message store properties can also be specified in server management messages, and stored on the server message store.

☞ See "About server management requests" on page 92.

Managing client message store properties in your application

The following QAManagerBase methods can be used to get and set client message store properties.

C++ methods to manage client message store properties

- qa_bool getBooleanStoreProperty(qa_const_string name, qa_bool * value)
- qa_bool setBooleanStoreProperty(qa_const_string name, qa_bool value)
- qa_bool getByteStoreProperty(qa_const_string name, qa_byte * value)
- qa_bool setByteStoreProperty(qa_const_string name, qa_byte value)
- qa_bool getShortStoreProperty(qa_const_string name, qa_short * value)
- qa_bool setShortStoreProperty(qa_const_string name, qa_short value)
- qa_bool getIntStoreProperty(qa_const_string name, qa_int * value)
- qa_bool setIntStoreProperty(qa_const_string name, qa_int value)
- qa_bool getLongStoreProperty(qa_const_string name, qa_long * value)
- qa_bool setLongStoreProperty(qa_const_string name, qa_long value)
- qa_bool getFloatStoreProperty(qa_const_string name, qa_float * value)
- qa_bool setFloatStoreProperty(qa_const_string name, qa_float value)
- qa_bool getDoubleStoreProperty(qa_const_string name, qa_double * value)
- qa_bool setDoubleStoreProperty(qa_const_string name, qa_double value)
- qa_int getStringStoreProperty(qa_const_string name, qa_string value, qa_int len)
- qa_bool setStringStoreProperty(qa_const_string name, qa_const_string value)
- qa_store_property_enum_handle QAManagerBase::beginEnumStorePropertyNames()
- virtual qa_int QAManagerBase::nextStorePropertyName(qa_store_property_enum_handle h, qa_string buffer, qa_int bufferLen)
- virtual void QAManagerBase::endEnumStorePropertyNames(qa_store_property_enum_handle h)

☞ See "QAManagerBase class" on page 432.

C# methods to manage client message store properties

- Object GetStoreProperty(String name)
- void SetStoreProperty(String name, Object value)
- boolean GetBooleanStoreProperty(String name)
- void SetBooleanStoreProperty(String name, boolean value)
- byte GetByteStoreProperty(String name)
- void SetByteStoreProperty(String name, byte value)
- short GetShortStoreProperty(String name)
- void SetShortStoreProperty(String name, short value)
- int GetIntStoreProperty(String name)
- void SetIntStoreProperty(String name, int value)
- long GetLongStoreProperty(String name)
- void SetLongStoreProperty(String name, long value)
- float GetFloatStoreProperty(String name)
- void SetFloatStoreProperty(String name, float value)
- double GetDoubleStoreProperty(String name)

- void SetDoubleStoreProperty(String name, double value)
- String GetStringStoreProperty(String name)
- void SetStringStoreProperty(String name, String value)
- IEnumerator GetStorePropertyNames()

☞ See "QAManagerBase interface" on page 275.

Java methods to manage client message store properties

- boolean getBooleanStoreProperty(String name)
- void setBooleanStoreProperty(String name, boolean value)
- byte getByteStoreProperty(String name)
- void setByteStoreProperty(String name, byte value)
- double getDoubleStoreProperty(String name)
- void setDoubleStoreProperty(String name, double value)
- float getFloatStoreProperty(String name)
- void setFloatStoreProperty(String name, float value)
- int getIntStoreProperty(String name)
- void setIntStoreProperty(String name, int value)
- long getLongStoreProperty(String name)
- void setLongStoreProperty(String name, long value)
- short getShortStoreProperty(String name)
- void setShortStoreProperty(String name, short value)
- void setStringStoreProperty(String name, String value)
- String getStringStoreProperty(String name)
- java.util.Enumeration getStorePropertyNames()

☞ See "Interface QAManagerBase" on page 540.

SQL stored procedures to manage client message store properties

- ml_qa_getstoreproperty
- ml_qa_setstoreproperty

☞ See "Message store properties" on page 664.

Server properties

You can set server properties in Sybase Central or with a server management request. In all cases, the server properties are stored in the database. See:

- "Setting server properties with a server management request" on page 114
- "Setting server properties with Sybase Central" on page 223

Server properties

- **ianywhere.qa.server.autoRulesEvaluationPeriod** The time in milliseconds between evaluations of rules, including message transmission and persistence rules. Since, typically, rules are evaluated on the fly as messages are transmitted to the server store, the rule evaluation period is only for rules that are timing-sensitive. The default value is **60000** (one minute).
- **ianywhere.qa.compressionLevel** The default amount of compression applied to each message received by a QAnywhere connector. The compression is an integer between 0 and 9, with 0 being no compression and 9 being the most compression. The default is **0**.

  If you also set the compression level for a connector in the connector properties file, this setting is overridden for that connector. For more information, see "JMS connector properties" on page 132.
- **ianywhere.qa.server.connectorPropertiesFiles**

  Deprecated feature
  Replaced by Sybase Central.

  A list of one or more files that specify the configuration of QAnywhere connectors to an external message system such as JMS. The default is no connectors.

  ☞ For more information, see "JMS Connectors" on page 127.
- **ianywhere.qa.server.disableNotifications** Set this to true to disable notification from the server about pending messages. This disables the processing on the server that is required to initiate notifications to clients when messages are waiting on the server for those clients. Set to true in any setup where notifications cannot be sent from the server, such as when firewall restrictions make notifications impossible. The default is false.
- **ianywhere.qa.server.logLevel** The logging level of the messaging. The property value may be one of 1, 2, 3, or 4. 1 indicates that only message errors are logged. 2 additionally causes warnings to be logged. 3 additionally causes informational messages to be logged. 4 additionally causes more verbose informational messages to be logged, including details about each QAnywhere message that is transmitted with the MobiLink server. The default is **2**.

  These logging messages are output to the MobiLink server console. If the mlsrv10 -o or -ot option was specified, the messages are output to the MobiLink server log file.
- **ianywhere.qa.server.id** Specifies the agent portion of the address to which to send server management requests. If this property is not set, this value is ianywhere.server.

- **ianywhere.qa.server.password.e** Specifies the password for authenticating server management requests. If this property is not set, the password is QAnywhere.

  ☞ See "About server management requests" on page 92.

- **ianywhere.qa.server.scheduleDateFormat** Specifies the date format used for server-side transmission rules. By default, the date format is yyyy-MM-dd.

Letter	Date component	Example
y	year	1996
M	month in year	July
d	day in month	10

- **ianywhere.qa.server.scheduleTimeFormat** Specifies the time format used for server-side transmission rules. By default, the time format is HH:mm:ss.

Letter	Date component	Example
a	AM/PM marker	PM
H	hour in day, a value between 0 and 23	0
k	hour in day, a value between 1 and 24	24
K	hour in AM/PM, a value between 0 and 11	0
h	hour in AM/PM, a value between 1 and 12	12
m	minute in hour	30
s	second in minute	55

- **ianywhere.qa.server.transmissionRulesFile**

Deprecated feature
Replaced by Sybase Central.

A file used to specify rules for governing the transmission and persistence of messages. By default, there are no filters for messages, and messages are deleted when the final status of the message has been transmitted to the message originator.

Setting server properties with Sybase Central

- **To set server properties with Sybase Central**

1. Start Sybase Central:

- Choose Start ► Programs ► SQL Anywhere 10 ► Sybase Central.
- From Connections, choose Connect with QAnywhere 10.
- Specify an ODBC data source name or file, and a user ID and password if required. Click OK.

2. Under Server Store tasks in the left pane, choose Change Properties of this message store.

   The message store Properties dialog appears.

CHAPTER 11

QAnywhere Transmission and Delete Rules

Contents

About this chapter

This chapter describes how to write transmission rules. You can create rules on the server to define which messages should be downloaded to the client and when they should be transmitted, and rules on the client to define which messages should be uploaded to the server and when they should be uploaded.

Rule syntax

Rules have two parts: a schedule and a condition. The schedule defines when an event is to occur. The condition defines which messages are to be part of the event. For example, if the event is message transmission, then the schedule indicates when transmission will occur and the condition defines which messages will be included in the transmission. If the event is message deletion, then the schedule indicates when deleting will occur and the condition indicates which messages will be deleted.

Rule syntax

Each rule has the following form:

schedules=*condition*

Schedule syntax

Schedule syntax

schedules : { **AUTOMATIC** | *schedule-spec* ,… }

schedule-spec :
{ **START TIME** *start-time* | **BETWEEN** *start-time* **AND** *end-time* }
[**EVERY** *period* { **HOURS** | **MINUTES** | **SECONDS** }]
[**ON** { **(** *day-of-week*, … **)** | **(** *day-of-month*, … **)** }]
[**START DATE** *start-date*]

Parameters

- **AUTOMATIC** For transmission rules, rules are evaluated when a message changes state or there is a change in network status. For delete rules, messages that satisfy the delete rule condition are deleted when a message transmission is initiated.
- **schedule-spec** Schedule specifications other than AUTOMATIC specify times when conditions are to be evaluated. At those scheduled times, the corresponding condition is evaluated.
- **START TIME** The first scheduled time for each day on which the event is scheduled. If a START DATE is specified, the START TIME refers to that date. If no START DATE is specified, the START TIME is on the current day (unless the time has passed) and each subsequent day (if the schedule includes EVERY or ON).
- **BETWEEN ... AND ...** A range of times during the day outside of which no scheduled times occur. If a START DATE is specified, the scheduled times do not occur until that date.
- **EVERY** An interval between successive scheduled events. Scheduled events occur only after the START TIME for the day, or in the range specified by BETWEEN … AND.
- **ON** A list of days on which the scheduled events occur. The default is every day if EVERY is specified. Days can be specified as days of the week or days of the month.

  Days of the week are Mon, Tues, and so on. You may also use the full forms of the day, such as Monday. You must use the full forms of the day names if the language you are using is not English, is not the

language requested by the client in the connection string, and is not the language that appears in the server window.

Days of the month are integers from 0 to 31. A value of 0 represents the last day of any month.

- **START DATE** The date on which scheduled events are to start occurring. The default is the current date.

Usage

You can create more than one schedule for a given condition. This permits complex schedules to be implemented.

A schedule specification is recurring if its definition includes EVERY or ON; if neither of these reserved words is used, the schedule specifies at most a single time. An attempt to create a non-recurring schedule for which the start time has passed generates an error.

Each time a scheduled time occurs, the associated condition is evaluated and then the next scheduled time and date is calculated.

The next scheduled time is computed by inspecting the schedule or schedules, and finding the next schedule time that is in the future. If a schedule specifies every minute, and it takes 65 seconds to evaluate the conditions, it runs every two minutes. If you want execution to overlap, you must create more than one rule.

1. If the EVERY clause is used, find whether the next scheduled time falls on the current day, and is before the end of the BETWEEN … AND range. If so, that is the next scheduled time.
2. If the next scheduled time does not fall on the current day, find the next date on which the event is to be executed.
3. Find the START TIME for that date, or the beginning of the BETWEEN … AND range.

The QAnywhere schedule syntax is derived from the SQL Anywhere CREATE EVENT schedule syntax.

Keywords are case insensitive.

See also

- "CREATE EVENT statement" [*SQL Anywhere Server - SQL Reference*]

Example

The following sample server transmission rules file applies to the client identified by the client message store ID sample_store_id. It creates a dual schedule: high priority messages are sent once an hour. The schedule is `every 1 hours` and the condition is `ias_priority=9`. Also, between the hours of 8 A.M. and 9 A.M., high priority messages are sent every minute.

```
[sample_store_id]
; This rule governs when messages are transmitted to the client
; store with id sample_store_id.
;
   every 1 hours = ias_priority=9
   between 8:00 and 9:00 every 1 minutes = iasPriority=9
```


Condition syntax

QAnywhere conditions use a SQL-like syntax. Conditions are evaluated against messages in the message store. A condition evaluates to true, false, or unknown. If a condition is empty, all messages are judged to satisfy the condition. Conditions can be used in transmission rules, delete rules, and the QAnywhere programming APIs.

Keywords and string comparisons are case insensitive.

Syntax

condition :
expression **IS** [**NOT**] **NULL**
| *expression compare expression*
| *expression* [**NOT**] **BETWEEN** *expression* **AND** *expression*
| *expression* [**NOT**] **LIKE** *string* [**ESCAPE** *character*]
| *expression* [**NOT**] **IN (** *string*, ... **)**
| **NOT** *condition*
| *condition* **AND** *condition*
| *condition* **OR** *condition*
| **(** *condition* **)**

compare: = **>** | **<** | **>=** | **<=** | **<>**

expression:
constant
| *rule-variable*
| **-***expression*
| *expression operator expression*
| **(** *expression* **)**
| *rule-function* **(** *expression*, ... **)**

integer: An integer in the range -2**63 to 2**63-1

number: A n umber in scientific notation in the range 2.2250738585072e-308 to 1.79769313486231e+308

string: A sequence of characters enclosed in single quotes. A single quote in a string is represented by two consecutive single quotes.

constant: *integer* | *number* | *string* | **TRUE** | **FALSE**

operator: **+** | **-** | ***** | **/**

rule-variable:

See “Rule variables” on page 231

rule-function:

See “Rule functions” on page 230

Parameters

- **BETWEEN** The BETWEEN condition can evaluate as true, false, or unknown. Without the NOT keyword, the condition evaluates as TRUE if *expression* is greater than or equal to the start expression and less than or equal to the end expression.

The NOT keyword reverses the meaning of the condition but leaves UNKNOWN unchanged.

The BETWEEN condition is equivalent to a combination of two inequalities:

[**NOT**] **(** *expression* **>=** *start-expression* **AND** *arithmetic-expression* **<=** *end-expr* **)**

For example:

- `age BETWEEN 15 AND 19` is equivalent to `age >=15 AND age <= 19`
- `age NOT BETWEEN 15 AND 19` is equivalent to `age < 15 OR age > 19`.

- **IN** The IN condition evaluates according to the following rules:
  - True if *expression* is not null and equals at least one of the values in the list.
  - Unknown if *expression* is null and the values list is not empty, or if at least one of the values is null and expression does not equal any of the other values.
  - False if none of the values are null, and *expression* does not equal any of the values in the list.

  The NOT keyword interchanges true and false.

  For example:

  - `Country IN ( 'UK', 'US', 'France' )` is true for `'UK'` and false for `'Peru'`. It is equivalent to the following:

    ```
    ( Country = 'UK' )       \
    OR ( Country = 'US' )    \
    OR ( Country = 'France' )
    ```

  - `Country NOT IN ( 'UK', 'US', 'France' )` is false for `'UK'` and true for `'Peru'`. It is equivalent to the following:

    ```
    NOT (    ( Country = 'UK' )      \
          OR ( Country = 'US' )      \
          OR ( Country = 'France' ) )
    ```

- **LIKE** The LIKE condition can evaluate as true, false, or unknown.

  Without the NOT keyword, the condition evaluates as TRUE if *expression* matches the *like expression*. If either *expression* or *like expression* is NULL, this condition is unknown.

  The NOT keyword reverses the meaning of the condition, but leaves UNKNOWN unchanged.

  The *like expression* may contain any number of wildcards. The wildcards are:

Wildcard	Matches
_ (underscore)	Any one character
% (percent)	Any string of zero or more characters

  For example:

  - `phone LIKE 12%3` is true for `'123'` or `'12993'` and false for `'1234'`

- `word LIKE 's_d'` is true for `'sad'` and false for `'said'`
- `phone NOT LIKE '12%3'` is false for `'123'` or `'12993'` and true for `'1234'`

- **ESCAPE CHARACTER** The ESCAPE CHARACTER is a single character string literal whose character is used to escape the special meaning of the wildcard characters (_, **%**) in *pattern*. For example:
  - `underscored LIKE '\_%' ESCAPE '\'` is true for `'_myvar'` and false for `'myvar'`.
- **IS NULL** The IS NULL condition evaluates to true if the rule-variable is unknown; otherwise it evaluates to false. The NOT keyword reverses the meaning of the condition. This condition cannot evaluate to unknown.

Rule functions

You can use the following functions in transmission rules:

Syntax	Description
DATEADD(*datepart*, *count*, *datetime* **)**	Returns a datetime produced by adding a number of date parts to a datetime. The *datepart* can be one of year, quarter, month, week, day, hour, minute, or second. For example, the following example adds two months, resulting in the value 2006-07-02 00:00:00.0: `DATEADD( month, 2, '2006/05/02' )`
DATEPART(*datepart*, *date* **)**	Returns the value of part of a datetime value. The datepart can be one of year, quarter, month, week, day, dayofyear, weekday, hour, minute, or second. For example, the following example gets the month May as a number, resulting in the value 5: `DATEPART( month, '2006/05/02' )`
DATETIME(*string* **)**	Converts a string value to a datetime. The string must have the format 'yyyy-mm-dd hh:nn:ss'.
LENGTH(*string* **)**	Returns the number of characters in a string.
SUBSTRING(*string*, *start*, *length* **)**	Returns a substring of a string. The *start* is the start position of the substring to return, in characters. The *length* is the length of the substring to return, in characters.

Example

The following delete rule deletes all messages that entered a final state more than 10 days ago:

```
every 1 hours = ias_Status >= ias_FinalState
   AND ias_StatusTime < DATEADD( day, -10, ias_CurrentTimestamp )
   AND ias_TransmissionStatus = 1
```

Rule variables

QAnywhere rule variables can be used in the condition part of rules. You can use the following as rule variables:

- "Message properties" on page 209
- "Client message store properties" on page 215
- "Variables defined by the rule engine" on page 231

Using properties as rule variables

Message properties and message store properties can be used as transmission rule variables. In both cases you can use pre-defined properties or you can create custom properties. If you have a message property and a message store property with the same name, the message property is used. To override this precedence, you can explicitly reference the property as follows:

- Preface a message store property name with ias_Store.
- Preface a message property name with ias_Message.

For example, the following automatic transmission rule selects all messages with the custom message property called urgent set to yes:

```
automatic = ias_Message.urgent = 'yes'
```

The following automatic transmission rule selects messages when the custom message store property transmitNow is set to yes:

```
automatic = ias_Store.transmitNow = 'yes'
```

Variables defined by the rule engine

The following variables are defined by the rule engine:

- **ias_Address** The address of the message. For example, myclient\myqueue.
- **ias_Content** The body of the message. Character comparisons are done in the character set of the message store default. Searching content can significantly slow rule evaluation.
- **ias_ContentSize** The size of the message content. If the message is a text message, this is the number of characters. If the message is binary, this is the number of bytes.
- **ias_ContentType** The type of message:

IAS_TEXT_CONTENT	The message content consists of unicode characters.
IAS_BINARY_CONTENT	The message content is treated as an uninterpreted sequence of bytes.

- **ias_Expires** The date and time when the message will expire if it is not delivered.

- **ias_Priority** The priority of message: an integer between 0 and 9, where 0 indicates less priority and 9 indicates more priority.
- **ias_TransmissionStatus** The synchronization status of the message. It can be one of:

IAS_UNTRANSMITTED	The message has not been transmitted to its intended recipient message store.
IAS_TRANSMITTED	The message has been transmitted to its intended recipient message store.
IAS_DO_NOT_TRANSMIT	The recipient and originating message stores are the same so no transmission is necessary.
IAS_TRANSMITTING	The message has been transmitted to its intended recipient, but that transmission has yet to be confirmed. There is a possibility that the message transmission was interrupted, and that QAnywhere may transmit the message again.

Status constants defined by the rule engine

- **ias_CurrentDate** The current date.

  A string can be compared against ias_currentDate if it is supplied in one of two ways:

  - as a string of format, which is interpreted unambiguously.
  - as a string according to the date_format database option set for the client message store database.

    For more information, see "Setting options" [*SQL Anywhere Server - Database Administration*] and "date_format option [compatibility]" [*SQL Anywhere Server - Database Administration*].

- **ias_Status** The current status of the message. The values can be:
  - **ias_CancelledState** The message has been cancelled.
  - **ias_ExpiredState** The message expired before it could be received by the intended recipient.
  - **ias_FinalState** The message is received or expired. Therefore, `>=ias_FinalState` means that the message is received or expired, and `<ias_FinalState` means that the message is neither received nor expired.
  - **ias_PendingState** The message has not yet been received by the intended recipient.
  - **ias_ReceivedState** The message was received by the intended recipient.
  - **ias_UnreceivableState** The message has been marked as unreceivable because it is either malformed or there were too many failed attempts to deliver it.
- **ias_Network** Information about the current network in use. ias_Network is a special transmission variable. It has a number of built-in attributes that provide information regarding the current network that is being used by the device.

- **ias_CurrentTime** The current time.

  A string can be compared against ias_CurrentTime if the hours, minutes, and seconds are separated by colons in the format hh:mm:ss:sss. A 24-hour clock is assumed unless **am** or **pm** is specified. See "time_format option [compatibility]" [*SQL Anywhere Server - Database Administration*].

- **ias_CurrentTimestamp** The current timestamp (current date and time). See "time_format option [compatibility]" [*SQL Anywhere Server - Database Administration*].

Example

For an example of how to create client store properties and use them in transmission rules, see "Using custom client message store property attributes" on page 217.

Message transmission rules

Message transmission is the action of moving messages from a client message store to a server message store, or vice versa.

Message transmission is handled by the QAnywhere Agent and the MobiLink server:

- The QAnywhere Agent is connected to the client message store. It transmits messages to and from the MobiLink server.
- The MobiLink server is connected to the server message store. It receives message transmissions from QAnywhere Agents and transmits them to other QAnywhere Agents.

Message transmission can only take place between a client message store and a server message store. A message transmission can only occur when a QAnywhere Agent is connected to a MobiLink server.

Transmission rules allow you to specify when message transmission is to occur and which messages to transmit. Delete rules allow you to specify when messages should be deleted from the message stores, if you do not want to use the default behavior.

You can specify transmission rules on the server and on the client. For more information, see:

- "Client transmission rules" on page 234
- "Server transmission rules" on page 235

Client transmission rules

Client transmission rules govern the behavior of messages going from the client to the server. Client transmission rules are handled by the QAnywhere Agent.

By default, the QAnywhere Agent uses the automatic policy. You can change and customize this behavior by specifying a transmission rules file as the transmission policy for the QAnywhere Agent.

The following partial qaagent command line shows how to specify a rules file for the QAnywhere Agent:

```
qaagent -policy myrules.txt ...
```

☞ For a complete description of how to write transmission rules, see "Rule syntax" on page 226.

☞ For more information about policies, see:

- "Determining when message transmission should occur on the client" on page 36
- "-policy option" on page 163

☞ For information about client delete rules, see "Client delete rules" on page 239.

The transmission rules file holds the following kinds of entry:

- **Rules** No more than one rule can be entered per line.

Each rule must be entered on a single line, but you can use **** as a line continuation character.

- **Comments** Comments are indicated by a line beginning with either a **#** or **;** character. Any characters on that line are ignored.

☞ For more information, see "Rule syntax" on page 226 and "Condition syntax" on page 228.

You can also use transmission rules files to determine when messages are to be deleted from the message stores.

☞ For more information, see "Message delete rules" on page 239.

You can also use the Sybase Central QAnywhere plug-in to create a QAnywhere Agent rules file.

Example

For example, the following client transmission rules file specifies that during business hours only small high priority messages should be transmitted, while outside of business hours, any message can be transmitted. This rule is automatic, which indicates that if the condition is satisfied, the message is transmitted immediately. This example demonstrates that conditions can use information derived from the message as well as other information such as the current time.

```
automatic = ( ias_ContentSize < 100000 and ias_Priority > 7 ) \
    or datepart(Weekday,Ias_CurrentDate) in ( 1, 7 ) \
    or ias_CurrentTime < '8:00:00' or ias_CurrentTime > '18:00:00'
```

Server transmission rules

Server transmission rules govern the behavior of messages going from the server to the client. Server transmission rules are handled by the MobiLink server. They apply both when you are using push notifications and when you are not using notifications.

There are several ways to set server transmission rules:

- Write a server management request to set the transmission rule.

  See "Specifying transmission rules with a server management request" on page 116.

- Use Sybase Central to set the rules.

  See "Specifying server transmission rules using Sybase Central" on page 236.

- Create a server transmission rules file and specify it when you start the MobiLink server. This method is deprecated.

  See "Specifying server transmission rules with a transmission rules file (deprecated)" on page 236.

Setting default rules

You can specify server transmission rules for a particular message store or destination alias, or you can set default rules for all clients. Every user that does not have an explicit transmission rule will use the default rule.

To set default rules, you use the special client name ianywhere.server.defaultClient.

Specifying server transmission rules using Sybase Central

You can create and edit transmission rules in Sybase Central.

♦ To specify default server transmission rules

1. Start Sybase Central:
   - ♦ Choose Start ► Programs ► SQL Anywhere 10 ► Sybase Central.
   - ♦ From Connections, choose Connect with QAnywhere 10.
   - ♦ Specify an ODBC data source name or file, and a user ID and password if required. Click OK.
2. From Server Store Tasks, click Change Properties of this Message Store.

   The Properties dialog appears.
3. Open the Transmission Rules tab and select Customize the Default Transmission Rules.
4. Click New to add a rule.

   The Rule Editor appears.
5. Add conditions either by typing them into the text field or by choosing Message Variables or Status Constants from the drop-down lists.
6. Click OK to exit.

Specifying server transmission rules with a transmission rules file (deprecated)

You can create a server transmission rules file and specify it with the ianywhere.qa.server.transmissionRulesFile property in your QAnywhere messaging properties file.

☞ For more information about the messaging properties file, see "-m option" [*MobiLink - Server Administration*].

To specify transmission rules for a particular client, precede a section of rules with the client message store ID in square brackets.

Default server transmission rules can be created that apply to all users.

To specify default transmission rules, start a section with the following line:


```
[ianywhere.server.defaultClient]
```

For new transmission rules to take effect, you must restart the MobiLink server. This only applies to transmission rules specified in a transmission rules file. Server transmission rules specified using Sybase Central or a server management request take effect immediately.

☞ For information about server delete rules, see "Server delete rules" on page 239.

Example

The following section of a server transmission rules file creates the default rule that only high priority messages should be sent:

```
[ianywhere.server.defaultClient]
auto = ias_Priority > 6
```

In the following sample server transmission rules file, the rules apply only to the client identified by the client message store ID sample_store_id.

```
[sample_store_id]
; This rule governs when messages are transmitted to the client
; store with id sample_store_id.
;
;    ias_Priority >= 7
;
; Messages with priority 7 or greater should always be
; transmitted.
;
;    ias_ContentSize < 100
;
; Small messages (messages less than 100 characters or
; bytes in size) should always be transmitted.
;
;    ias_CurrentTime < '8:00am' or ias_CurrentTime > '6:00pm'
;
; Outside of business hours messages should always be
; transmitted

auto = ias_Priority >= 7 or ias_ContentSize < 100 \
    or ias_CurrentTime < '8:00:00' or ias_CurrentTime > '18:00:00'
```

In the following example, the rules apply only to the client identified by the client message store ID qanywhere.

```
[qanywhere]
; This rule governs when messages are transmitted to the client
; store with id qanywhere.
;
;    tm_Subject not like '%non-business%'
;
; Messages with the property tm_Subject set to a value that
; includes the phrase 'non-business' should not be transmitted
;
;    ias_CurrentTime < '8:00:00' or ias_CurrentTime > '18:00:00'
;
; Outside of business hours, messages should always be
; transmitted
```



```
auto = tm_Subject not like '%non-business%' \
    or ias_CurrentTime < '8:00am' or ias_CurrentTime > '6:00pm'
```

Message delete rules

Delete rules determine the persistence of messages in the client message store and the server message store.

Client delete rules

By default, messages are deleted from the client message store when the status of the message is determined to be received, expired, cancelled, or undeliverable and the final state has been transmitted to the server message store. You may want messages to be deleted faster than that, or to hold on to messages longer. You do that by creating a delete section in your client transmission rules file. The delete section must be prefaced by **[system:delete]**.

For more information about acknowledgement, see:

- .NET: "AcknowledgementMode enumeration" on page 244
- C++: "AcknowledgementMode class" on page 392
- Java: "Interface AcknowledgementMode" on page 504

☞ For more information about client transmission rules, see "Client transmission rules" on page 234.

Following is an example of the delete rules section in a client transmission rules file:

```
[system:delete]

; This rule governs when messages are deleted from the client
; store.
;
;    start time '1:00:00' on ( 'Sunday' )
;
; Messages are deleted every Sunday at 1:00 A.M.
;
;    ias_Status >= ias_FinalState
;
; Typically, messages are deleted when they reach a final
; state: received, unreceivable, expired, or cancelled.

start time '1:00:00' on ( 'Sunday' ) = ias_Status >= ias_FinalState
```

☞ For an explanation of ias_Status, see "Rule variables" on page 231.

Server delete rules

By default, messages are deleted from the server message store when the status of the message is determined to be received, expired, cancelled, or undeliverable and the final state has been transmitted back to the message originator. You may want to keep messages longer for purposes such as auditing.

Server-side delete rules apply to all messages in the server message store.

☞ For more information about server transmission rules, see "Server transmission rules" on page 235.

☞ For an explanation of ias_Status, see "Rule variables" on page 231.

Part II. QAnywhere API Reference

This part provides reference documentation of the QAnywhere APIs.

CHAPTER 12

QAnywhere .NET API Reference

Contents

About this chapter

The iAnywhere.QAnywhere.Client namespace contains classes and enumerations for building applications that handle QAnywhere messages.

iAnywhere.QAnywhere.Client namespace (.NET)

AcknowledgementMode enumeration

Indicates how messages should be acknowledged by QAnywhere client applications.

Prototypes

' Visual Basic
Public Enum **AcknowledgementMode**

// C#
public enum **AcknowledgementMode**

Remarks

The implicit and explicit acknowledgement modes are assigned to a QAManager instance using the Open method.

For more information, see "Initializing a QAnywhere API" on page 54.

With implicit acknowledgement, messages are acknowledged as soon as they are received by a client application. With explicit acknowledgement, you must call one of the QAManager acknowledgement methods. The server propagates all status changes from client to client.

For more information, see "Receiving messages synchronously" on page 75 and "Receiving messages asynchronously" on page 76.

Member name

Member name	Description
EXPLICIT_ACKNOWLEDGE-MENT	Indicates that received messages are acknowledged using one of the QAManager interface acknowledge methods.
IMPLICIT_ACKNOWLEDGE-MENT	Indicates that all messages are acknowledged as soon as they are received by a client application. If you receive messages synchronously, messages are acknowledged as soon as the GetMessage method returns. If you receive messages asynchronously, the message is acknowledged as soon as the event handling function returns.
TRANSACTIONAL	This mode indicates that messages are only acknowledged as part of the on going transaction. This mode is automatically assigned to QA-TransactionalManager interface instances.

See also

- "QAManager interface" on page 271
- "QATransactionalManager interface" on page 337
- "QAManagerBase interface" on page 275

MessageListener delegate

MessageListener delegate definition. You pass a MessageListener to the SetMessageListener method.

Prototypes

```
' Visual Basic
Public Delegate Sub MessageListener( _
  ByVal msg As QAMessage _
)

// C#
public delegate void MessageListener(
  QAMessage msg
);
```

Parameters

- **msg** The message that was received.

MessageProperties class

Provides fields storing standard message property names.

Prototypes

```
' Visual Basic
Public Class MessageProperties

// C#
public class MessageProperties
```

Remarks

The MessageProperties class provides standard message property names. You can pass MessageProperties fields to QAMessage methods used to get and set message properties.

For more information, see "Message headers and message properties" on page 206.

See also

- "MessageProperties members" on page 245
- "QAMessage interface" on page 314

MessageProperties members

Public static fields (shared)

Member name	Description
ADAPTER field	For "system" queue messages, the network adapter that is being used to connect to the QAnywhere server.

Member name	Description
ADAPTERS field	This property name refers to a delimited list of network adapters that can be used to connect to the QAnywhere server.
DELIVERY_COUNT field	This property name refers to the number of attempts that have been made so far to deliver the message.
IP field	For "system" queue messages, the IP address of the network adapter that is being used to connect to the QAnywhere server.
MAC field	For "system" queue messages, the MAC address of the network adapter that is being used to connect to the QAnywhere server.
MSG_TYPE field	This property name refers to MessageType enumeration values associated with a QAnywhere message.
NETWORK_STATUS field	This property name refers to the state of the network connection.
ORIGINATOR field	This property name refers to the message store ID of the originator of the message.
RAS field	For "system" queue messages, the RAS entry name that is being used to connect to the QAnywhere server.
RASNAMES field	For "system" queue messages, a delimited list of RAS entry names that can be used to connect to the QAnywhere server.
STATUS field	This property name refers to the current status of the message.
STATUS_TIME field	This property name refers to the time at which the message became its current status.
TRANSMISSION_STATUS field	This property name refers to the current transmission status of the message.

ADAPTER field

For "system" queue messages, the network adapter that is being used to connect to the QAnywhere server.

Prototypes

' Visual Basic
Public Shared **ADAPTER** As String

// C#
public const string **ADAPTER**;

Remarks

The value of this field is "ias_Network.Adapter".

For more information, see “Pre-defined client message store properties” on page 215.

You can pass MessageProperties.ADAPTER in the QAMessage interfaceGetStringProperty method to access the associated property.

See also

- "MessageProperties class" on page 245
- "MessageProperties members" on page 245
- "MessageProperties class" on page 245

ADAPTERS field

This property name refers to a delimited list of network adapters that can be used to connect to the QAnywhere server.

Prototypes

' Visual Basic
Public Shared **ADAPTERS** As String

// C#
public const string **ADAPTERS**;

Remarks

It is used for system queue messages.

You can pass MessageProperties.ADAPTERS in the QAMessage interfaceGetStringProperty method to access the associated property.

For more information, see "Pre-defined client message store properties" on page 215.

See also

- "MessageProperties class" on page 245
- "MessageProperties members" on page 245
- "MessageProperties class" on page 245

DELIVERY_COUNT field

This property name refers to the number of attempts that have been made so far to deliver the message.

Prototypes

' Visual Basic
Public Shared **DELIVERY_COUNT** As String

// C#
public const string **DELIVERY_COUNT**;

Remarks

The value of this field is "ias_DeliveryCount".

You can pass MessageProperties.DELIVERY_COUNT in the QAMessage interfaceGetIntProperty method to access the associated property.

See also

- "MessageProperties class" on page 245
- "MessageProperties members" on page 245
- "MessageProperties class" on page 245

IP field

For "system" queue messages, the IP address of the network adapter that is being used to connect to the QAnywhere server.

Prototypes

' Visual Basic
Public Shared **IP** As String

// C#
public const string **IP**;

Remarks

The value of this field is "ias_Network.IP".

For more information, see "Pre-defined client message store properties" on page 215.

You can pass MessageProperties.IP in the QAMessage interfaceGetStringProperty method to access the associated property.

See also

- "MessageProperties class" on page 245
- "MessageProperties members" on page 245
- "MessageProperties class" on page 245

MAC field

For "system" queue messages, the MAC address of the network adapter that is being used to connect to the QAnywhere server.

Prototypes

' Visual Basic
Public Shared **MAC** As String

// C#
public const string **MAC**;

Remarks

The value of this field is "ias_Network.MAC".

For more information, see “Pre-defined client message store properties” on page 215.

You can pass MessageProperties.MAC in the QAMessage interfaceGetStringProperty method to access the associated property.

See also

- “MessageProperties class” on page 245
- “MessageProperties members” on page 245
- “MessageProperties class” on page 245

MSG_TYPE field

This property name refers to MessageType enumeration values associated with a QAnywhere message.

Prototypes

' Visual Basic
Public Shared **MSG_TYPE** As String

// C#
public const string **MSG_TYPE**;

Remarks

The value of this field is "ias_MessageType".

You can pass MessageProperties.MSG_TYPE in the QAMessage interfaceGetIntProperty method to access the associated property.

See also

- “MessageProperties class” on page 245
- “MessageProperties members” on page 245
- “MessageProperties class” on page 245

NETWORK_STATUS field

This property name refers to the state of the network connection.

Prototypes

' Visual Basic
Public Shared **NETWORK_STATUS** As String

// C#
public const string **NETWORK_STATUS**;

Remarks

The value is 1 if the network is accessible and 0 otherwise.

The network status is used for system queue messages (for example, network status changes).

For more information, see “Pre-defined client message store properties” on page 215.

You can pass MessageProperties.NETWORK_STATUS in the QAMessage interfaceGetIntProperty method to access the associated property.

See also

- “MessageProperties class” on page 245
- “MessageProperties members” on page 245
- “MessageProperties class” on page 245

ORIGINATOR field

This property name refers to the message store ID of the originator of the message.

Prototypes

' Visual Basic
Public Shared **ORIGINATOR** As String

// C#
public const string **ORIGINATOR**;

Remarks

The value of this field is "ias_Originator".

You can pass MessageProperties.ORIGINATOR in the QAMessage interfaceGetStringProperty method to access the associated property.

See also

- “MessageProperties class” on page 245
- “MessageProperties members” on page 245
- “MessageProperties class” on page 245

RAS field

For "system" queue messages, the RAS entry name that is being used to connect to the QAnywhere server.

Prototypes

' Visual Basic
Public Shared **RAS** As String

// C#
public const string **RAS**;

Remarks

The value of this field is "ias_Network.RAS".

For more information, see “Pre-defined client message store properties” on page 215.

You can pass MessageProperties.RAS in the QAMessage interfaceGetStringProperty method to access the associated property.

See also

- "MessageProperties class" on page 245
- "MessageProperties members" on page 245
- "MessageProperties class" on page 245

RASNAMES field

For "system" queue messages, a delimited list of RAS entry names that can be used to connect to the QAnywhere server.

Prototypes

' Visual Basic
Public Shared **RASNAMES** As String

// C#
public const string **RASNAMES**;

Remarks

The value of this field is "ias_RASNames".

For more information, see "Pre-defined client message store properties" on page 215.

You can pass MessageProperties.RASNAMES in the QAMessage interfaceGetStringProperty method to access the associated property.

See also

- "MessageProperties class" on page 245
- "MessageProperties members" on page 245
- "MessageProperties class" on page 245

STATUS field

This property name refers to the current status of the message.

Prototypes

' Visual Basic
Public Shared **STATUS** As String

// C#
public const string **STATUS**;

Remarks

For a list of property values, see the StatusCodes enumeration class.

The value of this field is "ias_Status".

You can pass MessageProperties.STATUS in the QAMessage interfaceGetIntProperty method to access the associated property.

See also

- "MessageProperties class" on page 245
- "MessageProperties members" on page 245
- "StatusCodes enumeration" on page 340
- "MessageProperties class" on page 245

STATUS_TIME field

This property name refers to the time at which the message became its current status.

Prototypes

' Visual Basic
Public Shared **STATUS_TIME** As String

// C#
public const string **STATUS_TIME**;

Remarks

It is a local time. When STATUS_TIME is passed to GetProperty method, it returns a DateTime object. The value of this field is "ias_StatusTime".

See also

- "MessageProperties class" on page 245
- "MessageProperties members" on page 245
- "GetProperty method" on page 324
- "MessageProperties class" on page 245

TRANSMISSION_STATUS field

This property name refers to the current transmission status of the message.

Prototypes

' Visual Basic
Public Shared **TRANSMISSION_STATUS** As String

// C#
public const string **TRANSMISSION_STATUS**;

Remarks

For a list of property values, see the StatusCodes enumeration class.

The value of this field is "ias_TransmissionStatus".

You can pass MessageProperties.TRANSMISSION_STATUS in the QAMessage interfaceGetIntProperty method to access the associated property.

See also

- "MessageProperties class" on page 245
- "MessageProperties members" on page 245
- "StatusCodes enumeration" on page 340
- "MessageProperties class" on page 245

MessageStoreProperties class

This class defines constant values for useful message store property names. The MessageStoreProperties class provides standard message property names. You can pass MessageProperties fields to QAManagerBase methods used to get and set pre-defined or custom message store properties. For more information, see "Client message store properties" on page 215.

Prototypes

' Visual Basic
Public Class **MessageStoreProperties**

// C#
public class **MessageStoreProperties**

MessageStoreProperties members

Public static fields (shared)

Member name	Description
MAX_DELIVERY_ATTEMPTS field	This property name refers to the maximum number of times that a message can be received, without explicit acknowledgement, before its status is set to StatusCodes enumeration.UNRECEIVABLE. The value of this field is "ias_MaxDeliveryAttempts".

Public constructors

Member name	Description
MessageStoreProperties constructor	Initializes a new instance of the MessageStoreProperties class class.

MessageStoreProperties constructor

Initializes a new instance of the MessageStoreProperties class class.

Prototypes

' Visual Basic
Public Sub **New()**


```
// C#
public MessageStoreProperties();
```

MAX_DELIVERY_ATTEMPTS field

This property name refers to the maximum number of times that a message can be received, without explicit acknowledgement, before its status is set to StatusCodes enumeration.UNRECEIVABLE. The value of this field is "ias_MaxDeliveryAttempts".

Prototypes

```
' Visual Basic
Public Shared MAX_DELIVERY_ATTEMPTS As String

// C#
public const string  MAX_DELIVERY_ATTEMPTS;
```

MessageType enumeration

Defines constant values for the MessageProperties class.MSG_TYPE message property.

Prototypes

```
' Visual Basic
Public Enum MessageType

// C#
public enum MessageType
```

Member name

Member name	Description
NETWORK_STATUS_NOTI-FICATION	Identifies a QAnywhere system message used to notify QAnywhere client applications of network status changes.
PUSH_NOTIFICATION	Identifies a QAnywhere system message used to notify QAnywhere client applications of push notifications.
REGULAR	If no message type property exists then the message type is assumed to be REGULAR.

PropertyType enumeration

QAMessage property type enumeration, corresponding naturally to the C# types.

Prototypes

```
' Visual Basic
Public Enum PropertyType
```


// C#
public enum **PropertyType**

Member name

Member name	Description
BOOLEAN	Indicates a boolean property.
BYTE	Indicates a signed byte property.
DOUBLE	Indicates a double property.
FLOAT	Indicates a float property.
INT	Indicates an int property.
LONG	Indicates an long property.
SHORT	Indicates a short property.
STRING	Indicates a string property.
UNKNOWN	Indicates an unknown property type, usually because the property is unknown.

QABinaryMessage interface

An QABinaryMessage object is used to send a message containing a stream of uninterpreted bytes. It inherits from the QAMessage interface class and adds a bytes message body. QABinaryMessage provides a variety of functions to read from and write to the bytes message body.

When the message is first created, the body of the message is in write-only mode. After a message has been sent, the client that sent it can retain and modify it without affecting the message that has been sent. The same message object can be sent multiple times.

When a message is received, the provider has called Reset method so that the message body is in read-only mode and reading of values starts from the beginning of the message body.

Prototypes

' Visual Basic
Public Interface **QABinaryMessage**

// C#
public interface **QABinaryMessage**

QABinaryMessage members

Public properties

Member name	Description
BodyLength property	Returns the size of the message body in bytes.

Public methods

Member name	Description
ReadBinary method	Reads a specified number of bytes starting from the unread portion of a QABinaryMessage instance body.
ReadBoolean method	Reads a boolean value starting from the unread portion of the QABinaryMessage instance's message body.
ReadChar method	Reads a char value starting from the unread portion of a QABinaryMessage message body.
ReadDouble method	Reads a double value starting from the unread portion of a QABinaryMessage message body.
ReadFloat method	Reads a float value starting from the unread portion of a QABinaryMessage message body.
ReadInt method	Reads an integer value starting from the unread portion of a QABinaryMessage message body.
ReadLong method	Reads a long value starting from the unread portion of a QABinaryMessage message body.
ReadSbyte method	Reads a signed byte value starting from the unread portion of a QABinaryMessage message body.
ReadShort method	Reads a short value starting from the unread portion of a QABinaryMessage message body.
ReadString method	Reads a string value starting from the unread portion of a QABinaryMessage message body.
Reset method	Resets a message so that the reading of values starts from the beginning of the message body. The Reset method also puts the QABinaryMessage message body in read-only mode.
WriteBinary method	Appends a byte array value to the QABinaryMessage instance's message body.
WriteBoolean method	Appends a boolean value to the QABinaryMessage instance's message body. The boolean is represented as a one byte value. True is represented as 1; false is represented as 0.

Member name	Description
WriteChar method	Appends a char value to the QABinaryMessage instance's message body. The char is represented as a two byte value and the high order byte is appended first.
WriteDouble method	Appends a double value to the QABinaryMessage instance's message body. The double is converted to a representive 8-byte long and higher order bytes are appended first.
WriteFloat method	Appends a float value to the QABinaryMessage instance's message body. The float parameter is converted to a representative 4-byte integer and the higher order bytes are appended first.
WriteInt method	Appends an integer value to the QABinaryMessage instance's message body. The integer parameter is represented as a 4 byte value and higher order bytes are appended first.
WriteLong method	Appends a long value to the QABinaryMessage instance's message body. The long parameter is represented using 8-bytes value and higher order bytes are appended first.
WriteSbyte method	Appends a signed byte value to the QABinaryMessage instance's message body. The signed byte is represented as a one byte value.
WriteShort method	Appends a short value to the QABinaryMessage instance's message body. The short parameter is represented as a two byte value and the higher order byte is appended first.
WriteString method	Appends a string value to the QABinaryMessage instance's message body. Note: the receving application needs to invoke ReadString method for each WriteString invocation. Note: The UTF-8 representation of the string to be written can be at most 32767 bytes.

BodyLength property

Returns the size of the message body in bytes.

Prototypes

' Visual Basic
Public Readonly Property **BodyLength** As Long

// C#
public long **BodyLength** {get;}

See also

- “QABinaryMessage interface” on page 255
- “QABinaryMessage members” on page 256
- “QABinaryMessage interface” on page 255

ReadBinary method

Reads a specified number of bytes starting from the unread portion of a QABinaryMessage instance body.

Prototypes

```
' Visual Basic
Public Function ReadBinary( _
   ByVal bytes As Byte(), _
   ByVal len As Integer _
) As Integer

// C#
public int ReadBinary(
   byte[] bytes,
   int len
);
```

Parameters

- **bytes** the byte array that will contain the read bytes.
- **len** the maximum number of bytes to read

Return value

the number of bytes read from the message body.

Exceptions

- QAException class - if there was a conversion error reading the value or if there is no more input.

See also

- "QABinaryMessage interface" on page 255
- "QABinaryMessage members" on page 256
- "QABinaryMessage interface" on page 255
- "WriteBinary method" on page 263

ReadBoolean method

Reads a boolean value starting from the unread portion of the QABinaryMessage instance's message body.

Prototypes

```
' Visual Basic
Public Function ReadBoolean() As Boolean

// C#
public bool ReadBoolean();
```

Return value

the boolean value read from the message body.

Exceptions

- QAException class - if there was a conversion error reading the value or if there is no more input.

See also

- "QABinaryMessage interface" on page 255
- "QABinaryMessage members" on page 256
- "QABinaryMessage interface" on page 255
- "WriteBoolean method" on page 264

ReadChar method

Reads a char value starting from the unread portion of a QABinaryMessage message body.

Prototypes

```
' Visual Basic
Public Function ReadChar() As Char

// C#
public char ReadChar();
```

Return value

the character value read from the message body.

Exceptions

- QAException class - if there was a conversion error reading the value or if there is no more input.

See also

- "QABinaryMessage interface" on page 255
- "QABinaryMessage members" on page 256
- "QABinaryMessage interface" on page 255
- "WriteChar method" on page 264

ReadDouble method

Reads a double value starting from the unread portion of a QABinaryMessage message body.

Prototypes

```
' Visual Basic
Public Function ReadDouble() As Double

// C#
public double ReadDouble();
```

Return value

the double value read from the message body.

Exceptions

- QAException class - if there was a conversion error reading the value or if there is no more input.

See also

- "QABinaryMessage interface" on page 255
- "QABinaryMessage members" on page 256
- "QABinaryMessage interface" on page 255
- "WriteDouble method" on page 265

ReadFloat method

Reads a float value starting from the unread portion of a QABinaryMessage message body.

Prototypes

```
' Visual Basic
Public Function ReadFloat() As Single

// C#
public float  ReadFloat();
```

Return value

the float value read from the message body.

Exceptions

- QAException class - if there was a conversion error reading the value or if there is no more input.

See also

- "QABinaryMessage interface" on page 255
- "QABinaryMessage members" on page 256
- "QABinaryMessage interface" on page 255
- "WriteFloat method" on page 265

ReadInt method

Reads an integer value starting from the unread portion of a QABinaryMessage message body.

Prototypes

```
' Visual Basic
Public Function ReadInt() As Integer

// C#
public int ReadInt();
```

Return value

the int value read from the message body.

Exceptions

- QAException class - if there was a conversion error reading the value or if there is no more input.

See also

- “QABinaryMessage interface” on page 255
- “QABinaryMessage members” on page 256
- “QABinaryMessage interface” on page 255
- “WriteInt method” on page 266

ReadLong method

Reads a long value starting from the unread portion of a QABinaryMessage message body.

Prototypes

```
' Visual Basic
Public Function ReadLong() As Long

// C#
public long ReadLong();
```

Return value

the long value read from the message body.

Exceptions

- QAException class - if there was a conversion error reading the value or if there is no more input.

See also

- “QABinaryMessage interface” on page 255
- “QABinaryMessage members” on page 256
- “QABinaryMessage interface” on page 255
- “WriteLong method” on page 266

ReadSbyte method

Reads a signed byte value starting from the unread portion of a QABinaryMessage message body.

Prototypes

```
' Visual Basic
Public Function ReadSbyte() As System.SByte

// C#
public System.Sbyte ReadSbyte();
```

Return value

the signed byte value read from the message body.

Exceptions

- QAException class - if there was a conversion error reading the value or if there is no more input.

See also

- “QABinaryMessage interface” on page 255
- “QABinaryMessage members” on page 256
- “QABinaryMessage interface” on page 255
- “WriteSbyte method” on page 267

ReadShort method

Reads a short value starting from the unread portion of a QABinaryMessage message body.

Prototypes

```
' Visual Basic
Public Function ReadShort() As Short

// C#
public short ReadShort();
```

Return value

the short value read from the message body.

Exceptions

- QAException class - if there was a conversion error reading the value or if there is no more input.

See also

- “QABinaryMessage interface” on page 255
- “QABinaryMessage members” on page 256
- “QABinaryMessage interface” on page 255
- “WriteShort method” on page 267

ReadString method

Reads a string value starting from the unread portion of a QABinaryMessage message body.

Prototypes

```
' Visual Basic
Public Function ReadString() As String

// C#
public string  ReadString();
```

Return value

the string value read from the message body.

Exceptions

- QAException class - if there was a conversion error reading the value or if there is no more input.

See also

- "QABinaryMessage interface" on page 255
- "QABinaryMessage members" on page 256
- "QABinaryMessage interface" on page 255
- "WriteString method" on page 268

Reset method

Resets a message so that the reading of values starts from the beginning of the message body. The Reset method also puts the QABinaryMessage message body in read-only mode.

Prototypes

```
' Visual Basic
Public Sub Reset()

// C#
public void Reset();
```

See also

- "QABinaryMessage interface" on page 255
- "QABinaryMessage members" on page 256
- "QABinaryMessage interface" on page 255

WriteBinary method

Appends a byte array value to the QABinaryMessage instance's message body.

Prototypes

```
' Visual Basic
Public Sub WriteBinary( _
  ByVal val As Byte(), _
  ByVal offset As Integer, _
  ByVal len As Integer _
)

// C#
public void WriteBinary(
  byte[] val,
  int offset,
  int len
);
```

Parameters

- **val** the byte array value to write to the message body.

- **len** the number of bytes to write.
- **offset** the offset within the byte array to begin writing.

See also

- "QABinaryMessage interface" on page 255
- "QABinaryMessage members" on page 256
- "QABinaryMessage interface" on page 255
- "ReadBinary method" on page 258

WriteBoolean method

Appends a boolean value to the QABinaryMessage instance's message body. The boolean is represented as a one byte value. True is represented as 1; false is represented as 0.

Prototypes

```
' Visual Basic
Public Sub WriteBoolean( _
  ByVal val As Boolean _
)

// C#
public void WriteBoolean(
  bool val
);
```

Parameters

- **val** the boolean value to write to the message body.

See also

- "QABinaryMessage interface" on page 255
- "QABinaryMessage members" on page 256
- "QABinaryMessage interface" on page 255
- "ReadBoolean method" on page 258

WriteChar method

Appends a char value to the QABinaryMessage instance's message body. The char is represented as a two byte value and the high order byte is appended first.

Prototypes

```
' Visual Basic
Public Sub WriteChar( _
  ByVal val As Char _
)

// C#
public void WriteChar(
```



```
  char val
);
```

Parameters

- **val** the char value to write to the message body.

See also

- "QABinaryMessage interface" on page 255
- "QABinaryMessage members" on page 256
- "QABinaryMessage interface" on page 255
- "ReadChar method" on page 259

WriteDouble method

Appends a double value to the QABinaryMessage instance's message body. The double is converted to a representive 8-byte long and higher order bytes are appended first.

Prototypes

```
' Visual Basic
Public Sub WriteDouble( _
  ByVal val As Double _
)

// C#
public void WriteDouble(
  double val
);
```

Parameters

- **val** the double value to write to the message body.

See also

- "QABinaryMessage interface" on page 255
- "QABinaryMessage members" on page 256
- "QABinaryMessage interface" on page 255
- "ReadDouble method" on page 259

WriteFloat method

Appends a float value to the QABinaryMessage instance's message body. The float parameter is converted to a representative 4-byte integer and the higher order bytes are appended first.

Prototypes

```
' Visual Basic
Public Sub WriteFloat( _
  ByVal val As Single _
)
```



```
// C#
public void WriteFloat(
  float  val
);
```

Parameters

- **val** the float value to write to the message body.

See also

- "QABinaryMessage interface" on page 255
- "QABinaryMessage members" on page 256
- "QABinaryMessage interface" on page 255
- "ReadFloat method" on page 260

WriteInt method

Appends an integer value to the QABinaryMessage instance's message body. The integer parameter is represented as a 4 byte value and higher order bytes are appended first.

Prototypes

```
' Visual Basic
Public Sub WriteInt( _
  ByVal val As Integer _
)

// C#
public void WriteInt(
  int val
);
```

Parameters

- **val** the int value to write to the message body.

See also

- "QABinaryMessage interface" on page 255
- "QABinaryMessage members" on page 256
- "QABinaryMessage interface" on page 255
- "ReadInt method" on page 260

WriteLong method

Appends a long value to the QABinaryMessage instance's message body. The long parameter is represented using 8-bytes value and higher order bytes are appended first.

Prototypes

```
' Visual Basic
Public Sub WriteLong( _
```



```
  ByVal val As Long _
)

// C#
public void WriteLong(
  long val
);
```

Parameters

- **val** the long value to write to the message body.

See also

- "QABinaryMessage interface" on page 255
- "QABinaryMessage members" on page 256
- "QABinaryMessage interface" on page 255
- "ReadLong method" on page 261

WriteSbyte method

Appends a signed byte value to the QABinaryMessage instance's message body. The signed byte is represented as a one byte value.

Prototypes

```
' Visual Basic
Public Sub WriteSbyte( _
  ByVal val As System.SByte _
)

// C#
public void WriteSbyte(
  System.Sbyte val
);
```

Parameters

- **val** the signed byte value to write to the message body.

See also

- "QABinaryMessage interface" on page 255
- "QABinaryMessage members" on page 256
- "QABinaryMessage interface" on page 255
- "ReadSbyte method" on page 261

WriteShort method

Appends a short value to the QABinaryMessage instance's message body. The short parameter is represented as a two byte value and the higher order byte is appended first.

Prototypes

```
' Visual Basic
Public Sub WriteShort( _
  ByVal val As Short _
)

// C#
public void WriteShort(
  short val
);
```

Parameters

- **val** the short value to write to the message body.

See also

- "QABinaryMessage interface" on page 255
- "QABinaryMessage members" on page 256
- "QABinaryMessage interface" on page 255
- "ReadShort method" on page 262

WriteString method

Appends a string value to the QABinaryMessage instance's message body. Note: the receving application needs to invoke ReadString method for each WriteString invocation. Note: The UTF-8 representation of the string to be written can be at most 32767 bytes.

Prototypes

```
' Visual Basic
Public Sub WriteString( _
  ByVal val As String _
)

// C#
public void WriteString(
  string  val
);
```

Parameters

- **val** the string value to write to the message body.

See also

- "QABinaryMessage interface" on page 255
- "QABinaryMessage members" on page 256
- "QABinaryMessage interface" on page 255
- "ReadString method" on page 262

QAException class

Encapsulates QAnywhere client application exceptions. You can use the QAException class to catch QAnywhere exceptions.

Prototypes

```
' Visual Basic
Public Class QAException
  Inherits ApplicationException

// C#
public class QAException :
  ApplicationException
```

QAException members

Public constructors

Member name	Description
QAException constructor	Create a QAException instance providing the error message text.
QAException constructor	Create a QAException instance providing the error code and the error message text.

Public properties

Member name	Description
ErrorCode property	The error code of the exception.
HelpLink (inherited from Exception)	Gets or sets a link to the help file associated with this exception.
InnerException (inherited from Exception)	Gets the System.Exception instance that caused the current exception.
Message (inherited from Exception)	Gets a message that describes the current exception.
Source (inherited from Exception)	Gets or sets the name of the application or the object that causes the error.
StackTrace (inherited from Exception)	Gets a string representation of the frames on the call stack at the time the current exception was thrown.
TargetSite (inherited from Exception)	Gets the method that throws the current exception.

Public methods

Member name	Description
GetBaseException (inherited from Exception)	When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.
GetObjectData (inherited from Exception)	When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.
ToString (inherited from Exception)	Creates and returns a string representation of the current exception.

QAException constructor

Create a QAException instance providing the error message text.

Prototypes

```
' Visual Basic
Overloads Public Sub New( _
  ByVal msg As String _
)

// C#
public QAException(
  string  msg
);
```

Parameters

- **msg** the text description of the exception.

QAException constructor

Create a QAException instance providing the error code and the error message text.

Prototypes

```
' Visual Basic
Overloads Public Sub New( _
  ByVal msg As String, _
  ByVal errCode As Integer _
)

// C#
public QAException(
  string  msg,
  int errCode
);
```


Parameters

- **msg** the text description of the exception.
- **errCode** the error code.

ErrorCode property

The error code of the exception.

Prototypes

' Visual Basic
Public Readonly Property **ErrorCode** As Integer

// C#
public int **ErrorCode** {get;}

QAManager interface

The QAManager class derives from QAManagerBase interface and manages non-transactional QAnywhere messaging operations.

Prototypes

' Visual Basic
Public Interface **QAManager**

// C#
public interface **QAManager**

Remarks

For a detailed description of derived behavior, see QAManagerBase interface.

The QAManager can be configured for implicit or explicit acknowledgement as defined in the AcknowledgementMode enumeration. To acknowledge messages as part of a transaction, use QATransactionalManager interface. Use the QAManagerFactory class to create QAManager and QATransactionalManager interface objects.

QAManager members

Public methods

Member name	Description
Acknowledge method	Acknowledges that the client application successfully received a QAnywhere message.
AcknowledgeAll method	Acknowledges that the client application successfully received QAnywhere messages. All unacknowledged messages are acknowledged.

Member name	Description
AcknowledgeUntil method	Acknowledges the given QAMessage instance and all unacknowledged messages received before the given message.
Open method	Open the QAManager with the given AcknowledgementMode enumeration value.
Recover method	Forces all unacknowledged messages into a state of unreceived.

Acknowledge method

Acknowledges that the client application successfully received a QAnywhere message.

Prototypes

```
' Visual Basic
Public Sub Acknowledge( _
  ByVal msg As QAMessage _
)

// C#
public void Acknowledge(
  QAMessage msg
);
```

Parameters

- **msg** the message to acknowledge.

Remarks

Note: when a QAMessage is acknowledged, its status property changes to StatusCodes enumeration.RECEIVED. When a QAMessage STATUS field message property changes to StatusCodes enumeration.RECEIVED, it can be deleted using the default delete rule.

For more information about delete rules, see "Message delete rules" on page 239.

Exceptions

- QAException class - Thrown if there is a problem acknowledging the message.

See also

- "QAManager interface" on page 271
- "QAManager members" on page 271
- "QAManager interface" on page 271
- "AcknowledgeUntil method" on page 273
- "AcknowledgeAll method" on page 273

AcknowledgeAll method

Acknowledges that the client application successfully received QAnywhere messages. All unacknowledged messages are acknowledged.

Prototypes

```
' Visual Basic
Public Sub AcknowledgeAll()

// C#
public void AcknowledgeAll();
```

Remarks

Note: when a QAMessage is acknowledged, its STATUS field property changes to StatusCodes enumeration.RECEIVED. When a QAMessage status changes to StatusCodes enumeration.RECEIVED, it can be deleted using the default delete rule.

For more information about delete rules, see "Message delete rules" on page 239.

Exceptions

- QAException class - Thrown if there is a problem acknowledging the messages.

See also

- "QAManager interface" on page 271
- "QAManager members" on page 271
- "QAManager interface" on page 271
- "Acknowledge method" on page 272
- "AcknowledgeUntil method" on page 273

AcknowledgeUntil method

Acknowledges the given QAMessage instance and all unacknowledged messages received before the given message.

Prototypes

```
' Visual Basic
Public Sub AcknowledgeUntil( _
  ByVal msg As QAMessage _
)

// C#
public void AcknowledgeUntil(
  QAMessage msg
);
```

Parameters

- **msg** The last message to acknowledge. All earlier unacknowledged messages are also acknowledged.

Remarks

Note: when a QAMessage is acknowledged, its MessageProperties class.STATUS property changes to StatusCodes.RECEIVED. When a QAMessage status changes to StatusCodes enumeration.RECEIVED, it can be deleted using the default delete rule.

For more information about delete rules, see "Message delete rules" on page 239.

Exceptions

- QAException class - Thrown if there is a problem acknowledging the messages.

See also

- "QAManager interface" on page 271
- "QAManager members" on page 271
- "QAManager interface" on page 271
- "Acknowledge method" on page 272
- "AcknowledgeAll method" on page 273

Open method

Open the QAManager with the given AcknowledgementMode enumeration value.

Prototypes

```
' Visual Basic
Public Sub Open( _
  ByVal mode As AcknowledgementMode _
)

// C#
public void Open(
  AcknowledgementMode mode
);
```

Parameters

- **mode** The acknowledgement mode, one of AcknowledgementMode enumeration.EXPLICIT_ACKNOWLEDGEMENT or AcknowledgementMode enumeration.IMPLICIT_ACKNOWLEDGEMENT.

Remarks

The Open method must be the first method called after creating a QAManager.

Exceptions

- QAException class - Thrown if there is a problem opening the QAManager instance.

See also

- "QAManager interface" on page 271
- "QAManager members" on page 271
- "QAManager interface" on page 271

Recover method

Forces all unacknowledged messages into a state of unreceived.

Prototypes

' Visual Basic
Public Sub **Recover()**

// C#
public void **Recover();**

Remarks

That is, these messages must be received again using GetMessage method.

Exceptions

- QAException class - Thrown if there is a problem recovering.

See also

- "QAManager interface" on page 271
- "QAManager members" on page 271
- "QAManager interface" on page 271

QAManagerBase interface

This class acts as a base class for QATransactionalManager interface and QAManager interface, which manage transactional and non-transactional messaging, respectively.

Prototypes

' Visual Basic
Public Interface **QAManagerBase**

// C#
public interface **QAManagerBase**

Remarks

Use the Start method to allow a QAManagerBase instance to listen for messages. There must be only a single instance of QAManagerBase per thread in your application.

You can use instances of this class to create and manage QAnywhere messages. Use the CreateBinaryMessage method and CreateTextMessage method to create appropriate QAMessage interface instances. QAMessage interface instances provide a variety of methods to set message content and properties.

To send QAnywhere messages, use the PutMessage method to place the addressed message in the local message store queue. The message is transmitted by the QAnywhere Agent based on its transmission policies or when you call TriggerSendReceive method.

For more information about qaagent transmission policies, see "Determining when message transmission should occur on the client" on page 36.

Messages are released from memory when you close a QAManagerBase instance using the Close method.

You can use LastError property and LastErrorMessage property to return error information when a QAException class occurs. You may also obtain the error information from the QAException class object.

QAManagerBase also provides methods to set and get message store properties.

For more information, see "Client message store properties" on page 215 and the MessageStoreProperties class.

QAManagerBase members

Public properties

Member name	Description
LastError property	The error code associated with the last excecuted QAManagerBase method.
LastErrorMessage property	The error text associated with the last executed QAManagerBase method.
Mode property	Returns the QAManager acknowledgement mode for received messages.

Public methods

Member name	Description
BrowseMessages method	Browses all available messages in the message store.
BrowseMessages method	This method is deprecated. Use the BrowseMessagesByQueue method instead.
BrowseMessagesByID method	Browses the message with the given message ID.
BrowseMessagesByQueue method	Browses the next available messages waiting that have been sent to the given address.
BrowseMessagesBySelector method	Browses messages queued in the message store that satisfy the given selector.
CancelMessage method	Cancels the message with the given message ID.
Close method	Closes the connection to the QAnywhere message system and releases any resources used by the QAManagerBase interface.
CreateBinaryMessage method	Creates a QABinaryMessage interface object.
CreateTextMessage method	Creates a QATextMessage interface object.
GetBooleanStoreProperty method	Gets a boolean value for a pre-defined or custom message store property.

Member name	Description
GetDoubleStoreProperty method	Gets a double value for a pre-defined or custom message store property.
GetFloatStoreProperty method	Gets a float value for a pre-defined or custom message store property.
GetIntStoreProperty method	Gets a int value for a pre-defined or custom message store property.
GetLongStoreProperty method	Gets a long value for a pre-defined or custom message store property.
GetMessage method	Returns the next available QAMessage interface sent to the specified address.
GetMessageBySelector method	Returns the next available QAMessage interface sent to the specified address that satisfies the given selector.
GetMessageBySelectorNoWait method	Returns the next available QAMessage interface sent to the given address that satisfies the given selector.
GetMessageBySelectorTimeout method	Returns the next available QAMessage interface sent to the given address that satisfies the given selector.
GetMessageNoWait method	Returns the next available QAMessage interface sent to the given address.
GetMessageTimeout method	Returns the next available QAMessage interface sent to the given address.
GetQueueDepth method	Returns the depth of a queue, based on a given filter.
GetQueueDepth method	Returns the total depth of all queues, based on a given filter.
GetSbyteStoreProperty method	Gets a signed byte value for a pre-defined or custom message store property.
GetShortStoreProperty method	Gets a short value for a pre-defined or custom message store property.
GetStoreProperty method	Gets a System.Object representing a message store property.
GetStorePropertyNames method	Gets an enumerator over the message store property names.
GetStringStoreProperty method	Gets a string value for a pre-defined or custom message store property.
PutMessage method	Prepares a message to send to another QAnywhere client.
PutMessageTimeToLive method	Prepares a message to send to another QAnywhere client.
SetBooleanStoreProperty method	Sets a pre-defined or custom message store property to a boolean value.
SetDoubleStoreProperty method	Sets a pre-defined or custom message store property to a double value.
SetFloatStoreProperty method	Sets a pre-defined or custom message store property to a float value.

Member name	Description
SetIntStoreProperty method	Sets a pre-defined or custom message store property to a int value.
SetLongStoreProperty method	Sets a pre-defined or custom message store property to a long value.
SetMessageListener method	Sets a MessageListener delegate delegate to recieve QAnywhere messages asynchronously.
SetMessageListenerBySelector method	Sets a MessageListener delegate delegate to recieve QAnywhere messages asynchronously, with a message selector.
SetProperty method	Allows you to set QAnywhere Manager configuration properties programmatically.
SetSbyteStoreProperty method	Sets a pre-defined or custom message store property to a sbyte value.
SetShortStoreProperty method	Sets a pre-defined or custom message store property to a short value.
SetStoreProperty method	Sets a pre-defined or custom message store property to a System.Object value.
SetStringStoreProperty method	Sets a pre-defined or custom message store property to a string value.
Start method	Starts the QAManagerBase for receiving incoming messages in message listeners. The QAManagerBase does not need to be started if there are no message listeners set, ie. if messages are received with the GetMessage methods. It is not recommended to use the GetMessage methods as well as message listeners for receiving messages, one should use one or the other of the asynchronous (message listener) or synchronous (GetMessage) models. Any calls to Start() beyond the first without an intervening Stop method call are ignored.
Stop method	Stops the QAManagerBase's reception of incoming messages.
TriggerSendReceive method	Causes a synchronization with the QAnywhere message server, uploading any messages addressed to other clients, and downloading any messages addressed to the local client.

LastError property

The error code associated with the last excecuted QAManagerBase method.

Prototypes

```
' Visual Basic
Public Readonly Property LastError As Integer

// C#
public int LastError {get;}
```

Return value

The error code.

Remarks

A value of 0 indicates no error. You can retrieve this property after catching a QAException class.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "QAException class" on page 269

LastErrorMessage property

The error text associated with the last executed QAManagerBase method.

Prototypes

```
' Visual Basic
Public Readonly Property LastErrorMessage As String

// C#
public string  LastErrorMessage {get;}
```

Remarks

This value is null if the LastError property is 0. You can retrieve this property after catching a QAException class.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "QAException class" on page 269

Mode property

Returns the QAManager acknowledgement mode for received messages.

Prototypes

```
' Visual Basic
Public Readonly Property Mode As AcknowledgementMode

// C#
public AcknowledgementMode Mode {get;}
```

Remarks

For a list of possible values, see AcknowledgementMode enumeration. AcknowledgementMode enumeration.EXPLICIT_ACKNOWLEDGEMENT and AcknowledgementMode enumeration.IMPLICIT_ACKNOWLEDGEMENT apply to QAManager instances; AcknowledgementMode enumeration.TRANSACTIONAL is the mode for QATransactionalManager instances.

BrowseMessages method

Browses all available messages in the message store.

Prototypes

```
' Visual Basic
Overloads Public Function BrowseMessages() As System.Collections.IEnumerator

// C#
public System.Collections.IEnumerator BrowseMessages();
```

Return value

An enumerator over the available messages.

Remarks

The messages are just being browsed, so they cannot be acknowledged. Because browsing messages allocates native resources, you should call the Reset() method of the enumerator when you are done with it. If it is not called, the native resources will not be freed until this QAManagerBase object is freed.

Use GetMessage method to receive messages so they can be acknowledged.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "BrowseMessagesByQueue method" on page 282
- "BrowseMessagesByID method" on page 281
- "BrowseMessages method" on page 280

BrowseMessages method

This method is deprecated. Use the BrowseMessagesByQueue method instead.

Prototypes

```
' Visual Basic
Overloads Public Function BrowseMessages( _
  ByVal address As String _
) As System.Collections.IEnumerator

// C#
public System.Collections.IEnumerator BrowseMessages(
  string  address
);
```

Parameters

- **address** The address of the messages.

Return value

An enumerator over the available messages.

Remarks

Browses the next available messages waiting that have been sent to a given address. The address parameter takes the form 'store-id\queue-name' or 'queue-name'. The messages are just being browsed, so they cannot be acknowledged.

Because browsing messages allocates native resources, you should call the Reset() method of the enumerator when you are done with it. If it is not called, the native resources will not be freed until this QAManagerBase interface object is freed.

Use GetMessage method to receive messages so they can be acknowledged.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "BrowseMessagesByQueue method" on page 282
- "BrowseMessagesByID method" on page 281
- "BrowseMessagesBySelector method" on page 282
- "BrowseMessages method" on page 280

BrowseMessagesByID method

Browses the message with the given message ID.

Prototypes

```
' Visual Basic
Public Function BrowseMessagesByID( _
   ByVal msgid As String _
) As System.Collections.IEnumerator

// C#
public System.Collections.IEnumerator BrowseMessagesByID(
   string  msgid
);
```

Parameters

- **msgid** The message id of the message.

Return value

An enumerator containing 0 or 1 messages.

Remarks

The message is just being browsed, so it cannot be acknowledged. Because browsing messages allocates native resources, you should call the Reset() method of the enumerator when you are done with it. If it is not called, the native resources will not be freed until this QAManagerBase object is freed.

Use GetMessage method to receive messages so they can be acknowledged.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "BrowseMessagesByQueue method" on page 282
- "BrowseMessages method" on page 280
- "BrowseMessages method" on page 280

BrowseMessagesByQueue method

Browses the next available messages waiting that have been sent to the given address.

Prototypes

```
' Visual Basic
Public Function BrowseMessagesByQueue( _
  ByVal address As String _
) As System.Collections.IEnumerator

// C#
public System.Collections.IEnumerator BrowseMessagesByQueue(
  string  address
);
```

Parameters

- **address** The address of the messages.

Return value

An enumerator over the available messages.

Remarks

The messages are just being browsed, so they cannot be acknowledged. Because browsing messages allocates native resources, you should call the Reset() method of the enumerator when you are done with it. If it is not called, the native resources will not be freed until this QAManagerBase object is freed.

Use GetMessage method to receive messages so they can be acknowledged.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "BrowseMessagesByID method" on page 281
- "BrowseMessages method" on page 280
- "BrowseMessages method" on page 280

BrowseMessagesBySelector method

Browses messages queued in the message store that satisfy the given selector.

Prototypes

```
' Visual Basic
Public Function BrowseMessagesBySelector( _
  ByVal selector As String _
) As System.Collections.IEnumerator

// C#
public System.Collections.IEnumerator BrowseMessagesBySelector(
  string  selector
);
```

Parameters

- **selector** The selector.

Return value

An enumerator over the available messages.

Remarks

The message is just being browsed, so it cannot be acknowledged. Because browsing messages allocates native resources, you should call the Reset() method of the enumerator when you are done with it. If it is not called, the native resources will not be freed until this QAManagerBase object is freed.

Use GetMessage method to receive messages so they can be acknowledged.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "BrowseMessagesByQueue method" on page 282
- "BrowseMessages method" on page 280
- "BrowseMessages method" on page 280
- "BrowseMessagesByID method" on page 281

CancelMessage method

Cancels the message with the given message ID.

Prototypes

```
' Visual Basic
Public Sub CancelMessage( _
  ByVal msgid As String _
)

// C#
public void CancelMessage(
  string  msgid
);
```

Parameters

- **msgid** The message ID of the message to cancel.

Remarks

CancelMessage puts a message into a cancelled state before it is transmitted. With the default delete rules of the QAnywhere Agent, cancelled messages will eventually be deleted from the message store.

CancelMessage will fail if the message is already in a final state, or if it has been transmitted to the central messaging server.

For more information about delete rules, see "Message delete rules" on page 239.

Exceptions

- QAException class - Thrown if there is a problem cancelling the message.

Close method

Closes the connection to the QAnywhere message system and releases any resources used by the QAManagerBase interface.

Prototypes

```
' Visual Basic
Public Sub Close()

// C#
public void Close();
```

Remarks

Additional calls to Close() following the first are ignored. Any subsequent calls to a QAManagerBase method, other than Close(), will result in a QAException class. You must create and open a new QAManagerBase interface instance in this case.

Exceptions

- QAException class - Thrown if there is a problem closing the QAManagerBase instance.

CreateBinaryMessage method

Creates a QABinaryMessage interface object.

Prototypes

```
' Visual Basic
Public Function CreateBinaryMessage() As QABinaryMessage

// C#
public QABinaryMessage CreateBinaryMessage();
```

Return value

A new QABinaryMessage interface instance.

Remarks

A QABinaryMessage object is used to send a message containing a message body of uninterpreted bytes.

Exceptions

- QAException class - Thrown if there is a problem creating the message.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "QABinaryMessage interface" on page 255

CreateTextMessage method

Creates a QATextMessage interface object.

Prototypes

```
' Visual Basic
Public Function CreateTextMessage() As QATextMessage

// C#
public QATextMessage CreateTextMessage();
```

Return value

A new QATextMessage interface instance.

Remarks

A QATextMessage interface object is used to send a message containing a string message body.

Exceptions

- QAException class - Thrown if there is a problem creating the message.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "QATextMessage interface" on page 335

GetBooleanStoreProperty method

Gets a boolean value for a pre-defined or custom message store property.

Prototypes

```
' Visual Basic
Public Function GetBooleanStoreProperty( _
   ByVal propName As String _
) As Boolean
```



```
// C#
public bool GetBooleanStoreProperty(
  string  propName
);
```

Parameters

- **propName** The pre-defined or custom property name.

Return value

The boolean property value.

Remarks

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class. For more information, see "Client message store properties" on page 215.

Exceptions

- QAException class - Thrown if there is a conversion error getting the property value or if the property does not exist.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "MessageStoreProperties class" on page 253

GetDoubleStoreProperty method

Gets a double value for a pre-defined or custom message store property.

Prototypes

```
' Visual Basic
Public Function GetDoubleStoreProperty( _
  ByVal propName As String _
) As Double

// C#
public double GetDoubleStoreProperty(
  string  propName
);
```

Parameters

- **propName** The pre-defined or custom property name.

Return value

The double property value.

Remarks

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

Exceptions

- QAException class - Thrown if there is a conversion error getting the property value or if the property does not exist.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "MessageStoreProperties class" on page 253

GetFloatStoreProperty method

Gets a float value for a pre-defined or custom message store property.

Prototypes

```
' Visual Basic
Public Function GetFloatStoreProperty( _
  ByVal propName As String _
) As Single

// C#
public float  GetFloatStoreProperty(
  string  propName
);
```

Parameters

- **propName** The pre-defined or custom property name.

Return value

The float property value.

Remarks

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

Exceptions

- QAException class - Thrown if there is a conversion error getting the property value or if the property does not exist.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "MessageStoreProperties class" on page 253

GetIntStoreProperty method

Gets a int value for a pre-defined or custom message store property.

Prototypes

```
' Visual Basic
Public Function GetIntStoreProperty( _
   ByVal propName As String _
) As Integer

// C#
public int GetIntStoreProperty(
   string  propName
);
```

Parameters

- **propName** The pre-defined or custom property name.

Return value

The integer property value.

Remarks

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

Exceptions

- QAException class - Thrown if there is a conversion error getting the property value or if the property does not exist.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "MessageStoreProperties class" on page 253

GetLongStoreProperty method

Gets a long value for a pre-defined or custom message store property.

Prototypes

```
' Visual Basic
Public Function GetLongStoreProperty( _
  ByVal propName As String _
) As Long

// C#
public long GetLongStoreProperty(
  string  propName
);
```

Parameters

- **propName** The pre-defined or custom property name.

Return value

The long property value.

Remarks

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215

Exceptions

- QAException class - Thrown if there is a conversion error getting the property value or if the property does not exist.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "MessageStoreProperties class" on page 253

GetMessage method

Returns the next available QAMessage interface sent to the specified address.

Prototypes

```
' Visual Basic
Public Function GetMessage( _
  ByVal address As String _
) As QAMessage

// C#
public QAMessage GetMessage(
  string  address
);
```


Parameters

- **address** Specifies the queue name used by the QAnywhere client to receive messages.

Return value

The next QAMessage interface, or null if no message is available.

Remarks

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name' or 'queue-name'.

If there is no message available, this call blocks indefinitely until a message is available. Use this method to receive messages synchronously.

For more information about receiving messages asynchronously (using a message event handler), see "Receiving messages asynchronously" on page 76.

Exceptions

- QAException class - Thrown if there is a problem getting the message.

GetMessageBySelector method

Returns the next available QAMessage interface sent to the specified address that satisfies the given selector.

Prototypes

```
' Visual Basic
Public Function GetMessageBySelector( _
   ByVal address As String, _
   ByVal selector As String _
) As QAMessage

// C#
public QAMessage GetMessageBySelector(
   string  address,
   string  selector
);
```

Parameters

- **address** This address specifies the queue name used by the QAnywhere client to receive messages.
- **selector** The selector.

Return value

The next QAMessage interface, or null if no message is available.

Remarks

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name' or 'queue-name'.

If there is no message available, this call blocks indefinitely until a message is available. Use this method to receive messages synchronously.

For more information about receiving messages asynchronously (using a message event handler), see "Receiving messages asynchronously" on page 76.

Exceptions

- QAException class - Thrown if there is a problem getting the message.

GetMessageBySelectorNoWait method

Returns the next available QAMessage interface sent to the given address that satisfies the given selector.

Prototypes

```
' Visual Basic
Public Function GetMessageBySelectorNoWait( _
   ByVal address As String, _
   ByVal selector As String _
) As QAMessage

// C#
public QAMessage GetMessageBySelectorNoWait(
   string  address,
   string  selector
);
```

Parameters

- **address** Specifies the queue name used by the QAnywhere client to receive messages.
- **selector** The selector.

Return value

The next available message or null there are no available message.

Remarks

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name' or 'queue-name'. If no message is available, this method returns immediately. Use this method to receive messages synchronously.

For more information about receiving messages asynchronously (using a message event handler), see "Receiving messages asynchronously" on page 76.

Exceptions

- QAException class - Thrown if there is a problem getting the message.

GetMessageBySelectorTimeout method

Returns the next available QAMessage interface sent to the given address that satisfies the given selector.

Prototypes

```
' Visual Basic
Public Function GetMessageBySelectorTimeout( _
  ByVal address As String, _
  ByVal selector As String, _
  ByVal timeout As Long _
) As QAMessage

// C#
public QAMessage GetMessageBySelectorTimeout(
  string  address,
  string  selector,
  long timeout
);
```

Parameters

- **address** Specifies the queue name used by the QAnywhere client to receive messages.
- **selector** The selector.
- **timeout** The time to wait, in milliseconds, for a message to become available.

Return value

The next QAMessage interface, or null if no message is available.

Remarks

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name' or 'queue-name'. If no message is available, this method waits for the specified timeout and then returns. Use this method to receive messages synchronously.

For more information about receiving messages asynchronously (using a message event handler), see "Receiving messages asynchronously" on page 76.

Exceptions

- QAException class - Thrown if there is a problem getting the message.

GetMessageNoWait method

Returns the next available QAMessage interface sent to the given address.

Prototypes

```
' Visual Basic
Public Function GetMessageNoWait( _
  ByVal address As String _
) As QAMessage

// C#
public QAMessage GetMessageNoWait(
  string  address
);
```


Parameters

- **address** this address specifies the queue name used by the QAnywhere client to receive messages.

Return value

The next available message or null there is no available message.

Remarks

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name' or 'queue-name'. If no message is available, this method returns immediately. Use this method to receive messages synchronously. For more information about receiving messages asynchronously (using a message event handler), see "Receiving messages asynchronously" on page 76.

Exceptions

- QAException class - Thrown if there is a problem getting the message.

GetMessageTimeout method

Returns the next available QAMessage interface sent to the given address.

Prototypes

```
' Visual Basic
Public Function GetMessageTimeout( _
   ByVal address As String, _
   ByVal timeout As Long _
) As QAMessage

// C#
public QAMessage GetMessageTimeout(
   string  address,
   long timeout
);
```

Parameters

- **address** Specifies the queue name used by the QAnywhere client to receive messages.
- **timeout** The time to wait, in milliseconds, for a message to become available.

Return value

The next QAMessage interface, or null if no message is available.

Remarks

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name' or 'queue-name'.

If no message is available, this method waits for the specified timeout and then returns. Use this method to receive messages synchronously.

For more information about receiving messages asynchronously (using a message event handler), see "Receiving messages asynchronously" on page 76.

Exceptions

- QAException class - Thrown if there is a problem getting the message.

GetQueueDepth method

Returns the depth of a queue, based on a given filter.

Prototypes

```
' Visual Basic
Overloads Public Function GetQueueDepth( _
  ByVal address As String, _
  ByVal filter As QueueDepthFilter _
) As Integer

// C#
public int GetQueueDepth(
  string  address,
  QueueDepthFilter filter
);
```

Parameters

- **filter** A filter indicating incoming messages, outgoing messages, or all messages.
- **address** The queue name.

Return value

The number of messages.

Remarks

The depth of the queue is the number of messages which have not been received (for example, using GetMessage method).

Exceptions

- QAException class - Thrown if there was an error.

GetQueueDepth method

Returns the total depth of all queues, based on a given filter.

Prototypes

```
' Visual Basic
Overloads Public Function GetQueueDepth( _
  ByVal filter As QueueDepthFilter _
) As Integer

// C#
public int GetQueueDepth(
```



```
  QueueDepthFilter filter
);
```

Parameters

- **filter** A filter indicating incoming messages, outgoing messages, or all messages.

Return value

The number of messages.

Remarks

The depth of the queue is the number of messages which have not been received (for example, using GetMessage method).

Exceptions

- QAException class - Thrown if there was an error.

GetSbyteStoreProperty method

Gets a signed byte value for a pre-defined or custom message store property.

Prototypes

```
' Visual Basic
Public Function GetSbyteStoreProperty( _
  ByVal propName As String _
) As System.SByte

// C#
public System.Sbyte GetSbyteStoreProperty(
  string  propName
);
```

Parameters

- **propName** The pre-defined or custom property name.

Return value

The signed byte property value.

Remarks

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see “Client message store properties” on page 215.

Exceptions

- QAException class - Thrown if there is a conversion error getting the property value or if the property does not exist.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "MessageStoreProperties class" on page 253

GetShortStoreProperty method

Gets a short value for a pre-defined or custom message store property.

Prototypes

```
' Visual Basic
Public Function GetShortStoreProperty( _
   ByVal propName As String _
) As Short

// C#
public short GetShortStoreProperty(
   string  propName
);
```

Parameters

- **propName** the pre-defined or custom property name.

Return value

The short property value.

Remarks

You can use this method to access pre-defined or user-defined client store properties. For a list of pre-defined properties, see MessageStoreProperties class. For more information, see "Client message store properties" on page 215.

Exceptions

- QAException class - Thrown if there is a conversion error getting the property value or if the property does not exist.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "MessageStoreProperties class" on page 253

GetStoreProperty method

Gets a System.Object representing a message store property.

Prototypes

```
' Visual Basic
Public Function GetStoreProperty( _
   ByVal propName As String _
) As Object

// C#
public object GetStoreProperty(
   string  propName
);
```

Parameters

- **propName** The pre-defined or custom property name.

Return value

The property value.

Remarks

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

Exceptions

- QAException class - Thrown if the property does not exist

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "MessageStoreProperties class" on page 253

GetStorePropertyNames method

Gets an enumerator over the message store property names.

Prototypes

```
' Visual Basic
Public Function GetStorePropertyNames() As System.Collections.IEnumerator

// C#
public System.Collections.IEnumerator GetStorePropertyNames();
```

Return value

An enumerator over the message store property names.

Remarks

For more information about client store properties, see "Client message store properties" on page 215.

GetStringStoreProperty method

Gets a string value for a pre-defined or custom message store property.

Prototypes

```
' Visual Basic
Public Function GetStringStoreProperty( _
  ByVal propName As String _
) As String

// C#
public string  GetStringStoreProperty(
  string  propName
);
```

Parameters

- **propName** The pre-defined or custom property name.

Return value

The string property value or null if the property does not exist.

Remarks

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "MessageStoreProperties class" on page 253

PutMessage method

Prepares a message to send to another QAnywhere client.

Prototypes

```
' Visual Basic
Public Sub PutMessage( _
  ByVal address As String, _
  ByVal msg As QAMessage _
)

// C#
public void PutMessage(
  string  address,
  QAMessage msg
);
```


Parameters

- **address** The address of the message specifying the destination queue name.
- **msg** The message to put in the local message store for transmission.

Remarks

The PutMessage method inserts a message and a destination address into your local message store. The time of message transmission depends on QAnywhere Agent transmission policies.

For more information, see "Determining when message transmission should occur on the client" on page 36.

The address takes the form 'id\queue-name', where 'id' is the destination message store ID and 'queue-name' identifies a queue that is used by the destination QAnywhere client to listen for or receive messages.

For more information about QAnywhere addresses, see "QAnywhere message addresses" on page 50.

Exceptions

- QAException class - Thrown if there is a problem putting the message.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "PutMessageTimeToLive method" on page 299

PutMessageTimeToLive method

Prepares a message to send to another QAnywhere client.

Prototypes

```
' Visual Basic
Public Sub PutMessageTimeToLive( _
   ByVal address As String, _
   ByVal msg As QAMessage, _
   ByVal ttl As Long _
)

// C#
public void PutMessageTimeToLive(
   string  address,
   QAMessage msg,
   long ttl
);
```

Parameters

- **address** The address of the message specifying the destination queue name.
- **msg** The message to put.

- **ttl** The delay, in milliseconds, before the message will expire if it has not been delivered. A value of 0 indicates the message will not expire.

Remarks

The PutMessageTimeToLive method inserts a message and a destination address into your local message store. The time of message transmission depends on QAnywhere Agent transmission policies. However, if the next message tranmission time exceeds the given time-to-live value, the message expires.

For more information, see "Determining when message transmission should occur on the client" on page 36.

The address takes the form 'id\queue-name', where 'id' is the destination message store id and 'queue-name' identifies a queue that is used by the destination QAnywhere client to listen for or receive messages.

For more information about QAnywhere addresses, see "QAnywhere message addresses" on page 50.

Exceptions

- QAException class - Thrown if there is a problem putting the message.

SetBooleanStoreProperty method

Sets a pre-defined or custom message store property to a boolean value.

Prototypes

```
' Visual Basic
Public Sub SetBooleanStoreProperty( _
  ByVal propName As String, _
  ByVal val As Boolean _
)

// C#
public void SetBooleanStoreProperty(
  string  propName,
  bool val
);
```

Parameters

- **propName** The pre-defined or custom property name.
- **val** The boolean property value.

Remarks

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See also

- "QAManagerBase interface" on page 275

- "QAManagerBase members" on page 276
- "MessageStoreProperties class" on page 253

SetDoubleStoreProperty method

Sets a pre-defined or custom message store property to a double value.

Prototypes

```
' Visual Basic
Public Sub SetDoubleStoreProperty( _
  ByVal propName As String, _
  ByVal val As Double _
)

// C#
public void SetDoubleStoreProperty(
  string  propName,
  double val
);
```

Parameters

- **propName** The pre-defined or custom property name.
- **val** The double property value.

Remarks

You can use this method to set pre-defined or user-defined client. store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "MessageStoreProperties class" on page 253

SetFloatStoreProperty method

Sets a pre-defined or custom message store property to a float value.

Prototypes

```
' Visual Basic
Public Sub SetFloatStoreProperty( _
  ByVal propName As String, _
  ByVal val As Single _
)
```



```
// C#
public void SetFloatStoreProperty(
  string  propName,
  float  val
);
```

Parameters

- **propName** The pre-defined or custom property name.
- **val** The float property value.

Remarks

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "MessageStoreProperties class" on page 253

SetIntStoreProperty method

Sets a pre-defined or custom message store property to a int value.

Prototypes

```
' Visual Basic
Public Sub SetIntStoreProperty( _
  ByVal propName As String, _
  ByVal val As Integer _
)

// C#
public void SetIntStoreProperty(
  string  propName,
  int val
);
```

Parameters

- **propName** The pre-defined or custom property name.
- **val** The int property value.

Remarks

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "MessageStoreProperties class" on page 253

SetLongStoreProperty method

Sets a pre-defined or custom message store property to a long value.

Prototypes

```
' Visual Basic
Public Sub SetLongStoreProperty( _
  ByVal propName As String, _
  ByVal val As Long _
)

// C#
public void SetLongStoreProperty(
  string  propName,
  long val
);
```

Parameters

- **propName** The pre-defined or custom property name.
- **val** The long property value

Remarks

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "MessageStoreProperties class" on page 253

SetMessageListener method

Sets a MessageListener delegate delegate to recieve QAnywhere messages asynchronously.

Prototypes

```
' Visual Basic
Public Sub SetMessageListener( _
  ByVal address As String, _
  ByVal listener As MessageListener _
)
```



```
// C#
public void SetMessageListener(
  string  address,
  MessageListener listener
);
```

Parameters

- **address** The address of messages.
- **listener** The listener to register.

Remarks

Use this method to receive message asynchronously.

MessageListener delegate accepts a single QAMessage parameter.

The SetMessageListener address parameter specifies a local queue name used to receive the message. You can only have one listener delegate assigned to a given queue.

If you want to listen for QAnywhere system messages, including push notifications and network status changes, specify "system" as the queue name.

For more information, see “Receiving messages asynchronously” on page 76.

SetMessageListenerBySelector method

Sets a MessageListener delegate delegate to recieve QAnywhere messages asynchronously, with a message selector.

Prototypes

```
' Visual Basic
Public Sub SetMessageListenerBySelector( _
  ByVal address As String, _
  ByVal selector As String, _
  ByVal listener As MessageListener _
)

// C#
public void SetMessageListenerBySelector(
  string  address,
  string  selector,
  MessageListener listener
);
```

Parameters

- **address** The address of messages.
- **listener** The listener to register.
- **selector** The selector to be used to filter the messages to be received.

Remarks

Use this method to receive message asynchronously.

MessageListener delegate accepts a single QAMessage parameter.

The SetMessageListener address parameter specifies a local queue name used to receive the message. You can only have one listener delegate assigned to a given queue. The selector parameter specifies a selector to be used to filter the messages to be received on the given address.

If you want to listen for QAnywhere system messages, including push notifications and network status changes, specify "system" as the queue name.

For more information, see "Receiving messages asynchronously" on page 76 and "System queue" on page 50.

SetProperty method

Allows you to set QAnywhere Manager configuration properties programmatically.

Prototypes

```
' Visual Basic
Public Sub SetProperty( _
   ByVal name As String, _
   ByVal val As String _
)

// C#
public void SetProperty(
   string  name,
   string  val
);
```

Parameters

- **name** The QAnywhere Manager configuration property name.
- **val** The QAnywhere Manager configuration property value

Remarks

You can use this method to override default QAnywhere Manager configuration properties by specifying a property name and value. For a list of properties, see "QAnywhere manager configuration properties" on page 62.

You can also set QAnywhere Manager configuration properties using a properties file and the CreateQAManager method.

For more information, see "Setting QAnywhere manager configuration properties in a file" on page 62. *Note:* you must set required properties before calling Open method or Open method.

Exceptions

- QAException class - Thrown if there is a problem setting the property.

SetSbyteStoreProperty method

Sets a pre-defined or custom message store property to a sbyte value.

Prototypes

```
' Visual Basic
Public Sub SetSbyteStoreProperty( _
  ByVal propName As String, _
  ByVal val As System.SByte _
)

// C#
public void SetSbyteStoreProperty(
  string  propName,
  System.Sbyte val
);
```

Parameters

- **propName** The pre-defined or custom property name.
- **val** The sbyte property value.

Remarks

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "MessageStoreProperties class" on page 253

SetShortStoreProperty method

Sets a pre-defined or custom message store property to a short value.

Prototypes

```
' Visual Basic
Public Sub SetShortStoreProperty( _
  ByVal propName As String, _
  ByVal val As Short _
)

// C#
public void SetShortStoreProperty(
  string  propName,
  short val
);
```


Parameters

- **propName** The pre-defined or custom property name.
- **val** The short property value.

Remarks

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class. For more information, see "Client message store properties" on page 215.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "MessageStoreProperties class" on page 253

SetStoreProperty method

Sets a pre-defined or custom message store property to a System.Object value.

Prototypes

```
' Visual Basic
Public Sub SetStoreProperty( _
   ByVal propName As String, _
   ByVal val As Object _
)

// C#
public void SetStoreProperty(
   string  propName,
   object val
);
```

Parameters

- **propName** The pre-defined or custom property name.
- **val** The property value.

Remarks

The property type must be one of the acceptable primitive types, or String. You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276

- "MessageStoreProperties class" on page 253

SetStringStoreProperty method

Sets a pre-defined or custom message store property to a string value.

Prototypes

```
' Visual Basic
Public Sub SetStringStoreProperty( _
  ByVal propName As String, _
  ByVal val As String _
)

// C#
public void SetStringStoreProperty(
  string  propName,
  string  val
);
```

Parameters

- **propName** The pre-defined or custom property name.
- **val** The string property value.

Remarks

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "MessageStoreProperties class" on page 253

Start method

Starts the QAManagerBase for receiving incoming messages in message listeners. The QAManagerBase does not need to be started if there are no message listeners set, ie. if messages are received with the GetMessage methods. It is not recommended to use the GetMessage methods as well as message listeners for receiving messages, one should use one or the other of the asynchronous (message listener) or synchronous (GetMessage) models. Any calls to Start() beyond the first without an intervening Stop method call are ignored.

Prototypes

```
' Visual Basic
Public Sub Start()
```



```
// C#
public void Start();
```

Exceptions

- QAException class - Thrown if there is a problem starting the QAManagerBase instance.

See also

- “QAManagerBase interface” on page 275
- “QAManagerBase members” on page 276
- “Stop method” on page 309

Stop method

Stops the QAManagerBase's reception of incoming messages.

Prototypes

```
' Visual Basic
Public Sub Stop()
```

```
// C#
public void Stop();
```

Remarks

The messages are not lost. They just won't be received until the manager is started again. Any calls to Stop () beyond the first without an intervening Start method are ignored.

Exceptions

- QAException class - Thrown if there is a problem stopping the QAManagerBase instance.

See also

- “QAManagerBase interface” on page 275
- “QAManagerBase members” on page 276
- “Start method” on page 308

TriggerSendReceive method

Causes a synchronization with the QAnywhere message server, uploading any messages addressed to other clients, and downloading any messages addressed to the local client.

Prototypes

```
' Visual Basic
Public Sub TriggerSendReceive()
```

```
// C#
public void TriggerSendReceive();
```


Remarks

QAManagerBase TriggerSendReceive results in immediate message synchronization between a QAnywhere Agent and the central messaging server. A manual TriggerSendReceive call results in immediate message transmission, independent of the QAnywhere Agent transmission policies.

QAnywhere Agent transmission policies determine how message transmission occurs. For example, message transmission can occur automatically at regular intervals, when your client receives a push notification, or when you call the PutMessage method to send a message.

For more information, see "Determining when message transmission should occur on the client" on page 36.

Exceptions

- QAException class - Thrown if there is a problem triggering the send/receive.

See also

- "QAManagerBase interface" on page 275
- "QAManagerBase members" on page 276
- "PutMessage method" on page 298

QAManagerFactory class

This class acts as a factory class for creating QATransactionalManager and QAManager objects.

Prototypes

```
' Visual Basic
MustInherit Public Class QAManagerFactory
  Inherits Component

// C#
public abstract class QAManagerFactory :
  Component
```

Remarks

You can only have one instance of QAManagerFactory.

QAManagerFactory members

Public static properties (shared)

Member name	Description
Instance property	A singleton QAManagerFactory instance.
InstanceCount property	Indicates the number of factory instances.

Public fields

Member name	Description
InstanceID field	Factory ID.

Public properties

Member name	Description
LastError property	The error code associated with the last excecuted QAManagerFactory method.
LastErrorMessage property	The error text associated with the last executed QAManagerFactory method.

Public methods

Member name	Description
CreateQAManager method	Returns a new QAManager instance with the specified properties.
CreateQATransactionalManager method	Returns a new QATransactionalManager instance with the specified properties.

InstanceID field

Factory ID.

Prototypes

' Visual Basic
Public**InstanceID** As Integer

// C#
public int **InstanceID**;

Instance property

A singleton QAManagerFactory instance.

Prototypes

' Visual Basic
Public Shared Readonly Property **Instance** As QAManagerFactory

// C#
public const QAManagerFactory **Instance** {get;}

Exceptions

- QAException class - Thrown if there is a problem creating the manager factory.

InstanceCount property

Indicates the number of factory instances.

Prototypes

' Visual Basic
Public Shared Readonly Property **InstanceCount** As Long

// C#
public const long **InstanceCount** {get;}

LastError property

The error code associated with the last excecuted QAManagerFactory method.

Prototypes

' Visual Basic
Public Readonly Property **LastError** As Integer

// C#
public int **LastError** {get;}

Return value

The error code.

Remarks

A value of 0 indicates no error. You can retrieve this property after catching a QAException class.

See also

- "QAManagerFactory class" on page 310
- "QAManagerFactory members" on page 310
- "QAException class" on page 269

LastErrorMessage property

The error text associated with the last executed QAManagerFactory method.

Prototypes

' Visual Basic
Public Readonly Property **LastErrorMessage** As String

// C#
public string **LastErrorMessage** {get;}

Return value

The error message.

Remarks

This value is null if the LastError property is 0. You can retrieve this property after catching a QAException class.

See also

- "QAManagerFactory class" on page 310
- "QAManagerFactory members" on page 310
- "QAException class" on page 269

CreateQAManager method

Returns a new QAManager instance with the specified properties.

Prototypes

```
' Visual Basic
Public Function CreateQAManager( _
   ByVal iniFile As String _
) As QAManager

// C#
public QAManager CreateQAManager(
   string  iniFile
);
```

Parameters

- **iniFile** A properties file for configuring the QAManager instance.

Return value

A new QAManager instance.

Remarks

If the properties file parameter is null, the QAManager is created using default properties. You can use the SetProperty method to set QAnywhere manager configuration properties programmatically after you create the instance.

For a list of QAnywhere manager configuration properties, see "QAnywhere manager configuration properties" on page 62.

For more information, see "Setting QAnywhere manager configuration properties in a file" on page 62.

Exceptions

- QAException class - Thrown if there is a problem creating the manager.

See also

- "QAManagerFactory class" on page 310
- "QAManagerFactory members" on page 310
- "QAManager interface" on page 271

CreateQATransactionalManager method

Returns a new QATransactionalManager instance with the specified properties.

Prototypes

```
' Visual Basic
Public Function CreateQATransactionalManager( _
  ByVal iniFile As String _
) As QATransactionalManager

// C#
public QATransactionalManager CreateQATransactionalManager(
  string  iniFile
);
```

Parameters

- **iniFile** A properties file for configuring the QATransactionalManager instance, or null to create the QATransactionalManager instance with default properties.

Return value

The configured QATransactionalManager.

Remarks

If the properties file parameter is null, the QATransactionalManager is created using default properties. You can use the SetProperty method to set QAnywhere Manager configuration properties programmatically after you create the instance.

For a list of QAnywhere Manager configuration properties, see "QAnywhere manager configuration properties" on page 62.

For more information, see "Setting QAnywhere manager configuration properties in a file" on page 62.

Exceptions

- QAException class - Thrown if there is a problem creating the manager.

See also

- "QAManagerFactory class" on page 310
- "QAManagerFactory members" on page 310
- "QATransactionalManager interface" on page 337

QAMessage interface

Provides an interface to set message properties and header fields.

Prototypes

```
' Visual Basic
Public Interface QAMessage
```



```
// C#
public interface QAMessage
```

Remarks

The derived classes QABinaryMessage interface and QATextMessage interface provide specialized methods to read and write to the message body. You can use QAMessage methods to set predefined or custom message properties.

For a list of pre-defined property names, see the MessageProperties class.

For more information about setting message properties and header fields, see "Message headers and message properties" on page 206.

QAMessage members

Public properties

Member name	Description
Address property	The destination address for the QAMessage instance.
Expiration property	Gets the message's expiration value.
InReplyToID property	The message id of the message for which this message is a reply.
MessageID property	The globally unique message id of the message.
Priority property	The priority of the message (ranging from 0 to 9).
Redelivered property	Indicates whether the message has been previously received but not acknowledged.
ReplyToAddress property	The reply to address of this message.
Timestamp property	The message timestamp.

Public methods

Member name	Description
ClearBody method	Clears the body of the message.
ClearProperties method	Clears all the properties of the message.
GetBooleanProperty method	Gets a boolean message property.
GetByteProperty method	Gets a byte message property.
GetDoubleProperty method	Gets a double message property.
GetFloatProperty method	Gets a float message property.

Member name	Description
GetIntProperty method	Gets an int message property.
GetLongProperty method	Gets a long message property.
GetProperty method	Gets a message property.
GetPropertyNames method	Gets an enumerator over the property names of the message.
GetPropertyType method	Returns the property type of the given property.
GetSbyteProperty method	Gets a signed byte message property.
GetShortProperty method	Gets a short message property.
GetStringProperty method	Gets a string message property.
PropertyExists method	Indicates whether the given property has been set for this message.
SetBooleanProperty method	Sets a boolean property.
SetByteProperty method	Sets a byte property.
SetDoubleProperty method	Sets a double property.
SetFloatProperty method	Sets a float property.
SetIntProperty method	Sets an int property.
SetLongProperty method	Sets a long property.
SetProperty method	Sets a property.
SetSbyteProperty method	Sets a signed byte property.
SetShortProperty method	Sets a short property.
SetStringProperty method	Sets a string property.

Address property

The destination address for the QAMessage instance.

Prototypes

' Visual Basic
Public Property **Address** As String

// C#
public string **Address** {get;set;}

Remarks

When a message is sent, this field is ignored. After completion of a send operation, the field holds the destination address specified in PutMessage method.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Expiration property

Gets the message's expiration value.

Prototypes

' Visual Basic
Public Readonly Property **Expiration** As Date

// C#
public DateTime **Expiration** {get;}

Remarks

When a message is sent, the Expiration header field is left unassigned. After completion of the send method, it holds the expiration time of the message.

This is a read-only property because the expiration time of a message is set by adding the time-to-live argument of QAManagerBase::PutMessageTimeToLive to the current time.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

InReplyToID property

The message id of the message for which this message is a reply.

Prototypes

' Visual Basic
Public Property **InReplyToID** As String

// C#
public string **InReplyToID** {get;set;}

Remarks

May be null.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

MessageID property

The globally unique message id of the message.

Prototypes

' Visual Basic
Public Readonly Property **MessageID** As String

// C#
public string **MessageID** {get;}

Remarks

This property is null until a message is put.

When a message is sent using PutMessage method, the MessageID is null and can be ignored. When the send method returns, it contains an assigned value.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Priority property

The priority of the message (ranging from 0 to 9).

Prototypes

' Visual Basic
Public Property **Priority** As Integer

// C#
public int **Priority** {get;set;}

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Redelivered property

Indicates whether the message has been previously received but not acknowledged.

Prototypes

' Visual Basic
Public Readonly Property **Redelivered** As Boolean

// C#
public bool **Redelivered** {get;}

Remarks

Redelivered is set by a receiving QAManager when it detects that a message being received was received before.

For example, an application receives a message using a QAManager interface opened with AcknowledgementMode enumeration.EXPLICIT_ACKNOWLEDGEMENT, and shuts down without acknowledging the message. When the application starts again and receives the same message the Redelivered header will be true.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

ReplyToAddress property

The reply to address of this message.

Prototypes

' Visual Basic
Public Property **ReplyToAddress** As String

// C#
public string **ReplyToAddress** {get;set;}

Remarks

May be null.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Timestamp property

The message timestamp.

Prototypes

' Visual Basic
Public Readonly Property **Timestamp** As Date

// C#
public DateTime **Timestamp** {get;}

Remarks

This Timestamp header field contains the time a message was created.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

ClearBody method

Clears the body of the message.

Prototypes

```
' Visual Basic
Public Sub ClearBody()

// C#
public void ClearBody();
```

ClearProperties method

Clears all the properties of the message.

Prototypes

```
' Visual Basic
Public Sub ClearProperties()

// C#
public void ClearProperties();
```

GetBooleanProperty method

Gets a boolean message property.

Prototypes

```
' Visual Basic
Public Function GetBooleanProperty( _
  ByVal propName As String _
) As Boolean

// C#
public bool GetBooleanProperty(
  string  propName
);
```

Parameters

- **propName** The property name.

Return value

The property value.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Exceptions

- QAException class - Thrown if there is a conversion error getting the property value or if the property does not exist.

See also

- "QAMessage interface" on page 314
- "QAMessage members" on page 315
- "MessageProperties class" on page 245

GetByteProperty method

Gets a byte message property.

Prototypes

```
' Visual Basic
Public Function GetByteProperty( _
   ByVal propName As String _
) As Byte

// C#
public byte GetByteProperty(
   string  propName
);
```

Parameters

- **propName** The property name.

Return value

The property value.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Exceptions

- QAException class - Thrown if there is a conversion error getting the property value or if the property does not exist.

See also

- "QAMessage interface" on page 314
- "QAMessage members" on page 315
- "MessageProperties class" on page 245

GetDoubleProperty method

Gets a double message property.

Prototypes

```
' Visual Basic
Public Function GetDoubleProperty( _
  ByVal propName As String _
) As Double

// C#
public double GetDoubleProperty(
  string  propName
);
```

Parameters

- **propName** The property name.

Return value

The property value.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Exceptions

- QAException class - Thrown if there is a conversion error getting the property value or if the property does not exist.

See also

- "QAMessage interface" on page 314
- "QAMessage members" on page 315
- "MessageProperties class" on page 245

GetFloatProperty method

Gets a float message property.

Prototypes

```
' Visual Basic
Public Function GetFloatProperty( _
  ByVal propName As String _
) As Single

// C#
public float  GetFloatProperty(
  string  propName
);
```

Parameters

- **propName** The property name.

Return value

The property value.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Exceptions

- QAException class - Thrown if there is a conversion error getting the property value or if the property does not exist.

See also

- "QAMessage interface" on page 314
- "QAMessage members" on page 315
- "MessageProperties class" on page 245

GetIntProperty method

Gets an int message property.

Prototypes

```
' Visual Basic
Public Function GetIntProperty( _
   ByVal propName As String _
) As Integer

// C#
public int GetIntProperty(
   string  propName
);
```

Parameters

- **propName** The property name.

Return value

The property value.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Exceptions

- QAException class - Thrown if there is a conversion error getting the property value or if the property does not exist.

See also

- "QAMessage interface" on page 314

- "QAMessage members" on page 315
- "MessageProperties class" on page 245

GetLongProperty method

Gets a long message property.

Prototypes

```
' Visual Basic
Public Function GetLongProperty( _
  ByVal propName As String _
) As Long

// C#
public long GetLongProperty(
  string  propName
);
```

Parameters

- **propName** The property name.

Return value

The property value.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Exceptions

- QAException class - Thrown if there is a conversion error getting the property value or if the property does not exist.

See also

- "QAMessage interface" on page 314
- "QAMessage members" on page 315
- "MessageProperties class" on page 245

GetProperty method

Gets a message property.

Prototypes

```
' Visual Basic
Public Function GetProperty( _
  ByVal propName As String _
) As Object
```

```
// C#
public object GetProperty(
   string  propName
);
```

Parameters

- **propName** The property name.

Return value

The property value.

Remarks

The property must be one of the acceptable primitive types, string, or DateTime.

Exceptions

- QAException class - Thrown if the property does not exist.

GetPropertyNames method

Gets an enumerator over the property names of the message.

Prototypes

```
' Visual Basic
Public Function GetPropertyNames() As System.Collections.IEnumerator

// C#
public System.Collections.IEnumerator GetPropertyNames();
```

Return value

An enumerator over the message property names.

GetPropertyType method

Returns the property type of the given property.

Prototypes

```
' Visual Basic
Public Function GetPropertyType( _
   ByVal propName As String _
) As PropertyType

// C#
public PropertyType GetPropertyType(
   string  propName
);
```

Parameters

- **propName** The name of the property.

Return value

The property type.

GetSbyteProperty method

Gets a signed byte message property.

Prototypes

```
' Visual Basic
Public Function GetSbyteProperty( _
   ByVal propName As String _
) As System.SByte

// C#
public System.Sbyte GetSbyteProperty(
   string  propName
);
```

Parameters

- **propName** the property name.

Return value

The property value.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Exceptions

- QAException class - Thrown if there is a conversion error getting the property value or if the property does not exist.

See also

- "QAMessage interface" on page 314
- "QAMessage members" on page 315
- "MessageProperties class" on page 245

GetShortProperty method

Gets a short message property.

Prototypes

```
' Visual Basic
Public Function GetShortProperty( _
   ByVal propName As String _
) As Short
```



```
// C#
public short GetShortProperty(
  string  propName
);
```

Parameters

- **propName** The property name.

Return value

The property value.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Exceptions

- QAException class - Thrown if there is a conversion error getting the property value or if the property does not exist.

See also

- "QAMessage interface" on page 314
- "QAMessage members" on page 315
- "MessageProperties class" on page 245

GetStringProperty method

Gets a string message property.

Prototypes

```
' Visual Basic
Public Function GetStringProperty( _
  ByVal propName As String _
) As String

// C#
public string  GetStringProperty(
  string  propName
);
```

Parameters

- **propName** The property name.

Return value

The property value or null if the property does not exist.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See also

- "QAMessage interface" on page 314
- "QAMessage members" on page 315
- "MessageProperties class" on page 245

PropertyExists method

Indicates whether the given property has been set for this message.

Prototypes

```
' Visual Basic
Public Function PropertyExists( _
  ByVal propName As String _
) As Boolean

// C#
public bool PropertyExists(
  string  propName
);
```

Parameters

- **propName** The property name.

Return value

True if the property exists.

SetBooleanProperty method

Sets a boolean property.

Prototypes

```
' Visual Basic
Public Sub SetBooleanProperty( _
  ByVal propName As String, _
  ByVal val As Boolean _
)

// C#
public void SetBooleanProperty(
  string  propName,
  bool val
);
```

Parameters

- **propName** The property name.
- **val** The property value.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See also

- "QAMessage interface" on page 314
- "QAMessage members" on page 315
- "MessageProperties class" on page 245

SetByteProperty method

Sets a byte property.

Prototypes

```
' Visual Basic
Public Sub SetByteProperty( _
  ByVal propName As String, _
  ByVal val As Byte _
)

// C#
public void SetByteProperty(
  string  propName,
  byte val
);
```

Parameters

- **propName** The property name.
- **val** The property value.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See also

- "QAMessage interface" on page 314
- "QAMessage members" on page 315
- "MessageProperties class" on page 245

SetDoubleProperty method

Sets a double property.

Prototypes

```
' Visual Basic
Public Sub SetDoubleProperty( _
  ByVal propName As String, _
```

```
  ByVal val As Double _
)

// C#
public void SetDoubleProperty(
  string  propName,
  double val
);
```

Parameters

- **propName** The property name.
- **val** The property value.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See also

- "QAMessage interface" on page 314
- "QAMessage members" on page 315
- "MessageProperties class" on page 245

SetFloatProperty method

Sets a float property.

Prototypes

```
' Visual Basic
Public Sub SetFloatProperty( _
  ByVal propName As String, _
  ByVal val As Single _
)

// C#
public void SetFloatProperty(
  string  propName,
  float  val
);
```

Parameters

- **propName** The property name.
- **val** The property value.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See also

- "QAMessage interface" on page 314
- "QAMessage members" on page 315
- "MessageProperties class" on page 245

SetIntProperty method

Sets an int property.

Prototypes

```
' Visual Basic
Public Sub SetIntProperty( _
  ByVal propName As String, _
  ByVal val As Integer _
)

// C#
public void SetIntProperty(
  string  propName,
  int val
);
```

Parameters

- **propName** The property name.
- **val** The property value.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See also

- "QAMessage interface" on page 314
- "QAMessage members" on page 315
- "MessageProperties class" on page 245

SetLongProperty method

Sets a long property.

Prototypes

```
' Visual Basic
Public Sub SetLongProperty( _
  ByVal propName As String, _
  ByVal val As Long _
)
```



```
// C#
public void SetLongProperty(
  string  propName,
  long val
);
```

Parameters

- **propName** The property name.
- **val** The property value.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See also

- "QAMessage interface" on page 314
- "QAMessage members" on page 315
- "MessageProperties class" on page 245

SetProperty method

Sets a property.

Prototypes

```
' Visual Basic
Public Sub SetProperty( _
  ByVal propName As String, _
  ByVal val As Object _
)

// C#
public void SetProperty(
  string  propName,
  object val
);
```

Parameters

- **propName** The property name.
- **val** The property value.

Remarks

The property type must be one of the acceptable primitive types, or String.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See also

- "QAMessage interface" on page 314

- "QAMessage members" on page 315
- "MessageProperties class" on page 245

SetSbyteProperty method

Sets a signed byte property.

Prototypes

```
' Visual Basic
Public Sub SetSbyteProperty( _
  ByVal propName As String, _
  ByVal val As System.SByte _
)

// C#
public void SetSbyteProperty(
  string  propName,
  System.Sbyte val
);
```

Parameters

- **propName** The property name.
- **val** The property value.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See also

- "QAMessage interface" on page 314
- "QAMessage members" on page 315
- "MessageProperties class" on page 245

SetShortProperty method

Sets a short property.

Prototypes

```
' Visual Basic
Public Sub SetShortProperty( _
  ByVal propName As String, _
  ByVal val As Short _
)

// C#
public void SetShortProperty(
  string  propName,
```

```
  short val
);
```

Parameters

- **propName** The property name.
- **val** The property value.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See also

- "QAMessage interface" on page 314
- "QAMessage members" on page 315
- "MessageProperties class" on page 245

SetStringProperty method

Sets a string property.

Prototypes

```
' Visual Basic
Public Sub SetStringProperty( _
  ByVal propName As String, _
  ByVal val As String _
)

// C#
public void SetStringProperty(
  string  propName,
  string  val
);
```

Parameters

- **propName** The property name.
- **val** The property value.

Remarks

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See also

- "QAMessage interface" on page 314
- "QAMessage members" on page 315
- "MessageProperties class" on page 245

QATextMessage interface

QATextMessage inherits from the QAMessage interface class and adds a text message body. QATextMessage provides methods to read from and write to the text message body.

Prototypes

' Visual Basic
Public Interface **QATextMessage**

// C#
public interface **QATextMessage**

Remarks

When the message is first created, the body of the message is in write-only mode. After a message has been sent, the client that sent it can retain and modify it without affecting the message that has been sent. The same message object can be sent multiple times.

When a message is received, the provider has called Reset method so that the message body is in read-only mode and reading of values starts from the beginning of the message body.

See also

- "QATextMessage members" on page 335
- "QABinaryMessage interface" on page 255

QATextMessage members

Public properties

Member name	Description
Text property	The message text.
TextLength property	The length, in characters, of the message.

Public methods

Member name	Description
ReadText method	Read unread text into the given buffer.
Reset method	Resets the text position of the message to the beginning.
WriteText method	Append text to the text of the message.

Text property

The message text.

Prototypes

```
' Visual Basic
Public Property Text As String

// C#
public string  Text {get;set;}
```

Remarks

If the message exceeds the maximum size specified by the QAManager MAX_IN_MEMORY_MESSAGE_SIZE, this property is null. In this case, use the ReadText method to read the text.

For more information about QAManager properties, see "QAnywhere manager configuration properties" on page 62.

TextLength property

The length, in characters, of the message.

Prototypes

```
' Visual Basic
Public Readonly Property TextLength As Long

// C#
public long TextLength {get;}
```

ReadText method

Read unread text into the given buffer.

Prototypes

```
' Visual Basic
Public Function ReadText( _
  ByVal buf As System.Text.StringBuilder _
) As Integer

// C#
public int ReadText(
  System.Text.string Builder buf
);
```

Parameters

- **buf** Target buffer for any read text.

Return value

The number of characters read or -1 if there are no more characters to read.

Remarks

Any additional unread text must be read by subsequent calls to this method. Text is read from the beginning of any unread text.

Reset method

Resets the text position of the message to the beginning.

Prototypes

```
' Visual Basic
Public Sub Reset()

// C#
public void Reset();
```

WriteText method

Append text to the text of the message.

Prototypes

```
' Visual Basic
Public Sub WriteText( _
  ByVal val As String _
)

// C#
public void WriteText(
  string  val
);
```

Parameters

- **val** The text to append.

QATransactionalManager interface

The QATransactionalManager class derives from QAManagerBase interface and manages transactional QAnywhere messaging operations.

Prototypes

```
' Visual Basic
Public Interface QATransactionalManager

// C#
public interface QATransactionalManager
```

Remarks

For a detailed description of derived behavior, see QAManagerBase interface.

The QATransactionalManager can only be used for transactional acknowledgement. Use the Commit method to commit all PutMessage method and GetMessage method invocations.

For more information, see "Implementing transactional messaging" on page 68.

See also

- "QATransactionalManager members" on page 338
- "QATransactionalManager interface" on page 337

QATransactionalManager members

Public methods

Member name	Description
Commit method	Commits the current transaction and begins a new transaction.
Open method	Opens a QATransactionalManager instance.
Rollback method	Rolls back the current transaction and begins a new transaction.

Commit method

Commits the current transaction and begins a new transaction.

Prototypes

```
' Visual Basic
Public Sub Commit()

// C#
public void Commit();
```

Remarks

This method commits all PutMessage method and GetMessage method invocations. *Note:* The first transaction begins with the call to Open method.

Exceptions

- QAException class - Thrown if there is a problem committing.

See also

- "QATransactionalManager interface" on page 337
- "QATransactionalManager members" on page 338
- "QATransactionalManager interface" on page 337

Open method

Opens a QATransactionalManager instance.

Prototypes

```
' Visual Basic
Public Sub Open()

// C#
public void Open();
```

Remarks

The Open method must be the first method called after creating a manager.

Exceptions

- QAException class - Thrown if there is a problem opening the manager

See also

- “QATransactionalManager interface” on page 337
- “QATransactionalManager members” on page 338
- “QATransactionalManager interface” on page 337

Rollback method

Rolls back the current transaction and begins a new transaction.

Prototypes

```
' Visual Basic
Public Sub Rollback()

// C#
public void Rollback();
```

Remarks

This method rolls back all uncommited PutMessage method and GetMessage method invocations.

Exceptions

- QAException class - Thrown if there is a problem rolling back

See also

- “QATransactionalManager interface” on page 337
- “QATransactionalManager members” on page 338
- “QATransactionalManager interface” on page 337

QueueDepthFilter enumeration

Provides queue depth filter values for GetQueueDepth method and GetQueueDepth method.

Prototypes

' Visual Basic
Public Enum **QueueDepthFilter**

// C#
public enum **QueueDepthFilter**

Member name

Member name	Description
ALL	Count both incoming and outgoing messages. System messages and expired messages are not included in any queue depth counts.
INCOMING	Count only incoming messages. An incoming message is defined as a message whose originator is different than the agent ID of the message store.
OUTGOING	Count only outgoing messages. An outgoing message is defined as a message whose originator is the agent ID of the message store, and whose destination is not the agent ID of the message store.

StatusCodes enumeration

This enumeration defines a set of codes for the status of a message.

Prototypes

' Visual Basic
Public Enum **StatusCodes**

// C#
public enum **StatusCodes**

Member name

Member name	Description
CANCELLED	The message has been cancelled. This code applies to the MessageProperties.STATUS.
EXPIRED	The message has expired because it was not received before its expiration time had passed. This code applies to the MessageProperties.STATUS.
FINAL	The message has acheived a final state. This code applies to the MessageProperties.STATUS.

Member name	Description
LOCAL	The message is addressed to the local message store and will not be transmitted to the server. This code applies to the MessageProperties.TRANSMISSION_STATUS.
PENDING	The message has been sent but not received. This code applies to the MessageProperties.STATUS.
RECEIVED	The message has been received and acknowledged by the receiver. This code applies to the MessageProperties.STATUS.
RECEIVING	The message is in the process of being received, or it was received but not acknowledged. This code applies to the MessageProperties.STATUS.
TRANSMITTED	The message has been transmitted to the server. This code applies to the MessageProperties.TRANSMISSION_STATUS.
TRANSMITTING	The message is in the process of being transmitted to the server. This code applies to the MessageProperties.TRANSMISSION_STATUS.
UNRECEIVABLE	The message has been marked as unreceivable. The message is either malformed, or there were too many failed attempts to deliver it. This code applies to the MessageProperties.STATUS.
UNTRANSMITTED	The message has not been transmitted to the server. This code applies to the MessageProperties.TRANSMISSION_STATUS.

iAnywhere.QAnywhere.WS namespace (.NET)

The iAnywhere.QAnywhere.WS namespace contains classes and enumerations for building applications that handle messaging between QAnywhere and mobile web services.

WSBase class

This is the base class for the main web service proxy class generated by the mobile web service compiler.

Prototypes

' Visual Basic
Public Class **WSBase**

// C#
public class **WSBase**

WSBase members

Public constructors

Member name	Description
WSBase constructor	Constructs a WSBase instance with the properties specified by a configuration property file.
WSBase constructor	Constructs a WSBase instance with default properties.

Public methods

Member name	Description
ClearRequestProperties method	Clears all request properties that have been set for this WSBase.
GetResult method	Gets a WSResult object that represents the results of a web service request.
GetServiceID method	Gets the service ID for this instance of WSBase.
SetListener method	Sets a listener for the results of a given web service request.
SetListener method	Sets a listener for the results of all web service requests made by this instance of WSBase.
SetProperty method	Sets a configuration property for this instance of WSBase.
SetQAManager method	Sets the QAManagerBase that is used by this web service client to do web service requests.
SetRequestProperty method	Sets a request property for webservice requests made by this WSBase.

Member name	Description
SetServiceID method	Sets a user-defined ID for this instance of WSBase.

WSBase constructor

Constructs a WSBase instance with the properties specified by a configuration property file.

Prototypes

```
' Visual Basic
Overloads Public Sub New( _
  ByVal iniFile As String _
)

// C#
public WSBase(
  string  iniFile
);
```

Parameters

- **iniFile** A file containing configuration properties.

Remarks

Valid configuration properties are:

LOG_FILE a file to which to log runtime information.

LOG_LEVEL a value between 0 and 6 that controls the verbosity of information logged, with 6 being the highest verbosity.

WS_CONNECTOR_ADDRESS the address of the web service connector in the MobiLink server.

The default WS_CONNECTOR_ADDRESS is "ianywhere.connector.webservices\\".

Exceptions

- WSException class - Thrown if there is a problem constructing the WSBase.

WSBase constructor

Constructs a WSBase instance with default properties.

Prototypes

```
' Visual Basic
Overloads Public Sub New()

// C#
public WSBase();
```


Exceptions

- WSException class - Thrown if there is a problem constructing the WSBase.

ClearRequestProperties method

Clears all request properties that have been set for this WSBase.

Prototypes

```
' Visual Basic
Public Sub ClearRequestProperties()

// C#
public void ClearRequestProperties();
```

GetResult method

Gets a WSResult object that represents the results of a web service request.

Prototypes

```
' Visual Basic
Public Function GetResult( _
   ByVal requestID As String _
) As iAnywhere.QAnywhere.WS.WSResult

// C#
public iAnywhere.QAnywhere.WS.WSResult GetResult(
   string  requestID
);
```

Parameters

- **requestID** The ID of the web service request.

Return value

A WSResult instance representing the results of the web service request.

See also

- "WSBase class" on page 342
- "WSBase members" on page 342
- "WSStatus enumeration" on page 388

GetServiceID method

Gets the service ID for this instance of WSBase.

Prototypes

```
' Visual Basic
Public Function GetServiceID() As String
```



```
// C#
public string  GetServiceID();
```

Return value

The service ID.

SetListener method

Sets a listener for the results of a given web service request.

Prototypes

```
' Visual Basic
Overloads Public Sub SetListener( _
  ByVal requestID As String, _
  ByVal listener As iAnywhere.QAnywhere.WS.WSListener _
)

// C#
public void SetListener(
  string  requestID,
  iAnywhere.QAnywhere.WS.WSListener listener
);
```

Parameters

- **requestID** The ID of the web service request to which to listen for results.
- **listener** The listener object that gets called when the result of the given web service request is available.

Remarks

Listeners are typically used to get results of the asyncXYZ methods of the service.

To remove a listener, call SetListener with null as the listener.

Note: This method replaces the listener set by any previous call to SetListener.

SetListener method

Sets a listener for the results of all web service requests made by this instance of WSBase.

Prototypes

```
' Visual Basic
Overloads Public Sub SetListener( _
  ByVal listener As iAnywhere.QAnywhere.WS.WSListener _
)

// C#
public void SetListener(
  iAnywhere.QAnywhere.WS.WSListener listener
);
```


Parameters

- **listener** The listener object that gets called when the result of a web service request is available.

Remarks

Listeners are typically used to get results of the asyncXYZ methods of the service.

To remove a listener, call SetListener with null as the listener.

Note: This method replaces the listener set by any previous call to SetListener.

SetProperty method

Sets a configuration property for this instance of WSBase.

Prototypes

```
' Visual Basic
Public Sub SetProperty( _
   ByVal property As String, _
   ByVal val As String _
)

// C#
public void SetProperty(
   string  property,
   string  val
);
```

Parameters

- **property** The property name to set.
- **val** The property value.

Remarks

Configuration properties must be set before any asynchronous or synchronous web service request is made. This method has no effect if it is called after a web service request has been made.

Valid configuration properties are:

LOG_FILE a file to which to log runtime information.

LOG_LEVEL a value between 0 and 6 that controls the verbosity of information logged, with 6 being the highest verbosity.

WS_CONNECTOR_ADDRESS the address of the web service connector in the MobiLink server. The default is: "ianywhere.connector.webservices\\".

SetQAManager method

Sets the QAManagerBase that is used by this web service client to do web service requests.

Prototypes

```
' Visual Basic
Public Sub SetQAManager( _
  ByVal mgr As QAManagerBase _
)

// C#
public void SetQAManager(
  QAManagerBase mgr
);
```

Parameters

- **mgr** The QAManagerBase to use.

Remarks

Note: If you use an EXPLICIT_ACKNOWLEDGEMENT QAManager, you can acknowledge the result of an asynchronous web service request by calling the acknowledge() method of WSResult. The result of a synchronous web service request is automatically acknowledged, even in the case of an EXPLICIT_ACKNOWLEDGEMENT QAManager. If you use an IMPLICIT_ACKNOWLEDGEMENT QAManager, the result of any web service request is acknowledged automatically.

SetRequestProperty method

Sets a request property for webservice requests made by this WSBase.

Prototypes

```
' Visual Basic
Public Sub SetRequestProperty( _
  ByVal name As String, _
  ByVal value As Object _
)

// C#
public void SetRequestProperty(
  string  name,
  object value
);
```

Parameters

- **name** The property name to set.
- **value** The property value.

Remarks

A request property is set on each QAMessage that is sent by this WSBase, until the property is cleared. A request property is cleared by setting it to a null value. The type of the message property is determined by the class of the value parameter. For example, if value is an instance of Int32, then SetIntProperty is used to set the property on the QAMessage.

SetServiceID method

Sets a user-defined ID for this instance of WSBase.

Prototypes

```
' Visual Basic
Public Sub SetServiceID( _
  ByVal serviceID As String _
)

// C#
public void SetServiceID(
  string  serviceID
);
```

Parameters

- **serviceID** The service ID.

Remarks

The service ID should be set to a value unique to this instance of WSBase. It is used internally to form a queue name for sending and receiving web service requests. Therefore, the service ID should be persisted between application sessions, in order to retrieve results of web service requests made in a previous session.

WSException class

This class represents an exception that occurred during processing of a web service request.

Prototypes

```
' Visual Basic
Public Class WSException
  Inherits Exception

// C#
public class WSException :
  Exception
```

WSException members

Public static fields (shared)

Member name	Description
WS_STATUS_HTTP_ERROR field	Error code indicating that there was an error in the web service HTTP request made by the web services connector.
WS_STATUS_HTTP_OK field	Error code indicating that the webservice HTTP request by the web services connector was successful.

Member name	Description
WS_STATUS_HTTP_RETRIES_EXCEEDED field	Error code indicating that the number of HTTP retries was exceeded the web services connector.
WS_STATUS_SOAP_PARSE_ERROR field	Error code indicating that there was an error in the web services runtime or in the webservices connector in parsing a SOAP response or request.

Public constructors

Member name	Description
WSException constructor	Constructs a new exception with the specified error message.
WSException constructor	Constructs a new exception with the specified error message and error code.
WSException constructor	Constructs a new exception.

Public properties

Member name	Description
ErrorCode property	The error code associated with this exception.
HelpLink (inherited from Exception)	Gets or sets a link to the help file associated with this exception.
InnerException (inherited from Exception)	Gets the System.Exception instance that caused the current exception.
Message (inherited from Exception)	Gets a message that describes the current exception.
Source (inherited from Exception)	Gets or sets the name of the application or the object that causes the error.
StackTrace (inherited from Exception)	Gets a string representation of the frames on the call stack at the time the current exception was thrown.
TargetSite (inherited from Exception)	Gets the method that throws the current exception.

Public methods

Member name	Description
GetBaseException (inherited from Exception)	When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

Member name	Description
GetObjectData (inherited from Exception)	When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.
ToString (inherited from Exception)	Creates and returns a string representation of the current exception.

WSException constructor

Constructs a new exception with the specified error message.

Prototypes

```
' Visual Basic
Overloads Public Sub New( _
  ByVal msg As String _
)

// C#
public WSException(
  string  msg
);
```

Parameters

- **msg** The error message.

WSException constructor

Constructs a new exception with the specified error message and error code.

Prototypes

```
' Visual Basic
Overloads Public Sub New( _
  ByVal msg As String, _
  ByVal errorCode As Integer _
)

// C#
public WSException(
  string  msg,
  int errorCode
);
```

Parameters

- **msg** The error message.
- **errorCode** The error code.

WSException constructor

Constructs a new exception.

Prototypes

```
' Visual Basic
Overloads Public Sub New( _
  ByVal ex As System.Exception _
)

// C#
public WSException(
  System.Exception ex
);
```

Parameters

- **ex** The exception.

WS_STATUS_HTTP_ERROR field

Error code indicating that there was an error in the web service HTTP request made by the web services connector.

Prototypes

```
' Visual Basic
Public Shared WS_STATUS_HTTP_ERROR As Integer

// C#
public const int WS_STATUS_HTTP_ERROR;
```

WS_STATUS_HTTP_OK field

Error code indicating that the webservice HTTP request by the web services connector was successful.

Prototypes

```
' Visual Basic
Public Shared WS_STATUS_HTTP_OK As Integer

// C#
public const int WS_STATUS_HTTP_OK;
```

WS_STATUS_HTTP_RETRIES_EXCEEDED field

Error code indicating that the number of HTTP retries was exceeded the web services connector.

Prototypes

```
' Visual Basic
Public Shared WS_STATUS_HTTP_RETRIES_EXCEEDED As Integer
```



```
// C#
public const int WS_STATUS_HTTP_RETRIES_EXCEEDED;
```

WS_STATUS_SOAP_PARSE_ERROR field

Error code indicating that there was an error in the web services runtime or in the webservices connector in parsing a SOAP response or request.

Prototypes

```
' Visual Basic
Public Shared WS_STATUS_SOAP_PARSE_ERROR As Integer

// C#
public const int WS_STATUS_SOAP_PARSE_ERROR;
```

ErrorCode property

The error code associated with this exception.

Prototypes

```
' Visual Basic
Public Property ErrorCode As Integer

// C#
public int ErrorCode {get;set;}
```

WSFaultException class

This class represents a SOAP Fault exception from the web service connector.

Prototypes

```
' Visual Basic
Public Class WSFaultException
  Inherits WSException

// C#
public class WSFaultException :
  WSException
```

WSFaultException members

Public constructors

Member name	Description
WSFaultException constructor	Constructs a new exception with the specified error message.

Public properties

Member name	Description
ErrorCode property (inherited from WSException)	The error code associated with this exception.
HelpLink (inherited from Exception)	Gets or sets a link to the help file associated with this exception.
InnerException (inherited from Exception)	Gets the System.Exception instance that caused the current exception.
Message (inherited from Exception)	Gets a message that describes the current exception.
Source (inherited from Exception)	Gets or sets the name of the application or the object that causes the error.
StackTrace (inherited from Exception)	Gets a string representation of the frames on the call stack at the time the current exception was thrown.
TargetSite (inherited from Exception)	Gets the method that throws the current exception.

Public methods

Member name	Description
GetBaseException (inherited from Exception)	When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.
GetObjectData (inherited from Exception)	When overridden in a derived class, sets the System.Runtime.Serialization.SerializationInfo with information about the exception.
ToString (inherited from Exception)	Creates and returns a string representation of the current exception.

WSFaultException constructor

Constructs a new exception with the specified error message.

Prototypes

```
' Visual Basic
Public Sub New( _
   ByVal msg As String _
)

// C#
public WSFaultException(
   string  msg
);
```


Parameters

- **msg** The error message.

WSListener interface

This class represents a listener for results of web service requests.

Prototypes

```
' Visual Basic
Public Interface WSListener
```

```
// C#
public interface WSListener
```

WSListener members

Public methods

Member name	Description
OnException method	Called when an exception occurs during processing of the result of an asynchronous web service request.
OnResult method	Called with the result of an asynchronous web service request.

OnException method

Called when an exception occurs during processing of the result of an asynchronous web service request.

Prototypes

```
' Visual Basic
Public Sub OnException( _
  ByVal e As iAnywhere.QAnywhere.WS.WSException, _
  ByVal wsResult As iAnywhere.QAnywhere.WS.WSResult _
)
```

```
// C#
public void OnException(
  iAnywhere.QAnywhere.WS.WSException e,
  iAnywhere.QAnywhere.WS.WSResult wsResult
);
```

Parameters

- **e** The WSException that occurred during processing of the result.
- **wsResult** A WSResult, from which the request ID may be obtained. Values of this WSResult are not defined.

OnResult method

Called with the result of an asynchronous web service request.

Prototypes

```
' Visual Basic
Public Sub OnResult( _
   ByVal wsResult As iAnywhere.QAnywhere.WS.WSResult _
)

// C#
public void OnResult(
   iAnywhere.QAnywhere.WS.WSResult wsResult
);
```

Parameters

- **wsResult** The WSResult describing the result of a web service request.

WSResult class

This class represents the results of a web service request.

Prototypes

```
' Visual Basic
Public Class WSResult

// C#
public class WSResult
```

Remarks

A WSResult object is obtained in one of three ways:

- It is passed to the WSListener.onResult.

- It is returned by an asyncXYZ method of the service proxy generated by the compiler.

- It is obtained by calling WSBase.getResult with a specific request ID.

WSResult members

Public methods

Member name	Description
Acknowledge method	Acknowledges that this WSResult has been processed.
GetArrayValue method	Gets an array of complex types value from this WSResult.
GetBoolArrayValue method	Gets an array of bool values from this WSResult.

Member name	Description
GetBooleanArrayValue method	Gets an array of Boolean values from this WSResult.
GetBooleanValue method	Gets a Boolean value from this WSResult.
GetBoolValue method	Gets a bool value from this WSResult.
GetByteArrayValue method	Gets an array of byte values from this WSResult.
GetByteValue method	Gets a byte value from this WSResult.
GetCharArrayValue method	Gets an array of char values from this WSResult.
GetCharValue method	Gets a char value from this WSResult.
GetDecimalArrayValue method	Gets an array of decimal values from this WSResult.
GetDecimalValue method	Gets a decimal value from this WSResult.
GetDoubleArrayValue method	Gets an array of double values from this WSResult.
GetDoubleValue method	Gets a double value from this WSResult.
GetErrorMessage method	Gets the error message.
GetFloatArrayValue method	Gets an array of float values from this WSResult.
GetFloatValue method	Gets a float value from this WSResult.
GetInt16ArrayValue method	Gets an array of Int16 values from this WSResult.
GetInt16Value method	Gets an Int16 value from this WSResult.
GetInt32ArrayValue method	Gets an array of Int32 values from this WSResult.
GetInt32Value method	Gets an Int32 value from this WSResult.
GetInt64ArrayValue method	Gets an array of Int64 values from this WSResult.
GetInt64Value method	Gets an Int64 value from this WSResult.
GetIntArrayValue method	Gets an array of int values from this WSResult.
GetIntValue method	Gets an int value from this WSResult.
GetLongArrayValue method	Gets an array of long values from this WSResult.
GetLongValue method	Gets a long value from this WSResult.
GetNullableBoolArrayValue method	Gets an array of bool values from this WSResult.
GetNullableBoolValue method	Gets a bool value from this WSResult.

Member name	Description
GetNullableDoubleArrayValue method	Gets an array of double values from this WSResult.
GetNullableDoubleValue method	Gets a double value from this WSResult.
GetNullableFloatArrayValue method	Gets an array of float values from this WSResult.
GetNullableFloatValue method	Gets a float value from this WSResult.
GetNullableIntArrayValue method	Gets an array of int values from this WSResult.
GetNullableIntValue method	Gets an int value from this WSResult.
GetNullableLongArrayValue method	Gets an array of long values from this WSResult.
GetNullableLongValue method	Gets an Int64 value from this WSResult.
GetNullableSByteArrayValue method	Gets an array of byte values from this WSResult.
GetNullableSByteValue method	Gets a byte value from this WSResult.
GetNullableShortArrayValue method	Gets an array of short values from this WSResult.
GetNullableShortValue method	Gets a short value from this WSResult.
GetObjectArrayValue method	Gets an array of Object values from this WSResult.
GetObjectValue method	Gets an object value from this WSResult.
GetRequestID method	Gets the request ID that this WSResult represents.
GetSByteArrayValue method	Gets an array of sbyte values from this WSResult.
GetSByteValue method	Gets an sbyte value from this WSResult.
GetSByteValue method	Gets an sbyte value from this WSResult.
GetShortArrayValue method	Gets an array of short values from this WSResult.
GetShortValue method	Gets a short value from this WSResult.
GetSingleArrayValue method	Gets an array of Single values from this WSResult.
GetSingleValue method	Gets a Single value from this WSResult.
GetStatus method	Gets the status of this WSResult.

Member name	Description
GetStringArrayValue method	Gets an array of string values from this WSResult.
GetStringValue method	Gets a string value from this WSResult.
GetUIntArrayValue method	Gets an array of unsigned int values from this WSResult.
GetUIntValue method	Gets a unsigned int value from this WSResult.
GetULongArrayValue method	Gets an array of unsigned long values from this WSResult.
GetULongValue method	Gets a unsigned long value from this WSResult.
GetUShortArrayValue method	Gets an array of unsigned short values from this WSResult.
GetUShortValue method	Gets a unsigned short value from this WSResult.
GetValue method	Gets the value of a complex type from this WSResult.
SetLogger method	Turns debug on or off.

Acknowledge method

Acknowledges that this WSResult has been processed.

Prototypes

```
' Visual Basic
Public Sub Acknowledge()

// C#
public void Acknowledge();
```

Remarks

This method is only useful when an EXPLICIT_ACKNOWLEDGEMENT QAManager is being used.

GetArrayValue method

Gets an array of complex types value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetArrayValue( _
   ByVal parentName As String _
) As iAnywhere.QAnywhere.WS.WSSerializable()

// C#
public iAnywhere.QAnywhere.WS.WSSerializable[] GetArrayValue(
   string  parentName
);
```

Parameters

- **parentName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetBoolArrayValue method

Gets an array of bool values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetBoolArrayValue( _
   ByVal elementName As String _
) As Boolean()

// C#
public bool[] GetBoolArrayValue(
   string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetBooleanArrayValue method

Gets an array of Boolean values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetBooleanArrayValue( _
   ByVal elementName As String _
) As Boolean()

// C#
public bool[] GetBooleanArrayValue(
   string  elementName
);
```


Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetBooleanValue method

Gets a Boolean value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetBooleanValue( _
  ByVal childName As String _
) As Boolean

// C#
public bool GetBooleanValue(
  string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetBoolValue method

Gets a bool value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetBoolValue( _
  ByVal childName As String _
) As Boolean

// C#
public bool GetBoolValue(
  string  childName
);
```


Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetByteArrayValue method

Gets an array of byte values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetByteArrayValue( _
   ByVal elementName As String _
) As Byte()

// C#
public byte[] GetByteArrayValue(
   string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetByteValue method

Gets a byte value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetByteValue( _
   ByVal childName As String _
) As Byte

// C#
public byte GetByteValue(
   string  childName
);
```


Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetCharArrayValue method

Gets an array of char values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetCharArrayValue( _
  ByVal elementName As String _
) As Char()

// C#
public char[] GetCharArrayValue(
  string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetCharValue method

Gets a char value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetCharValue( _
  ByVal childName As String _
) As Char

// C#
public char GetCharValue(
  string  childName
);
```


Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetDecimalArrayValue method

Gets an array of decimal values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetDecimalArrayValue( _
  ByVal elementName As String _
) As Decimal()

// C#
public decimal[] GetDecimalArrayValue(
  string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetDecimalValue method

Gets a decimal value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetDecimalValue( _
  ByVal childName As String _
) As Decimal

// C#
public decimal GetDecimalValue(
  string  childName
);
```


Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetDoubleArrayValue method

Gets an array of double values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetDoubleArrayValue( _
  ByVal elementName As String _
) As Double()

// C#
public double[] GetDoubleArrayValue(
  string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetDoubleValue method

Gets a double value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetDoubleValue( _
  ByVal childName As String _
) As Double

// C#
public double GetDoubleValue(
  string  childName
);
```


Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetErrorMessage method

Gets the error message.

Prototypes

```
' Visual Basic
Public Function GetErrorMessage() As String

// C#
public string  GetErrorMessage();
```

Return value

The error message.

GetFloatArrayValue method

Gets an array of float values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetFloatArrayValue( _
   ByVal elementName As String _
) As Single()

// C#
public float [] GetFloatArrayValue(
   string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetFloatValue method

Gets a float value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetFloatValue( _
  ByVal childName As String _
) As Single

// C#
public float  GetFloatValue(
  string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetInt16ArrayValue method

Gets an array of Int16 values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetInt16ArrayValue( _
  ByVal elementName As String _
) As Short()

// C#
public short[] GetInt16ArrayValue(
  string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetInt16Value method

Gets an Int16 value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetInt16Value( _
   ByVal childName As String _
) As Short

// C#
public short GetInt16Value(
   string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetInt32ArrayValue method

Gets an array of Int32 values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetInt32ArrayValue( _
   ByVal elementName As String _
) As Integer()

// C#
public int[] GetInt32ArrayValue(
   string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetInt32Value method

Gets an Int32 value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetInt32Value( _
  ByVal childName As String _
) As Integer

// C#
public int GetInt32Value(
  string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetInt64ArrayValue method

Gets an array of Int64 values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetInt64ArrayValue( _
  ByVal elementName As String _
) As Long()

// C#
public long[] GetInt64ArrayValue(
  string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetInt64Value method

Gets an Int64 value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetInt64Value( _
   ByVal childName As String _
) As Long

// C#
public long GetInt64Value(
   string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetIntArrayValue method

Gets an array of int values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetIntArrayValue( _
   ByVal elementName As String _
) As Integer()

// C#
public int[] GetIntArrayValue(
   string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetIntValue method

Gets an int value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetIntValue( _
  ByVal childName As String _
) As Integer

// C#
public int GetIntValue(
  string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetLongArrayValue method

Gets an array of long values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetLongArrayValue( _
  ByVal elementName As String _
) As Long()

// C#
public long[] GetLongArrayValue(
  string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetLongValue method

Gets a long value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetLongValue( _
   ByVal childName As String _
) As Long

// C#
public long GetLongValue(
   string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetNullableBoolArrayValue method

Gets an array of bool values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetNullableBoolArrayValue( _
   ByVal elementName As String _
) As iAnywhere.QAnywhere.WS.NullableBool()

// C#
public iAnywhere.QAnywhere.WS.NullableBool[] GetNullableBoolArrayValue(
   string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetNullableBoolValue method

Gets a bool value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetNullableBoolValue( _
  ByVal childName As String _
) As iAnywhere.QAnywhere.WS.NullableBool

// C#
public iAnywhere.QAnywhere.WS.NullableBool GetNullableBoolValue(
  string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetNullableDoubleArrayValue method

Gets an array of double values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetNullableDoubleArrayValue( _
  ByVal elementName As String _
) As iAnywhere.QAnywhere.WS.NullableDouble()

// C#
public iAnywhere.QAnywhere.WS.Nullabledouble[] GetNullableDoubleArrayValue(
  string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetNullableDoubleValue method

Gets a double value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetNullableDoubleValue( _
  ByVal childName As String _
) As iAnywhere.QAnywhere.WS.NullableDouble

// C#
public iAnywhere.QAnywhere.WS.Nullabledouble GetNullableDoubleValue(
  string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetNullableFloatArrayValue method

Gets an array of float values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetNullableFloatArrayValue( _
  ByVal elementName As String _
) As iAnywhere.QAnywhere.WS.NullableFloat()

// C#
public iAnywhere.QAnywhere.WS.NullableFloat[] GetNullableFloatArrayValue(
  string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetNullableFloatValue method

Gets a float value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetNullableFloatValue( _
  ByVal childName As String _
) As iAnywhere.QAnywhere.WS.NullableFloat

// C#
public iAnywhere.QAnywhere.WS.NullableFloat GetNullableFloatValue(
  string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetNullableIntArrayValue method

Gets an array of int values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetNullableIntArrayValue( _
  ByVal elementName As String _
) As iAnywhere.QAnywhere.WS.NullableInt()

// C#
public iAnywhere.QAnywhere.WS.NullableInt[] GetNullableIntArrayValue(
  string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetNullableIntValue method

Gets an int value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetNullableIntValue( _
   ByVal childName As String _
) As iAnywhere.QAnywhere.WS.NullableInt

// C#
public iAnywhere.QAnywhere.WS.NullableInt GetNullableIntValue(
   string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetNullableLongArrayValue method

Gets an array of long values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetNullableLongArrayValue( _
   ByVal elementName As String _
) As iAnywhere.QAnywhere.WS.NullableLong()

// C#
public iAnywhere.QAnywhere.WS.NullableLong[] GetNullableLongArrayValue(
   string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetNullableLongValue method

Gets an Int64 value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetNullableLongValue( _
  ByVal childName As String _
) As iAnywhere.QAnywhere.WS.NullableLong

// C#
public iAnywhere.QAnywhere.WS.NullableLong GetNullableLongValue(
  string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetNullableSByteArrayValue method

Gets an array of byte values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetNullableSByteArrayValue( _
  ByVal elementName As String _
) As iAnywhere.QAnywhere.WS.NullableSByte()

// C#
public iAnywhere.QAnywhere.WS.NullableSbyte[] GetNullableSByteArrayValue(
  string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetNullableSByteValue method

Gets a byte value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetNullableSByteValue( _
  ByVal childName As String _
) As iAnywhere.QAnywhere.WS.NullableSByte

// C#
public iAnywhere.QAnywhere.WS.NullableSbyte GetNullableSByteValue(
  string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetNullableShortArrayValue method

Gets an array of short values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetNullableShortArrayValue( _
  ByVal elementName As String _
) As iAnywhere.QAnywhere.WS.NullableShort()

// C#
public iAnywhere.QAnywhere.WS.NullableShort[] GetNullableShortArrayValue(
  string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetNullableShortValue method

Gets a short value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetNullableShortValue( _
  ByVal childName As String _
) As iAnywhere.QAnywhere.WS.NullableShort

// C#
public iAnywhere.QAnywhere.WS.NullableShort GetNullableShortValue(
  string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetObjectArrayValue method

Gets an array of Object values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetObjectArrayValue( _
  ByVal elementName As String _
) As Object()

// C#
public object[] GetObjectArrayValue(
  string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetObjectValue method

Gets an object value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetObjectValue( _
   ByVal childName As String _
) As Object

// C#
public object GetObjectValue(
   string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetRequestID method

Gets the request ID that this WSResult represents.

Prototypes

```
' Visual Basic
Public Function GetRequestID() As String

// C#
public string  GetRequestID();
```

Return value

The request ID.

Remarks

This request ID should be persisted between runs of the application if it is desired to obtain a WSResult corresponding to a web service request in a run of the application different from when the request was made.

GetSByteArrayValue method

Gets an array of sbyte values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetSByteArrayValue( _
  ByVal elementName As String _
) As System.SByte()

// C#
public System.Sbyte[] GetSByteArrayValue(
  string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetSByteValue method

Gets an sbyte value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetSByteValue( _
  ByVal childName As String _
) As System.SByte

// C#
public System.Sbyte GetSByteValue(
  string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetSbyteValue method

Gets an sbyte value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetSbyteValue( _
   ByVal childName As String _
) As System.SByte

// C#
public System.Sbyte GetSbyteValue(
   string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetShortArrayValue method

Gets an array of short values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetShortArrayValue( _
   ByVal elementName As String _
) As Short()

// C#
public short[] GetShortArrayValue(
   string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetShortValue method

Gets a short value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetShortValue( _
  ByVal childName As String _
) As Short

// C#
public short GetShortValue(
  string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetSingleArrayValue method

Gets an array of Single values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetSingleArrayValue( _
  ByVal elementName As String _
) As Single()

// C#
public float [] GetSingleArrayValue(
  string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetSingleValue method

Gets a Single value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetSingleValue( _
   ByVal childName As String _
) As Single

// C#
public float  GetSingleValue(
   string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetStatus method

Gets the status of this WSResult.

Prototypes

```
' Visual Basic
Public Function GetStatus() As iAnywhere.QAnywhere.WS.WSStatus

// C#
public iAnywhere.QAnywhere.WS.WSStatus GetStatus();
```

Return value

The status code.

See also

- “WSResult class” on page 355
- “WSResult members” on page 355
- “WSStatus enumeration” on page 388

GetStringArrayValue method

Gets an array of string values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetStringArrayValue( _
   ByVal elementName As String _
) As String()
```



```
// C#
public string [] GetStringArrayValue(
  string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetStringValue method

Gets a string value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetStringValue( _
  ByVal childName As String _
) As String

// C#
public string  GetStringValue(
  string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetUIntArrayValue method

Gets an array of unsigned int values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetUIntArrayValue( _
  ByVal elementName As String _
) As UInt32()
```



```
// C#
public uint[] GetUIntArrayValue(
   string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetUIntValue method

Gets a unsigned int value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetUIntValue( _
   ByVal childName As String _
) As UInt32

// C#
public uint GetUIntValue(
   string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetULongArrayValue method

Gets an array of unsigned long values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetULongArrayValue( _
   ByVal elementName As String _
) As UInt64()
```



```
// C#
public ulong[] GetULongArrayValue(
  string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetULongValue method

Gets a unsigned long value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetULongValue( _
  ByVal childName As String _
) As UInt64

// C#
public ulong GetULongValue(
  string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetUShortArrayValue method

Gets an array of unsigned short values from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetUShortArrayValue( _
  ByVal elementName As String _
) As UInt16()
```



```
// C#
public ushort[] GetUShortArrayValue(
   string  elementName
);
```

Parameters

- **elementName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetUShortValue method

Gets a unsigned short value from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetUShortValue( _
   ByVal childName As String _
) As UInt16

// C#
public ushort GetUShortValue(
   string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

GetValue method

Gets the value of a complex type from this WSResult.

Prototypes

```
' Visual Basic
Public Function GetValue( _
   ByVal childName As String _
) As Object
```



```
// C#
public object GetValue(
  string  childName
);
```

Parameters

- **childName** The element name in the WSDL document of this value.

Return value

The value.

Exceptions

- WSException class - Thrown if there is a problem getting the value.

SetLogger method

Turns debug on or off.

Prototypes

```
' Visual Basic
Public Sub SetLogger( _
  ByVal wsLogger As iAnywhere.QAnywhere.WS.WSLogger _
)

// C#
public void SetLogger(
  iAnywhere.QAnywhere.WS.WSLogger wsLogger
);
```

WSStatus enumeration

This class defines codes for the status of a web service. request.

Prototypes

```
' Visual Basic
Public Enum WSStatus

// C#
public enum WSStatus
```

Member name

Member name	Description
STATUS_ERROR	There was an error processing the request.
STATUS_QUEUED	The request has been queued for delivery to the server.

Member name	Description
STATUS_RESULT_AVAIL-ABLE	The result of the request is available.
STATUS_SUCCESS	The request was successful.

CHAPTER 13

QAnywhere C++ API Reference

Contents

About this chapter

This chapter describes the QAnywhere C++ API.

AcknowledgementMode class

Synopsis

public **AcknowledgementMode**

Remarks

Indicates how messages should be acknowledged by QAnywhere client applications.

See Also

QAManager class

QATransactionalManager class

QAManagerBase class

You can determine the mode of a QAManagerBase class instance using the QAManagerBase::Mode property.

For transactional messaging, use the QATransactionalManager class. In this case, you use the commit function to acknowledge messages belonging to a transaction.

For more information, see “Receiving messages synchronously” on page 75and “Receiving messages asynchronously” on page 76.

In implicit acknowledgement mode, messages are acknowledged as soon as they are received by a client application. In explicit acknowledgement mode, you must call one of the QAManager class acknowledgement methods. In transactional mode, you must call commit function to acknowledge all outstanding messages. The server propagates all status changes from client to client.

For more information, see “Initializing a QAnywhere API” on page 54.

The IMPLICIT_ACKNOWLEDGEMENT and EXPLICIT_ACKNOWLEDGEMENT modes are assigned to a QAManager class instance using the open function. The TRANSACTIONAL mode is implicitly assigned to QATransactionalManager class instances.

Members

All members of AcknowledgementMode, including all inherited members.

EXPLICIT_ACKNOWLEDGEMENT variable

Synopsis

const qa_short **AcknowledgementMode::EXPLICIT_ACKNOWLEDGEMENT**

Remarks

Indicates that received messages are acknowledged using one of the QAManager class acknowledge methods.

IMPLICIT_ACKNOWLEDGEMENT variable

Synopsis

const qa_short **AcknowledgementMode::IMPLICIT_ACKNOWLEDGEMENT**

Remarks

Indicates that all messages are acknowledged as soon as they are received by a client application.

If you receive messages synchronously, messages are acknowledged as soon as the getMessage function returns. If you receive messages asynchronously, the message is acknowledged as soon as the event handling function returns.

TRANSACTIONAL variable

Synopsis

const qa_short **AcknowledgementMode::TRANSACTIONAL**

Remarks

Indicates that messages are only acknowledged as part of the ongoing transaction.

This mode is automatically assigned to QATransactionalManager class instances.

MessageProperties class

Synopsis

public **MessageProperties**

Remarks

Provides fields storing standard message property names.

The MessageProperties class provides standard message property names. You can pass MessageProperties fields to QAMessage class methods used to get and set message properties.

For more information, see "Message headers and message properties" on page 206

```
QATextMessage * t_msg;
```

The following example gets the value corresponding to MSG_TYPE variable using the getIntProperty function. The MessageType class enumeration maps the integer result to an appropriate message type.

```
int msg_type;
t_msg->getIntProperty( MessageProperties::MSG_TYPE, &msg_type)
```

The following example, evaluates the message type and RAS names using MSG_TYPE variable and RASNAMES variable respectively.

```
void SystemQueueListener::onMessage(QAMessage * msg) {
    QATextMessage *        t_msg;
    TCHAR                buffer[512];
    int                    len;
    int                    msg_type;

    t_msg = msg->castToTextMessage();
    if( t_msg != NULL ) {
     t_msg->getIntProperty( MessageProperties::MSG_TYPE, &msg_type );

  if( msg_type == MessageType::NETWORK_STATUS_NOTIFICATION ) {

   // get RAS names using MessageProperties::RASNAMES
   len = t_msg->getStringProperty(MessageProperties::RASNAMES,buffer,sizeof
(buffer));

    }

  //...
  }
 }
```

See Also

QAMessage class

Members

All members of MessageProperties, including all inherited members.

ADAPTER variable

Synopsis

const qa_string **MessageProperties::ADAPTER**

Remarks

This property name refers to the currently active network adapter that is being used to connect to the QAnywhere server.

It is used for system queue messages.

The value of this field is "ias_Network.Adapter".

Pass ADAPTER variable as the first parameter to getStringProperty function to access the associated message property.

For more information, see "Message properties" on page 209.

See Also

MessageProperties

ADAPTERS variable

Synopsis

const qa_string **MessageProperties::ADAPTERS**

Remarks

This property name refers to a delimited list of network adapters that can be used to connect to the QAnywhere server.

It is used for system queue messages.

The value of this field is "ias_Adapters".

Pass ADAPTERS variable as the first parameter to getStringProperty function to access the associated message property.

For more information, see “Message properties” on page 209.

See Also

MessageProperties

DELIVERY_COUNT variable

Synopsis

const qa_string **MessageProperties::DELIVERY_COUNT**

Remarks

This property name refers to the number of attempts that have been made so far to deliver the message.

The value of this field is "ias_DeliveryCount".

Pass DELIVERY_COUNT variable as the first parameter in setStringProperty function or getStringProperty function to access the associated message property.

See Also

MessageProperties

IP variable

Synopsis

const qa_string **MessageProperties::IP**

Remarks

This property name refers to the IP address of the currently active network adapter that is being used to connect to the QAnywhere server.

It is used for system queue messages.

The value of this field is "ias_Network.IP".

Pass IP variable as the first parameter to getStringProperty function to access the associated message property.

For more information, see “Message properties” on page 209.

See Also

MessageProperties

MAC variable

Synopsis

const qa_string **MessageProperties::MAC**

Remarks

This property name refers to the MAC address of the currently active network adapter that is being used to connect to the QAnywhere server.

It is used for system queue messages.

The value of this field is "ias_Network.MAC".

Pass MAC variable as the first parameter to getStringProperty function to access the associated message property.

For more information, see "Message properties" on page 209.

See Also

MessageProperties

MSG_TYPE variable

Synopsis

const qa_string **MessageProperties::MSG_TYPE**

Remarks

This property name refers to MessageType class enumeration values associated with a QAnywhere message.

The value of this field is "ias_MessageType". Pass MSG_TYPE variable as the first parameter in setIntProperty function or getIntProperty function to determine the associated property.

See Also

MessageType class

MessageProperties

NETWORK_STATUS variable

Synopsis

const qa_string **MessageProperties::NETWORK_STATUS**

Remarks

This property name refers to the state of the network connection.

The value of this field is "ias_NetworkStatus".

The value of this property is 1 if the network is accessible and 0 otherwise. The network status is used for system queue messages (for example, network status changes).

For more information, see "Message properties" on page 209.

Pass NETWORK_STATUS variable as the first parameter in setStringProperty function or getStringProperty function to access the associated message property.

See Also

MessageProperties

ORIGINATOR variable

Synopsis

const qa_string **MessageProperties::ORIGINATOR**

Remarks

This property name refers to the message store ID of the originator of the message.

The value of this field is "ias_Originator".

Pass ORIGINATOR variable as the first parameter in setStringProperty function or getStringProperty function to access the associated message property.

See Also

MessageProperties

RAS variable

Synopsis

const qa_string **MessageProperties::RAS**

Remarks

This property name refers to the currently active RAS name that is being used to connect to the QAnywhere server.

It is used for system queue messages.

The value of this field is "ias_Network.RAS".

Pass RAS variable as the first parameter to getStringProperty function to access the associated message property.

For more information, see "Message properties" on page 209.

See Also

MessageProperties

RASNAMES variable

Synopsis

const qa_string **MessageProperties::RASNAMES**

Remarks

This property name refers to a delimited list of RAS entry names that can be used to connect to the QAnywhere server.

It is used for system queue messages.

The value of this field is "ias_RASNames".

For more information, see "Message properties" on page 209.

Pass RASNAMES variable as the first parameter in setStringProperty function or getStringProperty function to access the associated message property.

See Also

MessageProperties

STATUS variable

Synopsis

const qa_string **MessageProperties::STATUS**

Remarks

This property name refers to the current status of the message.

For a list of values, see the StatusCodes class enumeration. The value of this field is "ias_Status".

Pass STATUS variable as the first parameter in setIntProperty function or getIntProperty function to access the associated message property.

See Also

StatusCodes class

MessageProperties

STATUS_TIME variable

Synopsis

const qa_string **MessageProperties::STATUS_TIME**

Remarks

This property name refers to the time at which the message received its current status.

It is in units that are natural for the platform. For Windows/PocketPC platforms, the timestamp is the SYSTEMTIME, converted to a FILETIME, which is copied to a qa_long value. It is a local time. The value of this field is "ias_StatusTime".

Pass STATUS_TIME variable as the first parameter in getLongProperty function to access the associated read-only message property.

See Also

MessageProperties

TRANSMISSION_STATUS variable

Synopsis

const qa_string **MessageProperties::TRANSMISSION_STATUS**

Remarks

This property name refers to the current transmission status of the message.

For a list of values, see the StatusCodes class enumeration. The value of this field is "ias_TransmissionStatus".

Pass TRANSMISSION_STATUS variable as the first parameter in setIntProperty function or getIntProperty function to access the associated message property.

See Also

StatusCodes class

MessageProperties

MessageStoreProperties class

Synopsis

public **MessageStoreProperties**

Remarks

The MessageStoreProperties class provides standard message property names.

You can pass MessageProperties class fields to QAManagerBase class methods used to get and set pre-defined or custom message store properties. For more information, see "Client message store properties" on page 215.

Members

All members of MessageStoreProperties, including all inherited members.

- "MAX_DELIVERY_ATTEMPTS variable" on page 401

MAX_DELIVERY_ATTEMPTS variable

Synopsis

const qa_string **MessageStoreProperties::MAX_DELIVERY_ATTEMPTS**

Remarks

This property name refers to the maximum number of times that a message can be received, without explicit acknowledgement, before its status is set to UNRECEIVABLE variable.

The value of this field is "ias_MaxDeliveryAttempts".

See Also

StatusCodes class.

MessageStoreProperties

MessageType class

Synopsis

public **MessageType**

Remarks

Defines constant values for the MSG_TYPE variable message property.

The following example shows the onSystemMessage method which is used to handle QAnywhere system messages. The message type is compared to NETWORK_STATUS_NOTIFICATION variable.

```
    void SystemQueueListener::onMessage(QAMessage * msg)
    {
        QATextMessage *        t_msg;
        TCHAR                buffer[512];
        int                    len;
        int                    msg_type;

        t_msg = msg->castToTextMessage();
        if( t_msg != NULL ) {
         t_msg->getIntProperty( MessageProperties::MSG_TYPE, &msg_type );
        if( msg_type == MessageType::NETWORK_STATUS_NOTIFICATION ) {

         // get network names using MessageProperties::NETWORK
        len = t_msg->getStringProperty(MessageProperties::NETWORK,buffer,sizeof
  (buffer));

    }

    //...
    }
   }
```

Members

All members of MessageType, including all inherited members.

- “NETWORK_STATUS_NOTIFICATION variable” on page 402
- “PUSH_NOTIFICATION variable” on page 403
- “REGULAR variable” on page 403

NETWORK_STATUS_NOTIFICATION variable

Synopsis

const qa_int **MessageType::NETWORK_STATUS_NOTIFICATION**

Remarks

Identifies a QAnywhere system message used to notify QAnywhere client applications of network status changes.

Network status changes apply to the device receiving the system message. Use ADAPTER variable, MessageProperties::NETWORK,and NETWORK_STATUS variable fields to identify new network status information.

For more information, see “Pre-defined message properties” on page 209.

PUSH_NOTIFICATION variable

Synopsis

const qa_int **MessageType::PUSH_NOTIFICATION**

Remarks

Identifies a QAnywhere system message used to notify QAnywhere client applications of push notifications.

If you use the on-demand qaagent policy, a typical response is to call triggerSendReceive function to receive messages waiting with the central message server.

For more information, see “Pre-defined message properties” on page 209.

REGULAR variable

Synopsis

const qa_int **MessageType::REGULAR**

Remarks

If no message type property exists then the message type is assumed to be REGULAR.

This type of message is not treated specially by the message system.

QABinaryMessage class

Synopsis

public **QABinaryMessage**

Base classes

- "QAMessage class" on page 466

Remarks

A QABinaryMessage object is used to send a message containing a stream of uninterpreted bytes.

It inherits from the QAMessage class and adds a bytes message body. QABinaryMessage provides a variety of functions to read from and write to the bytes message body.

When the message is first created, the body of the message is write-only. After a message has been sent, the client that sent it can retain and modify it without affecting the message that has been sent. The same message object can be sent multiple times.

When a message is received, the provider has called reset function so that the message body is in read-only mode and reading of values starts from the beginning of the message body. If a client attempts to write a message in read-only mode, a COMMON_MSG_NOT_WRITEABLE_ERROR is set.

The following example uses the QABinaryMessage writeString function to write the string "Q" followed by the string "Anywhere" to a QABinaryMessage instances message body.

```
// create a binary message instance
QABinaryMessage * binary_message;
binary_message = qa_manager->createBinaryMessage();

// set optional message properties ...
binary_message->setReplyToAddress( "my-queue-name" );

// write to the message body
binary_message->writeString("Q");
binary_messge->writeString("Anywhere");

// put the message in the local database, ready for sending

if( !qa_manager->putMessage( "store-id\\queue-name", msg ) ) {
        handleError();
    }
```

Note: on the receiving end, the first readString function invocation returns "Q" and the next readString function invocation returns "Anywhere".

The message is sent by the QAnywhere Agent. For more information, see "Determining when message transmission should occur on the client" on page 36and "Writing QAnywhere Client Applications" on page 45.

Members

All members of QABinaryMessage, including all inherited members.

getBodyLength function

Synopsis

virtual qa_long **QABinaryMessage::getBodyLength()**

Remarks

Returns the size of the message body in bytes.

See Also

QABinaryMessage

readBinary function

Synopsis

virtual qa_int **QABinaryMessage::readBinary(**
  qa_bytes *value*,
  qa_int *length*
)

Parameters

- **value** The buffer into which the data is read.
- **length** The maximum number of bytes to read.

Remarks

Reads a specified number of bytes starting from the unread portion of the QABinaryMessage instance's message body.

See Also

QABinaryMessage

writeBinary function

Returns

The total number of bytes read into the buffer, or -1 if there is no more data because the end of the stream has been reached.

readBoolean function

Synopsis

virtual qa_bool **QABinaryMessage::readBoolean(**
  qa_bool * *value*
)

Parameters

- **value** The destination of the `qa_bool` value read from the bytes message stream.

Remarks

Reads a boolean value starting from the unread portion of the QABinaryMessage instance's message body.

See Also

QABinaryMessage

writeBoolean function

Returns

True if and only if the operation succeeded.

readByte function

Synopsis

virtual qa_bool **QABinaryMessage::readByte(**
  qa_byte * *value*
)

Parameters

- **value** The destination of the `qa_byte` value read from the bytes message stream.

Remarks

Reads a signed 8-bit value starting from the unread portion of the QABinaryMessage instance's message body.

See Also

QABinaryMessage

writeByte function

Returns

True if and only if the operation succeeded.

readChar function

Synopsis

virtual qa_bool **QABinaryMessage::readChar(**
  qa_char * *value*
)

Parameters

- **value** The destination of the `qa_char` value read from the bytes message stream.

Remarks

Reads a character value starting from the unread portion of the QABinaryMessage instance's message body.

See Also

QABinaryMessage

writeChar function

Returns

The character value read.

readDouble function

Synopsis

virtual qa_bool **QABinaryMessage::readDouble(**
  qa_double * *value*
)

Parameters

- **value** The destination of the double value read from the bytes message stream.

Remarks

Reads a double value starting from the unread portion of the QABinaryMessage instance's message body.

See Also

QABinaryMessage

writeDouble function

Returns

True if and only if the operation succeeded.

readFloat function

Synopsis

virtual qa_bool **QABinaryMessage::readFloat(**
  qa_float * *value*
)

Parameters

- **value** The destination of the float value read from the bytes message stream.

Remarks

Reads a float value starting from the unread portion of the QABinaryMessage instance's message body.

See Also

QABinaryMessage

writeFloat function

Returns

True if and only if the operation succeeded.

readInt function

Synopsis

virtual qa_bool **QABinaryMessage::readInt(**
  qa_int * *value*
)

Parameters

- **value** The destination of the `qa_int` value read from the bytes message stream.

Remarks

Reads a signed 32-bit integer value starting from the unread portion of the QABinaryMessage instance's message body.

See Also

QABinaryMessage

writeInt function

Returns

True if and only if the operation succeeded.

readLong function

Synopsis

virtual qa_bool **QABinaryMessage::readLong(**
  qa_long * *value*
)

Parameters

- **value** The destination of the long value read from the bytes message stream.

Remarks

Reads a signed 64-bit integer value starting from the unread portion of the QABinaryMessage instance's message body.

See Also

QABinaryMessage

writeLong function

Returns

True if and only if the operation succeeded.

readShort function

Synopsis

virtual qa_bool **QABinaryMessage::readShort(**
  qa_short * *value*
)

Parameters

- **value** The destination of the `qa_short` value read from the bytes message stream.

Remarks

Reads a signed 16-bit value starting from the unread portion of the QABinaryMessage instance's message body.

See Also

QABinaryMessage

writeShort function

Returns

True if and only if the operation succeeded.

readString function

Synopsis

virtual qa_int **QABinaryMessage::readString(**
  qa_string *dest*,
  qa_int *maxLen*
)

Parameters

- **dest** The destination of the `qa_string` value read from the bytes message stream.
- **maxLen** The maximum number of characters to read, including the null terminator character.

Remarks

Reads a string value starting from the unread portion of the QABinaryMessage instance's message body.

See Also

QABinaryMessage

writeString function

Returns

The total number of non-null qa_chars read into the buffer, -1 if there is no more data or an error occurred, or -2 if the buffer is too small.

reset function

Synopsis

virtual void **QABinaryMessage::reset()**

Remarks

Resets a message so that the reading of values starts from the beginning of the message body.

The Reset method also puts the QABinaryMessage message body in read-only mode.

See Also

QABinaryMessage

writeBinary function

Synopsis

virtual void **QABinaryMessage::writeBinary(**
  qa_const_bytes *value*,
  qa_int *offset*,
  qa_int *length*
)

Parameters

- **value** The byte array value to write to the message body.
- **offset** The offset within the byte array to begin writing.
- **length** The number of bytes to write.

Remarks

Appends a byte array value to the QABinaryMessage instance's message body.

See Also

QABinaryMessage

readBinary function

writeBoolean function

Synopsis

virtual void **QABinaryMessage::writeBoolean(**
  qa_bool *value*
)

Parameters

- **value** The boolean value to write to the message body.

Remarks

Appends a boolean value to the QABinaryMessage instance's message body.

The boolean is represented as a one-byte value. True is represented as 1; false is represented as 0.

See Also

QABinaryMessage

readBoolean function

writeByte function

Synopsis

virtual void **QABinaryMessage::writeByte(**
qa_byte *value*
)

Parameters

- **value** The byte array value to write to the message body.

Remarks

Appends a byte value to the QABinaryMessage instance's message body.

The byte is represented as a one-byte value.

See Also

QABinaryMessage

readByte function

writeChar function

Synopsis

virtual void **QABinaryMessage::writeChar(**
qa_char *value*
)

Parameters

- **value** the char value to write to the message body.

Remarks

Appends a char value to the QABinaryMessage instance's message body.

The char parameter is represented as a two-byte value and the high order byte is appended first.

See Also

QABinaryMessage

readChar function

writeDouble function

Synopsis

virtual void **QABinaryMessage::writeDouble(**
qa_double *value*
)

Parameters

- **value** The double value to write to the message body.

Remarks

Appends a double value to the QABinaryMessage instance's message body.

The double parameter is converted to a representive eight-byte long value. Higher order bytes are appended first.

See Also

QABinaryMessage

readDouble function

writeFloat function

Synopsis

virtual void **QABinaryMessage::writeFloat(**
qa_float *value*
)

Parameters

- **value** The float value to write to the message body.

Remarks

Appends a float value to the QABinaryMessage instance's message body.

The float parameter is converted to a representative 4-byte integer and the higher order bytes are appended first.

See Also

QABinaryMessage

readFloat function

writeInt function

Synopsis

virtual void **QABinaryMessage::writeInt(**
qa_int *value*
)

Parameters

- **value** the int value to write to the message body.

Remarks

Appends an integer value to the QABinaryMessage instance's message body.

The integer parameter is represented as a four-byte value and higher order bytes are appended first.

See Also

QABinaryMessage

readInt function

writeLong function

Synopsis

virtual void **QABinaryMessage::writeLong(**
qa_long *value*
)

Parameters

- **value** The long value to write to the message body.

Remarks

Appends a long value to the QABinaryMessage instance's message body.

The long parameter is represented as an eight-byte value and higher order bytes are appended first.

See Also

QABinaryMessage

readLong function

writeShort function

Synopsis

virtual void **QABinaryMessage::writeShort(**
qa_short *value*
)

Parameters

- **value** The short value to write to the message body.

Remarks

Appends a short value to the QABinaryMessage instance's message body.

The short parameter is represented as a two-byte value and the higher order byte is appended first.

See Also

QABinaryMessage

readShort function

writeString function

Synopsis

virtual void **QABinaryMessage::writeString(**
qa_const_string *value*
)

Parameters

- **value** The string value to write to the message body.

Remarks

Appends a string value to the QABinaryMessage instance's message body.

Note: The receving application needs to invoke readString function for each writeString invocation.

Note: The UTF-8 representation of the string can be at most 32767 bytes.

See Also

QABinaryMessage

readString function

~QABinaryMessage function

Synopsis

virtual **QABinaryMessage::~QABinaryMessage()**

Remarks

Virtual destructor.

QAError class

Synopsis

public **QAError**

Remarks

This class defines error constants associated with a QAnywhere client application.

A QAError object is used internally by the QAManager class object to keep track of errors associated with messaging operations. The application programmer should not need to create an instance of this class. The error constants should be used by the application programmer to interpret error codes returned by getLastError function

See Also

getLastErrorMsg function

```
if (qa_mgr->getLastError() != QAError::QA_NO_ERROR)
{
  //process error
}
```

Members

All members of QAError, including all inherited members.

- "COMMON_ALREADY_OPEN_ERROR variable" on page 419
- "COMMON_GET_INIT_FILE_ERROR variable" on page 420
- "COMMON_GETQUEUEDEPTH_ERROR variable" on page 419
- "COMMON_GETQUEUEDEPTH_ERROR_INVALID_ARG variable" on page 419
- "COMMON_GETQUEUEDEPTH_ERROR_NO_STORE_ID variable" on page 420
- "COMMON_INIT_ERROR variable" on page 420
- "COMMON_INIT_THREAD_ERROR variable" on page 420
- "COMMON_INVALID_PROPERTY variable" on page 421
- "COMMON_MSG_ACKNOWLEDGE_ERROR variable" on page 421
- "COMMON_MSG_CANCEL_ERROR variable" on page 421
- "COMMON_MSG_CANCEL_ERROR_SENT variable" on page 421
- "COMMON_MSG_NOT_WRITEABLE_ERROR variable" on page 422
- "COMMON_MSG_RETRIEVE_ERROR variable" on page 422
- "COMMON_MSG_STORE_ERROR variable" on page 422
- "COMMON_MSG_STORE_NOT_INITIALIZED variable" on page 423
- "COMMON_MSG_STORE_TOO_LARGE variable" on page 423
- "COMMON_NO_DEST_ERROR variable" on page 423
- "COMMON_NO_IMPLEMENTATION variable" on page 424
- "COMMON_NOT_OPEN_ERROR variable" on page 423
- "COMMON_OPEN_ERROR variable" on page 424
- "COMMON_OPEN_LOG_FILE_ERROR variable" on page 424
- "COMMON_OPEN_MAXTHREADS_ERROR variable" on page 424
- "COMMON_SELECTOR_SYNTAX_ERROR variable" on page 425

- "COMMON_TERMINATE_ERROR variable" on page 425
- "COMMON_UNEXPECTED_EOM_ERROR variable" on page 425
- "COMMON_UNREPRESENTABLE_TIMESTAMP variable" on page 425
- "QA_NO_ERROR variable" on page 426

COMMON_ALREADY_OPEN_ERROR variable

Synopsis

const qa_int **QAError::COMMON_ALREADY_OPEN_ERROR**

Remarks

The QAManager class is already open.

See Also

QAError

COMMON_GETQUEUEDEPTH_ERROR variable

Synopsis

const qa_int **QAError::COMMON_GETQUEUEDEPTH_ERROR**

Remarks

Error getting queue depth.

See Also

QAError

COMMON_GETQUEUEDEPTH_ERROR_INVALID_ARG variable

Synopsis

const qa_int **QAError::COMMON_GETQUEUEDEPTH_ERROR_INVALID_ARG**

Remarks

Cannot use getQueueDepth function on a given destination when filter is ALL.

See Also

QAError

COMMON_GETQUEUEDEPTH_ERROR_NO_STORE_ID variable

Synopsis

const qa_int **QAError::COMMON_GETQUEUEDEPTH_ERROR_NO_STORE_ID**

Remarks

Cannot use getQueueDepth function when the message store ID has not been set.

See Also

QAError

COMMON_GET_INIT_FILE_ERROR variable

Synopsis

const qa_int **QAError::COMMON_GET_INIT_FILE_ERROR**

Remarks

Unable to access the client properties file.

See Also

QAError

COMMON_INIT_ERROR variable

Synopsis

const qa_int **QAError::COMMON_INIT_ERROR**

Remarks

Initialization error.

See Also

QAError

COMMON_INIT_THREAD_ERROR variable

Synopsis

const qa_int **QAError::COMMON_INIT_THREAD_ERROR**

Remarks

Error initializing the background thread.

See Also

QAError

COMMON_INVALID_PROPERTY variable

Synopsis

const qa_int **QAError::COMMON_INVALID_PROPERTY**

Remarks

There is an invalid property in the client properties file.

See Also

QAError

COMMON_MSG_ACKNOWLEDGE_ERROR variable

Synopsis

const qa_int **QAError::COMMON_MSG_ACKNOWLEDGE_ERROR**

Remarks

Error acknowledging the message.

See Also

QAError

COMMON_MSG_CANCEL_ERROR variable

Synopsis

const qa_int **QAError::COMMON_MSG_CANCEL_ERROR**

Remarks

Error cancelling message.

See Also

QAError

COMMON_MSG_CANCEL_ERROR_SENT variable

Synopsis

const qa_int **QAError::COMMON_MSG_CANCEL_ERROR_SENT**

Remarks

Error cancelling message.

Cannot cancel a message that has already been sent.

See Also

QAError

COMMON_MSG_NOT_WRITEABLE_ERROR variable

Synopsis

const qa_int **QAError::COMMON_MSG_NOT_WRITEABLE_ERROR**

Remarks

You cannot write to a message as it is in read-only mode.

See Also

QAError

COMMON_MSG_RETRIEVE_ERROR variable

Synopsis

const qa_int **QAError::COMMON_MSG_RETRIEVE_ERROR**

Remarks

Error retrieving a message from the client message store.

See Also

QAError

COMMON_MSG_STORE_ERROR variable

Synopsis

const qa_int **QAError::COMMON_MSG_STORE_ERROR**

Remarks

Error storing a message in the client message store.

See Also

QAError

COMMON_MSG_STORE_NOT_INITIALIZED variable

Synopsis

const qa_int **QAError::COMMON_MSG_STORE_NOT_INITIALIZED**

Remarks

The message store has not been initialized for messaging.

See Also

QAError

COMMON_MSG_STORE_TOO_LARGE variable

Synopsis

const qa_int **QAError::COMMON_MSG_STORE_TOO_LARGE**

Remarks

The message store is too large relative to the free disk space on the device.

See Also

QAError

COMMON_NOT_OPEN_ERROR variable

Synopsis

const qa_int **QAError::COMMON_NOT_OPEN_ERROR**

Remarks

The QAManager class is not open.

See Also

QAError

COMMON_NO_DEST_ERROR variable

Synopsis

const qa_int **QAError::COMMON_NO_DEST_ERROR**

Remarks

No destination.

See Also

QAError

COMMON_NO_IMPLEMENTATION variable

Synopsis

const qa_int **QAError::COMMON_NO_IMPLEMENTATION**

Remarks

The function is not implemented.

See Also

QAError

COMMON_OPEN_ERROR variable

Synopsis

const qa_int **QAError::COMMON_OPEN_ERROR**

Remarks

Error opening a connection to the message store.

See Also

QAError

COMMON_OPEN_LOG_FILE_ERROR variable

Synopsis

const qa_int **QAError::COMMON_OPEN_LOG_FILE_ERROR**

Remarks

Error opening the log file.

See Also

QAError

COMMON_OPEN_MAXTHREADS_ERROR variable

Synopsis

const qa_int **QAError::COMMON_OPEN_MAXTHREADS_ERROR**

Remarks

Cannot open the QAManager class because the maximum number of concurrent server requests is not high enough (see database server -gn option).

COMMON_SELECTOR_SYNTAX_ERROR variable

Synopsis

const qa_int **QAError::COMMON_SELECTOR_SYNTAX_ERROR**

Remarks

The given selector has a syntax error.

See Also

QAError

COMMON_TERMINATE_ERROR variable

Synopsis

const qa_int **QAError::COMMON_TERMINATE_ERROR**

Remarks

Termination error.

See Also

QAError

COMMON_UNEXPECTED_EOM_ERROR variable

Synopsis

const qa_int **QAError::COMMON_UNEXPECTED_EOM_ERROR**

Remarks

Unexpected end of message reached.

See Also

QAError

COMMON_UNREPRESENTABLE_TIMESTAMP variable

Synopsis

const qa_int **QAError::COMMON_UNREPRESENTABLE_TIMESTAMP**

Remarks

The timestamp is outside of the acceptable range.

See Also

QAError

QA_NO_ERROR variable

Synopsis

const qa_int **QAError::QA_NO_ERROR**

Remarks

No error.

See Also

QAError

QAManager class

Synopsis

public **QAManager**

Base classes

- "QAManagerBase class" on page 432

Remarks

The QAManager class derives from QAManagerBase class and manages non-transactional QAnywhere messaging operations.

For a detailed description of derived behavior, see QAManagerBase class.

The QAManager can be configured for implicit or explicit acknowledgement as defined in the AcknowledgementMode class enumeration. To acknowledge messages as part of a transaction, use QATransactionalManager class.

Use the QAManagerFactory class to create QAManager and QATransactionalManager class objects.

Members

All members of QAManager, including all inherited members.

acknowledge function

Synopsis

virtual qa_bool **QAManager::acknowledge(**
  QAMessage * *msg*
)

Parameters

- **msg** The message to acknowledge.

Remarks

Acknowledges that the client application successfully received a QAnywhere message.

Note: when a QAMessage class is acknowledged, its STATUS variable property changes to RECEIVED variable. When a QAMessage class status changes to RECEIVED variable, it can be deleted using the default delete rule.

For more information about delete rules, see "Message delete rules" on page 239.

See Also

QAManager

acknowledgeAll function

acknowledgeUntil function

Returns

True if and only if the operation succeeded.

acknowledgeAll function

Synopsis

virtual qa_bool **QAManager::acknowledgeAll()**

Remarks

Acknowledges that the client application successfully received all unacknowledged QAnywhere messages.

Note: when a QAMessage class is acknowledged, its STATUS variable property changes to RECEIVED variable. When a QAMessage class status changes to RECEIVED variable, it can be deleted using the default delete rule.

For more information about delete rules, see "Message delete rules" on page 239.

See Also

QAManager

acknowledge function

acknowledgeUntil function

Returns

True if and only if the operation succeeded.

acknowledgeUntil function

Synopsis

virtual qa_bool **QAManager::acknowledgeUntil(**
  QAMessage * *msg*
)

Parameters

- **msg** The last message to acknowledge. All earlier unacknowledged messages are also acknowledged.

Remarks

Acknowledges the given QAMessage class instance and all unacknowledged messages received before the given message.

Note: when a QAMessage class is acknowledged, its STATUS variable property changes to RECEIVED variable. When a QAMessage class status changes to RECEIVED variable, it can be deleted using the default delete rule.

For more information about delete rules, see "Message delete rules" on page 239.

See Also

QAManager

acknowledge function

acknowledgeAll function

Returns

True if and only if the operation succeeded.

open function

Synopsis

```
virtual qa_bool QAManager::open(
  qa_short mode
)
```

Parameters

- **mode** The acknowledgement mode.

Remarks

Opens the QAManager with the given AcknowledgementMode class value.

The open function must be the first method called after creating a QAManager.

See Also

QAManager

AcknowledgementMode class

Returns

True if and only if the operation succeeded.

recover function

Synopsis

virtual qa_bool **QAManager::recover()**

Remarks

Force all unacknowledged messages into a state of unreceived.

That is, these messages must be received again using getMessage function.

Returns

True if and only if the operation succeeded.

QAManagerBase class

Synopsis

public **QAManagerBase**

Derived classes

- "QAManager class" on page 427
- "QATransactionalManager class" on page 492

Remarks

This class acts as a base class for QATransactionalManager class and QAManager class, which manage transactional and non-transactional messaging, respectively.

Use the start function to allow a QAManagerBase instance to listen for messages. There must be only a single instance of QAManagerBase per thread in your application. For more information about qaagent transmission policies, see "Determining when message transmission should occur on the client" on page 36.

You can use instances of this class to create and manage QAnywhere messages. Use the createBinaryMessage function and the createTextMessage function to create appropriate QAMessage class instances. QAMessage class instances provide a variety of methods to set message content and properties. To send QAnywhere messages, use the putMessage function to place the addressed message in the local message store queue. The message is transmitted by the QAnywhere Agent based on its transmission policies or when you call the triggerSendReceive function.

Messages are released from memory when you close a QAManagerBase instance using the close function. For more information, see "Client message store properties" on page 215and the MessageStoreProperties class.

You can use getLastError function and QAManagerBase::getLastErrorMessage to return error information when a QAException occurs. QAManagerBase also provides methods to set and get message store properties.

Members

All members of QAManagerBase, including all inherited members.

- "beginEnumStorePropertyNames function" on page 433
- "browseClose function" on page 434
- "browseMessages function" on page 434
- "browseMessagesByID function" on page 435
- "browseMessagesByQueue function" on page 436
- "browseMessagesBySelector function" on page 436
- "browseNextMessage function" on page 437
- "cancelMessage function" on page 438
- "close function" on page 438
- "createBinaryMessage function" on page 439
- "createTextMessage function" on page 439
- "deleteMessage function" on page 440
- "endEnumStorePropertyNames function" on page 440

beginEnumStorePropertyNames function

Synopsis

virtual qa_store_property_enum_handle **QAManagerBase::beginEnumStorePropertyNames()**

Remarks

Begins an enumeration of message store property names.

The handle returned by this method is supplied to the nextStorePropertyName function. This method and the nextStorePropertyName function can be used to enumerate the message store property names at the time this method was called. Message store properties cannot be set between the beginEnumStorePropertyNames and the endEnumStorePropertyNames function calls.

See Also

QAManagerBase

Returns

A handle that is supplied to nextStorePropertyName function.

browseClose function

Synopsis

virtual void **QAManagerBase::browseClose(**
  qa_browse_handle *handle*
)

Parameters

- **handle** A handle returned by one of the begin browse operations.

Remarks

Frees the resources associated with a browse operation.

See Also

QAManagerBase

browseMessages function

Synopsis

virtual qa_browse_handle **QAManagerBase::browseMessages()**

Remarks

Begins a browse of messages queued in the message store.

The handle returned by this method is supplied to browseNextMessage function. This method and the browseNextMessage function can be used to enumerate the messages in the message store at the time this method was called.

The messages are just being browsed, so they cannot be acknowledged. Use getMessage function to receive messages so they can be acknowledged.

See Also

browseNextMessage function

browseMessagesByQueue function

browseMessagesByID function

browseClose function

QAManagerBase

Returns

A handle that is supplied to browseNextMessage function

browseMessagesByID function

Synopsis

virtual qa_browse_handle **QAManagerBase::browseMessagesByID(**
qa_const_string *msgid*
)

Parameters

- **msgid** The message ID.

Remarks

Begins a browse of the message that is queued in the message store, with the given message ID.

The handle returned by this method is supplied to browseNextMessage function. This method and browseNextMessage function can be used to enumerate the messages in the message store at the time this method was called.

The messages are just being browsed, so they cannot be acknowledged. Use getMessage function to receive messages so they can be acknowledged.

See Also

browseNextMessage function

browseMessagesByQueue function

QAManagerBase::BrowseMessages()

browseClose function

QAManagerBase

Returns

A handle that is supplied to browseNextMessage.

browseMessagesByQueue function

Synopsis

virtual qa_browse_handle **QAManagerBase::browseMessagesByQueue(**
qa_const_string *address*
)

Parameters

- **address** The queue in which to browse.

Remarks

Begins a browse of messages queued in the message store for the given queue.

The handle returned by this method is supplied to browseNextMessage function. This method and browseNextMessage function can be used to enumerate the messages in the message store at the time this method was called.

The messages are just being browsed, so they cannot be acknowledged. Use getMessage function to receive messages so they can be acknowledged.

See Also

browseNextMessage function

browseMessagesByID function

QAManagerBase::BrowseMessages()

browseClose function

QAManagerBase

Returns

A handle that is supplied to browseNextMessage.

browseMessagesBySelector function

Synopsis

virtual qa_browse_handle **QAManagerBase::browseMessagesBySelector(**
qa_const_string *selector*
)

Parameters

- **selector** The selector.

Remarks

Begins a browse of messages queued in the message store that satisfy the given selector.

The handle returned by this method is supplied to browseNextMessage function. This method and browseNextMessage function can be used to enumerate the messages in the message store at the time this method was called.

The messages are just being browsed, so they cannot be acknowledged.

Use getMessage function to receive messages so they can be acknowledged.

See Also

browseNextMessage function

browseMessagesByID function

browseMessagesByQueue function

QAManagerBase::BrowseMessages()

browseClose function

QAManagerBase

Returns

A handle that is supplied to browseNextMessage.

browseNextMessage function

Synopsis

virtual QAMessage * **QAManagerBase::browseNextMessage(**
  qa_browse_handle *handle*
)

Parameters

- **handle** A handle returned by one of the begin browse operations.

Remarks

Returns the next message for the given browse operation, returning null if there are no more messages.

To obtain the handle to browsed messages, use browseMessages function or other QAManagerBase methods which allow you to browse messages by queue or message ID.

See Also

QAManagerBase::BrowseMessages()

browseMessagesByQueue function

browseMessagesByID function

browseClose function

QAManagerBase

Returns

The next message, or qa_null if there are no more messages.

cancelMessage function

Synopsis

virtual qa_bool **QAManagerBase::cancelMessage(**
  qa_const_string *msgid*
)

Parameters

- **msgid** The ID of the message to cancel.

Remarks

Cancels the message with the given message ID.

cancelMessage puts a message into a cancelled state before it is transmitted. With the default delete rules of the QAnywhere Agent, cancelled messages are eventually deleted from the message store. cancelMessage fails if the message is already in a final state, or if it has been transmitted to the central messaging server.

For more information about delete rules, see "Message delete rules" on page 239.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

close function

Synopsis

virtual qa_bool **QAManagerBase::close()**

Remarks

Closes the connection to the QAnywhere message system and releases any resources used by the QAManagerBase.

Subsequent calls to close function are ignored. When an instance of QAManagerBase is closed, it cannot be re-opened; you must create and open a new QAManagerBase instance in this case.

See Also

open function

open function

QAManagerBase

Returns

True if and only if the operation succeeded.

createBinaryMessage function

Synopsis

virtual QABinaryMessage * **QAManagerBase::createBinaryMessage()**

Remarks

Creates a QABinaryMessage class instance.

A QABinaryMessage class instance is used to send a message containing a message body of uninterpreted bytes.

See Also

QABinaryMessage class

QAManagerBase

Returns

A new QABinaryMessage class instance.

createTextMessage function

Synopsis

virtual QATextMessage * **QAManagerBase::createTextMessage()**

Remarks

Creates a QATextMessage class instance.

A QATextMessage class object is used to send a message containing a string message body.

See Also

QATextMessage class

QAManagerBase

Returns

A new QATextMessage class instance.

deleteMessage function

Synopsis

virtual void **QAManagerBase::deleteMessage(**
QAMessage * *msg*
)

Parameters

- **msg** The message to delete.

Remarks

Deletes a QAMessage class object.

By default, messages created by QAManagerBase:createTextMessage or createBinaryMessage function are deleted automatically when the QAManagerBase is closed. This method allows more control over when messages are deleted.

See Also

QAManagerBase

endEnumStorePropertyNames function

Synopsis

virtual void **QAManagerBase::endEnumStorePropertyNames(**
qa_store_property_enum_handle *h*
)

Parameters

- **h** A handle returned by beginEnumStorePropertyNames.

Remarks

Frees the resources associated with a message store property name enumeration.

See Also

QAManagerBase

getAllQueueDepth function

Synopsis

virtual qa_int **QAManagerBase::getAllQueueDepth(**
qa_short *filter*
)

Parameters

- **filter** A filter indicating incoming messages, outgoing messages, or all messages.

Remarks

Returns the total depth of all queues, based on a given filter.

The depth of a queue is the number of messages which have not been received (for example, using getMessage function).

See Also

QAManagerBase

Returns

The number of messages, or -1 if an error occurs.

See Also

QueueDepthFilter class.

getBooleanStoreProperty function

Synopsis

```
virtual qa_bool QAManagerBase::getBooleanStoreProperty(
  qa_const_string name,
  qa_bool * value
)
```

Parameters

- **name** The pre-defined or custom property name.
- **value** The destination for the boolean value.

Remarks

Gets a boolean value for a pre-defined or custom message store property.

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

getByteStoreProperty function

Synopsis

virtual qa_bool **QAManagerBase::getByteStoreProperty(**
  qa_const_string *name*,
  qa_byte * *value*
)

Parameters

- **name** The pre-defined or custom property name.
- **value** The destination for the byte value.

Remarks

Gets a byte value for a pre-defined or custom message store property.

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

getDoubleStoreProperty function

Synopsis

virtual qa_bool **QAManagerBase::getDoubleStoreProperty(**
  qa_const_string *name*,
  qa_double * *value*
)

Parameters

- **name** The pre-defined or custom property name.
- **value** The destination for the double value.

Remarks

Gets a double value for a pre-defined or custom message store property.

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

getFloatStoreProperty function

Synopsis

```
virtual qa_bool QAManagerBase::getFloatStoreProperty(
  qa_const_string name,
  qa_float * value
)
```

Parameters

- **name** The pre-defined or custom property name.
- **value** The destination for the float value.

Remarks

Gets a float value for a pre-defined or custom message store property.

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

getIntStoreProperty function

Synopsis

```
virtual qa_bool QAManagerBase::getIntStoreProperty(
  qa_const_string name,
  qa_int * value
)
```

Parameters

- **name** The pre-defined or custom property name.
- **value** The destination for the int value.

Remarks

Gets an int value for a pre-defined or custom message store property.

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

getLastError function

Synopsis

virtual qa_int **QAManagerBase::getLastError()**

Remarks

The error code associated with the last excecuted QAManagerBase method.

0 indicates no error.

For a list of values, see the QAError class.

See Also

getLastErrorMsg function

QAError class

QAManagerBase

Returns

The error code.

getLastErrorMsg function

Synopsis

virtual qa_string **QAManagerBase::getLastErrorMsg()**

Remarks

The error text associated with the last executed QAManagerBase method.

This method returns null if getLastError function returns 0. You can retrieve this property after catching a QAError class.

See Also

getLastError function

QAError class

QAManagerBase

Returns

The error message.

getLongStoreProperty function

Synopsis

```
virtual qa_bool QAManagerBase::getLongStoreProperty(
  qa_const_string name,
  qa_long * value
)
```

Parameters

- **name** The pre-defined or custom property name.
- **value** The destination for the long value.

Remarks

Gets a long value for a pre-defined or custom message store property.

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

getMessage function

Synopsis

```
virtual QAMessage * QAManagerBase::getMessage(
  qa_const_string address
)
```


Parameters

- **address** The destination.

Remarks

Returns the next available QAMessage class sent to the specified address.

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name'or 'queue-name'. If there is no message available, this call blocks indefinitely until a message is available. Use this method to receive messages synchronously.

For more information about receiving messages asynchronously (using a message event handler), see "Receiving messages asynchronously" on page 76.

See Also

QAManagerBase

Returns

The next QAMessage class, or null if no message is available.

getMessageBySelector function

Synopsis

```
virtual QAMessage * QAManagerBase::getMessageBySelector(
  qa_const_string address,
  qa_const_string selector
)
```

Parameters

- **address** The destination.
- **selector** The selector.

Remarks

Returns the next available QAMessage class sent to the specified address that satisfies the given selector.

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name'or 'queue-name'. If there is no message available, this call blocks indefinitely until a message is available. Use this method to receive messages synchronously.

For more information about receiving messages asynchronously (using a message event handler), see "Receiving messages asynchronously" on page 76.

See Also

QAManagerBase

Returns

The next QAMessage class, or null if no message is available.

getMessageBySelectorNoWait function

Synopsis

virtual QAMessage * **QAManagerBase::getMessageBySelectorNoWait(**
  qa_const_string *address*,
  qa_const_string *selector*
)

Parameters

- **address** The destination.
- **selector** The selector.

Remarks

Returns the next available QAMessage class sent to the given address that satisfies the given selector.

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name'or 'queue-name'. If no message is available, this method returns immediately. Use this method to receive messages synchronously. For more information about receiving messages asynchronously (using a message event handler), see "Receiving messages asynchronously" on page 76.

See Also

QAManagerBase

Returns

the next message, or qa_null if no message is available.

getMessageBySelectorTimeout function

Synopsis

virtual QAMessage * **QAManagerBase::getMessageBySelectorTimeout(**
  qa_const_string *address*,
  qa_const_string *selector*,
  qa_long *timeout*
)

Parameters

- **address** The destination.
- **selector** The selector.
- **timeout** the maximum time, in milliseconds, to wait

Remarks

Returns the next available QAMessage class sent to the given address that satisfies the given selector.

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name'or 'queue-name'. If no message is available, this method waits for the specified timeout and then returns. Use

this method to receive messages synchronously. For more information about receiving messages asynchronously (using a message event handler), see "Receiving messages asynchronously" on page 76.

See Also

QAManagerBase

Returns

The next QAMessage class, or null if no message is available.

getMessageNoWait function

Synopsis

```
virtual QAMessage * QAManagerBase::getMessageNoWait(
  qa_const_string address
)
```

Parameters

- **address** The destination.

Remarks

Returns the next available QAMessage class sent to the given address.

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name'or 'queue-name'. If no message is available, this method returns immediately. Use this method to receive messages synchronously.

For more information about receiving messages asynchronously (using a message event handler), see "Receiving messages asynchronously" on page 76.

See Also

QAManagerBase

Returns

The next message, or qa_null if no message is available.

getMessageTimeout function

Synopsis

```
virtual QAMessage * QAManagerBase::getMessageTimeout(
  qa_const_string address,
  qa_long timeout
)
```

Parameters

- **address** The destination

- **timeout** The maximum time, in milliseconds, to wait

Remarks

Returns the next available QAMessage class sent to the given address.

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name'or 'queue-name'. If no message is available, this method waits for the specified timeout and then returns. Use this method to receive messages synchronously.

For more information about receiving messages asynchronously (using a message event handler), see "Receiving messages asynchronously" on page 76.

See Also

QAManagerBase

Returns

The next QAMessage class, or null if no message is available.

getMode function

Synopsis

virtual qa_short **QAManagerBase::getMode()**

Remarks

Returns the QAManager class acknowledgement mode for received messages.

For a list of values, see the AcknowledgementMode class.

EXPLICIT_ACKNOWLEDGEMENT variable and IMPLICIT_ACKNOWLEDGEMENT variable apply to QAManager class instances; TRANSACTIONAL variable is the mode for QATransactionalManager class instances.

See Also

QAManagerBase

Returns

the acknowledge mode

See Also

AcknowledgementMode class

getQueueDepth function

Synopsis

virtual qa_int **QAManagerBase::getQueueDepth(**
  qa_const_string *address*,
  qa_short *filter*
)

Parameters

- **address** The queue name.
- **filter** a filter indicating incoming messages, outgoing messages, or all messages.

Remarks

Returns the depth of a queue, based on a given filter.

The depth of the queue is the number of messages which have not been received (for example, using getMessage function).

See Also

QAManagerBase

Returns

The number of messages in the queue, or -1 if an error occurs.

See Also

QueueDepthFilter class

getShortStoreProperty function

Synopsis

virtual qa_bool **QAManagerBase::getShortStoreProperty(**
  qa_const_string *name*,
  qa_short * *value*
)

Parameters

- **name** The pre-defined or custom property name.
- **value** The destination for the short value.

Remarks

Gets a short value for a pre-defined or custom message store property.

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

getStringStoreProperty function

Synopsis

```
virtual qa_int QAManagerBase::getStringStoreProperty(
  qa_const_string name,
  qa_string address,
  qa_int maxlen
)
```

Parameters

- **name** The pre-defined or custom property name.
- **address** The destination for the qa_string value.
- **maxlen** The maximum number of qa_chars of the value to copy, including the null terminator character.

Remarks

Gets a string value for a pre-defined or custom message store property.

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See Also

QAManagerBase

Returns

The number of non-null qa_chars actually copied, or -1 if the operation failed.

nextStorePropertyName function

Synopsis

```
virtual qa_int QAManagerBase::nextStorePropertyName(
  qa_store_property_enum_handle h,
  qa_string buffer,
  qa_int bufferLen
)
```


Parameters

- **h** A handle returned by beginEnumStorePropertyNames.
- **buffer** The buffer into which to write the property name.
- **bufferLen** The length of the buffer to store the property name. This length must include space for the null terminator.

Remarks

Returns the message store property name for the given enumeration.

If there are no more property names, returns -1.

See Also

QAManagerBase

Returns

The length of the property name, or -1 if there are no more property names. property names

putMessage function

Synopsis

```
virtual qa_bool QAManagerBase::putMessage(
  qa_const_string address,
  QAMessage * msg
)
```

Parameters

- **address** The destination.
- **msg** The message.

Remarks

Puts a message into the queue for the given destination.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

putMessageTimeToLive function

Synopsis

virtual qa_bool **QAManagerBase::putMessageTimeToLive(**
qa_const_string *address*,
QAMessage * *msg*,
qa_long *ttl*
)

Parameters

- **address** The destination.
- **msg** The message.
- **ttl** The time-to-live, in milliseconds.

Remarks

Puts a message into the queue for the given destination and a given time-to-live in milliseconds.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

setBooleanStoreProperty function

Synopsis

virtual qa_bool **QAManagerBase::setBooleanStoreProperty(**
qa_const_string *name*,
qa_bool *value*
)

Parameters

- **name** The pre-defined or custom property name.
- **value** The qa_bool value of the property.

Remarks

Sets a pre-defined or custom message store property to a boolean value.

You can use this method to set pre-defined or user-defined client. store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

setByteStoreProperty function

Synopsis

```
virtual qa_bool QAManagerBase::setByteStoreProperty(
  qa_const_string name,
  qa_byte value
)
```

Parameters

- **name** The pre-defined or custom property name.
- **value** The qa_byte value of the property.

Remarks

Sets a pre-defined or custom message store property to a byte value.

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

setDoubleStoreProperty function

Synopsis

```
virtual qa_bool QAManagerBase::setDoubleStoreProperty(
  qa_const_string name,
  qa_double value
)
```

Parameters

- **name** The pre-defined or custom property name.
- **value** The qa_double value of the property.

Remarks

Sets a pre-defined or custom message store property to a double value.

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

setFloatStoreProperty function

Synopsis

```
virtual qa_bool QAManagerBase::setFloatStoreProperty(
  qa_const_string name,
  qa_float value
)
```

Parameters

- **name** The pre-defined or custom property name.
- **value** The qa_float value of the property.

Remarks

Sets a pre-defined or custom message store property to a float value.

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

setIntStoreProperty function

Synopsis

virtual qa_bool **QAManagerBase::setIntStoreProperty(**
  qa_const_string *name*,
  qa_int *value*
)

Parameters

- **name** The pre-defined or custom property name.
- **value** The qa_int value of the property.

Remarks

Sets a pre-defined or custom message store property to a int value.

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

setLongStoreProperty function

Synopsis

virtual qa_bool **QAManagerBase::setLongStoreProperty(**
  qa_const_string *name*,
  qa_long *value*
)

Parameters

- **name** The pre-defined or custom property name.
- **value** The qa_long value of the property.

Remarks

Sets a pre-defined or custom message store property to a long value.

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

setMessageListener function

Synopsis

```
virtual void QAManagerBase::setMessageListener(
  qa_const_string address,
  QAMessageListener * listener
)
```

Parameters

- **address** The destination address that the listener applies to.
- **listener** The message listener to associate with destination address.

Remarks

Sets a message listener class to recieve QAnywhere messages asynchronously.

The listener is an instance of a class implementing onMessage function, the only method defined in the QAMessageListener class interface. onMessage function accepts a single QAMessage class parameter.

The setMessageListener address parameter specifies a local queue name used to receive the message. You can only have one listener assigned to a given queue. If you want to listen for QAnywhere system messages, including push notifications and network status changes, specify "system" as the queue name. Use this method to receive message asynchronously.

For more information, see “Receiving messages asynchronously” on page 76and “System queue” on page 50.

See Also

QAManagerBase

setMessageListenerBySelector function

Synopsis

```
virtual void QAManagerBase::setMessageListenerBySelector(
  qa_const_string address,
  qa_const_string selector,
  QAMessageListener * listener
)
```


Parameters

- **address** The destination address that the listener applies to.
- **selector** The selector to be used to filter the messages to be received.
- **listener** The message listener to associate with destination address.

Remarks

Sets a message listener class to recieve QAnywhere messages asynchronously, with a message selector.

The listener is an instance of a class implementing onMessage function, the only method defined in the QAMessageListener class interface. onMessage function accepts a single QAMessage class parameter.

The setMessageListener address parameter specifies a local queue name used to receive the message. You can only have one listener assigned to a given queue. The selector parameter specifies a selector to be used to filter the messages to be received on the given address. If you want to listen for QAnywhere system messages, including push notifications and network status changes, specify "system" as the queue name. Use this method to receive message asynchronously.

For more information, see "Receiving messages asynchronously" on page 76and "System queue" on page 50.

See Also

QAManagerBase

setProperty function

Synopsis

```
virtual qa_bool QAManagerBase::setProperty(
  qa_const_string name,
  qa_const_string value
)
```

Parameters

- **name** The pre-defined or custom QAnywhere Manager configuration property name.
- **value** The value of the QAnywhere Manager configuration property.

Remarks

Allows you to set QAnywhere Manager configuration properties programmatically.

You can use this method to override default QAnywhere Manager configuration properties by specifying a property name and value.

For a list of QAnywhere Manager configuration properties, see "QAnywhere manager configuration properties" on page 62.

You can also set QAnywhere Manager configuration properties using a properties file and the createQAManager function.

For more information, see "Setting QAnywhere manager configuration properties in a file" on page 62.

Note: you must set required properties before calling open function or open function.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

setShortStoreProperty function

Synopsis

```
virtual qa_bool QAManagerBase::setShortStoreProperty(
  qa_const_string name,
  qa_short value
)
```

Parameters

- **name** The pre-defined or custom property name.
- **value** The qa_short value of the property.

Remarks

Sets a pre-defined or custom message store property to a short value.

You can use this method to set pre-defined or user-defined client store properties. For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

setStringStoreProperty function

Synopsis

```
virtual qa_bool QAManagerBase::setStringStoreProperty(
  qa_const_string name,
  qa_const_string value
)
```


Parameters

- **name** The pre-defined or custom property name.
- **value** The qa_string value of the property.

Remarks

Sets a pre-defined or custom message store property to a string value.

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see MessageStoreProperties class.

For more information, see "Client message store properties" on page 215.

See Also

QAManagerBase

Returns

True if and only if the operation succeeded.

start function

Synopsis

virtual qa_bool **QAManagerBase::start()**

Remarks

Starts the QAManagerBase for receiving incoming messages in message listeners.

The QAManagerBase does not need to be started if there are no message listeners set, ie. if messages are received with the getMessage methods. It is not recommended to use the getMessage methods as well as message listeners for receiving messages, one should use one or the other of the asynchronous (message listener) or synchronous (getMessage) models. Any calls to start beyond the first without an intervening stop function call are ignored.

See Also

stop function

QAManagerBase

Returns

True if and only if the operation succeeded.

stop function

Synopsis

virtual qa_bool **QAManagerBase::stop()**

Remarks

Stops the QAManagerBase's reception of incoming messages.

The messages are not lost. They are not received until the manager is started again. Any calls to stop beyond the first without an intervening start function are ignored.

See Also

start function

QAManagerBase

Returns

True if and only if the operation succeeded.

triggerSendReceive function

Synopsis

virtual qa_bool **QAManagerBase::triggerSendReceive()**

Remarks

Causes a synchronization with the QAnywhere message server, uploading any messages addressed to other clients, and downloading any messages addressed to the local client.

QAManagerBase triggerSendReceive results in immediate message synchronization between a QAnywhere Agent and the central messaging server. A manual TriggerSendReceive call results in immediate message transmission, independent of the QAnywhere Agent transmission policies. QAnywhere Agent transmission policies determine how message transmission occurs. For example, message transmission can occur automatically at regular intervals, when your client receives a push notification, or when you call the putMessage function to send a message.

For more information, see "Determining when message transmission should occur on the client" on page 36.

See Also

putMessage function

QAManagerBase

Returns

True if and only if the operation succeeded.

QAManagerFactory class

Synopsis

public **QAManagerFactory**

Remarks

This class acts as a factory class for creating QATransactionalManager class and QAManager class objects.

You can only have one instance of QAManagerFactory.

Members

All members of QAManagerFactory, including all inherited members.

createQAManager function

Synopsis

virtual QAManager * **QAManagerFactory::createQAManager(**
  qa_const_string *iniFile*
)

Parameters

- **iniFile** The path of the properties file.

Remarks

Returns a new QAManager class instance with the specified properties.

If the properties file parameter is null, the QATransactionalManager class is created using default properties. You can use the setProperty function to set QATransactionalManager class properties programmatically after you create the instance.

For a list of QAManagerBase class properties, see "QAnywhere manager configuration properties" on page 62.

See Also

QAManager class

Returns

The QAManager class instance.

createQATransactionalManager function

Synopsis

virtual QATransactionalManager * **QAManagerFactory::createQATransactionalManager(**
  qa_const_string *iniFile*
)

Parameters

- **iniFile** The path of the properties file.

Remarks

Returns a new QAManager class instance with the specified properties.

If the properties file parameter is null, the QAManager class is created using default properties. You can use the setProperty function to set QAManager class properties programmatically after you create the instance.

For a list of QAManagerBase class properties, see "QAnywhere manager configuration properties" on page 62.

See Also

QATransactionalManager class

Returns

The QATransactionalManager class instance.

deleteQAManager function

Synopsis

virtual void **QAManagerFactory::deleteQAManager(**
  QAManager * *mgr*
)

Parameters

- **mgr** The "QAManager class" on page 427 instance to destroy.

Remarks

Destroys a QAManager class, freeing its resources.

It is not necessary to use this method, since all created QAManager's are destroyed when QAnywhereFactory_term() is called. It is provided as a convenience for when it is desirable to free resources in a timely manner. "Shutting down QAnywhere" on page 88

deleteQATransactionalManager function

Synopsis

virtual void **QAManagerFactory::deleteQATransactionalManager(**
QATransactionalManager * *mgr*
)

Parameters

- **mgr** The "QATransactionalManager class" on page 492 instance to destroy.

Remarks

Destroys a QATransactionalManager class, freeing its resources.

It is not necessary to use this method, since all created QATransactionalManager's are destroyed when QAnywhereFactory_term() is called. It is provided as a convenience for when it is desirable to free resources in a timely manner.

For more information, see "Shutting down QAnywhere" on page 88

getLastError function

Synopsis

virtual qa_int **QAManagerFactory::getLastError()**

Remarks

The error code associated with the last excecuted QAManagerFactory method.

0 indicates no error.

For a list of values, see the QAError class.

See Also

getLastErrorMsg function

QAError class

Returns

The error code.

getLastErrorMsg function

Synopsis

virtual qa_string **QAManagerFactory::getLastErrorMsg()**

Remarks

The error text associated with the last executed QAManagerFactory method.

This method returns null if getLastError function returns 0. You can retrieve this property after catching a QAError class.

See Also

getLastError function

QAError class

Returns

The error message.

QAMessage class

Synopsis

public **QAMessage**

Derived classes

- "QABinaryMessage class" on page 404
- "QATextMessage class" on page 488

Remarks

QAMessage provides an interface to set message properties and header fields.

The derived classes QABinaryMessage class and QATextMessage class provide specialized functions to read and write to the message body. You can use QAMessage functions to set predefined or custom message properties.

For a list of pre-defined property names, see the MessageProperties class.

For more information about setting message properties and header fields, see "Message headers and message properties" on page 206.

Members

All members of QAMessage, including all inherited members.

- "beginEnumPropertyNames function" on page 468
- "castToBinaryMessage function" on page 468
- "castToTextMessage function" on page 468
- "clearProperties function" on page 469
- "DEFAULT_PRIORITY variable" on page 467
- "DEFAULT_TIME_TO_LIVE variable" on page 467
- "endEnumPropertyNames function" on page 469
- "getAddress function" on page 469
- "getBooleanProperty function" on page 470
- "getByteProperty function" on page 470
- "getDoubleProperty function" on page 471
- "getExpiration function" on page 471
- "getFloatProperty function" on page 472
- "getInReplyToID function" on page 472
- "getIntProperty function" on page 473
- "getLongProperty function" on page 473
- "getMessageID function" on page 474
- "getPriority function" on page 474
- "getPropertyType function" on page 475
- "getRedelivered function" on page 475
- "getReplyToAddress function" on page 476
- "getShortProperty function" on page 476
- "getStringProperty function" on page 477

DEFAULT_PRIORITY variable

Synopsis

const qa_int **QAMessage::DEFAULT_PRIORITY**

Remarks

The default message priority.

This value is 4. This is normal priority as values 0-4 are gradations of normal priority and values 5-9 are gradations of expedited priority.

DEFAULT_TIME_TO_LIVE variable

Synopsis

const qa_long **QAMessage::DEFAULT_TIME_TO_LIVE**

Remarks

The default message time-to-live value.

This value is 0, which indicates that the message does not expire.

beginEnumPropertyNames function

Synopsis

virtual qa_property_enum_handle **QAMessage::beginEnumPropertyNames()**

Remarks

Begins an enumeration of message property names.

The handle returned by this method is supplied to nextPropertyName. This method and nextPropertyName can be used to enumerate the message property names at the time this method was called. Message properties cannot be set between beginEnumPropertyNames and endEnumPropertyNames.

Returns

A handle that is supplied to nextPropertyName.

castToBinaryMessage function

Synopsis

virtual QABinaryMessage * **QAMessage::castToBinaryMessage()**

Remarks

Casts this QAMessage to a QABinaryMessage class.

You can also use the conversion operator to convert this QAMessage to a QABinaryMessage class.

To convert a QAMessage to a QABinaryMessage class using the conversion operator, do the following:

```
QAMessage *msg;
QABinaryMessage *bmsg;
...
bmsg = (QABinaryMessage *)(*msg);
```

Returns

A pointer to the QABinaryMessage class, or NULL if this message is not an instance of QABinaryMessage class.

castToTextMessage function

Synopsis

virtual QATextMessage * **QAMessage::castToTextMessage()**

Remarks

Casts this QAMessage to a QATextMessage class.

You can also use the conversion operator to convert this QAMessage to a QATextMessage class.

For example, to convert a QAMessage to a QATextMessage class using the conversion operator, do the following:

```
QAMessage *msg;
QATextMessage *bmsg;
...
bmsg = (QATextMessage *)(*msg);
```

Returns

A pointer to the QATextMessage class, or NULL if this message is not an instance of QATextMessage class.

clearProperties function

Synopsis

virtual void **QAMessage::clearProperties()**

Remarks

Clears a message's properties.

Note: The message's header fields and body are not cleared.

endEnumPropertyNames function

Synopsis

virtual void **QAMessage::endEnumPropertyNames(**
qa_property_enum_handle *h*
)

Parameters

- **h** A handle returned by beginEnumPropertyNames.

Remarks

Frees the resources associated with a message property name enumeration.

getAddress function

Synopsis

virtual qa_const_string **QAMessage::getAddress()**

Remarks

Gets the destination address for the QAMessage instance.

When a message is sent, this field is ignored. After completion of the send method, the field holds the destination address specified in putMessage function.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Returns

The destination address.

getBooleanProperty function

Synopsis

```
virtual qa_bool QAMessage::getBooleanProperty(
  qa_const_string name,
  qa_bool * value
)
```

Parameters

- **name** The name of the property to get.
- **value** The destination for the qa_bool value.

Remarks

Gets the value of the qa_bool property with the specified name.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

MessageProperties class

Returns

True if and only if the operation succeeded.

getByteProperty function

Synopsis

```
virtual qa_bool QAMessage::getByteProperty(
  qa_const_string name,
  qa_byte * value
)
```

Parameters

- **name** The name of the property to get.
- **value** The destination for the qa_byte value.

Remarks

Gets the value of the qa_byte property with the specified name.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

MessageProperties class

Returns

True if and only if the operation succeeded.

getDoubleProperty function

Synopsis

virtual qa_bool **QAMessage::getDoubleProperty(**
  qa_const_string *name*,
  qa_double * *value*
)

Parameters

- **name** The name of the property to get.
- **value** The destination for the qa_double value.

Remarks

Gets the value of the qa_double property with the specified name.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

MessageProperties class

Returns

True if and only if the operation succeeded.

getExpiration function

Synopsis

virtual qa_long **QAMessage::getExpiration()**

Remarks

Gets the message's expiration time.

When a message is sent, the Expiration header field is left unassigned. After the send method completes, the Expiration header holds the expiration time of the message.

This property is read-only because the expiration time of a message is set by adding the time-to-live argument of QAManagerBase::PutMessageTimeToLive to the current time.

The expiration time is in units that are natural for the platform. For Windows/PocketPC platforms, expiration is a SYSTEMTIME, converted to a FILETIME, which is copied to an qa_long value.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Returns

The expiration time.

See Also

getTimestamp function

getFloatProperty function

Synopsis

virtual qa_bool **QAMessage::getFloatProperty(**
  qa_const_string *name*,
  qa_float * *value*
)

Parameters

- **name** The name of the property to get.
- **value** The destination for the qa_float value.

Remarks

Gets the value of the qa_float property with the specified name.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

MessageProperties class

Returns

True if and only if the operation succeeded.

getInReplyToID function

Synopsis

virtual qa_const_string **QAMessage::getInReplyToID()**

Remarks

Gets the ID of the message that this message is in reply to.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Returns

The In-Reply-To ID.

getIntProperty function

Synopsis

```
virtual qa_bool QAMessage::getIntProperty(
  qa_const_string name,
  qa_int * value
)
```

Parameters

- **name** The name of the property to get.
- **value** The destination for the qa_int value.

Remarks

Gets the value of the qa_int property with the specified name.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

MessageProperties class

Returns

True if and only if the operation succeeded.

getLongProperty function

Synopsis

```
virtual qa_bool QAMessage::getLongProperty(
  qa_const_string name,
  qa_long * value
)
```

Parameters

- **name** The name of the property to get.
- **value** The destination for the qa_long value.

Remarks

Gets the value of the qa_long property with the specified name.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

MessageProperties class

Returns

True if and only if the operation succeeded.

getMessageID function

Synopsis

virtual qa_const_string **QAMessage::getMessageID()**

Remarks

Gets the message ID.

The MessageID header field contains a value that uniquely identifies each message sent by the QAnywhere client.

When a message is sent using putMessage function, the MessageID header is null and can be ignored. When the send method returns, it contains an assigned value.

A MessageID is a qa_string value that should function as a unique key for identifying messages in a historical repository.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Returns

The message ID.

getPriority function

Synopsis

virtual qa_int **QAMessage::getPriority()**

Remarks

Gets the message priority level.

The QAnywhere client API defines ten levels of priority value, with 0 as the lowest priority and 9 as the highest. Clients should consider priorities 0-4 as gradations of normal priority and priorities 5-9 as gradations of expedited priority.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Returns

The message priority.

getPropertyType function

Synopsis

virtual qa_short **QAMessage::getPropertyType(**
  qa_const_string *name*
)

Parameters

- **name** The name of the property.

Remarks

Returns the type of a property with the given name.

One of PROPERTY_TYPE_BOOLEAN, PROPERTY_TYPE_BYTE, PROPERTY_TYPE_SHORT, PROPERTY_TYPE_INT, PROPERTY_TYPE_LONG, PROPERTY_TYPE_FLOAT, PROPERTY_TYPE_DOUBLE, PROPERTY_TYPE_STRING, PROPERTY_TYPE_UNKNOWN.

Returns

The type of the property.

getRedelivered function

Synopsis

virtual qa_bool **QAMessage::getRedelivered()**

Remarks

Indicates whether the message has been previously received but not acknowledged.

The Redelivered header is set by a receiving QAManager class when it detects that a message being received was received before.

For example, an application receives a message using a QAManager class opened with EXPLICIT_ACKNOWLEDGEMENT variable, and shuts down without acknowledging the message. When the application starts again and receives the same message the Redelivered header is true.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Returns

True if and only if the message was redelivered.

getReplyToAddress function

Synopsis

virtual qa_const_string **QAMessage::getReplyToAddress()**

Remarks

Gets the address to which a reply to this message should be sent.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Returns

The reply-to address.

getShortProperty function

Synopsis

virtual qa_bool **QAMessage::getShortProperty(**
  qa_const_string *name*,
  qa_short * *value*
)

Parameters

- **name** The name of the property to get.
- **value** The destination for the qa_short value.

Remarks

Gets the value of the qa_short property with the specified name.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

MessageProperties class

Returns

True if and only if the operation succeeded.

getStringProperty function

Synopsis

virtual qa_int **QAMessage::getStringProperty(**
  qa_const_string *name*,
  qa_string *dest*,
  qa_int *maxlen*
)

Parameters

- **name** The name of the property to get.
- **dest** The destination for the qa_string value.
- **maxlen** The maximum number of qa_chars of the value to copy. This value includes the null terminator qa_char.

Remarks

Gets the value of the qa_string property with the specified name.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

MessageProperties class

Returns

The number of non-null qa_chars actually copied, or -1 if the operation failed.

getStringProperty function

Synopsis

virtual qa_int **QAMessage::getStringProperty(**
  qa_const_string *name*,
  qa_int *offset*,
  qa_string *dest*,
  qa_int *maxlen*
)

Parameters

- **name** The name of the property to get.
- **offset** The starting offset into the property value from which to copy.
- **dest** The destination for the qa_string value.
- **maxlen** The maximum number of qa_chars of the value to copy. This value includes the null terminator qa_char.

Remarks

Gets the value of the qa_string property (starting at offset) with the specified name.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

MessageProperties class

Returns

The number of non-null qa_chars actually copied, or -1 if the operation failed.

getTimestamp function

Synopsis

virtual qa_long **QAMessage::getTimestamp()**

Remarks

Gets the message timestamp.

This Timestamp header field contains the time a message was created. It is a coordinated universal time (UTC). It is not the time the message was actually transmitted, because the actual send may occur later due to transactions or other client-side queuing of messages. It is in units that are natural for the platform. For Windows/PocketPC platforms, the timestamp is a SYSTEMTIME, converted to a FILETIME, which is copied to a qa_long value.

To convert a timestamp ts to SYSTEMTIME for displaying to a user, run the following code:

```
SYSTEMTIME stime;
FILETIME   ftime;
ULARGE_INTEGER time;
time.QuadPart = ts;
memcpy(&ftime, &time, sizeof(FILETIME));
FileTimeToSystemTime(&ftime, &stime);
```

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Returns

The message timestamp.

getTimestampAsString function

Synopsis

virtual qa_int **QAMessage::getTimestampAsString(**
  qa_string *buffer*,
  qa_int *bufferLen*
)

Parameters

- **buffer** The buffer for the formatted timestamp.
- **bufferLen** The size of the buffer.

Remarks

Gets the message timestamp as a formatted string.

The format is: "dow, MMM dd, yyyy hh:mm:ss.nnn GMT".

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

Returns

The number of non-null qa_chars written to the buffer.

nextPropertyName function

Synopsis

virtual qa_int **QAMessage::nextPropertyName(**
  qa_property_enum_handle *h*,
  qa_string *buffer*,
  qa_int *bufferLen*
)

Parameters

- **h** A handle returned by beginEnumPropertyNames.
- **buffer** The buffer into which to write the property name.
- **bufferLen** The length of the buffer to store the property name. This length must include space for the null terminator

Remarks

Returns the message property name for the given enumeration, returning -1 if there are no more property names.

Returns

The length of the property name, or -1 if there are no more property names.

propertyExists function

Synopsis

virtual qa_bool **QAMessage::propertyExists(**
  qa_const_string *name*
)

Parameters

- **name** The name of the property.

Remarks

Indicates whether a property value exists.

Returns

True if and only if the property exists.

setAddress function

Synopsis

virtual void **QAMessage::setAddress(**
  qa_const_string *destination*
)

Parameters

- **destination** The destination address.

Remarks

Sets the destination address for this message.

This method can be used to change the value for a message that has been received.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

setBooleanProperty function

Synopsis

virtual void **QAMessage::setBooleanProperty(**
  qa_const_string *name*,
  qa_bool *value*
)

Parameters

- **name** the name of the property to set.

- **value** the qa_bool value of the property.

Remarks

Sets the qa_bool property with the specified name to the specified value.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

MessageProperties class

setByteProperty function

Synopsis

virtual void **QAMessage::setByteProperty(**
  qa_const_string *name*,
  qa_byte *value*
)

Parameters

- **name** The name of the property to set.
- **value** The qa_byte value of the property.

Remarks

Sets a qa_byte property with the specified name to the specified value.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

MessageProperties class

setDoubleProperty function

Synopsis

virtual void **QAMessage::setDoubleProperty(**
  qa_const_string *name*,
  qa_double *value*
)

Parameters

- **name** The name of the property to set.
- **value** The qa_double value of the property.

Remarks

Sets the qa_double property with the specified name to the specified value.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

MessageProperties class.

setFloatProperty function

Synopsis

virtual void **QAMessage::setFloatProperty(**
  qa_const_string *name*,
  qa_float *value*
)

Parameters

- **name** The name of the property to set.
- **value** The qa_float value of the property.

Remarks

Sets the qa_float property with the specified name to the specified value.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

MessageProperties class.

setInReplyToID function

Synopsis

virtual void **QAMessage::setInReplyToID(**
  qa_const_string *id*
)

Parameters

- **id** The In-Reply-To ID.

Remarks

Sets the In-Reply-To ID for the message.

A client can use the InReplyToID header field to link one message with another. A typical use is to link a response message with its request message.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

setIntProperty function

Synopsis

virtual void **QAMessage::setIntProperty(**
  qa_const_string *name*,
  qa_int *value*
)

Parameters

- **name** The name of the property to set.
- **value** The qa_int value of the property.

Remarks

Sets the qa_int property with the specified name to the specified value.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

MessageProperties class.

setLongProperty function

Synopsis

virtual void **QAMessage::setLongProperty(**
  qa_const_string *name*,
  qa_long *value*
)

Parameters

- **name** The name of the property to set.
- **value** The qa_long value of the property.

Remarks

Sets the qa_long property with the specified name to the specified value.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

MessageProperties class.

setMessageID function

Synopsis

virtual void **QAMessage::setMessageID(**
  qa_const_string *id*
)

Parameters

- **id** The message ID.

Remarks

Sets the message ID.

This method can be used to change the value for a message that has been received.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

setPriority function

Synopsis

virtual void **QAMessage::setPriority(**
  qa_int *priority*
)

Parameters

- **priority** The message priority.

Remarks

Sets the priority level for this message.

This method can be used to change the value for a message that has been received.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

setRedelivered function

Synopsis

virtual void **QAMessage::setRedelivered(**
  qa_bool *redelivered*
)

Parameters

- **redelivered** The redelivered indication.

Remarks

Sets an indication of whether this message was redelivered.

This method can be used to change the value for a message that has been received.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

setReplyToAddress function

Synopsis

virtual void **QAMessage::setReplyToAddress(**
  qa_const_string *replyTo*
)

Parameters

- **replyTo** The reply-to address.

Remarks

Sets the address to which a reply to this message should be sent.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

setShortProperty function

Synopsis

virtual void **QAMessage::setShortProperty(**
  qa_const_string *name*,
  qa_short *value*
)

Parameters

- **name** The name of the property to set.
- **value** The qa_short value of the property.

Remarks

Sets e qa_short property with the specified name to the specified value.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

MessageProperties class.

setStringProperty function

Synopsis

virtual void **QAMessage::setStringProperty(**
  qa_const_string *name*,
  qa_const_string *value*
)

Parameters

- **name** The name of the property to set.
- **value** The qa_string value of the property.

Remarks

Sets a qa_string property with the specified name to the specified value.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

MessageProperties class.

setTimestamp function

Synopsis

virtual void **QAMessage::setTimestamp(**
  qa_long *timestamp*
)

Parameters

- **timestamp** The message timestamp, a coordinated universal time (UTC).

Remarks

Sets the message timestamp.

This method can be used to change the value for a message that has been received.

For more information about getting and setting message headers and properties, see "Message headers and message properties" on page 206.

See Also

getTimestamp function

QAMessageListener class

Synopsis

public **QAMessageListener**

Remarks

A QAMessageListener object is used to receive asynchronously delivered messages.

Members

All members of QAMessageListener, including all inherited members.

onMessage function

Synopsis

virtual void **QAMessageListener::onMessage(**
QAMessage * *message*
)

Parameters

- **message** the message passed to the listener

Remarks

Passes a message to the listener.

~QAMessageListener function

Synopsis

virtual **QAMessageListener::~QAMessageListener()**

Remarks

Virtual destructor.

QATextMessage class

Synopsis

public **QATextMessage**

Base classes

- "QAMessage class" on page 466

Remarks

QATextMessage inherits from the QAMessage class and adds a text message body.

QATextMessage provides methods to read from and write to the text message body.

When the message is first created, the body of the message is in write-only mode. After a message has been sent, the client that sent it can retain and modify it without affecting the message that has been sent. The same message object can be sent multiple times.

When a message is received, the provider has called reset function so that the message body is in read-only mode and reading of values starts from the beginning of the message body. If a client attempts to write a message in read-only mode, a COMMON_MSG_NOT_WRITEABLE_ERROR is set.

See Also

QABinaryMessage class

Members

All members of QATextMessage, including all inherited members.

- "beginEnumPropertyNames function" on page 468
- "castToBinaryMessage function" on page 468
- "castToTextMessage function" on page 468
- "clearProperties function" on page 469
- "DEFAULT_PRIORITY variable" on page 467
- "DEFAULT_TIME_TO_LIVE variable" on page 467
- "endEnumPropertyNames function" on page 469
- "getAddress function" on page 469
- "getBooleanProperty function" on page 470
- "getByteProperty function" on page 470
- "getDoubleProperty function" on page 471
- "getExpiration function" on page 471
- "getFloatProperty function" on page 472
- "getInReplyToID function" on page 472
- "getIntProperty function" on page 473
- "getLongProperty function" on page 473
- "getMessageID function" on page 474
- "getPriority function" on page 474
- "getPropertyType function" on page 475
- "getRedelivered function" on page 475

getText function

Synopsis

virtual qa_string **QATextMessage::getText()**

Remarks

Gets the string containing this message's data.

The default value is null.

If the message exceeds the maximum size specified by the QAManager class MAX_IN_MEMORY_MESSAGE_SIZE property, this function returns null. In this case, use the readText function to read the text.

For more information about QAManager class properties, see "QAnywhere manager configuration properties" on page 62.

Returns

the string containing the message's data.

getTextLength function

Synopsis

virtual qa_long **QATextMessage::getTextLength()**

Remarks

Returns the text length.

Note: If the text length is non-zero and getText function returns qa_null then the text does not fit in memory, and must be read in pieces using the readText.

readText function

Synopsis

virtual qa_int **QATextMessage::readText(**
  qa_string *string*,
  qa_int *length*
)

Parameters

- **string** The destination for the text.
- **length** The maximum number of qa_chars to read into the destination. buffer, including the null termination character.

Remarks

Reads the requested length of text from the current text position into a buffer.

Returns

The actual number of non-null qa_chars read, or -1 if the entire text stream has been read.

reset function

Synopsis

virtual void **QATextMessage::reset()**

Remarks

Repositions the current text position to the beginning.

setText function

Synopsis

virtual void **QATextMessage::setText(**
  qa_const_string *string*
)

Parameters

- **string** A string containing the message data to set.

Remarks

Sets the string containing this message's data.

writeText function

Synopsis

virtual void **QATextMessage::writeText(**
  qa_const_string *string*,
  qa_int *offset*,
  qa_int *length*
)

Parameters

- **string** The source text to concatenate.
- **offset** The offset into the source text at which to start reading.
- **length** The number of qa_chars of the source text to read.

Remarks

Concatenates text to the current text.

~QATextMessage function

Synopsis

virtual **QATextMessage::~QATextMessage()**

Remarks

Virtual destructor.

QATransactionalManager class

Synopsis

public **QATransactionalManager**

Base classes

- "QAManagerBase class" on page 432

Remarks

This class is the manager for transactional messaging.

The QATransactionalManager class derives from QAManagerBase class and manages transactional QAnywhere messaging operations.

For a detailed description of derived behavior, see QAManagerBase class.

The QATransactionalManager can only be used for transactional acknowledgement. Use the commit function to commit all putMessage function and getMessage function invocations.

For more information, see "Implementing transactional messaging" on page 68

See Also

QATransactionalManager.

Members

All members of QATransactionalManager, including all inherited members.

- "beginEnumStorePropertyNames function" on page 433
- "browseClose function" on page 434
- "browseMessages function" on page 434
- "browseMessagesByID function" on page 435
- "browseMessagesByQueue function" on page 436
- "browseMessagesBySelector function" on page 436
- "browseNextMessage function" on page 437
- "cancelMessage function" on page 438
- "close function" on page 438
- "commit function" on page 493
- "createBinaryMessage function" on page 439
- "createTextMessage function" on page 439
- "deleteMessage function" on page 440
- "endEnumStorePropertyNames function" on page 440
- "getAllQueueDepth function" on page 440
- "getBooleanStoreProperty function" on page 441
- "getByteStoreProperty function" on page 442
- "getDoubleStoreProperty function" on page 442
- "getFloatStoreProperty function" on page 443
- "getIntStoreProperty function" on page 443

commit function

Synopsis

virtual qa_bool **QATransactionalManager::commit()**

Remarks

Commits the current transaction and begins a new transaction.

This method commits all putMessage function and getMessage function invocations.

Note: The first transaction begins with the call to open function.

See Also

QATransactionalManager

Returns

True if and only if the commit operation was successful.

open function

Synopsis

virtual qa_bool **QATransactionalManager::open()**

Remarks

Opens a QATransactionalManager intance.

The open method must be the first method called after creating a manager.

See Also

QATransactionalManager

Returns

True if and only if the operation was successful.

rollback function

Synopsis

virtual qa_bool **QATransactionalManager::rollback()**

Remarks

Rolls back the current transaction and begins a new transaction.

This method rolls back all uncommited putMessage function and getMessage function invocations.

See Also

QATransactionalManager

Returns

True if and only if the open operation was successful.

~QATransactionalManager function

Synopsis

virtual **QATransactionalManager::~QATransactionalManager()**

Remarks

Virtual destructor.

QueueDepthFilter class

Synopsis

public **QueueDepthFilter**

Remarks

QueueDepthFilter values for Queue depth APIs of QAManagerBase class.

Members

All members of QueueDepthFilter, including all inherited members.

ALL variable

Synopsis

const qa_short **QueueDepthFilter::ALL**

Remarks

Count both incoming and outgoing messages.

System messages and expired messages are not included in any queue depth counts.

INCOMING variable

Synopsis

const qa_short **QueueDepthFilter::INCOMING**

Remarks

Count only incoming messages.

An incoming message is defined as a message whose originator is different than the agent ID of the message store.

OUTGOING variable

Synopsis

const qa_short **QueueDepthFilter::OUTGOING**

Remarks

Count only outgoing messages.

An outgoing message is defined as a message whose originator is the agent ID of the message store, and whose destination is not the agent ID of the message store.

StatusCodes class

Synopsis

public **StatusCodes**

Remarks

This interface defines a set of codes for the status of a message.

Members

All members of StatusCodes, including all inherited members.

CANCELLED variable

Synopsis

const qa_int **StatusCodes::CANCELLED**

Remarks

The message has been cancelled.

This code has value 40. This code applies to the MEssageProperties::STATUS.

EXPIRED variable

Synopsis

const qa_int **StatusCodes::EXPIRED**

Remarks

The message has expired, i.e.

the message was not received before its expiration time passed. This code has value 30. This code applies to the MEssageProperties::STATUS.

FINAL variable

Synopsis

const qa_int **StatusCodes::FINAL**

Remarks

The message has acheived a final state.

This code has value 20. This code applies to the MEssageProperties::STATUS.

LOCAL variable

Synopsis

const qa_int **StatusCodes::LOCAL**

Remarks

The message is addressed to the local message store and will not be transmitted to the server.

This code has value 2. This code applies to the MEssageProperties::TRANSMISSION_STATUS.

PENDING variable

Synopsis

const qa_int **StatusCodes::PENDING**

Remarks

The message has been sent but not received.

This code has value 1. This code applies to the MEssageProperties::STATUS.

RECEIVED variable

Synopsis

const qa_int **StatusCodes::RECEIVED**

Remarks

The message has been received and acknowledged by the receiver.

This code has value 60. This code applies to the MEssageProperties::STATUS.

RECEIVING variable

Synopsis

const qa_int **StatusCodes::RECEIVING**

Remarks

The message is in the process of being received, or it was received but not acknowledged.

This code has value 10. This code applies to the MEssageProperties::STATUS.

TRANSMITTED variable

Synopsis

const qa_int **StatusCodes::TRANSMITTED**

Remarks

The message has been transmitted to the server.

This code has value 1. This code applies to the MEssageProperties::TRANSMISSION_STATUS.

TRANSMITTING variable

Synopsis

const qa_int **StatusCodes::TRANSMITTING**

Remarks

The message is in the process of being transmitted to the server.

This code has value 3. This code applies to the MEssageProperties::TRANSMISSION_STATUS.

UNRECEIVABLE variable

Synopsis

const qa_int **StatusCodes::UNRECEIVABLE**

Remarks

The message has been marked as unreceivable.

The message is either malformed, or there were too many failed attempts to deliver it. This code has value 50. This code applies to the MEssageProperties::STATUS.

UNTRANSMITTED variable

Synopsis

const qa_int **StatusCodes::UNTRANSMITTED**

Remarks

The message has not been transmitted to the server.

This code has value 0. This code applies to the MEssageProperties::TRANSMISSION_STATUS.

CHAPTER 14

QAnywhere Java API Reference

Contents

About this chapter

This chapter describes the QAnywhere Java API.

ianywhere.qanywhere.client package

Interface AcknowledgementMode

Synopsis

public **ianywhere.qanywhere.client.AcknowledgementMode**

Remarks

Indicates how messages should be acknowledged by QAnywhere client applications.

The implicit and explicit acknowledgement modes are assigned to a Interface QAManager instance using the open method.

With implicit acknowledgement, messages are acknowledged as soon as they are received by a client application. With explicit acknowledgement, you must call one of the Interface QAManager acknowledgement methods. The server propagates all status changes from client to client.

See Also

Interface QAManager

Interface QATransactionalManager

Interface QAManagerBase

Members

All members of ianywhere.qanywhere.client.AcknowledgementMode, including all inherited members.

- "EXPLICIT_ACKNOWLEDGEMENT variable" on page 504
- "IMPLICIT_ACKNOWLEDGEMENT variable" on page 505
- "TRANSACTIONAL variable" on page 505

EXPLICIT_ACKNOWLEDGEMENT variable

Synopsis

final short **ianywhere.qanywhere.client.AcknowledgementMode.EXPLICIT_ACKNOWLEDGEMENT**

Remarks

Indicates that received messages are acknowledged using one of the Interface QAManager acknowledge methods.

See Also

Interface QAManager

IMPLICIT_ACKNOWLEDGEMENT variable

Synopsis

final short **ianywhere.qanywhere.client.AcknowledgementMode.IMPLICIT_ACKNOWLEDGEMENT**

Remarks

Indicates that all messages are acknowledged as soon as they are received by a client application.

If you receive messages synchronously, messages are acknowledged as soon as the getMessage method returns. If you receive messages asynchronously, the message is acknowledged as soon as the event handling function returns.

TRANSACTIONAL variable

Synopsis

final short **ianywhere.qanywhere.client.AcknowledgementMode.TRANSACTIONAL**

Remarks

This mode indicates that messages are only acknowledged as part of the on going transaction.

This mode is automatically assigned to Interface QATransactionalManager instances.

See Also

Interface QATransactionalManager

Interface MessageProperties

Synopsis

public **ianywhere.qanywhere.client.MessageProperties**

Remarks

Provides fields storing standard message property names.

The Interface MessageProperties class provides standard message property names. You can pass Interface MessageProperties fields to Interface QAMessage methods used to get and set message properties.

For example, assume you have the following Interface QAMessage instance: `QAMessage msg = mgr.createTextMessage();`

The following example gets the value corresponding to MSG_TYPE variable using the getIntProperty method. The Interface MessageType enumeration maps the integer result to an appropriate message type.

```
int msg_type = t_msg.getIntProperty( MessageProperties.MSG_TYPE );
```

The following example shows the onSystemMessage(QAMessage) method, which is used to handle QAnywhere system messages.


```
 private void onSystemMessage(QAMessage msg) {
   QATextMessage    t_msg;
   int              msg_type;
   String           network_adapters;
   String           network_names;
   String           network_info;

   t_msg = (QATextMessage)msg;
   if( t_msg != null ) {
       // Evaluate the message type.
       msg_type = (MessageType)t_msg.getIntProperty
( MessageProperties.MSG_TYPE );
       if( msg_type == MessageType.NETWORK_STATUS_NOTIFICATION ) {
           // Handle network status notification.
           network_info = "";
           network_adapters = t_msg.getStringProperty
( MessageProperties.ADAPTERS );
           if( network_adapters != null && network_adapters.length > 0) {
             network_info += network_adapters;
           }
           network_names = t_msg.getStringProperty
( MessageProperties.RASNAMES );

            //...
       }
   }
 }
```

Members

All members of ianywhere.qanywhere.client.MessageProperties, including all inherited members.

ADAPTER variable

Synopsis

final String **ianywhere.qanywhere.client.MessageProperties.ADAPTER**

Remarks

For "system" queue messages, the network adapter that is being used to connect to the QAnywhere server.

The value of this field is "ias_Network.Adapter".

You can pass ADAPTER variable in the getStringProperty method to access the associated property. This property is read-only.

See Also

Interface MessageProperties

ADAPTERS variable

Synopsis

final String **ianywhere.qanywhere.client.MessageProperties.ADAPTERS**

Remarks

This property name refers to a delimited list of network adapters that can be used to connect to the QAnywhere server.

It is used for system queue messages.

You can pass ADAPTERS variable in the getStringProperty method to access the associated property. This property is read-only.

See Also

Interface MessageProperties

DELIVERY_COUNT variable

Synopsis

final String **ianywhere.qanywhere.client.MessageProperties.DELIVERY_COUNT**

Remarks

This property name refers to the number of attempts that have been made so far to deliver the message.

IP variable

Synopsis

final String **ianywhere.qanywhere.client.MessageProperties.IP**

Remarks

For "system" queue messages, the IP address of the network adapter that is being used to connect to the QAnywhere server.

The value of this field is "ias_Network.IP".

You can pass IP variable in the getStringProperty method to access the associated property. This property is read-only.

See Also

Interface MessageProperties

MAC variable

Synopsis

final String **ianywhere.qanywhere.client.MessageProperties.MAC**

Remarks

For "system" queue messages, the MAC address of the network adapter that is being used to connect to the QAnywhere server.

The value of this field is "ias_Network.MAC".

You can pass MAC variable in the getStringProperty method to access the associated property. This property is read-only.

See Also

Interface MessageProperties

MSG_TYPE variable

Synopsis

final String **ianywhere.qanywhere.client.MessageProperties.MSG_TYPE**

Remarks

This property name refers to Interface MessageType enumeration values associated with a QAnywhere message.

The value of this field is "ias_MessageType".

You can pass MSG_TYPE variable in the getIntProperty method to access the associated property. This property is read-only.

See Also

Interface MessageProperties

NETWORK_STATUS variable

Synopsis

final String **ianywhere.qanywhere.client.MessageProperties.NETWORK_STATUS**

Remarks

This property name refers to the state of the network connection.

The value is 1 if the network is accessible and 0 otherwise. The network status is used for system queue messages (for example, network status changes).

You can pass NETWORK_STATUS variable in the getIntProperty method to access the associated property. This property is read-only.

See Also

Interface MessageProperties

ORIGINATOR variable

Synopsis

final String **ianywhere.qanywhere.client.MessageProperties.ORIGINATOR**

Remarks

This property name refers to the message store ID of the originator of the message.

RAS variable

Synopsis

final String **ianywhere.qanywhere.client.MessageProperties.RAS**

Remarks

For "system" queue messages, the RAS entry name that is being used to connect to the QAnywhere server.

The value of this field is "ias_Network.RAS".

You can pass RAS variable in the getStringProperty method to access the associated property. This property is read-only.

See Also

Interface MessageProperties

RASNAMES variable

Synopsis

final String **ianywhere.qanywhere.client.MessageProperties.RASNAMES**

Remarks

For "system" queue messages, a delimited list of RAS entry names that can be used to connect to the QAnywhere server.

The value of this field is "ias_RASNames".

You can pass RASNAMES variable in the getStringProperty method to access the associated property. This property is read-only.

See Also

Interface MessageProperties

STATUS variable

Synopsis

final String **ianywhere.qanywhere.client.MessageProperties.STATUS**

Remarks

This property name refers to the current status of the message.

See Also

Interface StatusCodes

STATUS_TIME variable

Synopsis

final String **ianywhere.qanywhere.client.MessageProperties.STATUS_TIME**

Remarks

This property name refers to the time at which the message assumed its current status.

If you pass MessageProperties.StatusTime to the getProperty method, it returns a java.util.Date instance.

TRANSMISSION_STATUS variable

Synopsis

final String **ianywhere.qanywhere.client.MessageProperties.TRANSMISSION_STATUS**

Remarks

This property name refers to the current transmission status of the message.

See Also

Interface StatusCodes

Interface MessageStoreProperties

Synopsis

public **ianywhere.qanywhere.client.MessageStoreProperties**

Remarks

This class defines constant values for useful message store property names.

The Interface MessageStoreProperties class provides standard message property names. You can pass Interface MessageProperties fields to Interface QAManagerBase methods used to get and set pre-defined or custom message store properties.

Members

All members of ianywhere.qanywhere.client.MessageStoreProperties, including all inherited members.

- "MAX_DELIVERY_ATTEMPTS variable" on page 511

MAX_DELIVERY_ATTEMPTS variable

Synopsis

final String **ianywhere.qanywhere.client.MessageStoreProperties.MAX_DELIVERY_ATTEMPTS**

Remarks

This property name refers to the maximum number of times that a message can be received without being acknowledged before its status is set to UNRECEIVABLE variable.

Interface MessageType

Synopsis

public **ianywhere.qanywhere.client.MessageType**

Remarks

Defines constant values for the MSG_TYPE variable message property.

The following example shows the onSystemMessage(QAMessage) method, which is used to handle QAnywhere system messages. The message type is compared to NETWORK_STATUS_NOTIFICATION variable.

```
private void onSystemMessage(QAMessage msg)
{
  QATextMessage    t_msg;
  int              msg_type;
  String           network_adapters;
  String           network_names;
  String           network_info;

  t_msg = (QATextMessage)msg;
  if( t_msg != null )
  {
      // Evaluate message type.
      msg_type = t_msg.getIntProperty( MessageProperties.MSG_TYPE );
      if( msg_type == MessageType.NETWORK_STATUS_NOTIFICATION )
      {
          // Handle network status notification.

      }
  }
}
```


Members

All members of ianywhere.qanywhere.client.MessageType, including all inherited members.

NETWORK_STATUS_NOTIFICATION variable

Synopsis

final int **ianywhere.qanywhere.client.MessageType.NETWORK_STATUS_NOTIFICATION**

Remarks

Identifies a QAnywhere system message used to notify QAnywhere client applications of network status changes.

Network status changes apply to the device receiving the system message. Use the ADAPTER variable, MessageProperties.NETWORK,and NETWORK_STATUS variable fields to identify new network status information.

PUSH_NOTIFICATION variable

Synopsis

final int **ianywhere.qanywhere.client.MessageType.PUSH_NOTIFICATION**

Remarks

Identifies a QAnywhere system message used to notify QAnywhere client applications of push notifications.

If you use the on-demand QAnywhere Agent policy, a typical response is to call the triggerSendReceive method to receive messages waiting with the central message server.

REGULAR variable

Synopsis

final int **ianywhere.qanywhere.client.MessageType.REGULAR**

Remarks

If no message type property exists, the message type is assumed to be REGULAR.

This type of message is not treated specially by the message system.

Interface PropertyType

Synopsis

public **ianywhere.qanywhere.client.PropertyType**

Remarks

Interface QAMessage property type enumeration, corresponding naturally to the Java types.

Members

All members of ianywhere.qanywhere.client.PropertyType, including all inherited members.

PROPERTY_TYPE_BOOLEAN variable

Synopsis

final short **ianywhere.qanywhere.client.PropertyType.PROPERTY_TYPE_BOOLEAN**

Remarks

Indicates a boolean property.

PROPERTY_TYPE_BYTE variable

Synopsis

final short **ianywhere.qanywhere.client.PropertyType.PROPERTY_TYPE_BYTE**

Remarks

Indicates a signed byte property.

PROPERTY_TYPE_DOUBLE variable

Synopsis

final short **ianywhere.qanywhere.client.PropertyType.PROPERTY_TYPE_DOUBLE**

Remarks

Indicates a double property.

PROPERTY_TYPE_FLOAT variable

Synopsis

final short **ianywhere.qanywhere.client.PropertyType.PROPERTY_TYPE_FLOAT**

Remarks

Indicates a float property.

PROPERTY_TYPE_INT variable

Synopsis

final short **ianywhere.qanywhere.client.PropertyType.PROPERTY_TYPE_INT**

Remarks

Indicates an int property.

PROPERTY_TYPE_LONG variable

Synopsis

final short **ianywhere.qanywhere.client.PropertyType.PROPERTY_TYPE_LONG**

Remarks

Indicates an long property.

PROPERTY_TYPE_SHORT variable

Synopsis

final short **ianywhere.qanywhere.client.PropertyType.PROPERTY_TYPE_SHORT**

Remarks

Indicates a short property.

PROPERTY_TYPE_STRING variable

Synopsis

final short **ianywhere.qanywhere.client.PropertyType.PROPERTY_TYPE_STRING**

Remarks

Indicates a String property.

PROPERTY_TYPE_UNKNOWN variable

Synopsis

final short **ianywhere.qanywhere.client.PropertyType.PROPERTY_TYPE_UNKNOWN**

Remarks

Indicates an unknown property type, usually because the property is unknown.

Interface QABinaryMessage

Synopsis

public **ianywhere.qanywhere.client.QABinaryMessage**

Base classes

- "Interface QAMessage" on page 570

Remarks

A Interface QABinaryMessage object is used to send a message containing a stream of uninterpreted bytes.

Interface QABinaryMessage inherits from the Interface QAMessage class and adds a bytes message body. Interface QABinaryMessage provides a variety of functions to read from and write to the bytes message body.

When the message is first created, the body of the message is in write-only mode. After a message has been sent, the client that sent it can retain and modify it without affecting the message that has been sent. The same message object can be sent multiple times.

When a message is received, the provider has called reset method so that the message body is in read-only mode and reading of values starts from the beginning of the message body.

The following example uses the writeString method to write the string "Q" followed by the string "Anywhere" to a Interface QABinaryMessage instance's message body.

```
// Create a binary message instance.
QABinaryMessage binary_message;
binary_message = qa_manager.createBinaryMessage();

// Set optional message properties.
binary_message.setReplyToAddress("my-queue-name");

// Write to the message body.
binary_message.writeString("Q");
binary_messge.writeString("Anywhere");

// Put the message in the local database, ready for sending.
try {
    qa_manager.putMessage( "store-id\\queue-name", binary_message );
}
catch ( QAException e ) {
    handleError();
}
```

Note: On the receiving end, the first readString method invocation returns "Q" and the next readString method invocation returns "Anywhere".

The message is sent by the QAnywhere Agent.

Members

All members of ianywhere.qanywhere.client.QABinaryMessage, including all inherited members.

- "setPriority method" on page 586
- "setProperty method" on page 586
- "setReplyToAddress method" on page 587
- "setShortProperty method" on page 587
- "setStringProperty method" on page 588
- "writeBinary method" on page 523
- "writeBinary method" on page 524
- "writeBinary method" on page 524
- "writeBoolean method" on page 525
- "writeByte method" on page 526
- "writeChar method" on page 526
- "writeDouble method" on page 527
- "writeFloat method" on page 527
- "writeInt method" on page 528
- "writeLong method" on page 528
- "writeShort method" on page 529
- "writeString method" on page 529

getBodyLength method

Synopsis

long **ianywhere.qanywhere.client.QABinaryMessage.getBodyLength()**
throws **QAException**

Throws

- Thrown if there is a problem retrieving the size of the message body.

Remarks

Returns the size of the message body in bytes.

See Also

Interface QABinaryMessage

Returns

The size of the message body in bytes.

readBinary method

Synopsis

int **ianywhere.qanywhere.client.QABinaryMessage.readBinary(**
  byte[] *dest*
)
throws **QAException**

Parameters

- **dest** The byte array to hold the read bytes.

Throws

- Thrown if there was a conversion error reading the value or if there is no more input.

Remarks

Reads a specified number of bytes starting from the unread portion of a Interface QABinaryMessage instance body.

See Also

Interface QABinaryMessage

writeBinary method

Returns

The number of bytes read from the message body.

readBinary method

Synopsis

```
int ianywhere.qanywhere.client.QABinaryMessage.readBinary(
  byte[] dest,
  int length
)
throws QAException
```

Parameters

- **dest** The byte array to hold the read bytes.
- **length** The maximum number of bytes to read.

Throws

- Thrown if there was a conversion error reading the value or if there is no more input.

Remarks

Reads a specified number of bytes starting from the unread portion of a Interface QABinaryMessage instance body.

See Also

Interface QABinaryMessage

writeBinary method

Returns

The number of bytes read from the message body.

readBoolean method

Synopsis

boolean **ianywhere.qanywhere.client.QABinaryMessage.readBoolean()**
throws **QAException**

Throws

- Thrown if there was a conversion error reading the value or if there is no more input.

Remarks

Reads a boolean value starting from the unread portion of the Interface QABinaryMessage instance's message body.

See Also

Interface QABinaryMessage

writeBoolean method

Returns

The boolean value read from the message body.

readByte method

Synopsis

byte **ianywhere.qanywhere.client.QABinaryMessage.readByte()**
throws **QAException**

Throws

- Thrown if there was a conversion error reading the value or if there is no more input.

Remarks

Reads a signed byte value starting from the unread portion of a Interface QABinaryMessage message body.

See Also

Interface QABinaryMessage

writeByte method

Returns

The signed byte value read from the message body.

readChar method

Synopsis

char **ianywhere.qanywhere.client.QABinaryMessage.readChar()**
throws **QAException**

Throws

- Thrown if there was a conversion error reading the value or if there is no more input.

Remarks

Reads a char value starting from the unread portion of a Interface QABinaryMessage message body.

See Also

Interface QABinaryMessage

writeChar method

Returns

The character value read from the message body.

readDouble method

Synopsis

double **ianywhere.qanywhere.client.QABinaryMessage.readDouble()**
throws **QAException**

Throws

- Thrown if there was a conversion error reading the value or if there is no more input.

Remarks

Reads a double value starting from the unread portion of a Interface QABinaryMessage message body.

See Also

Interface QABinaryMessage

writeDouble method

Returns

The double value read from the message body.

readFloat method

Synopsis

float **ianywhere.qanywhere.client.QABinaryMessage.readFloat()**
throws **QAException**

Throws

- Thrown if there was a conversion error reading the value or if there is no more input.

Remarks

Reads a float value starting from the unread portion of a Interface QABinaryMessage message body.

See Also

Interface QABinaryMessage

writeFloat method

Returns

The float value read from the message body.

readInt method

Synopsis

int **ianywhere.qanywhere.client.QABinaryMessage.readInt()**
throws **QAException**

Throws

- Thrown if there was a conversion error reading the value or if there is no more input.

Remarks

Reads an integer value starting from the unread portion of a Interface QABinaryMessage message body.

See Also

Interface QABinaryMessage

writeInt method

Returns

The int value read from the message body.

readLong method

Synopsis

long **ianywhere.qanywhere.client.QABinaryMessage.readLong()**
throws **QAException**

Throws

- Thrown if there was a conversion error reading the value or if there is no more input.

Remarks

Reads a long value starting from the unread portion of a Interface QABinaryMessage message body.

See Also

Interface QABinaryMessage

writeLong method

Returns

The long value read from the message body.

readShort method

Synopsis

short **ianywhere.qanywhere.client.QABinaryMessage.readShort()**
throws **QAException**

Throws

- Thrown if there was a conversion error reading the value or if there is no more input.

Remarks

Reads a short value starting from the unread portion of a Interface QABinaryMessage message body.

See Also

Interface QABinaryMessage

writeShort method

Returns

The short value read from the message body.

readString method

Synopsis

String **ianywhere.qanywhere.client.QABinaryMessage.readString()**
throws **QAException**

Throws

- Thrown if there was a conversion error reading the value or if there is no more input.

Remarks

Reads a string value starting from the unread portion of a Interface QABinaryMessage message body.

See Also

Interface QABinaryMessage

writeString method

Returns

The string value read from the message body.

reset method

Synopsis

void **ianywhere.qanywhere.client.QABinaryMessage.reset()**
throws **QAException**

Throws

- Thrown if there is a problem resetting the message.

Remarks

Resets a message so that the reading of values starts from the beginning of the message body.

The reset method also puts the Interface QABinaryMessage message body in read-only mode.

See Also

Interface QABinaryMessage

writeBinary method

Synopsis

void **ianywhere.qanywhere.client.QABinaryMessage.writeBinary(**
byte[] *val*
)
throws **QAException**

Parameters

- **val** The byte array value to write to the message body.

Throws

- Thrown if there is a problem appending the byte array to the message body.

Remarks

Appends a byte array value to the Interface QABinaryMessage instance's message body.

See Also

Interface QABinaryMessage

readBinary method

writeBinary method

Synopsis

void **ianywhere.qanywhere.client.QABinaryMessage.writeBinary(**
byte[] *val*,
int *len*
)
throws **QAException**

Parameters

- **val** The byte array value to write to the message body.
- **len** The number of bytes to write.

Throws

- Thrown if there is a problem appending the byte array to the message body.

Remarks

Appends a byte array value to the Interface QABinaryMessage instance's message body.

See Also

Interface QABinaryMessage

readBinary method

writeBinary method

Synopsis

void **ianywhere.qanywhere.client.QABinaryMessage.writeBinary(**
byte[] *val*,

```
  int offset,
  int len
)
throws QAException
```

Parameters

- **val** The byte array value to write to the message body.
- **offset** The offset within the byte array to begin writing.
- **len** The number of bytes to write.

Throws

- Thrown if there is a problem appending the byte array to the message body.

Remarks

Appends a byte array value to the Interface QABinaryMessage instance's message body.

See Also

Interface QABinaryMessage

readBinary method

writeBoolean method

Synopsis

```
void ianywhere.qanywhere.client.QABinaryMessage.writeBoolean(
  boolean val
)
throws QAException
```

Parameters

- **val** The boolean value to write to the message body.

Throws

- Thrown if there is a problem appending the boolean value to the message body.

Remarks

Appends a boolean value to the Interface QABinaryMessage instance's message body.

The boolean is represented as a one byte value. True is represented as 1; false is represented as 0.

See Also

Interface QABinaryMessage

readBoolean method

writeByte method

Synopsis

void **ianywhere.qanywhere.client.QABinaryMessage.writeByte(**
byte *val*
)
throws **QAException**

Parameters

- **val** The signed byte value to write to the message body.

Throws

- Thrown if there is a problem appending the signed byte value to the message body.

Remarks

Appends a signed byte value to the Interface QABinaryMessage instance's message body.

The signed byte is represented as a one byte value.

See Also

Interface QABinaryMessage

readByte method

writeChar method

Synopsis

void **ianywhere.qanywhere.client.QABinaryMessage.writeChar(**
char *val*
)
throws **QAException**

Parameters

- **val** The char value to write to the message body.

Throws

- Thrown if there is a problem appending the char value to the message body.

Remarks

Appends a char value to the Interface QABinaryMessage instance's message body.

The char is represented as a two byte value and the high order byte is appended first.

See Also

Interface QABinaryMessage

readChar method

writeDouble method

Synopsis

void **ianywhere.qanywhere.client.QABinaryMessage.writeDouble(**
double *val*
)
throws **QAException**

Parameters

- **val** the double value to write to the message body.

Throws

- Thrown if there is a problem appending the double value to the message body.

Remarks

Appends a double value to the Interface QABinaryMessage instance's message body.

The double is converted to a representative 8-byte long and higher order bytes are appended first.

See Also

Interface QABinaryMessage

readDouble method

writeFloat method

Synopsis

void **ianywhere.qanywhere.client.QABinaryMessage.writeFloat(**
float *val*
)
throws **QAException**

Parameters

- **val** The float value to write to the message body.

Throws

- Thrown if there is a problem appending the float value to the message body.

Remarks

Appends a float value to the Interface QABinaryMessage instance's message body.

The float is converted to a representative 4-byte integer and the higher order bytes are appended first.

See Also

Interface QABinaryMessage

readFloat method

writeInt method

Synopsis

void **ianywhere.qanywhere.client.QABinaryMessage.writeInt(**
int *val*
)
throws **QAException**

Parameters

- **val** The int value to write to the message body.

Throws

- Thrown if there is a problem appending the integer value to the message body.

Remarks

Appends an integer value to the Interface QABinaryMessage instance's message body.

The integer parameter is represented as a 4 byte value and higher order bytes are appended first.

See Also

Interface QABinaryMessage

readInt method

writeLong method

Synopsis

void **ianywhere.qanywhere.client.QABinaryMessage.writeLong(**
long *val*
)
throws **QAException**

Parameters

- **val** The long value to write to the message body.

Throws

- Thrown if there is a problem appending the long value to the message body.

Remarks

Appends a long value to the Interface QABinaryMessage instance's message body.

The long parameter is represented using 8-bytes value and higher order bytes are appended first.

See Also

Interface QABinaryMessage

readLong method

writeShort method

Synopsis

void **ianywhere.qanywhere.client.QABinaryMessage.writeShort(**
short *val*
)
throws **QAException**

Parameters

- **val** The short value to write to the message body.

Throws

- Thrown if there is a problem appending the short value to the message body.

Remarks

Appends a short value to the Interface QABinaryMessage instance's message body.

The short parameter is represented as a two byte value and the higher order byte is appended first.

See Also

Interface QABinaryMessage

readShort method

writeString method

Synopsis

void **ianywhere.qanywhere.client.QABinaryMessage.writeString(**
String *val*
)
throws **QAException**

Parameters

- **val** The string value to write to the message body.

Throws

- Thrown if there is a problem appending the string value to the message body.

Remarks

Appends a string value to the Interface QABinaryMessage instance's message body.

Note: The receiving application needs to invoke readString method for each WriteString invocation. *Note:* The UTF-8 representation of the string to be written can be at most 32767 bytes.

See Also

Interface QABinaryMessage

readString method

Class QAException

Synopsis

public **ianywhere.qanywhere.client.QAException**

Remarks

Encapsulates QAnywhere client application exceptions.

You can use the Class QAException class to catch QAnywhere exceptions.

```
    try
    {
        _qaManager = QAManagerFactory.getInstance().CreateQAManager();
        _qaManager.open( AcknowledgementMode.EXPLICIT_ACKNOWLEDGEMENT );
        _qaManager.start();
    }
    catch( QAException e )
    {
        // Handle exception.
        System.err.println("Error code: " + e.getErrorCode() );
        System.err.println("Error message: " + e.getMessage() );
    }
}
```

Members

All members of ianywhere.qanywhere.client.QAException, including all inherited members.

- "COMMON_ALREADY_OPEN_ERROR variable" on page 531
- "COMMON_GET_INIT_FILE_ERROR variable" on page 532
- "COMMON_GETQUEUEDEPTH_ERROR variable" on page 531
- "COMMON_GETQUEUEDEPTH_ERROR_INVALID_ARG variable" on page 531
- "COMMON_GETQUEUEDEPTH_ERROR_NO_STORE_ID variable" on page 531
- "COMMON_INIT_ERROR variable" on page 532
- "COMMON_INIT_THREAD_ERROR variable" on page 532
- "COMMON_INVALID_PROPERTY variable" on page 532
- "COMMON_MSG_ACKNOWLEDGE_ERROR variable" on page 532
- "COMMON_MSG_CANCEL_ERROR variable" on page 532
- "COMMON_MSG_CANCEL_ERROR_SENT variable" on page 533
- "COMMON_MSG_NOT_WRITEABLE_ERROR variable" on page 533
- "COMMON_MSG_RETRIEVE_ERROR variable" on page 533
- "COMMON_MSG_STORE_ERROR variable" on page 533
- "COMMON_MSG_STORE_NOT_INITIALIZED variable" on page 533
- "COMMON_MSG_STORE_TOO_LARGE variable" on page 534
- "COMMON_NO_DEST_ERROR variable" on page 534
- "COMMON_NO_IMPLEMENTATION variable" on page 534
- "COMMON_NOT_OPEN_ERROR variable" on page 534

COMMON_ALREADY_OPEN_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_ALREADY_OPEN_ERROR**

Remarks

The Interface QAManager is already open.

COMMON_GETQUEUEDEPTH_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_GETQUEUEDEPTH_ERROR**

Remarks

Error getting the queue depth.

COMMON_GETQUEUEDEPTH_ERROR_INVALID_ARG variable

Synopsis

final int
ianywhere.qanywhere.client.QAException.COMMON_GETQUEUEDEPTH_ERROR_INVALID_ARG

Remarks

Cannot use getQueueDepth method on a given destination when filter is ALL.

COMMON_GETQUEUEDEPTH_ERROR_NO_STORE_ID variable

Synopsis

final int
ianywhere.qanywhere.client.QAException.COMMON_GETQUEUEDEPTH_ERROR_NO_STORE_ID

Remarks

Cannot use getQueueDepth method when the message store ID has not been set.

COMMON_GET_INIT_FILE_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_GET_INIT_FILE_ERROR**

Remarks

Unable to access the client properties file.

COMMON_INIT_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_INIT_ERROR**

Remarks

Initialization error.

COMMON_INIT_THREAD_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_INIT_THREAD_ERROR**

Remarks

Error initializing the background thread.

COMMON_INVALID_PROPERTY variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_INVALID_PROPERTY**

Remarks

There is an invalid property in the client properties file.

COMMON_MSG_ACKNOWLEDGE_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_MSG_ACKNOWLEDGE_ERROR**

Remarks

Error acknowledging the message.

COMMON_MSG_CANCEL_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_MSG_CANCEL_ERROR**

Remarks

Error cancelling message.

COMMON_MSG_CANCEL_ERROR_SENT variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_MSG_CANCEL_ERROR_SENT**

Remarks

Error cancelling message.

Cannot cancel a message that has already been sent.

COMMON_MSG_NOT_WRITEABLE_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_MSG_NOT_WRITEABLE_ERROR**

Remarks

You cannot write to a message that is in read-only mode.

COMMON_MSG_RETRIEVE_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_MSG_RETRIEVE_ERROR**

Remarks

Error retrieving a message from the client message store.

COMMON_MSG_STORE_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_MSG_STORE_ERROR**

Remarks

Error storing a message in the client message store.

COMMON_MSG_STORE_NOT_INITIALIZED variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_MSG_STORE_NOT_INITIALIZED**

Remarks

The message store has not been initialized for messaging.

COMMON_MSG_STORE_TOO_LARGE variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_MSG_STORE_TOO_LARGE**

Remarks

The message store is too large relative to the free disk space on the device.

COMMON_NOT_OPEN_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_NOT_OPEN_ERROR**

Remarks

The Interface QAManager is not open.

COMMON_NO_DEST_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_NO_DEST_ERROR**

Remarks

No destination.

COMMON_NO_IMPLEMENTATION variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_NO_IMPLEMENTATION**

Remarks

The method is not implemented.

COMMON_OPEN_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_OPEN_ERROR**

Remarks

Error opening a connection to the message store.

COMMON_OPEN_LOG_FILE_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_OPEN_LOG_FILE_ERROR**

Remarks

Error opening the log file.

COMMON_SELECTOR_SYNTAX_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_SELECTOR_SYNTAX_ERROR**

Remarks

The given selector has a syntax error.

COMMON_TERMINATE_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_TERMINATE_ERROR**

Remarks

Termination error.

COMMON_UNEXPECTED_EOM_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_UNEXPECTED_EOM_ERROR**

Remarks

Unexpected end of message reached.

COMMON_UNREPRESENTABLE_TIMESTAMP variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.COMMON_UNREPRESENTABLE_TIMESTAMP**

Remarks

The timestamp is outside of the acceptable range.

QAException method

Synopsis

ianywhere.qanywhere.client.QAException.QAException(
String *message*,
int *errorCode*
)

Parameters

- **message** The text description of the exception.
- **errorCode** The error code.

Remarks

Creates a Class QAException instance with the provided error code and error message text.

QA_NO_ERROR variable

Synopsis

final int **ianywhere.qanywhere.client.QAException.QA_NO_ERROR**

Remarks

No error.

getErrorCode method

Synopsis

int **ianywhere.qanywhere.client.QAException.getErrorCode()**

Remarks

Returns the error code of the last exception.

Returns

The error code of the last exception.

Interface QAManager

Synopsis

public **ianywhere.qanywhere.client.QAManager**

Base classes

- "Interface QAManagerBase" on page 540

Remarks

Interface QAManager derives from Interface QAManagerBase.

It manages non-transactional QAnywhere messaging operations.

For a detailed description of derived behavior, see Interface QAManagerBase.

The Interface QAManager instance can be configured for implicit or explicit acknowledgement, as defined in the Interface AcknowledgementMode class. To acknowledge messages as part of a transaction, use Interface QATransactionalManager.

Use the Class QAManagerFactory class to create Interface QAManager and Interface QATransactionalManager objects.

Members

All members of ianywhere.qanywhere.client.QAManager, including all inherited members.

- "triggerSendReceive method" on page 566

acknowledge method

Synopsis

void **ianywhere.qanywhere.client.QAManager.acknowledge(**
QAMessage *msg*
)
throws **QAException**

Parameters

- **msg** The message to acknowledge.

Throws

- Thrown if there is a problem acknowledging the message.

Remarks

Acknowledges that the client application successfully received a QAnywhere message.

Note: When a Interface QAMessage is acknowledged, its status property changes to RECEIVED variable. It can then be deleted using the default delete rule.

See Also

Interface QAManager

acknowledgeUntil method

acknowledgeAll method

acknowledgeAll method

Synopsis

void **ianywhere.qanywhere.client.QAManager.acknowledgeAll()**
throws **QAException**

Throws

- Thrown if there is a problem acknowledging the messages.

Remarks

Acknowledges that the client application successfully received QAnywhere messages.

All unacknowledged messages are acknowledged.

Note: When a Interface QAMessage is acknowledged, its status property changes to RECEIVED variable. It can then be deleted using the default delete rule.

See Also

Interface QAManager

acknowledge method

acknowledgeUntil method

acknowledgeUntil method

Synopsis

void **ianywhere.qanywhere.client.QAManager.acknowledgeUntil(**
QAMessage *msg*
)
throws **QAException**

Parameters

- **msg** The last message to acknowledge. All earlier unacknowledged messages are also acknowledged.

Throws

- Thrown if there is a problem acknowledging the messages.

Remarks

Acknowledges the given Interface QAMessage instance and all unacknowledged messages received before the given message.

Note: When a Interface QAMessage is acknowledged, its status property changes to RECEIVED variable. It can then be deleted using the default delete rule.

See Also

Interface QAManager

acknowledge method

acknowledgeAll method

open method

Synopsis

void **ianywhere.qanywhere.client.QAManager.open(**
short *mode*
)
throws **QAException**

Parameters

- **mode** The acknowledgement mode, one of "EXPLICIT_ACKNOWLEDGEMENT variable" on page 504 or "IMPLICIT_ACKNOWLEDGEMENT variable" on page 505.

Throws

- Thrown if there is a problem opening the "Interface QAManager" on page 536 instance.

Remarks

Opens the Interface QAManager with the given Interface AcknowledgementMode value.

The open method must be the first method called after creating a Interface QAManager.

See Also

Interface AcknowledgementMode

Interface QAManager

recover method

Synopsis

void **ianywhere.qanywhere.client.QAManager.recover()**
throws **QAException**

Throws

- Thrown if there is a problem recovering.

Remarks

Forces all unacknowledged messages into a state of unreceived.

These messages must be received again using getMessage method.

See Also

Interface QAManager

Interface QAManagerBase

Synopsis

public **ianywhere.qanywhere.client.QAManagerBase**

Derived classes

- "Interface QAManager" on page 536
- "Interface QATransactionalManager" on page 594

Remarks

This class acts as a base class for Interface QATransactionalManager and Interface QAManager, which manage transactional and non-transactional messaging, respectively.

Use the start method to allow a Interface QAManagerBase instance to listen for messages. An instance of Interface QAManagerBase must be used only on the thread that created it.

You can use instances of this class to create and manage QAnywhere messages. Use the createBinaryMessage method and createTextMessage method to create appropriate Interface QAMessage instances. Interface QAMessage instances provide a variety of methods to set message content and properties. To send QAnywhere messages, use the putMessage method to place the addressed message in the local message store queue. The message is transmitted by the QAnywhere Agent based on its transmission policies or when you call triggerSendReceive method.

Interface QAManagerBase also provides methods to set and get message store properties.

Members

All members of ianywhere.qanywhere.client.QAManagerBase, including all inherited members.

browseMessages method

Synopsis

java.util.Enumeration **ianywhere.qanywhere.client.QAManagerBase.browseMessages()**
throws **QAException**

Throws

- Thrown if there is a problem browsing the messages.

Remarks

Browses all available messages in the message store.

The messages are just being browsed, so they cannot be acknowledged.

Use getMessage method to receive messages so that they can be acknowledged.

See Also

browseMessagesByQueue method

browseMessagesByID method

Returns

An enumerator over the available messages.

browseMessagesByID method

Synopsis

java.util.Enumeration **ianywhere.qanywhere.client.QAManagerBase.browseMessagesByID(**
String *id*
)
throws **QAException**

Parameters

- **id** The message ID of the message.

Throws

- Thrown if there is a problem browsing the messages.

Remarks

Browse the message with the given message ID.

The message is just being browsed, so it cannot be acknowledged. Use getMessage method to receive messages so that they can be acknowledged.

See Also

browseMessagesByQueue method

browseMessages method

Returns

An enumerator containing 0 or 1 messages.

browseMessagesByQueue method

Synopsis

java.util.Enumeration **ianywhere.qanywhere.client.QAManagerBase.browseMessagesByQueue(**
String *address*
)
throws **QAException**

Parameters

- **address** The address of the messages.

Throws

- Thrown if there is a problem browsing the messages.

Remarks

Browses the available messages waiting that have been sent to the given address.

The messages are just being browsed, so they cannot be acknowledged.

Use getMessage method to receive messages so they can be acknowledged.

See Also

browseMessagesByID method

browseMessages method

Returns

An enumerator over the available messages.

browseMessagesBySelector method

Synopsis

java.util.Enumeration **ianywhere.qanywhere.client.QAManagerBase.browseMessagesBySelector(**
String *selector*
)
throws **QAException**

Parameters

- **selector** The selector.

Throws

- Thrown if there is a problem browsing the messages.

Remarks

Browse messages queued in the message store that satisfy the given selector.

The message is just being browsed, so it cannot be acknowledged. Use getMessage method to receive messages so that they can be acknowledged.

See Also

browseMessagesByQueue method

browseMessages method

browseMessagesByID method

Returns

An enumerator over the available messages.

cancelMessage method

Synopsis

boolean **ianywhere.qanywhere.client.QAManagerBase.cancelMessage(**
String *id*
)
throws **QAException**

Parameters

- **id** The message ID of the message to cancel.

Throws

- Thrown if there is a problem cancelling the message.

Remarks

Cancels the message with the given message ID.

Puts a message into a cancelled state before it is transmitted.

With the default delete rules of the QAnywhere Agent, cancelled messages are eventually deleted from the message store.

Fails if the message is already in a final state, or if the message has been transmitted to the central messaging server.

close method

Synopsis

void **ianywhere.qanywhere.client.QAManagerBase.close()**
throws **QAException**

Throws

- Thrown if there is a problem closing the "Interface QAManagerBase" on page 540 instance.

Remarks

Closes the connection to the QAnywhere message system and releases any resources used by the Interface QAManagerBase.

Additional calls to close method following the first are ignored. Any subsequent calls to a Interface QAManagerBase method, other than close method, result in a Class QAException. You must create and open a new Interface QAManagerBase instance in this case.

createBinaryMessage method

Synopsis

QABinaryMessage **ianywhere.qanywhere.client.QAManagerBase.createBinaryMessage()**
throws **QAException**

Throws

- Thrown if there is a problem creating the message.

Remarks

Creates a Interface QABinaryMessage object.

A Interface QABinaryMessage object is used to send a message containing a message body of uninterpreted bytes.

See Also

Interface QABinaryMessage

Returns

A new Interface QABinaryMessage instance.

createTextMessage method

Synopsis

QATextMessage **ianywhere.qanywhere.client.QAManagerBase.createTextMessage()**
throws **QAException**

Throws

- Thrown if there is a problem creating the message.

Remarks

Creates a Interface QATextMessage object.

A Interface QATextMessage object is used to send a message containing a string message body.

See Also

Interface QATextMessage

Returns

A new Interface QATextMessage instance.

getBooleanStoreProperty method

Synopsis

boolean **ianywhere.qanywhere.client.QAManagerBase.getBooleanStoreProperty(**
String *name*
)
throws **QAException**

Parameters

- **name** The pre-defined or custom property name.

Throws

- Thrown if there is a conversion error getting the property value or if the property does not exist.

Remarks

Gets a boolean value for a pre-defined or custom message store property.

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

Returns

The boolean property value.

getByteStoreProperty method

Synopsis

byte **ianywhere.qanywhere.client.QAManagerBase.getByteStoreProperty(**
String *name*
)
throws **QAException**

Parameters

- **name** The pre-defined or custom property name.

Throws

- Thrown if there is a conversion error getting the property value or if the property does not exist.

Remarks

Gets a signed byte value for a pre-defined or custom message store property.

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

Returns

The signed byte property value.

getDoubleStoreProperty method

Synopsis

double **ianywhere.qanywhere.client.QAManagerBase.getDoubleStoreProperty(**
String *name*
)
throws **QAException**

Parameters

- **name** the pre-defined or custom property name.

Throws

- Thrown if there is a conversion error getting the property value or if the property does not exist.

Remarks

Gets a double value for a pre-defined or custom message store property.

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

Returns

The double property value.

getFloatStoreProperty method

Synopsis

```
float ianywhere.qanywhere.client.QAManagerBase.getFloatStoreProperty(
  String name
)
throws QAException
```

Parameters

- **name** The pre-defined or custom property name.

Throws

- Thrown if there is a conversion error getting the property value or if the property does not exist.

Remarks

Gets a float value for a pre-defined or custom message store property.

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

Returns

The float property value.

getIntStoreProperty method

Synopsis

```
int ianywhere.qanywhere.client.QAManagerBase.getIntStoreProperty(
  String name
)
throws QAException
```

Parameters

- **name** The pre-defined or custom property name.

Throws

- Thrown if there is a conversion error getting the property value or if the property does not exist.

Remarks

Gets a int value for a pre-defined or custom message store property.

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

Returns

The integer property value.

getLongStoreProperty method

Synopsis

long **ianywhere.qanywhere.client.QAManagerBase.getLongStoreProperty(**
String *name*
)
throws **QAException**

Parameters

- **name** The pre-defined or custom property name.

Throws

- Thrown if there is a conversion error getting the property value or if the property does not exist.

Remarks

Gets a long value for a pre-defined or custom message store property.

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

Returns

The long property value.

getMessage method

Synopsis

QAMessage **ianywhere.qanywhere.client.QAManagerBase.getMessage(**
  String *address*
)
throws **QAException**

Parameters

- **address** This address specifies the queue name used by the QAnywhere client to receive messages.

Throws

- Thrown if there is a problem getting the message.

Remarks

Returns the next available Interface QAMessage sent to the specified address.

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name' or 'queue-name'. If there is no message available, this call blocks indefinitely until a message is available. Use this method to receive messages synchronously.

Returns

The next Interface QAMessage, or null if no message is available.

getMessageBySelector method

Synopsis

QAMessage **ianywhere.qanywhere.client.QAManagerBase.getMessageBySelector(**
  String *address*,
  String *selector*
)
throws **QAException**

Parameters

- **address** This address specifies the queue name used by the QAnywhere client to receive messages.
- **selector** The selector.

Throws

- Thrown if there is a problem getting the message.

Remarks

Returns the next available Interface QAMessage sent to the specified address that satisfies the given selector.

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name' or 'queue-name'. If there is no message available, this call blocks indefinitely until a message is available.

Use this method to receive messages synchronously.

Returns

The next Interface QAMessage, or null if no message is available.

getMessageBySelectorNoWait method

Synopsis

QAMessage **ianywhere.qanywhere.client.QAManagerBase.getMessageBySelectorNoWait(**
String *address*,
String *selector*
)
throws **QAException**

Parameters

- **address** This address specifies the queue name used by the QAnywhere client to receive messages.
- **selector** The selector.

Throws

- Thrown if there is a problem getting the message.

Remarks

Returns the next available Interface QAMessage sent to the given address that satisfies the given selector.

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name' or 'queue-name'. If no message is available, this method returns immediately.

Use this method to receive messages synchronously.

Returns

The next available message or null there is no available message.

getMessageBySelectorTimeout method

Synopsis

QAMessage **ianywhere.qanywhere.client.QAManagerBase.getMessageBySelectorTimeout(**
String *address*,
String *selector*,
long *timeout*
)
throws **QAException**

Parameters

- **address** This address specifies the queue name used by the QAnywhere client to receive messages.

- **selector** The selector.
- **timeout** The time to wait, in milliseconds, for a message to become available.

Throws

- Thrown if there is a problem getting the message.

Remarks

Returns the next available Interface QAMessage sent to the given address that satisfies the given selector.

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name' or 'queue-name'. If no message is available, this method waits for the specified timeout and then returns.

Use this method to receive messages synchronously.

Returns

The next Interface QAMessage, or null if no message is available.

getMessageListener method

Synopsis

QAMessageListener **ianywhere.qanywhere.client.QAManagerBase.getMessageListener(**
String *address*
)
throws **QAException**

Parameters

- **address** A local queue name used to receive messages, or system.

Throws

- Thrown if there is a problem getting the listener.

Remarks

Returns the Interface QAMessageListener associated with the specified queue.

If there is no Interface QAMessageListener associated with the specified queue, returns null.

Returns

The listener.

getMessageNoWait method

Synopsis

QAMessage **ianywhere.qanywhere.client.QAManagerBase.getMessageNoWait(**
String *address*

)
throws **QAException**

Parameters

- **address** This address specifies the queue name used by the QAnywhere client to receive messages.

Throws

- Thrown if there is a problem getting the message.

Remarks

Returns the next available Interface QAMessage sent to the given address.

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name' or 'queue-name'. If no message is available, this method returns immediately.

Use this method to receive messages synchronously.

Returns

The next available message or null there is no available message.

getMessageTimeout method

Synopsis

QAMessage **ianywhere.qanywhere.client.QAManagerBase.getMessageTimeout(**
String *address*,
long *timeout*
)
throws **QAException**

Parameters

- **address** This address specifies the queue name used by the QAnywhere client to receive messages.
- **timeout** The time to wait, in milliseconds, for a message to become available.

Throws

- Thrown if there is a problem getting the message.

Remarks

Returns the next available Interface QAMessage sent to the given address.

The address parameter specifies a local queue name. The address can be in the form 'store-id\queue-name' or 'queue-name'. If no message is available, this method waits for the specified timeout and then returns. Use this method to receive messages synchronously.

Returns

the next Interface QAMessage, or null if no message is available.

getMode method

Synopsis

short **ianywhere.qanywhere.client.QAManagerBase.getMode()**
throws **QAException**

Throws

- Thrown if there is a problem retrieving the "Interface QAManager" on page 536 acknowledgement mode

Remarks

Returns the Interface QAManager acknowledgement mode for received messages.

For a list of return values, see Interface AcknowledgementMode.

EXPLICIT_ACKNOWLEDGEMENT variable and IMPLICIT_ACKNOWLEDGEMENT variable apply to Interface QAManager instances. TRANSACTIONAL variable is the mode for Interface QATransactionalManager instances.

Returns

The Interface QAManager acknowledgement mode for received messages.

getQueueDepth method

Synopsis

int **ianywhere.qanywhere.client.QAManagerBase.getQueueDepth(**
  short *filter*
)
throws **QAException**

Parameters

- **filter** A filter indicating incoming messages, outgoing messages, or all messages.

Throws

- Thrown if there was an error.

Remarks

Returns the total depth of all queues, based on a given filter.

The depth of the queue is the number of messages that have not been received (for example, using the getMessage method).

See Interface QueueDepthFilter for a list of possible filter values.

Returns

The number of messages in all queues for the given filter.

getQueueDepth method

Synopsis

int **ianywhere.qanywhere.client.QAManagerBase.getQueueDepth(**
String *queue*,
short *filter*
)
throws **QAException**

Parameters

- **queue** A filter indicating incoming messages, outgoing messages, or all messages.
- **filter** The queue name.

Throws

- Thrown if there was an error.

Remarks

Returns the depth of a queue, based on a given filter.

The depth of the queue is the number of messages that have not been received (for example, using the getMessage method).

See Interface QueueDepthFilter for a list of possible filter values.

Returns

the number of messages.

getShortStoreProperty method

Synopsis

short **ianywhere.qanywhere.client.QAManagerBase.getShortStoreProperty(**
String *name*
)
throws **QAException**

Parameters

- **name** The pre-defined or custom property name.

Throws

- Thrown if there is a conversion error getting the property value or if the property does not exist.

Remarks

Gets a short value for a pre-defined or custom message store property.

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

Returns

The short property value.

getStoreProperty method

Synopsis

Object **ianywhere.qanywhere.client.QAManagerBase.getStoreProperty(**
String *name*
)
throws **QAException**

Parameters

- **name** The pre-defined or custom property name.

Throws

- Thrown if there is a conversion error getting the property value or if the property does not exist.

Remarks

Gets an Object representing a message store property.

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

Returns

The property value.

getStorePropertyNames method

Synopsis

java.util.Enumeration **ianywhere.qanywhere.client.QAManagerBase.getStorePropertyNames()**
throws **QAException**

Throws

- Thrown if there is a problem retrieving the enumerator.

Remarks

Gets an enumerator over the message store property names.

Returns

An enumerator over the message store property names.

getStringStoreProperty method

Synopsis

String **ianywhere.qanywhere.client.QAManagerBase.getStringStoreProperty(**
String *name*
)
throws **QAException**

Parameters

- **name** The pre-defined or custom property name.

Throws

- Thrown if there is a problem retrieving the string value.

Remarks

Gets a string value for a pre-defined or custom message store property.

You can use this method to access pre-defined or user-defined client store properties.

For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

Returns

The string property value or null if the property does not exist.

putMessage method

Synopsis

void **ianywhere.qanywhere.client.QAManagerBase.putMessage(**
String *address*,
QAMessage *msg*
)
throws **QAException**

Parameters

- **address** The address of the message specifying the destination queue name.
- **msg** The message to put in the local message store for transmission.

Throws

- Thrown if there is a problem putting the message.

Remarks

Prepares a message to send to another QAnywhere client.

This method inserts a message and a destination address into your local message store. The time of message transmission depends on QAnywhere Agent transmission policies.

The address takes the form 'id\queue-name', where 'id' is the destination message store id and 'queue-name' identifies a queue that is used by the destination QAnywhere client to listen for or receive messages.

See Also

putMessageTimeToLive method

putMessageTimeToLive method

Synopsis

void **ianywhere.qanywhere.client.QAManagerBase.putMessageTimeToLive(**
  String *address*,
  QAMessage *msg*,
  long *ttl*
)
throws **QAException**

Parameters

- **address** The address of the message specifying the destination queue name.
- **msg** The message to put.
- **ttl** The delay, in milliseconds, before the message expires if it has not been delivered. A value of 0 indicates the message does not expire.

Throws

- Thrown if there is a problem putting the message.

Remarks

Prepares a message to send to another QAnywhere client.

This method inserts a message and a destination address into your local message store. The time of message transmission depends on QAnywhere Agent transmission policies. However, if the next message transmission time exceeds the given time-to-live value, the message expires.

The address takes the form 'id\queue-name', where 'id' is the destination message store id and 'queue-name' identifies a queue that is used by the destination QAnywhere client to listen for or receive messages.

setBooleanStoreProperty method

Synopsis

void **ianywhere.qanywhere.client.QAManagerBase.setBooleanStoreProperty(**
String *name*,
boolean *value*
)
throws **QAException**

Parameters

- **name** The pre-defined or custom property name.
- **value** The boolean property value.

Throws

- Thrown if there is a problem setting the message store property.

Remarks

Sets a pre-defined or custom message store property to a boolean value.

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

setByteStoreProperty method

Synopsis

void **ianywhere.qanywhere.client.QAManagerBase.setByteStoreProperty(**
String *name*,
byte *value*
)
throws **QAException**

Parameters

- **name** The pre-defined or custom property name.
- **value** The sbyte property value.

Throws

- Thrown if there is a problem setting the message store property.

Remarks

Sets a pre-defined or custom message store property to a sbyte value.

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

setDoubleStoreProperty method

Synopsis

void **ianywhere.qanywhere.client.QAManagerBase.setDoubleStoreProperty(**
String *name*,
double *value*
)
throws **QAException**

Parameters

- **name** The pre-defined or custom property name.
- **value** The double property value.

Throws

- Thrown if there is a problem setting the message store property.

Remarks

Sets a pre-defined or custom message store property to a double value.

You can use this method to set pre-defined or user-defined client. store properties.

For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

setFloatStoreProperty method

Synopsis

void **ianywhere.qanywhere.client.QAManagerBase.setFloatStoreProperty(**
String *name*,
float *value*
)
throws **QAException**

Parameters

- **name** The pre-defined or custom property name.
- **value** The float property value.

Throws

- Thrown if there is a problem setting the message store property.

Remarks

Sets a pre-defined or custom message store property to a float value.

You can use this method to set pre-defined or user-defined client store properties. For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

setIntStoreProperty method

Synopsis

void **ianywhere.qanywhere.client.QAManagerBase.setIntStoreProperty(**
String *name*,
int *value*
)
throws **QAException**

Parameters

- **name** The pre-defined or custom property name.
- **value** The int property value.

Throws

- Thrown if there is a problem setting the message store property.

Remarks

Sets a pre-defined or custom message store property to a int value.

You can use this method to set pre-defined or user-defined client store properties. For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

setLongStoreProperty method

Synopsis

void **ianywhere.qanywhere.client.QAManagerBase.setLongStoreProperty(**
String *name*,
long *value*

)
throws **QAException**

Parameters

- **name** The pre-defined or custom property name.
- **value** The long property value.

Throws

- Thrown if there is a problem setting the message store property.

Remarks

Sets a pre-defined or custom message store property to a long value.

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

setMessageListener method

Synopsis

void **ianywhere.qanywhere.client.QAManagerBase.setMessageListener(**
String *address*,
QAMessageListener *listener*
)
throws **QAException**

Parameters

- **address** The address of a local queue name used to receive messages, or system to listen for QAnywhere system messages.
- **listener** The listener.

Throws

- Thrown if there is a problem registering the "Interface QAMessageListener" on page 588 object, such as because there is already a listener object assigned to the given queue.

Remarks

Registers a Interface QAMessageListener object to receive QAnywhere messages asynchronously.

The address parameter specifies a local queue name used to receive the message. You can only have one listener object assigned to a given queue. If you want to listen for QAnywhere system messages, including push notifications and network status changes, specify "system" as the queue name.

Use this method to receive messages asynchronously.

setMessageListenerBySelector method

Synopsis

void **ianywhere.qanywhere.client.QAManagerBase.setMessageListenerBySelector(**
String *address*,
String *selector*,
QAMessageListener *listener*
)
throws **QAException**

Parameters

- **address** The address of a local queue name used to receive messages, or system to listen for QAnywhere system messages.
- **selector** The selector to be used to filter the messages to be received.
- **listener** The listener.

Throws

- Thrown if there is a problem registering the "Interface QAMessageListener" on page 588 object, such as because there is already a listener object assigned to the given queue.

Remarks

Registers a Interface QAMessageListener object to receive QAnywhere messages asynchronously, with a message selector.

The address parameter specifies a local queue name used to receive the message. You can only have one listener object assigned to a given queue. The selector parameter specifies a selector to be used to filter the messages to be received on the given address. If you want to listen for QAnywhere system messages, including push notifications and network status changes, specify "system" as the queue name.

Use this method to receive messages asynchronously.

setShortStoreProperty method

Synopsis

void **ianywhere.qanywhere.client.QAManagerBase.setShortStoreProperty(**
String *name*,
short *value*
)
throws **QAException**

Parameters

- **name** The pre-defined or custom property name.
- **value** The short property value.

Throws

- Thrown if there is a problem setting the message store property.

Remarks

Sets a pre-defined or custom message store property to a short value.

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

setStoreProperty method

Synopsis

void **ianywhere.qanywhere.client.QAManagerBase.setStoreProperty(**
String *name*,
Object *value*
)
throws **QAException**

Parameters

- **name** The pre-defined or custom property name.
- **value** The property value.

Throws

- Thrown if there is a problem setting the message store property to the value.

Remarks

Sets a pre-defined or custom message store property to a System.Object value.

The property type must correspond to one of the acceptable primitive types, or String. You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

setStringStoreProperty method

Synopsis

void **ianywhere.qanywhere.client.QAManagerBase.setStringStoreProperty(**
String *name*,
String *value*
)
throws **QAException**

Parameters

- **name** The pre-defined or custom property name.
- **value** The String property value.

Throws

- Thrown if there is a problem setting the message store property to a string value.

Remarks

Sets a pre-defined or custom message store property to a String value.

You can use this method to set pre-defined or user-defined client store properties.

For a list of pre-defined properties, see Interface MessageStoreProperties.

See Also

Interface MessageStoreProperties

start method

Synopsis

void **ianywhere.qanywhere.client.QAManagerBase.start()**
throws **QAException**

Throws

- Thrown if there is a problem starting the "Interface QAManagerBase" on page 540 instance.

Remarks

Starts the Interface QAManagerBase for receiving incoming messages.

Any calls to this method beyond the first without an intervening stop method call are ignored.

See Also

stop method

stop method

Synopsis

void **ianywhere.qanywhere.client.QAManagerBase.stop()**
throws **QAException**

Throws

- Thrown if there is a problem stopping the "Interface QAManagerBase" on page 540 instance.

Remarks

Halts the QAManagerBase's reception of incoming messages.

The messages are not lost. They just are not received until the manager is started again. Any calls to stop method beyond the first without an intervening start method call are ignored.

See Also

start method

triggerSendReceive method

Synopsis

void **ianywhere.qanywhere.client.QAManagerBase.triggerSendReceive()**
throws **QAException**

Throws

- Thrown if there is a problem triggering the send/receive.

Remarks

Causes a synchronization with the QAnywhere message server, uploading any messages addressed to other clients, and downloading any messages addressed to the local client.

A call to this method results in immediate message synchronization between a QAnywhere Agent and the central messaging server. A manual triggerSendReceive method call results in immediate message transmission, independent of the QAnywhere Agent transmission policies.

QAnywhere Agent transmission policies determine how message transmission occurs. For example, message transmission can occur automatically at regular intervals, when your client receives a push notification, or when you call the putMessage method to send a message.

See Also

putMessage method

Class QAManagerFactory

Synopsis

public **ianywhere.qanywhere.client.QAManagerFactory**

Remarks

This class acts as a factory class for creating Interface QATransactionalManager and Interface QAManager objects.

You can only have one instance of Class QAManagerFactory.

Members

All members of ianywhere.qanywhere.client.QAManagerFactory, including all inherited members.

createQAManager method

Synopsis

abstract QAManager **ianywhere.qanywhere.client.QAManagerFactory.createQAManager(**
String *iniFile*
)
throws **QAException**

Parameters

- **iniFile** A properties file for configuring the "Interface QAManager" on page 536 instance, or null to create the "Interface QAManager" on page 536 instance using default properties.

Throws

- Thrown if there is a problem creating the manager.

Remarks

Returns a new Interface QAManager instance with the specified properties.

If the iniFile parameter is null, the Interface QAManager is created using default properties. You can use the Interface QAManagerBase set property methods to set Interface QAManager properties programmatically after you create the instance.

See Also

Interface QAManager

Returns

A new Interface QAManager instance.

createQAManager method

Synopsis

abstract QAManager **ianywhere.qanywhere.client.QAManagerFactory.createQAManager(**
java.util.Hashtable *properties*

)
throws **QAException**

Parameters

- **properties** A hashtable for configuring the "Interface QAManager" on page 536 instance.

Throws

- Thrown if there is a problem creating the manager.

Remarks

Returns a new Interface QAManager instance with the specified properties as a Hashtable.

See Also

Interface QAManager

Returns

A new Interface QAManager instance.

createQAManager method

Synopsis

abstract QAManager **ianywhere.qanywhere.client.QAManagerFactory.createQAManager()**
throws **QAException**

Throws

- Thrown if there is a problem creating the manager.

Remarks

Returns a new Interface QAManager instance with default properties.

See Also

Interface QAManager

Returns

A new Interface QAManager instance.

createQATransactionalManager method

Synopsis

abstract QATransactionalManager
ianywhere.qanywhere.client.QAManagerFactory.createQATransactionalManager(
String *iniFile*

)
throws **QAException**

Parameters

- **iniFile** A properties file for configuring the "Interface QATransactionalManager" on page 594 instance.

Throws

- Thrown if there is a problem creating the manager.

Remarks

Returns a new Interface QATransactionalManager instance with the specified properties.

If the iniFile parameter is null, the Interface QATransactionalManager is created using default properties. You can use the Interface QAManagerBase set property methods to set Interface QATransactionalManager properties programmatically after you create the instance.

See Also

Interface QATransactionalManager

Returns

The configured Interface QATransactionalManager.

createQATransactionalManager method

Synopsis

abstract QATransactionalManager
ianywhere.qanywhere.client.QAManagerFactory.createQATransactionalManager(
java.util.Hashtable *properties*
)
throws **QAException**

Parameters

- **properties** A hashtable for configuring the "Interface QATransactionalManager" on page 594 instance.

Throws

- Thrown if there is a problem creating the manager.

Remarks

Returns a new Interface QAManager instance with the specified properties.

See Also

Interface QATransactionalManager

Returns

The configured Interface QATransactionalManager.

createQATransactionalManager method

Synopsis

abstract QATransactionalManager
ianywhere.qanywhere.client.QAManagerFactory.createQATransactionalManager()
throws **QAException**

Throws

- Thrown if there is a problem creating the manager.

Remarks

Returns a new Interface QAManager instance with default properties.

See Also

Interface QATransactionalManager

Returns

A new Interface QATransactionalManager.

getInstance method

Synopsis

QAManagerFactory **ianywhere.qanywhere.client.QAManagerFactory.getInstance()**
throws **QAException**

Throws

- Thrown if there is a problem creating the manager factory.

Remarks

Returns the singleton Class QAManagerFactory instance.

Returns

The singleton Class QAManagerFactory instance

Interface QAMessage

Synopsis

public **ianywhere.qanywhere.client.QAMessage**

Derived classes

- "Interface QABinaryMessage" on page 515
- "Interface QATextMessage" on page 589

Remarks

Interface QAMessage provides an interface to set message properties and header fields.

The derived classes Interface QABinaryMessage and Interface QATextMessage provide specialized functions to read and write to the message body. You can use Interface QAMessage functions to set predefined or custom message properties. For a list of pre-defined property names, see the Interface MessageProperties.

Members

All members of ianywhere.qanywhere.client.QAMessage, including all inherited members.

- "clearProperties method" on page 572
- "DEFAULT_PRIORITY variable" on page 572
- "DEFAULT_TIME_TO_LIVE variable" on page 572
- "getAddress method" on page 572
- "getBooleanProperty method" on page 573
- "getByteProperty method" on page 573
- "getDoubleProperty method" on page 574
- "getExpiration method" on page 574
- "getFloatProperty method" on page 575
- "getInReplyToID method" on page 575
- "getIntProperty method" on page 576
- "getLongProperty method" on page 576
- "getMessageID method" on page 577
- "getPriority method" on page 577
- "getProperty method" on page 578
- "getPropertyNames method" on page 578
- "getPropertyType method" on page 578
- "getRedelivered method" on page 579
- "getReplyToAddress method" on page 579
- "getShortProperty method" on page 580
- "getStringProperty method" on page 580
- "getTimestamp method" on page 581
- "propertyExists method" on page 581
- "setAddress method" on page 582
- "setBooleanProperty method" on page 582
- "setByteProperty method" on page 583
- "setDoubleProperty method" on page 583
- "setFloatProperty method" on page 584
- "setInReplyToID method" on page 584
- "setIntProperty method" on page 585
- "setLongProperty method" on page 585
- "setPriority method" on page 586

- "setProperty method" on page 586
- "setReplyToAddress method" on page 587
- "setShortProperty method" on page 587
- "setStringProperty method" on page 588

DEFAULT_PRIORITY variable

Synopsis

final int **ianywhere.qanywhere.client.QAMessage.DEFAULT_PRIORITY**

Remarks

The default message priority.

DEFAULT_TIME_TO_LIVE variable

Synopsis

final long **ianywhere.qanywhere.client.QAMessage.DEFAULT_TIME_TO_LIVE**

Remarks

The default time-to-live value.

clearProperties method

Synopsis

void **ianywhere.qanywhere.client.QAMessage.clearProperties()**
throws **QAException**

Throws

- Thrown if there is a problem clearing the message properties.

Remarks

Clear all the properties of the message.

getAddress method

Synopsis

String **ianywhere.qanywhere.client.QAMessage.getAddress()**
throws **QAException**

Throws

- Thrown if there is a problem retrieving the destination address.

Remarks

Returns the destination address for the Interface QAMessage instance.

When a message is sent, this field is ignored. After completion of a send operation, the field holds the destination address specified in putMessage method.

Returns

The destination address for the Interface QAMessage instance.

getBooleanProperty method

Synopsis

boolean **ianywhere.qanywhere.client.QAMessage.getBooleanProperty(**
String *name*
)
throws **QAException**

Parameters

- **name** The property name.

Throws

- Thrown if there is a conversion error getting the property value or if the property does not exist.

Remarks

Gets a boolean message property.

Returns

The property value.

See Also

Interface MessageProperties

getByteProperty method

Synopsis

byte **ianywhere.qanywhere.client.QAMessage.getByteProperty(**
String *name*
)
throws **QAException**

Parameters

- **name** The property name.

Throws

- Thrown if there is a conversion error getting the property value or if the property does not exist.

Remarks

Gets a signed byte message property.

Returns

The property value.

See Also

Interface MessageProperties

getDoubleProperty method

Synopsis

double **ianywhere.qanywhere.client.QAMessage.getDoubleProperty(**
String *name*
)
throws **QAException**

Parameters

- **name** The property name.

Throws

- Thrown if there is a conversion error getting the property value or if the property does not exist.

Remarks

Gets a double message property.

Returns

The property value.

See Also

Interface MessageProperties

getExpiration method

Synopsis

java.util.Date **ianywhere.qanywhere.client.QAMessage.getExpiration()**
throws **QAException**

Throws

- Thrown if there is a problem getting the expiration.

Remarks

Returns the message's expiration value, or null if the message does not expire or has not yet been sent.

When a message is sent, the expiration is left unassigned. After the send operation completes, it holds the expiration time of the message.

This is a read-only property because the expiration time of a message is set by adding the time-to-live argument of putMessageTimeToLive method to the current time.

Returns

The message's expiration value, or null if the message does not expire or has not yet been sent.

getFloatProperty method

Synopsis

float **ianywhere.qanywhere.client.QAMessage.getFloatProperty(**
String *name*
)
throws **QAException**

Parameters

- **name** The property name.

Throws

- Thrown if there is a conversion error getting the property value or if the property does not exist.

Remarks

Gets a float message property.

Returns

The property value.

See Also

Interface MessageProperties

getInReplyToID method

Synopsis

String **ianywhere.qanywhere.client.QAMessage.getInReplyToID()**
throws **QAException**

Throws

- Thrown if there is a problem getting the message ID of the message to which this message is a reply.

Remarks

Returns the message ID of the message to which this message is a reply.

Returns

The message ID of the message to which this message is a reply, or null if this message is not a reply.

getIntProperty method

Synopsis

int **ianywhere.qanywhere.client.QAMessage.getIntProperty(**
  String *name*
)
throws **QAException**

Parameters

- **name** The property name.

Throws

- Thrown if there is a conversion error getting the property value or if the property does not exist.

Remarks

Gets an int message property.

Returns

The property value.

See Also

Interface MessageProperties

getLongProperty method

Synopsis

long **ianywhere.qanywhere.client.QAMessage.getLongProperty(**
  String *name*
)
throws **QAException**

Parameters

- **name** The property name.

Throws

- Thrown if there is a conversion error getting the property value or if the property does not exist.

Remarks

Gets a long message property.

Returns

The property value.

See Also

Interface MessageProperties

getMessageID method

Synopsis

String **ianywhere.qanywhere.client.QAMessage.getMessageID()**
throws **QAException**

Throws

- Thrown if there is a problem getting the message ID.

Remarks

Returns the globally unique message ID of the message.

This property is null until a message is put.

When a message is sent using putMessage method the message ID is null and can be ignored. When the send method returns, it contains an assigned value.

Returns

The message ID of the message, or null if the message has not yet been put.

getPriority method

Synopsis

int **ianywhere.qanywhere.client.QAMessage.getPriority()**
throws **QAException**

Throws

- Thrown if there is a problem getting the message priority.

Remarks

Returns the priority of the message (ranging from 0 to 9).

Returns

The priority of the message.

getProperty method

Synopsis

Object **ianywhere.qanywhere.client.QAMessage.getProperty(**
String *name*
)
throws **QAException**

Parameters

- **name** The property name.

Throws

- Thrown if there is a conversion error getting the property value.

Remarks

Gets a message property.

Returns

The property value, or null if the property does not exist.

getPropertyNames method

Synopsis

java.util.Enumeration **ianywhere.qanywhere.client.QAMessage.getPropertyNames()**
throws **QAException**

Throws

- Thrown if there is a problem getting the enumerator over the property names of the message.

Remarks

Gets an enumerator over the property names of the message.

Returns

An enumerator over the message property names.

getPropertyType method

Synopsis

short **ianywhere.qanywhere.client.QAMessage.getPropertyType(**
String *name*
)
throws **QAException**

Parameters

- **name** The property name.

Throws

- Thrown if there is a problem retrieving the property type.

Remarks

Returns the property type of the given property.

See Also

Interface PropertyType

Returns

The property type.

getRedelivered method

Synopsis

boolean **ianywhere.qanywhere.client.QAMessage.getRedelivered()**
throws **QAException**

Throws

- Thrown if there is a problem retrieving the redelivered status.

Remarks

Indicates whether the message has been previously received but not acknowledged.

Redelivered is set by a receiving Interface QAManager when it detects that a message being received was received before. For example, an application receives a message using a Interface QAManager opened with EXPLICIT_ACKNOWLEDGEMENT variable, and shuts down without acknowledging the message. When the application starts again and receives the same message, the message will be marked as redelivered.

Returns

True if the message has been previously received but not acknowledged.

getReplyToAddress method

Synopsis

String **ianywhere.qanywhere.client.QAMessage.getReplyToAddress()**
throws **QAException**

Throws

- Thrown if there is a problem retrieving the reply-to address.

Remarks

Returns the reply-to address of this message.

Returns

The reply-to address of this message, or null if it does not exist.

getShortProperty method

Synopsis

short **ianywhere.qanywhere.client.QAMessage.getShortProperty(**
String *name*
)
throws **QAException**

Parameters

- **name** the property name.

Throws

- Thrown if there is a conversion error getting the property value or if the property does not exist.

Remarks

Gets a short message property.

Returns

The property value.

See Also

Interface MessageProperties

getStringProperty method

Synopsis

String **ianywhere.qanywhere.client.QAMessage.getStringProperty(**
String *name*
)
throws **QAException**

Parameters

- **name** The property name.

Throws

- Thrown if there is a problem retrieving the message property.

Remarks

Gets a String message property.

Returns

The property value, or null if the property does not exist.

See Also

Interface MessageProperties

getTimestamp method

Synopsis

java.util.Date **ianywhere.qanywhere.client.QAMessage.getTimestamp()**
throws **QAException**

Throws

- Thrown if there is a problem retrieving the message timestamp.

Remarks

Returns the message timestamp, which is the time the message was created.

Returns

The message timestamp.

propertyExists method

Synopsis

boolean **ianywhere.qanywhere.client.QAMessage.propertyExists(**
String *name*
)
throws **QAException**

Parameters

- **name** The property name

Throws

- Thrown if there is a problem checking if the property has been set.

Remarks

Indicates whether the given property has been set for this message.

Returns

True if the property exists.

setAddress method

Synopsis

void **ianywhere.qanywhere.client.QAMessage.setAddress(**
String *dest*
)
throws **QAException**

Parameters

- **dest** The destination address.

Throws

- Thrown if there is a problem setting the message destination address.

Remarks

Sets the message destination address.

setBooleanProperty method

Synopsis

void **ianywhere.qanywhere.client.QAMessage.setBooleanProperty(**
String *name*,
boolean *value*
)
throws **QAException**

Parameters

- **name** The property name.
- **value** The property value.

Throws

- Thrown if there is a problem setting the property.

Remarks

Sets a boolean property.

See Also

Interface MessageProperties

setByteProperty method

Synopsis

void **ianywhere.qanywhere.client.QAMessage.setByteProperty(**
String *name*,
byte *value*
)
throws **QAException**

Parameters

- **name** The property name.
- **value** The property value.

Throws

- Thrown if there is a problem setting the property.

Remarks

Sets a signed byte property.

See Also

Interface MessageProperties

setDoubleProperty method

Synopsis

void **ianywhere.qanywhere.client.QAMessage.setDoubleProperty(**
String *name*,
double *value*
)
throws **QAException**

Parameters

- **name** The property name.
- **value** The property value.

Throws

- Thrown if there is a problem setting the property.

Remarks

Sets a double property.

See Also

Interface MessageProperties

setFloatProperty method

Synopsis

void **ianywhere.qanywhere.client.QAMessage.setFloatProperty(**
String *name*,
float *value*
)
throws **QAException**

Parameters

- **name** The property name.
- **value** The property value.

Throws

- Thrown if there is a problem setting the property.

Remarks

Sets a float property.

See Also

Interface MessageProperties

setInReplyToID method

Synopsis

void **ianywhere.qanywhere.client.QAMessage.setInReplyToID(**
String *id*
)
throws **QAException**

Parameters

- **id** The ID of the message this message is in reply to.

Throws

- Thrown if there is a problem setting the in reply to ID.

Remarks

Sets the in reply to ID, which identifies the message this message is a reply to.

setIntProperty method

Synopsis

void **ianywhere.qanywhere.client.QAMessage.setIntProperty(**
String *name*,
int *value*
)
throws **QAException**

Parameters

- **name** The property name.
- **value** The property value.

Throws

- Thrown if there is a problem setting the property.

Remarks

Sets an int property.

See Also

Interface MessageProperties

setLongProperty method

Synopsis

void **ianywhere.qanywhere.client.QAMessage.setLongProperty(**
String *name*,
long *value*
)
throws **QAException**

Parameters

- **name** The property name.
- **value** The property value.

Throws

- Thrown if there is a problem setting the property.

Remarks

Sets a long property.

See Also

Interface MessageProperties

setPriority method

Synopsis

void **ianywhere.qanywhere.client.QAMessage.setPriority(**
int *priority*
)
throws **QAException**

Parameters

- **priority** The priority of the message.

Throws

- Thrown if there is a problem setting the priority.

Remarks

Sets the priority of the message (ranging from 0 to 9).

setProperty method

Synopsis

void **ianywhere.qanywhere.client.QAMessage.setProperty(**
String *name*,
Object *value*
)
throws **QAException**

Parameters

- **name** The property name.
- **value** The property value.

Throws

- Thrown if there is a problem setting the property.

Remarks

Sets a property.

The property type must correspond to one of the acceptable primitive types, or String.

See Also

Interface MessageProperties

setReplyToAddress method

Synopsis

void **ianywhere.qanywhere.client.QAMessage.setReplyToAddress(**
String *address*
)
throws **QAException**

Parameters

- **address** The reply-to address.

Throws

- Thrown if there is a problem setting the reply-to address.

Remarks

Sets the reply-to address.

setShortProperty method

Synopsis

void **ianywhere.qanywhere.client.QAMessage.setShortProperty(**
String *name*,
short *value*
)
throws **QAException**

Parameters

- **name** The property name.
- **value** The property value.

Throws

- Thrown if there is a problem setting the property.

Remarks

Sets a short property.

See Also

Interface MessageProperties

setStringProperty method

Synopsis

void **ianywhere.qanywhere.client.QAMessage.setStringProperty(**
String *name*,
String *value*
)
throws **QAException**

Parameters

- **name** The property name.
- **value** The property value.

Throws

- Thrown if there is a problem setting the property.

Remarks

Sets a string property.

See Also

Interface MessageProperties

Interface QAMessageListener

Synopsis

public **ianywhere.qanywhere.client.QAMessageListener**

Remarks

To listen for messages, implement this interface and register your implementation by calling QAMangerBase.setMessageListener(String,QAMessageListener).

Members

All members of ianywhere.qanywhere.client.QAMessageListener, including all inherited members.

- "onException method" on page 588
- "onMessage method" on page 589

onException method

Synopsis

void **ianywhere.qanywhere.client.QAMessageListener.onException(**
QAException *exception*,
QAMessage *message*
)

Parameters

- **exception** The exception that occurred.
- **message** If the exception occurred after the message was passed to "onMessage method" on page 589, the message that was processed. Otherwise, null.

Remarks

This method is called whenever an exception occurs while listening for messages.

Note that this method cannot be used to automatically close the Interface QAManagerBase instance, as the close method blocks until all message listeners are finished processing.

onMessage method

Synopsis

void **ianywhere.qanywhere.client.QAMessageListener.onMessage(**
QAMessage *message*
)

Parameters

- **message** The message that was received.

Remarks

This method is called whenever a message is received.

Interface QATextMessage

Synopsis

public **ianywhere.qanywhere.client.QATextMessage**

Base classes

- "Interface QAMessage" on page 570

Remarks

Interface QATextMessage inherits from the Interface QAMessage class and adds a text message body, and methods to read from and write to the text message body.

When the message is first created, the body of the message is in write-only mode. After a message has been sent, the client that sent it can retain and modify it without affecting the message that has been sent. The same message object can be sent multiple times.

When a message is received, the provider has called reset method so that the message body is in read-only mode and reading values starts from the beginning of the message body.

Members

All members of ianywhere.qanywhere.client.QATextMessage, including all inherited members.

getText method

Synopsis

String **ianywhere.qanywhere.client.QATextMessage.getText()**
throws **QAException**

Throws

- Thrown if there is a problem retrieving the message text.

Remarks

Returns the message text.

If the message text exceeds the maximum size specified by the Interface QAManager MAX_IN_MEMORY_MESSAGE_SIZE property, this method returns null. In this case, use the readText method to read the text.

Returns

The message text, or null .

getTextLength method

Synopsis

long **ianywhere.qanywhere.client.QATextMessage.getTextLength()**
throws **QAException**

Throws

- Thrown if there is a problem retrieving the length of the message.

Remarks

Returns the length, in characters, of the message.

Returns

The length in characters of the message.

readText method

Synopsis

String **ianywhere.qanywhere.client.QATextMessage.readText(**
int *maxLength*
)
throws **QAException**

Parameters

- **maxLength** The maximum number of characters to read.

Throws

- Thrown if there is a problem retrieving the unread text.

Remarks

Returns unread text from the message.

Any additional unread text must be read by subsequent calls to this method. Text is read from the beginning of any unread text.

Returns

The text.

reset method

Synopsis

void **ianywhere.qanywhere.client.QATextMessage.reset()**
throws **QAException**

Throws

- Thrown if there is a problem resetting the text position of the message.

Remarks

Resets the text position of the message to the beginning.

setText method

Synopsis

void **ianywhere.qanywhere.client.QATextMessage.setText(**
String *value*
)
throws **QAException**

Parameters

- **value** The text to write to the message body.

Throws

- Thrown if there is a problem overwriting the message text.

Remarks

Overwrites the message text.

writeText method

Synopsis

void **ianywhere.qanywhere.client.QATextMessage.writeText(**
String *value*
)
throws **QAException**

Parameters

- **value** The text to append.

Throws

- Thrown if there is a problem appending the message text.

Remarks

Appends text to the text of the message.

writeText method

Synopsis

void **ianywhere.qanywhere.client.QATextMessage.writeText(**
String *value*,
int *length*
)
throws **QAException**

Parameters

- **value** The text to append.
- **length** The number of characters of text to append.

Throws

- Thrown if there is a problem appending the message text.

Remarks

Appends text to the text of the message.

writeText method

Synopsis

void **ianywhere.qanywhere.client.QATextMessage.writeText(**
String *value*,
int *offset*,
int *length*

)
throws **QAException**

Parameters

- **value** The text to append.
- **offset** The offset into value of the text to append.
- **length** The number of characters of text to append.

Throws

- Thrown if there is a problem appending the message text.

Remarks

Appends text to the text of the message.

Interface QATransactionalManager

Synopsis

public **ianywhere.qanywhere.client.QATransactionalManager**

Base classes

- "Interface QAManagerBase" on page 540

Remarks

The Interface QATransactionalManager class derives from Interface QAManagerBase and manages transactional QAnywhere messaging operations.

For a detailed description of derived behavior, see Interface QAManagerBase.

Interface QATransactionalManager instances can only be used for transactional acknowledgement. Use the commit method to commit all putMessage method and getMessage method invocations.

Members

All members of ianywhere.qanywhere.client.QATransactionalManager, including all inherited members.

- "browseMessages method" on page 542
- "browseMessagesByID method" on page 542
- "browseMessagesByQueue method" on page 543
- "browseMessagesBySelector method" on page 544
- "cancelMessage method" on page 544
- "close method" on page 545
- "commit method" on page 595
- "createBinaryMessage method" on page 545
- "createTextMessage method" on page 546
- "getBooleanStoreProperty method" on page 546

commit method

Synopsis

void **ianywhere.qanywhere.client.QATransactionalManager.commit()**
throws **QAException**

Throws

- Thrown if there is a problem committing.

Remarks

Commits the current transaction and begins a new transaction.

This method commits all putMessage method and getMessage method invocations.

Note: The first transaction begins with the call to open method.

open method

Synopsis

void **ianywhere.qanywhere.client.QATransactionalManager.open()**
throws **QAException**

Throws

- Thrown if there is a problem opening the manager.

Remarks

Opens a Interface QATransactionalManager instance.

This method must be the first method called after creating a manager.

See Also

Interface QATransactionalManager

rollback method

Synopsis

void **ianywhere.qanywhere.client.QATransactionalManager.rollback()**
throws **QAException**

Throws

- Thrown if there is a problem rolling back.

Remarks

Rolls back the current transaction and begins a new transaction.

This method rolls back all uncommited putMessage method and getMessage method invocations.

See Also

Interface QATransactionalManager

Interface QueueDepthFilter

Synopsis

public **ianywhere.qanywhere.client.QueueDepthFilter**

Remarks

Provides queue depth filter values for getQueueDepth method and getQueueDepth method.

Members

All members of ianywhere.qanywhere.client.QueueDepthFilter, including all inherited members.

ALL variable

Synopsis

final short **ianywhere.qanywhere.client.QueueDepthFilter.ALL**

Remarks

This filter specifies both incoming and outgoing messages.

System messages and expired messages are not included in any queue depth counts.

INCOMING variable

Synopsis

final short **ianywhere.qanywhere.client.QueueDepthFilter.INCOMING**

Remarks

This filter specifies only incoming messages.

An incoming message is defined as a message whose originator is different than the agent ID of the message store.

OUTGOING variable

Synopsis

final short **ianywhere.qanywhere.client.QueueDepthFilter.OUTGOING**

Remarks

This filter specifies only outgoing messages.

An outgoing message is defined as a message whose originator is the agent ID of the message store, and whose destination is not the agent ID of the message store.

Interface StatusCodes

Synopsis

public **ianywhere.qanywhere.client.StatusCodes**

Remarks

This interface defines a set of codes for the status of a message.

Members

All members of ianywhere.qanywhere.client.StatusCodes, including all inherited members.

CANCELLED variable

Synopsis

final int **ianywhere.qanywhere.client.StatusCodes.CANCELLED**

Remarks

The message has been cancelled.

This code applies to the STATUS variable.

EXPIRED variable

Synopsis

final int **ianywhere.qanywhere.client.StatusCodes.EXPIRED**

Remarks

The message has expired; the message was not received before its expiration time had passed.

This code applies to the STATUS variable.

FINAL variable

Synopsis

final int **ianywhere.qanywhere.client.StatusCodes.FINAL**

Remarks

The message has achieved a final state.

This code applies to the STATUS variable.

LOCAL variable

Synopsis

final int **ianywhere.qanywhere.client.StatusCodes.LOCAL**

Remarks

The message is addressed to the local message store and will not be transmitted to the server.

This code applies to the TRANSMISSION_STATUS variable.

PENDING variable

Synopsis

final int **ianywhere.qanywhere.client.StatusCodes.PENDING**

Remarks

The message has been sent but not received.

This code applies to the STATUS variable.

RECEIVED variable

Synopsis

final int **ianywhere.qanywhere.client.StatusCodes.RECEIVED**

Remarks

The message has been received and acknowledged by the receiver.

This code applies to the STATUS variable.

RECEIVING variable

Synopsis

final int **ianywhere.qanywhere.client.StatusCodes.RECEIVING**

Remarks

The message is in the process of being received, or it was received but not acknowledged.

This code applies to the STATUS variable.

TRANSMITTED variable

Synopsis

final int **ianywhere.qanywhere.client.StatusCodes.TRANSMITTED**

Remarks

The message has been transmitted to the server.

This code applies to the TRANSMISSION_STATUS variable.

TRANSMITTING variable

Synopsis

final int **ianywhere.qanywhere.client.StatusCodes.TRANSMITTING**

Remarks

The message is in the process of being transmitted to the server.

This code applies to the TRANSMISSION_STATUS variable.

UNRECEIVABLE variable

Synopsis

final int **ianywhere.qanywhere.client.StatusCodes.UNRECEIVABLE**

Remarks

The message has been marked as unreceivable.

The message is either malformed, or there were too many failed attempts to deliver it. This code applies to the STATUS variable.

UNTRANSMITTED variable

Synopsis

final int **ianywhere.qanywhere.client.StatusCodes.UNTRANSMITTED**

Remarks

The message has not been transmitted to the server.

This code applies to the TRANSMISSION_STATUS variable.

ianywhere.qanywhere.ws package

Class WSBase

Synopsis

public **ianywhere.qanywhere.ws.WSBase**

Remarks

This is the base class for the main web service proxy class generated by the mobile web service compiler.

Members

All members of ianywhere.qanywhere.ws.WSBase, including all inherited members.

WSBase method

Synopsis

ianywhere.qanywhere.ws.WSBase.WSBase(
String *iniFile*
)
throws **WSException**

Parameters

- **iniFile** A file containing configuration properties.

Throws

- Thrown if there is a problem constructing the "Class WSBase" on page 602.

Remarks

Constructor with configuration property file.

Valid configuration properties are:

`LOG_FILE` a file to which to log runtime information.

LOG_LEVEL a value between 0 and 6 that controls the verbosity of information logged, with 6 being the highest verbosity.

WS_CONNECTOR_ADDRESS the address of the web service connector in the MobiLink server. The default WS_CONNECTOR_ADDRESS is "ianywhere.connector.webservices\\".

WSBase method

Synopsis

ianywhere.qanywhere.ws.WSBase.WSBase()
throws **WSException**

Throws

- Thrown if there is a problem constructing the "Class WSBase" on page 602.

Remarks

Constructor.

clearRequestProperties method

Synopsis

void **ianywhere.qanywhere.ws.WSBase.clearRequestProperties()**

Remarks

Clears all request properties that have been set for this Class WSBase.

getResult method

Synopsis

WSResult **ianywhere.qanywhere.ws.WSBase.getResult(**
String *requestID*
)

Parameters

- **requestID** The ID of the web service request.

Remarks

Gets a Class WSResult object that represents the results of a web service request.

Returns

A Class WSResult instance representing the results of the web service request.

See Also

Class WSStatus

getServiceID method

Synopsis

String **ianywhere.qanywhere.ws.WSBase.getServiceID()**

Remarks

Gets the service ID for this instance of Class WSBase.

Returns

The service ID.

setListener method

Synopsis

void **ianywhere.qanywhere.ws.WSBase.setListener(**
  String *requestID*,
  WSListener *listener*
)

Parameters

- **requestID** The ID of the web service request to which to listen for results.
- **listener** The listener object that gets called when the result of the given web service request is available.

Remarks

Sets a listener for the results of a given web service request.

Listeners are typically used to get results of the asyncXYZ methods of the service.

To remove a listener, call setListener with `null` as the listener.

Note: This method replaces the listener set by any previous call to setListener.

setListener method

Synopsis

void **ianywhere.qanywhere.ws.WSBase.setListener(**
  WSListener *listener*
)

Parameters

- **listener** The listener object that gets called when the result of a web service request is available.

Remarks

Sets a listener for the results of all web service requests made by this instance of Class WSBase.

Listeners are typically used to get results of the asyncXYZ methods of the service.

To remove a listener, call setListener with `null` as the listener.

Note: This method replaces the listener set by any previous call to setListener.

setProperty method

Synopsis

void **ianywhere.qanywhere.ws.WSBase.setProperty(**
String *property*,
String *val*
)

Parameters

- **property** The property name to set.
- **val** The property value.

Remarks

Sets a configuration property for this instance of Class WSBase.

Configuration properties must be set before any asynchronous or synchronous web service request is made; after which this method has no effect.

Valid configuration properties are:

`LOG_FILE` a file to which to log runtime information.

`LOG_LEVEL` a value between 0 and 6 that controls the verbosity of information logged, with 6 being the highest verbosity.

`WS_CONNECTOR_ADDRESS` the address of the web service connector in the MobiLink server. The default is: "ianywhere.connector.webservices\\".

setQAManager method

Synopsis

void **ianywhere.qanywhere.ws.WSBase.setQAManager(**
QAManagerBase *mgr*
)

Parameters

- **mgr** The QAManagerBase to use.

Remarks

Sets the QAManagerBase that is used by this web service client to do web service requests.

Note: If you use an EXPLICIT_ACKNOWLEDGEMENT QAManager, you can acknowledge the result of an asynchronous web service request by calling the acknowledge() method of Class WSResult. The result of a synchronous web service request is automatically acknowledged, even in the case of an EXPLICIT_ACKNOWLEDGEMENT QAManager. If you use an IMPLICIT_ACKNOWLEDGEMENT QAManager, the result of any web service request is acknowledged automatically.

setRequestProperty method

Synopsis

void **ianywhere.qanywhere.ws.WSBase.setRequestProperty(**
  String *name*,
  Object *value*
)

Parameters

- **name** The property name to set.
- **value** The property value.

Remarks

Sets a request property for webservice requests made by this Class WSBase.

A request property is set on each QAMessage that is sent by this Class WSBase, until the property is cleared. A request property is cleared by setting it to a null value. The type of the message property is determined by the class of the value parameter. For example, if value is an instance of Integer, then setIntProperty is used to set the property on the QAMessage.

setServiceID method

Synopsis

void **ianywhere.qanywhere.ws.WSBase.setServiceID(**
  String *serviceID*
)

Parameters

- **serviceID** The service ID.

Remarks

Sets a user-defined ID for this instance of Class WSBase.

The service ID should be set to a value unique to this instance of Class WSBase. It is used internally to form a queue name for sending and receiving web service requests. Therefore, the service ID should be persisted between application sessions, in order to retrieve results of web service requests made in a previous session.

Class WSException

Synopsis

public **ianywhere.qanywhere.ws.WSException**

Derived classes

- "Class WSFaultException" on page 608

Remarks

This class represents an exception that occurred during processing of a web service request.

Members

All members of ianywhere.qanywhere.ws.WSException, including all inherited members.

- "getErrorCode method" on page 608
- "WSException method" on page 607
- "WSException method" on page 607
- "WSException method" on page 608

WSException method

Synopsis

ianywhere.qanywhere.ws.WSException.WSException(
String *msg*
)

Parameters

- **msg** The error message.

Remarks

Constructs a new exception with the specified error message.

WSException method

Synopsis

ianywhere.qanywhere.ws.WSException.WSException(
String *msg*,
int *errorCode*
)

Parameters

- **msg** The error message.
- **errorCode** The error code.

Remarks

Constructs a new exception with the specified error message and error code.

WSException method

Synopsis

ianywhere.qanywhere.ws.WSException.WSException(
Exception *exception*
)

Parameters

- **exception** The exception.

Remarks

Constructs a new exception.

getErrorCode method

Synopsis

int **ianywhere.qanywhere.ws.WSException.getErrorCode()**

Remarks

Gets the error code associated with this exception.

Returns

The error code associated with this exception.

Class WSFaultException

Synopsis

public **ianywhere.qanywhere.ws.WSFaultException**

Base classes

- "Class WSException" on page 607

Remarks

This class represents a SOAP Fault exception from the web service connector.

Members

All members of ianywhere.qanywhere.ws.WSFaultException, including all inherited members.

- "getErrorCode method" on page 608
- "WSException method" on page 607
- "WSException method" on page 607
- "WSException method" on page 608

- "WSFaultException method" on page 609

WSFaultException method

Synopsis

ianywhere.qanywhere.ws.WSFaultException.WSFaultException(
String *msg*
)

Parameters

- **msg** The error message.

Remarks

Constructs a new exception with the specified error message.

Interface WSListener

Synopsis

public **ianywhere.qanywhere.ws.WSListener**

Remarks

This class represents a listener for results of web service requests.

Members

All members of ianywhere.qanywhere.ws.WSListener, including all inherited members.

- "onException method" on page 609
- "onResult method" on page 610

onException method

Synopsis

void **ianywhere.qanywhere.ws.WSListener.onException(**
WSException *e*,
WSResult *wsResult*
)

Parameters

- **e** The "Class WSException" on page 607 that occurred during processing of the result.
- **wsResult** A "Class WSResult" on page 610, from which the request ID may be obtained. Values of this "Class WSResult" on page 610 are not defined.

Remarks

Called when an exception occurs during processing of the result of an asynchronous web service request.

onResult method

Synopsis

void **ianywhere.qanywhere.ws.WSListener.onResult(**
WSResult *wsResult*
)

Parameters

- **wsResult** The "Class WSResult" on page 610 describing the result of a web service request.

Remarks

Called with the result of an asynchronous web service request.

Class WSResult

Synopsis

public **ianywhere.qanywhere.ws.WSResult**

Remarks

This class represents the results of a web service request.

- It is passed to the onResult method.
- It is returned by an asyncXYZ method of the service proxy generated by the compiler.
- It is obtained by calling getResult method with a specific request ID.

A Class WSResult object is obtained in one of three ways:

Members

All members of ianywhere.qanywhere.ws.WSResult, including all inherited members.

acknowledge method

Synopsis

void **ianywhere.qanywhere.ws.WSResult.acknowledge()**

Remarks

Acknowledges that this Class WSResult has been processed.

This method is only useful when an EXPLICIT_ACKNOWLEDGEMENT QAManager is being used.

getArrayValue method

Synopsis

WSSerializable[] **ianywhere.qanywhere.ws.WSResult.getArrayValue(**
String *parentName*
)
throws **WSException**

Parameters

- **parentName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets an array of complex types value from this Class WSResult.

Returns

The value.

getBigDecimalArrayValue method

Synopsis

BigDecimal[] **ianywhere.qanywhere.ws.WSResult.getBigDecimalArrayValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a BigDecimal array value from this Class WSResult.

Returns

The value.

getBigDecimalValue method

Synopsis

BigDecimal **ianywhere.qanywhere.ws.WSResult.getBigDecimalValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a BigDecimal value from this Class WSResult.

Returns

The value.

getBigIntegerArrayValue method

Synopsis

BigInteger[] **ianywhere.qanywhere.ws.WSResult.getBigIntegerArrayValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a BigInteger array value from this Class WSResult.

Returns

The value.

getBigIntegerValue method

Synopsis

BigInteger **ianywhere.qanywhere.ws.WSResult.getBigIntegerValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a BigInteger value from this Class WSResult.

Returns

The value.

getBooleanArrayValue method

Synopsis

Boolean[] **ianywhere.qanywhere.ws.WSResult.getBooleanArrayValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a java.lang.Boolean array value from this Class WSResult.

Returns

The value.

getBooleanValue method

Synopsis

Boolean **ianywhere.qanywhere.ws.WSResult.getBooleanValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a java.lang.Boolean value from this Class WSResult.

Returns

The value.

getByteArrayValue method

Synopsis

Byte[] **ianywhere.qanywhere.ws.WSResult.getByteArrayValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a java.lang.Byte array value from this Class WSResult.

Returns

The value.

getByteValue method

Synopsis

Byte **ianywhere.qanywhere.ws.WSResult.getByteValue(**
  String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a java.lang.Byte value from this Class WSResult.

Returns

The value.

getCharacterArrayValue method

Synopsis

Character[] **ianywhere.qanywhere.ws.WSResult.getCharacterArrayValue(**
  String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a java.lang.Character array value from this Class WSResult.

Returns

The value.

getCharacterValue method

Synopsis

Character **ianywhere.qanywhere.ws.WSResult.getCharacterValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a java.lang.Character value from this Class WSResult.

Returns

The value.

getDoubleArrayValue method

Synopsis

Double[] **ianywhere.qanywhere.ws.WSResult.getDoubleArrayValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a java.lang.Double array value from this Class WSResult.

Returns

The value.

getDoubleValue method

Synopsis

Double **ianywhere.qanywhere.ws.WSResult.getDoubleValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a java.lang.Double value from this Class WSResult.

Returns

The value.

getErrorMessage method

Synopsis

String **ianywhere.qanywhere.ws.WSResult.getErrorMessage()**

Remarks

Gets the error message.

Returns

The error message.

getFloatArrayValue method

Synopsis

Float[] **ianywhere.qanywhere.ws.WSResult.getFloatArrayValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a java.lang.Float array value from this Class WSResult.

Returns

The value.

getFloatValue method

Synopsis

Float **ianywhere.qanywhere.ws.WSResult.getFloatValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a java.lang.Float value from this Class WSResult.

Returns

The value.

getIntegerArrayValue method

Synopsis

Integer[] **ianywhere.qanywhere.ws.WSResult.getIntegerArrayValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a java.lang.Integer array value from this Class WSResult.

Returns

The value.

getIntegerValue method

Synopsis

Integer **ianywhere.qanywhere.ws.WSResult.getIntegerValue(**
  String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a java.lang.Integer value from this Class WSResult.

Returns

The value.

getLongArrayValue method

Synopsis

Long[] **ianywhere.qanywhere.ws.WSResult.getLongArrayValue(**
  String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a java.lang.Long array value from this Class WSResult.

Returns

The value.

getLongValue method

Synopsis

Long **ianywhere.qanywhere.ws.WSResult.getLongValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a java.lang.Long value from this Class WSResult.

Returns

The value.

getObjectArrayValue method

Synopsis

Object[] **ianywhere.qanywhere.ws.WSResult.getObjectArrayValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets an array of complex types value from this Class WSResult.

Returns

The value.

getObjectValue method

Synopsis

Object **ianywhere.qanywhere.ws.WSResult.getObjectValue(**
  String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets value of a complex type from this Class WSResult.

Returns

The value.

getPrimitiveBooleanArrayValue method

Synopsis

boolean[] **ianywhere.qanywhere.ws.WSResult.getPrimitiveBooleanArrayValue(**
  String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a boolean array value from this Class WSResult.

Returns

The value.

getPrimitiveBooleanValue method

Synopsis

boolean **ianywhere.qanywhere.ws.WSResult.getPrimitiveBooleanValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a boolean value from this Class WSResult.

Returns

The value.

getPrimitiveByteArrayValue method

Synopsis

byte[] **ianywhere.qanywhere.ws.WSResult.getPrimitiveByteArrayValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a byte array value from this Class WSResult.

Returns

The value.

getPrimitiveByteValue method

Synopsis

byte **ianywhere.qanywhere.ws.WSResult.getPrimitiveByteValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a byte value from this Class WSResult.

Returns

The value.

getPrimitiveCharArrayValue method

Synopsis

char[] **ianywhere.qanywhere.ws.WSResult.getPrimitiveCharArrayValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a char array value from this Class WSResult.

Returns

The value.

getPrimitiveCharValue method

Synopsis

char **ianywhere.qanywhere.ws.WSResult.getPrimitiveCharValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a char value from this Class WSResult.

Returns

The value.

getPrimitiveDoubleArrayValue method

Synopsis

double[] **ianywhere.qanywhere.ws.WSResult.getPrimitiveDoubleArrayValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a double array value from this Class WSResult.

Returns

The value.

getPrimitiveDoubleValue method

Synopsis

double **ianywhere.qanywhere.ws.WSResult.getPrimitiveDoubleValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a double value from this Class WSResult.

Returns

The value.

getPrimitiveFloatArrayValue method

Synopsis

float[] **ianywhere.qanywhere.ws.WSResult.getPrimitiveFloatArrayValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a float array value from this Class WSResult.

Returns

The value.

getPrimitiveFloatValue method

Synopsis

float **ianywhere.qanywhere.ws.WSResult.getPrimitiveFloatValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a float value from this Class WSResult.

Returns

The value.

getPrimitiveIntArrayValue method

Synopsis

int[] **ianywhere.qanywhere.ws.WSResult.getPrimitiveIntArrayValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets an int array value from this Class WSResult.

Returns

The value.

getPrimitiveIntValue method

Synopsis

int **ianywhere.qanywhere.ws.WSResult.getPrimitiveIntValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets an int value from this Class WSResult.

Returns

The value.

getPrimitiveLongArrayValue method

Synopsis

long[] **ianywhere.qanywhere.ws.WSResult.getPrimitiveLongArrayValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a long array value from this Class WSResult.

Returns

The value.

getPrimitiveLongValue method

Synopsis

long **ianywhere.qanywhere.ws.WSResult.getPrimitiveLongValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a long value from this Class WSResult.

Returns

The value.

getPrimitiveShortArrayValue method

Synopsis

short[] **ianywhere.qanywhere.ws.WSResult.getPrimitiveShortArrayValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a short array value from this Class WSResult.

Returns

The value.

getPrimitiveShortValue method

Synopsis

short **ianywhere.qanywhere.ws.WSResult.getPrimitiveShortValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a short value from this Class WSResult.

Returns

The value.

getRequestID method

Synopsis

String **ianywhere.qanywhere.ws.WSResult.getRequestID()**

Remarks

Gets the request ID that this Class WSResult represents.

This request ID should be persisted between runs of the application if it is desired to obtain a Class WSResult corresponding to a web service request in a run of the application different from when the request was made.

Returns

The request ID.

getShortArrayValue method

Synopsis

Short[] **ianywhere.qanywhere.ws.WSResult.getShortArrayValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a java.lang.Short array value from this Class WSResult.

Returns

The value.

getShortValue method

Synopsis

Short **ianywhere.qanywhere.ws.WSResult.getShortValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a java.lang.Short value from this Class WSResult.

Returns

The value.

getStatus method

Synopsis

int **ianywhere.qanywhere.ws.WSResult.getStatus()**

Remarks

Gets the status of this Class WSResult.

Returns

The status code.

See Also

Class WSStatus

getStringArrayValue method

Synopsis

String[] **ianywhere.qanywhere.ws.WSResult.getStringArrayValue(**
  String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a String array value from this Class WSResult.

Returns

The value.

getStringValue method

Synopsis

String **ianywhere.qanywhere.ws.WSResult.getStringValue(**
  String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets a String value from this Class WSResult.

Returns

The value.

getValue method

Synopsis

Object **ianywhere.qanywhere.ws.WSResult.getValue(**
String *elementName*
)
throws **WSException**

Parameters

- **elementName** The element name in the WSDL document of this value.

Throws

- Thrown if there is a problem getting the value.

Remarks

Gets the value of a complex type from this Class WSResult.

Returns

The value.

Class WSStatus

Synopsis

public **ianywhere.qanywhere.ws.WSStatus**

Remarks

This class defines codes for the status of a web service request.

Members

All members of ianywhere.qanywhere.ws.WSStatus, including all inherited members.

- "STATUS_ERROR variable" on page 633
- "STATUS_QUEUED variable" on page 634
- "STATUS_RESULT_AVAILABLE variable" on page 634
- "STATUS_SUCCESS variable" on page 634

STATUS_ERROR variable

Synopsis

final int **ianywhere.qanywhere.ws.WSStatus.STATUS_ERROR**

Remarks

There was an error processing the request.

STATUS_QUEUED variable

Synopsis

final int **ianywhere.qanywhere.ws.WSStatus.STATUS_QUEUED**

Remarks

The request has been queued for delivery to the server.

STATUS_RESULT_AVAILABLE variable

Synopsis

final int **ianywhere.qanywhere.ws.WSStatus.STATUS_RESULT_AVAILABLE**

Remarks

The result of the request is available.

STATUS_SUCCESS variable

Synopsis

final int **ianywhere.qanywhere.ws.WSStatus.STATUS_SUCCESS**

Remarks

The request was successful.

CHAPTER 15

QAnywhere SQL API Reference

Contents

About this chapter

This chapter describes the QAnywhere SQL API.

QAnywhere SQL is a collection of stored procedures that implement a messaging API in SQL.

Message properties, headers, and content

This section documents QAnywhere SQL stored procedures that help you set message headers, message content, and message properties.

Message headers

You can use the following stored procedures to get and set message header information.

☞ For more information about message headers, see "Message headers" on page 206.

ml_qa_getaddress

Description

Returns the QAnywhere address of the message.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.

Return value

The QAnywhere message address as VARCHAR(128). QAnywhere message addresses take the form *id* *queue-name*.

You can read this header after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "QAnywhere message addresses" on page 50
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666

Example

In the following example, a message is received and its address is output to the database console:

```
begin
    declare @msgid varchar(128);
    declare @addr varchar(128);
    set @msgid = ml_qa_getmessage( 'myaddress' );
    set @addr = ml_qa_getaddress( @msgid );
    message 'message to address ' || @addr || ' received';
    commit;
end
```


ml_qa_getexpiration

Description

Returns the expiration time of the message.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.

Return value

The expiration time as TIMESTAMP. Returns NULL if there is no expiration.

Remarks

After completion of ml_qa_putmessage, a message expires if it is not received by the intended recipient in the specified time. The message may then be deleted using default QAnywhere delete rules.

You can read this header after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "Message delete rules" on page 239
- "Sending QAnywhere messages" on page 66
- "ml_qa_setexpiration" on page 642
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666

Example

In the following example, a message is received and the message expiration is output to the database console:

```
begin
    declare @msgid varchar(128);
    declare @expires timestamp;
    set @msgid = ml_qa_getmessage( 'myaddress' );
    set @expires = ml_qa_getexpiration( @msgid );
    message 'message would have expired at ' || @expires || ' if it had not
been received';
    commit;
end
```

ml_qa_getinreplytoid

Description

Returns the in-reply-to ID for the message.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.

Return value

The in-reply-to ID as VARCHAR(128).

Remarks

A client can use the InReplyToID header field to link one message with another. A typical use is to link a response message with its request message.

The in-reply-to ID is the ID of the message that this message is replying to.

You can read this header after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "ml_qa_setinreplytoid" on page 643
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666

Example

In the following example, a message is received and the in-reply-to-id of the message is output to the database console:

```
begin
    declare @msgid varchar(128);
    declare @inreplytoid varchar(128);
    set @msgid = ml_qa_getmessage( 'myaddress' );
    set @inreplytoid = ml_qa_getinreplytoid( @msgid );
    message 'message is likely a reply to the message with id ' ||
@inreplytoid;
    commit;
end
```

ml_qa_getpriority

Description

Returns the priority level of the message.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.

Return value

The priority level as INTEGER.

Remarks

The QAnywhere API defines ten levels of priority value, with 0 as the lowest priority and 9 as the highest. Clients should consider priorities 0-4 as gradations of normal priority and priorities 5-9 as gradations of expedited priority.

You can read this header after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "ml_qa_setpriority" on page 644
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666

Example

In the following example, a message is received and the priority of the message is output to the database console:

```
begin
    declare @msgid varchar(128);
    declare @priority integer;
    set @msgid = ml_qa_getmessage( 'myaddress' );
    set @priority = ml_qa_getpriority( @msgid );
    message 'a message with priority ' || @priority || ' has been received';
    commit;
end
```

ml_qa_getredelivered

Description

Returns a value indicating whether this message has previously been received but not acknowledged.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.

Return value

The redelivered value as BIT. A value of **1** indicates that the message is being redelivered; **0** indicates that it is not being redelivered.

Remarks

A message may be redelivered if it was previously received but not acknowledged. For example, the message was received but the application receiving the message did not complete processing the message content

before it crashed. In these cases, QAnywhere marks the message as redelivered to alert the receiver that the message might be partly processed.

For example, assume that the receipt of a message occurs in three steps:

1. An application using a non-transactional QAnywhere manager receives the message.
2. The application writes the message content and message ID to a database table called T1, and commits the change.
3. The application acknowledges the message.

If the application fails between steps 1 and 2 or between steps 2 and 3, the message is redelivered when the application restarts.

If the failure occurs between steps 1 and 2, you should process the redelivered message by running steps 2 and 3. If the failure occurs between steps 2 and 3, then the message is already processed and you only need to acknowledge it.

To determine what happened when the application fails, you can have the application call ml_qa_getredelivered to check if the message has been previously redelivered. Only messages that are redelivered need to be looked up in table T1. This is more efficient than having the application access the received message's message ID to check whether the message is in the table T1, because application failures are rare.

You can read this header after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- “ml_qa_createmessage” on page 666
- “ml_qa_getmessage” on page 666

Example

In the following example, a message is received; if the message was previously delivered but not received, the message ID is output to the database console:

```
begin
    declare @msgid varchar(128);
    declare @redelivered bit;
    set @msgid = ml_qa_getmessage( 'myaddress' );
    set @redelivered = ml_qa_getredelivered( @msgid );
    if @redelivered = 1 then
        message 'message with message ID ' || @msgid || ' has been
redelivered';
    end if;
    commit;
end
```

ml_qa_getreplytoaddress

Description

Returns the address to which a reply to this message should be sent.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.

Return value

The reply address as VARCHAR(128).

Remarks

You can read this header after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "ml_qa_setreplytoaddress" on page 645
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666

Example

In the following example, if the received message has a reply-to address, then a message is sent to the reply-to-address with the content 'message received':

```
begin
    declare @msgid varchar(128);
    declare @rmsgid varchar(128);
    declare @replytoaddr varchar(128);
    set @msgid = ml_qa_getmessage( 'myaddress' );
    set @replytoaddr = ml_qa_getreplytoaddress( @msgid );
    if @replytoaddr is not null then
        set @rmsgid = ml_qa_createmessage();
        call ml_qa_settextcontent( @rmsgid, 'message received' );
        call ml_qa_putmessage( @rmsgid, @replytoaddr );
    end if;
    commit;
end
```

ml_qa_gettimestamp

Description

Returns the creation time of the message.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.

Return value

The message creation time as TIMESTAMP.

Remarks

The Timestamp header field contains the time a message was created. It is a coordinated universal time (UTC). It is not the time the message was actually transmitted, because the actual send may occur later due to transactions or other client-side queuing of messages.

You can read this header after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666

Example

In the following example, a message is received and the creation time of the message is output to the database console:

```
begin
    declare @msgid varchar(128);
    declare @ts timestamp;
    set @msgid = ml_qa_getmessage( 'myaddress' );
    set @ts = ml_qa_gettimestamp( @msgid );
    message 'message received with create time: ' || @ts ;
    commit;
end
```

ml_qa_setexpiration

Description

Sets the expiration time for a message.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Expiration	TIMESTAMP

Remarks

You can read this header after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "ml_qa_getexpiration" on page 637
- "ml_qa_createmessage" on page 666

- "ml_qa_getmessage" on page 666

Example

In the following example, a message is created so that if it is not delivered within the next 3 days it expires:

```
begin
    declare @msgid varchar(128);
    set @msgid = ml_qa_createmessage();
    call ml_qa_setexpiration( @msgid, dateadd( day, 3, current timestamp ) );
    call ml_qa_settextcontent( @msgid, 'time-limited offer' );
    call ml_qa_putmessage( @msgid, 'someclient\someaddress' );
    commit;
end
```

ml_qa_setinreplytoid

Description

Sets the in-reply-to ID of this message.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	in-reply-to ID	VARCHAR(128)

Remarks

An in-reply-to ID is similar to the in-reply-to IDs that are used by email systems to track replies.

Typically you set the in-reply-to ID to be the message ID of the message to which this message is replying, if any.

A client can use the InReplyToID header field to link one message with another. A typical use is to link a response message with its request message.

You cannot alter this header after the message has been sent.

See also

- "ml_qa_getinreplytoid" on page 637
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666

Example

In the following example, when a message is received that contains a reply-to-address, a reply message is created and sent containing the message ID in the in-reply-to-id:

```
begin
    declare @msgid varchar(128);
    declare @rmsgid varchar(128);
```



```
    declare @replyaddr varchar(128);
    set @msgid = ml_qa_getmessage( 'myaddress' );
    set @replyaddr = ml_qa_getreplyaddress( @msgid );
    if @replyaddr is not null then
        set @rmsgid = ml_qa_createmessage();
        call ml_qa_settextcontent( @rmsgid, 'message received' );
        call ml_qa_setinreplytoid( @rmsgid, @msgid );
        call ml_qa_putmessage( @rmsgid, @replyaddr );
    end if;
    commit;
end
```

ml_qa_setpriority

Description

Sets the priority of a message.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Priority	INTEGER

Remarks

The QAnywhere API defines ten levels of priority value, with 0 as the lowest priority and 9 as the highest. Clients should consider priorities 0-4 as gradations of normal priority and priorities 5-9 as gradations of expedited priority.

You cannot alter this header after the message has been sent.

See also

- “ml_qa_getpriority” on page 638
- “ml_qa_createmessage” on page 666
- “ml_qa_getmessage” on page 666

Example

The following example sends a high priority message:

```
begin
    declare @msgid varchar(128);
    set @msgid = ml_qa_createmessage();
    call ml_qa_setpriority( @msgid, 9 );
    call ml_qa_settextcontent( @msgid, 'priority content' );
    call ml_qa_putmessage( @msgid, 'someclient\someaddress' );
    commit;
end
```


ml_qa_setreplytoaddress

Description

Sets the reply-to address of the message.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Reply address	VARCHAR(128)

Remarks

You cannot alter this header after the message has been sent.

See also

- "ml_qa_getreplytoaddress" on page 640
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666

Example

In the following example, a reply-to-address is added to a message. The recipient of the message can then use that reply-to-address to create a reply.

```
begin
    declare @msgid varchar(128);
    set @msgid = ml_qa_createmessage();
    call ml_qa_setreplytoaddress( @msgid, 'myaddress' );
    call ml_qa_settextcontent( @msgid, 'some content' );
    call ml_qa_putmessage( @msgid, 'someclient\someaddress' );
    commit;
end
```

Message properties

You can use the following stored procedures to get and set your custom message properties, or to get pre-defined message properties.

☞ For more information about message properties, see "Message properties" on page 209.

ml_qa_getbooleanproperty

Description

Returns the specified message property as a SQL BIT data type.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Property name	VARCHAR(128)

Return value

The property value as BIT.

Remarks

If the message property value is out of range, then a SQL error with SQLSTATE 22003 occurs.

You can read this property after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "ml_qa_setbooleanproperty" on page 653
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666
- "Custom message properties" on page 211

Example

In the following example, a message is received and the value of the boolean property mybooleanproperty is output to the database console:

```
begin
    declare @msgid varchar(128);
    declare @prop bit;
    set @msgid = ml_qa_getmessage( 'myaddress' );
    set @prop = ml_qa_getbooleanproperty( @msgid, 'mybooleanproperty' );
    message 'message property mybooleanproperty is set to ' || @prop;
    commit;
end
```

ml_qa_getbyteproperty

Description

Returns the specified message property as a SQL TINYINT data type.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Property name	VARCHAR(128)

Return value

The property value as TINYINT.

Remarks

If the message property value is out of range, then a SQL error with SQLSTATE 22003 occurs.

You can read this property after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "ml_qa_setbyteproperty" on page 654
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666
- "Custom message properties" on page 211

Example

In the following example, a message is received and the value of byte property mybyteproperty is output to the database console:

```
begin
    declare @msgid varchar(128);
    declare @prop tinyint;
    set @msgid = ml_qa_getmessage( 'myaddress' );
    set @prop = ml_qa_getbyteproperty( @msgid, 'mybyteproperty' );
    message 'message property mybyteproperty is set to ' || @prop;
    commit;
end
```

ml_qa_getdoubleproperty

Description

Returns the specified message property as a SQL DOUBLE data type.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Property name	VARCHAR(128)

Return value

The property value as DOUBLE.

Remarks

If the message property value is out of range, then a SQL error with SQLSTATE 22003 occurs.

You can read this property after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "ml_qa_setdoubleproperty" on page 655
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666
- "Custom message properties" on page 211

Example

In the following example, a message is received and the value of double property mydoubleproperty is output to the database console:

```
begin
    declare @msgid varchar(128);
    declare @prop double;
    set @msgid = ml_qa_getmessage( 'myaddress' );
    set @prop = ml_qa_getdoubleproperty( @msgid, 'mydoubleproperty' );
    message 'message property mydoubleproperty is set to ' || @prop;
    commit;
end
```

ml_qa_getfloatproperty

Description

Returns the specified message property as a SQL FLOAT data type.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Property name	VARCHAR(128)

Return value

The property value as FLOAT.

Remarks

If the message property value is out of range, then a SQL error with SQLSTATE 22003 occurs.

You can read this property after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "ml_qa_setfloatproperty" on page 655
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666

- "Custom message properties" on page 211

Example

In the following example, a message is received and the value of float property myfloatproperty is output to the database console:

```
begin
    declare @msgid varchar(128);
    declare @prop float;
    set @msgid = ml_qa_getmessage( 'myaddress' );
    set @prop = ml_qa_getfloatproperty( @msgid, 'myfloatproperty' );
    message 'message property myfloatproperty is set to ' || @prop;
    commit;
end
```

ml_qa_getintproperty

Description

Returns the specified message property as a SQL INTEGER data type.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Property name	VARCHAR(128)

Return value

The property value as INTEGER.

Remarks

If the message property value is out of range, then a SQL error with SQLSTATE 22003 occurs.

You can read this property after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "ml_qa_setintproperty" on page 656
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666
- "Custom message properties" on page 211

Example

In the following example, a message is received and the value of integer property myintproperty is output to the database console:

```
begin
```

```
        declare @msgid varchar(128);
        declare @prop integer;
        set @msgid = ml_qa_getmessage( 'myaddress' );
        set @prop = ml_qa_getintproperty( @msgid, 'myintproperty' );
        message 'message property myintproperty is set to ' || @prop;
        commit;
    end
```

ml_qa_getlongproperty

Description

Returns the specified message property as a SQL BIGINT data type.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Property name	VARCHAR(128)

Return value

The property value as BIGINT.

Remarks

If the message property value is out of range, then a SQL error with SQLSTATE 22003 occurs.

You can read this property after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- “ml_qa_setlongproperty” on page 657
- “ml_qa_createmessage” on page 666
- “ml_qa_getmessage” on page 666
- “Custom message properties” on page 211

ml_qa_getpropertynames

Description

Retrieves the property names of the specified message.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.

Remarks

This stored procedure opens a result set over the property names of the specified message. The message ID parameter must be that of a message that has been received.

The result set is a single VARCHAR(128) column, where each row contains the name of a message property. QAnywhere reserved property names (those with the prefix "ias_" or "QA") are not returned.

You can read this property after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666
- "Custom message properties" on page 211

Example

The following example declares a cursor over the result set of property names for a message that has the message ID msgid. It then gets a message that has the address someAddress; opens a cursor to access the property names of the message; and finally fetches the next property name.

```
begin
    declare prop_name_cursor cursor for
        call ml_qa_getpropertynames( @msgid );
    declare @msgid varchar(128);
    declare @name varchar(128);

    set @msgid = ml_qa_getmessage( 'someAddress' );
    open prop_name_cursor;
    lp: loop
        fetch next prop_name_cursor into name;
        if sqlcode <> 0 then leave lp end if;
        ...
    end loop;
    close prop_name_cursor;
end
```

ml_qa_getshortproperty

Description

Returns the specified message property as a SQL SMALLINT data type.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Property name	VARCHAR(128)

Return value

The property value as SMALLINT.

Remarks

If the message property value is out of range, then a SQL error with SQLSTATE 22003 occurs.

You can read this property after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "ml_qa_setshortproperty" on page 658
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666
- "Custom message properties" on page 211

Example

In the following example, a message is received and the value of the short property myshortproperty is output to the database console:

```
begin
    declare @msgid varchar(128);
    declare @prop smallint;
    set @msgid = ml_qa_getmessage( 'myaddress' );
    set @prop = ml_qa_getshortproperty( @msgid, 'myshortproperty' );
    message 'message property myshortproperty is set to ' || @prop;
    commit;
end
```

ml_qa_getstringproperty

Description

Returns the specified message property as a SQL LONG VARCHAR data type.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Property name	VARCHAR(128)

Return value

The property value as LONG VARCHAR.

Remarks

If the message property value is out of range, then a SQL error with SQLSTATE 22003 occurs.

You can read this property after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "ml_qa_setstringproperty" on page 658
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666
- "Custom message properties" on page 211

Example

In the following example, a message is received and the value of the string property mystringproperty is output to the database console:

```
begin
    declare @msgid varchar(128);
    declare @prop long varchar;
    set @msgid = ml_qa_getmessage( 'myaddress' );
    set @prop = ml_qa_getstringproperty( @msgid, 'mystringproperty' );
    message 'message property mystringproperty is set to ' || @prop;
    commit;
end
```

ml_qa_setbooleanproperty

Description

Sets the specified message property from a SQL BIT data type.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Property name	VARCHAR(128)
3	Property value	BIT

Remarks

You cannot alter this property after the message has been sent.

See also

- "ml_qa_getbooleanproperty" on page 645
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666
- "Custom message properties" on page 211

Example

In the following example, a message is created, the boolean properties mybooleanproperty1 and mybooleanproperty2 are set, and the message is sent to the address someclient\someaddress:

```
begin
    declare @msgid varchar(128);
    set @msgid = ml_qa_createmessage();
    call ml_qa_setbooleanproperty( @msgid, 'mybooleanproperty1', 0 );
    call ml_qa_setbooleanproperty( @msgid, 'mybooleanproperty2', 1 );
    call ml_qa_putmessage( @msgid, 'someclient\someaddress' );
    commit;
end
```

ml_qa_setbyteproperty

Description

Sets the specified message property from a SQL TINYINT data type.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Property name	VARCHAR(128)
3	Property value	TINYINT

Remarks

You cannot alter this property after the message has been sent.

See also

- “ml_qa_getbyteproperty” on page 646
- “ml_qa_createmessage” on page 666
- “ml_qa_getmessage” on page 666
- “Custom message properties” on page 211

Example

In the following example, a message is created, the byte properties mybyteproperty1 and mybyteproperty2 are set, and the message is sent to the address someclient\someaddress:

```
begin
    declare @msgid varchar(128);
    set @msgid = ml_qa_createmessage();
    call ml_qa_setbyteproperty( @msgid, 'mybyteproperty1', 0 );
    call ml_qa_setbyteproperty( @msgid, 'mybyteproperty2', 255 );
    call ml_qa_putmessage( @msgid, 'someclient\someaddress' );
    commit;
end
```

ml_qa_setdoubleproperty

Description

Sets the specified message property from a SQL DOUBLE data type.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Property name	VARCHAR(128)
3	Property value	DOUBLE

Remarks

You cannot alter this property after the message has been sent.

See also

- "ml_qa_getdoubleproperty" on page 647
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666
- "Custom message properties" on page 211

Example

In the following example, a message is created, the double properties mydoubleproperty1 and mydoubleproperty2 are set, and the message is sent to the address someclient\someaddress:

```
begin
    declare @msgid varchar(128);
    set @msgid = ml_qa_createmessage();
    call ml_qa_setdoubleproperty( @msgid, 'mydoubleproperty1', -12.34e-56 );
    call ml_qa_setdoubleproperty( @msgid, 'mydoubleproperty2', 12.34e56 );
    call ml_qa_putmessage( @msgid, 'someclient\someaddress' );
    commit;
end
```

ml_qa_setfloatproperty

Description

Sets the specified message property from a SQL FLOAT data type.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.

Item	Description	Remarks
2	Property name	VARCHAR(128)
3	Property value	FLOAT

Remarks

You cannot alter this property after the message has been sent.

See also

- "ml_qa_getfloatproperty" on page 648
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666
- "Custom message properties" on page 211

Example

In the following example, a message is created, the float properties myfloatproperty1 and myfloatproperty2 are set, and the message is sent to the address someclient\someaddress:

```
begin
    declare @msgid varchar(128);
    set @msgid = ml_qa_createmessage();
    call ml_qa_setfloatproperty( @msgid, 'myfloatproperty1', -1.3e-5 );
    call ml_qa_setfloatproperty( @msgid, 'myfloatproperty2', 1.3e5 );
    call ml_qa_putmessage( @msgid, 'someclient\someaddress' );
    commit;
end
```

ml_qa_setintproperty

Description

Sets the specified message property from a SQL INTEGER data type.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Property name	VARCHAR(128)
3	Property value	INTEGER

Remarks

You cannot alter this property after the message has been sent.

See also

- "ml_qa_getintproperty" on page 649

- “ml_qa_createmessage” on page 666
- “ml_qa_getmessage” on page 666
- “Custom message properties” on page 211

Example

In the following example, a message is created, the integer properties myintproperty1 and myintproperty2 are set, and the message is sent to the address someclient\someaddress:

```
begin
    declare @msgid varchar(128);
    set @msgid = ml_qa_createmessage();
    call ml_qa_setintproperty( @msgid, 'myintproperty1', -1234567890 );
    call ml_qa_setintproperty( @msgid, 'myintproperty2',  1234567890 );
    call ml_qa_putmessage( @msgid, 'someclient\someaddress' );
    commit;
end
```

ml_qa_setlongproperty

Description

Sets the specified message property from a SQL BIGINT data type.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Property name	VARCHAR(128)
3	Property value	BIGINT

Remarks

You cannot alter this property after the message has been sent.

See also

- “ml_qa_getlongproperty” on page 650
- “ml_qa_createmessage” on page 666
- “ml_qa_getmessage” on page 666
- “Custom message properties” on page 211

Example

In the following example, a message is created, the long properties mylongproperty1 and mylongproperty2 are set, and the message is sent to the address someclient\someaddress:

```
begin
    declare @msgid varchar(128);
    set @msgid = ml_qa_createmessage();
    call ml_qa_setlongproperty( @msgid, 'mylongproperty1',
-12345678900987654321 );
```



```
    call ml_qa_setlongproperty( @msgid, 'mylongproperty2',
1234567890987654321 );
    call ml_qa_putmessage( @msgid, 'someclient\someaddress' );
    commit;
end
```

ml_qa_setshortproperty

Description

Sets the specified message property from a SQL SMALLINT data type.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Property name	VARCHAR(128)
3	Property value	SMALLINT

Remarks

You cannot alter this property after the message has been sent.

See also

- “ml_qa_getshortproperty” on page 651
- “ml_qa_createmessage” on page 666
- “ml_qa_getmessage” on page 666
- “Custom message properties” on page 211

Example

In the following example, a message is created, the short properties myshortproperty1 and myshortproperty2 are set, and the message is sent to the address someclient\someaddress:

```
begin
    declare @msgid varchar(128);
    set @msgid = ml_qa_createmessage();
    call ml_qa_setshortproperty( @msgid, 'myshortproperty1', -12345 );
    call ml_qa_setshortproperty( @msgid, 'myshortproperty2',  12345 );
    call ml_qa_putmessage( @msgid, 'someclient\someaddress' );
    commit;
end
```

ml_qa_setstringproperty

Description

Sets the specified message property from a SQL LONG VARCHAR data type.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Property name	VARCHAR(128)
3	Property value	LONG VARCHAR

Remarks

You cannot alter this property after the message has been sent.

See also

- “ml_qa_getstringproperty” on page 652
- “ml_qa_createmessage” on page 666
- “ml_qa_getmessage” on page 666
- “Custom message properties” on page 211

Example

In the following example, a message is created, the string properties mystringproperty1 and mystringproperty2 are set, and the message is sent to the address someclient\someaddress:

```
begin
    declare @msgid varchar(128);
    set @msgid = ml_qa_createmessage();
    call ml_qa_setstringproperty( @msgid, 'mystringproperty1', 'c:\\temp' );
    call ml_qa_setstringproperty( @msgid, 'mystringproperty2',  'first line
\nsecond line' );
    call ml_qa_putmessage( @msgid, 'someclient\someaddress' );
    commit;
end
```

Message content

You can use the following stored procedures to get and set message content.

ml_qa_getbinarycontent

Description

Returns the message content of a binary message.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.

Return value

The message content as LONG BINARY.

If the message has text content rather than binary content, this stored procedure returns NULL.

You can read this content after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "ml_qa_setbinarycontent" on page 662
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666
- "ml_qa_getcontentclass" on page 660

Example

In the following example, a message's encrypted content is decrypted and output to the database console:

```
begin
    declare @msgid varchar(128);
    declare @content long binary;
    declare @plaintext long varchar;
    set @msgid = ml_qa_getmessage( 'myaddress' );
    set @content = ml_qa_getbinarycontent( @msgid );
    set @plaintext = decrypt( @content, 'mykey' );
    message 'message content decrypted: ' || @plaintext;
    commit;
end
```

ml_qa_getcontentclass

Description

Returns the message type (text or binary).

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.

Return value

The content class as INTEGER.

The return value can be:

- **1** indicates that the message content is binary and should be read using the stored procedure ml_qa_getbinarycontent.
- **2** indicates that the message content is text and should be read using the stored procedure ml_qa_gettextcontent.

Remarks

You can read this content after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666
- "ml_qa_getbinarycontent" on page 659
- "ml_qa_gettextcontent" on page 661

Example

In the following example, a message is received and the content is output to the database console:

```
begin
    declare @msgid varchar(128);
    declare @contentclass integer;
    set @msgid = ml_qa_getmessage( 'myaddress' );
    set @contentclass = ml_qa_getcontentclass( @msgid );
    if @contentclass = 1 then
        message 'message  binary is ' || ml_qa_getbinarycontent( @msgid );
    elseif @contentclass = 2 then
        message 'message  text is ' || ml_qa_gettextcontent( @msgid );
    end if;
    commit;
end
```

ml_qa_gettextcontent

Description

Returns the message content of a text message.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.

Return value

The text content as LONG VARCHAR.

If the message has binary content rather than text content, this stored procedure returns NULL.

Remarks

You can read this content after a message is received and until a rollback or commit occurs; after that you cannot read it.

See also

- "ml_qa_settextcontent" on page 663
- "ml_qa_createmessage" on page 666

- "ml_qa_getmessage" on page 666
- "ml_qa_getcontentclass" on page 660

Example

In the following example, the content of a message is output to the database console:

```
begin
    declare @msgid varchar(128);
    declare @content long binary;
    set @msgid = ml_qa_getmessage( 'myaddress' );
    set @content = ml_qa_gettextcontent( @msgid );
    message 'message content: ' || @content ;
    commit;
end
```

ml_qa_setbinarycontent

Description

Sets the binary content of the message.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Content	LONG BINARY

Remarks

You cannot alter this content after the message has been sent.

See also

- "ml_qa_getbinarycontent" on page 659
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666

Example

In the following example, a message is created with encrypted content and sent:

```
begin
    declare @msgid varchar(128);
    set @msgid = ml_qa_createmessage();
    call ml_qa_setbinarycontent( @msgid, encrypt( 'my secret message',
'mykey' ) );
    call ml_qa_putmessage( @msgid, 'someclient\someaddress' );
    commit;
end
```


ml_qa_settextcontent

Description

Sets the text content of the message.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Content	LONG VARCHAR

Remarks

You cannot alter this content after the message has been sent.

See also

- “ml_qa_gettextcontent” on page 661
- “ml_qa_createmessage” on page 666
- “ml_qa_getmessage” on page 666

Example

In the following example, a message is created and then set with the given content:

```
begin
    declare @msgid varchar(128);
    set @msgid = ml_qa_createmessage();
    call ml_qa_settextcontent( @msgid, 'my simple message' );
    call ml_qa_putmessage( @msgid, 'someclient\someaddress' );
    commit;
end
```


Message store properties

You can use the following stored procedures to get and set properties for client message stores.

For more information about message store properties, see "Client message store properties" on page 215.

ml_qa_getstoreproperty

Description

Returns a client message store property.

Parameters

Item	Description	Remarks
1	Property name	VARCHAR(128)

Return value

The property value as LONG VARCHAR.

Remarks

Client message store properties are readable from every connection to this client message store.

See also

- "ml_qa_setstoreproperty" on page 664

Example

The following example gets the current synchronization policy of this message store and outputs it to the database console:

```
begin
    declare @policy varchar(128);
    set @policy = ml_qa_getstoreproperty( 'policy' );
    message 'the current policy for synchronizing this message store is ' ||
@policy;
end
```

ml_qa_setstoreproperty

Description

Sets a client message store property.

Parameters

Item	Description	Remarks
1	Property name	VARCHAR(128)

Item	Description	Remarks
2	Property value	SMALLINT

Remarks

Client message store properties are readable from every connection to this client message store. The values are synchronized up to the server, as well, where they can be used in transmission rules.

See also

- "ml_qa_getstoreproperty" on page 664

Example

The following example sets the synchronization policy to automatic for the message store:

```
begin
    call ml_qa_setstoreproperty( 'policy', 'automatic' );
    commit;
end
```

Message management

You can use the following stored procedures to manage your QAnywhere client transactions.

ml_qa_createmessage

Description

Returns the message ID of a new message.

Return value

The message ID of the new message.

Remarks

Use this stored procedure to create a message. Once created, you can associate content, properties, and headers with this message and then send the message.

You can associate content, properties, and headers using any of the QAnywhere stored procedures starting with ml_qa_set. For example, use ml_qa_setbinarycontent or ml_qa_settextcontent to create a binary or text message.

See also

- "Message headers" on page 636
- "Message properties" on page 645
- "Message content" on page 659

Example

The following example creates a message, sets the message content, and sends the message to the address someclient\someaddress:

```
begin
    declare @msgid varchar(128);
    set @msgid = ml_qa_createmessage();
    call ml_qa_settextcontent( @msgid, 'some content' );
    call ml_qa_putmessage( @msgid, 'someclient\someaddress' );
    commit;
end
```

ml_qa_getmessage

Description

Returns the message ID of the next message that is queued for the given address, blocking until one is queued.

Parameters

Item	Description	Remarks
1	Address	VARCHAR(128)

Return value

The message ID as VARCHAR(128).

Returns NULL if there is no queued message for this address.

Remarks

Use this stored procedure to check synchronously whether there is a message waiting for the specified QAnywhere message address. If you want a SQL procedure to be called asynchronously as soon as a message is available for a specified QAnywhere address, use the Listener.

This stored procedure blocks until a message is queued.

☞ For information about avoiding blocking, see "ml_qa_getmessagenowait" on page 668 or "ml_qa_getmessagetimeout" on page 669.

The message corresponding to the returned message ID is not considered to be received until the current transaction is committed. Once the receive is committed, the message cannot be received again by this or any other QAnywhere API. Similarly, a rollback of the current transaction means that the message is not received, so subsequent calls to ml_qa_getmessage may return the same message ID.

The properties and content of the received message can be read by the various ml_qa_get stored procedures until a commit or rollback is executed on the current transaction. Once a commit or rollback is executed on the current transaction, the message data is no longer readable. Before committing, you should store any data you need from the message as tabular data or in SQL variables.

See also

- "ml_qa_getmessagenowait" on page 668
- "ml_qa_getmessagetimeout" on page 669
- "Message headers" on page 636
- "Message properties" on page 645
- "Message content" on page 659

Example

The following example displays the content of all messages sent to the address myaddress:

```
begin
    declare @msgid varchar(128);
    loop
        set @msgid = ml_qa_getmessage( 'myaddress' );
        message 'a message with content ' || ml_qa_gettextcontent( @msgid )
|| ' has been received';
        commit;
    end loop;
end
```


ml_qa_getmessagenowait

Description

Returns the message ID of the next message that is currently queued for the given address.

Parameters

Item	Description	Remarks
1	Address	VARCHAR(128)

Return value

The message ID as VARCHAR(128).

Returns the message ID of the next message that is queued for the given address. Returns NULL if there is no queued message for this address.

Remarks

Use this stored procedure to check synchronously whether there is a message waiting for the specified QAnywhere message address. If you want a SQL procedure to be called asynchronously as soon as a message is available for a specified QAnywhere address, use the Listener.

☞ For information on blocking until a message is available, see "ml_qa_getmessage" on page 666 and "ml_qa_getmessagetimeout" on page 669.

The message corresponding to the returned message is not considered to be received until the current transaction is committed. Once the receive is committed, the message cannot be received again by this or any other QAnywhere API. Similarly, a rollback of the current transaction means that the message is not received, so subsequent calls to ml_qa_getmessage may return the same message ID.

The properties and content of the received message can be read by the various ml_qa_get stored procedures until a commit or rollback is executed on the current transaction. Once a commit or rollback is executed on the current transaction, the message data is no longer readable. Before committing, you should store any data you need from the message as tabular data or in SQL variables.

See also

- "QAnywhere message addresses" on page 50
- "Listeners" [*MobiLink - Server-Initiated Synchronization*]
- "ml_qa_getmessagetimeout" on page 669
- "Message headers" on page 636
- "Message properties" on page 645
- "Message content" on page 659

Example

The following example displays the content of all messages that are queued at the address myaddress until all such messages are read (it is generally more efficient to commit after the last message has been read, rather than after each message is read):


```
begin
    declare @msgid varchar(128);
    loop
        set @msgid = ml_qa_getmessagenowait( 'myaddress' );
        if @msgid is null then leave end if;
        message 'a message with content ' || ml_qa_gettextcontent( @msgid )
|| ' has been received';
    end loop;
    commit;
end
```

ml_qa_getmessagetimeout

Description

Waits for the specified timeout period to return the message ID of the next message that is queued for the given address.

Parameters

Item	Description	Remarks
1	Address	VARCHAR(128)
2	Timeout in milliseconds	INTEGER

Return value

The message ID as VARCHAR(128).

Returns NULL if there is no queued message for this address within the timeout period.

Remarks

Use this stored procedure to check synchronously whether there is a message waiting for the specified QAnywhere message address. If you want a SQL procedure to be called asynchronously as soon as a message is available for a specified QAnywhere address, use the Listener.

The message corresponding to the returned message is not considered to be received until the current transaction is committed. Once the receive is committed, the message cannot be received again by this or any other QAnywhere API. Similarly, a rollback of the current transaction means that the message is not received, so subsequent calls to ml_qa_getmessage may return the same message ID.

The properties and content of the received message can be read by the various ml_qa_get stored procedures until a commit or rollback is executed on the current transaction. Once a commit or rollback is executed on the current transaction, the message data is no longer readable. Before committing, you should store any data you need from the message as tabular data or in SQL variables.

See also

- “ml_qa_getmessage” on page 666
- “ml_qa_getmessagenowait” on page 668

Example

The following example outputs the content of all messages sent to the address myaddress to the database console, and updates the database console every 10 seconds if no message has been received:

```
begin
    declare @msgid varchar(128);
    loop
        set @msgid = ml_qa_getmessagetimeout( 'myaddress', 10000 );
        if @msgid is null then
            message 'waiting for a message...';
        else
          message 'a message with content ' || ml_qa_gettextcontent( @msgid )
|| ' has been received';
        commit;
        end if;
    end loop;
end
```

ml_qa_grant_messaging_permissions

Description

Grants permission to other users to use QAnywhere stored procedures.

Parameters

Item	Description	Remarks
1	Database user ID	VARCHAR(128)

Remarks

Only users with DBA privilege automatically have permission to execute the QAnywhere stored procedures. Other users must be granted permission by having a user with DBA privileges run this stored procedure.

This procedure adds the user to a group called ml_qa_message_group and gives them execute permissions on all QAnywhere stored procedures.

Example

For example, to grant messaging permissions to a user with the database ID user1, execute the following SQL code:

```
call dbo.ml_qa_grant_messaging_permissions( 'user1' )
```

ml_qa_listener_queue

Description

Create a stored procedure named **ml_qa_listener_*queue*** (where *queue* is the name of a message queue) to receive messages asynchronously.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from the QAnywhere Listener.

Remarks

Note
This procedure is different from all the other QAnywhere stored procedures in that the stored procedure is not provided. If you create a stored procedure named **ml_qa_listener_*queue***, where *queue* is a message queue, then it is used by QAnywhere.

Although messages can be received synchronously on a connection, it is often convenient to receive messages asynchronously. You can create a stored procedure that is called when a message has been queued on a particular address. The name of this procedure must be **ml_qa_listener_*queue***, where *queue* is the message queue. When this procedure exists, the procedure is called whenever a message is queued on the given address.

This procedure is called from a separate connection. As long as a SQL error does not occur while this procedure is executing, the message is automatically acknowledged and committed.

Do not commit or rollback within this procedure.

The queue name is part of the QAnywhere address. For more information, see "QAnywhere message addresses" on page 50.

See also

- "Receiving messages asynchronously" on page 76
- "Receiving messages synchronously" on page 75
- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666

Example

The following example creates a procedure that is called whenever a message is queued on the address named executesql. In this example, the procedure assumes that the content of the message is a SQL statement that it can execute against the current database.

```
CREATE PROCEDURE ml_qa_listener_executesql(IN @msgid VARCHAR(128))
begin
    DECLARE @execstr LONG VARCHAR;
    SET @execstr = ml_qa_gettextcontent( @msgid );
    EXECUTE IMMEDIATE @execstr;
end
```


ml_qa_putmessage

Description

Sends a message.

Parameters

Item	Description	Remarks
1	Message ID	VARCHAR(128). You can obtain the message ID from ml_qa_createmessage or ml_qa_getmessage.
2	Address	VARCHAR(128)

Remarks

The message ID you specify must have been previously created using ml_qa_createmessage. Only content, properties and headers associated with the message ID before the call to ml_qa_putmessage are sent with the message. Any added after the ml_qa_putmessage are ignored.

A commit is required before the message is actually queued for sending.

See also

- "ml_qa_createmessage" on page 666
- "ml_qa_getmessage" on page 666

Example

In the following example, a message is created with the content 'a simple message' and sent to the address someclient\someaddress:

```
begin
    declare @msgid varchar(128);
    set @msgid = ml_qa_createmessage();
    call ml_qa_settextcontent( @msgid, 'a simple message' );
    call ml_qa_putmessage( @msgid, 'someclient\someaddress' );
    commit;
end
```

ml_qa_triggersendreceive

Description

Triggers a synchronization of messages with the MobiLink server.

Remarks

Normally, message synchronization is handled by the QAnywhere Agent. However, if the synchronization policy is ondemand, then it is the application's responsibility to trigger the synchronization of messages. You can do so using this stored procedure. The trigger does not take effect until the current transaction is committed.

Example

In the following example, a message is sent and the transmission of the message is immediately initiated:

```
begin
    declare @msgid varchar(128);
    set @msgid = ml_qa_createmessage();
    call ml_qa_settextcontent( @msgid, 'my simple message' );
    call ml_qa_putmessage( @msgid, 'someclient\someaddress' );
    call ml_qa_triggersendreceive();
    commit;
end
```


Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

