

# **Netscape Application Server Foundation Class Reference (C++)**

Netscape Application Server

Version 4.0

Netscape Communications Corporation ("Netscape") and its licensors retain all ownership rights to the software programs offered by Netscape (referred to herein as "Software") and related documentation. Use of the Software and related documentation is governed by the license agreement accompanying the Software and applicable copyright law.

Your right to copy this documentation is limited by copyright law. Making unauthorized copies, adaptations, or compilation works is prohibited and constitutes a punishable violation of the law. Netscape may revise this documentation from time to time without notice.

THIS DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. IN NO EVENT SHALL NETSCAPE BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY KIND ARISING FROM ANY ERROR IN THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION ANY LOSS OR INTERRUPTION OF BUSINESS, PROFITS, USE, OR DATA.

The Software and documentation are copyright © 1999 Netscape Communications Corp.

Netscape, Netscape Navigator, Netscape Certificate Server, Netscape DevEdge, Netscape FastTrack Server, Netscape ONE, SuiteSpot, and the Netscape N and Ship's Wheel logos are registered trademarks of Netscape Communications Corporation in the United States and other countries. Other Netscape logos, product names, and service names are also trademarks of Netscape Communications Corporation, which may be registered in other countries. Other product and brand names are trademarks of their respective owners.

The downloading, exporting, or reexporting of Netscape software or any underlying information or technology must be in full compliance with all United States and other applicable laws and regulations. Any provision of Netscape software or documentation to the U.S. Government is with restricted rights as described in the license agreement accompanying Netscape software.



Recycled and Recyclable Paper

Version 4.0

Part Number 151-07623-00

Copyright ©1999 Netscape Communications Corp. All rights reserved.

Printed in the United States of America. 00 99 5 4 3 2 1

Netscape Communications Corporation, 501 East Middlefield Road, Mountain View, CA 94043

# Contents

|   |    |
|---|----|
| <b>Preface</b> .....  | 15 |
| Using the Documentation .....                                       | 15 |
| About This Guide .....  | 18 |
| Naming Conventions .....  | 19 |
| <b>Chapter 1 Class Library Components by Programming Task</b> ..... | 21 |
| Running AppLogics .....   | 22 |
| Securing AppLogics .....  | 22 |
| Managing User Sessions .....  | 22 |
| Managing Application States .....                                   | 23 |
| Working with Databases .....  | 23 |
| Creating Reports Using Templates .....                              | 24 |
| Creating and Managing Application Events .....                      | 25 |
| Sending and Receiving Electronic Mail .....                         | 25 |
| Managing Object Lifetime .....                                      | 25 |
| <b>Chapter 2 Classes</b> .....                                      | 27 |
| GXAppLogic class .....  | 29 |
| CreateDataConn( ) .....   | 32 |
| CreateDataConnSet( ) .....  | 35 |
| CreateHierQuery( ) .....  | 36 |
| CreateMailbox( ) .....  | 37 |
| CreateQuery( ) .....  | 39 |
| CreateSession( ) .....  | 40 |
| CreateTrans( ) .....  | 42 |
| DeleteCache( ) .....  | 44 |
| DestroySession( ) .....   | 45 |
| EvalOutput( ) .....   | 45 |
| EvalTemplate( ) .....   | 48 |
| Execute( ) .....  | 51 |

|                                 |    |
|---------------------------------|----|
| GetAppEvent( ) .....            | 52 |
| GetSession( ) .....             | 53 |
| GetStateTreeRoot( ) .....       | 54 |
| IsAuthorized( ) .....           | 55 |
| IsCached( ) .....               | 57 |
| LoadHierQuery( ) .....          | 58 |
| LoadQuery( ) .....              | 61 |
| Log( ) .....                    | 64 |
| LoginSession( ) .....           | 65 |
| LogoutSession( ) .....          | 68 |
| NewRequest( ) .....             | 69 |
| NewRequestAsync( ) .....        | 71 |
| RemoveAllCachedResults( ) ..... | 74 |
| RemoveCachedResult( ) .....     | 75 |
| Result( ) .....                 | 76 |
| SaveSession( ) .....            | 77 |
| SetCacheCriteria( ) .....       | 78 |
| SetSessionVisibility( ) .....   | 81 |
| SetVariable( ) .....            | 82 |
| SkipCache( ) .....              | 83 |
| StreamResult( ) .....           | 84 |
| StreamResultBinary( ) .....     | 85 |
| StreamResultHeader( ) .....     | 86 |
| GXSession2 class .....          | 89 |
| GXTemplateDataBasic class ..... | 91 |
| GroupAppend( ) .....            | 92 |
| RowAppend( ) .....              | 93 |
| GXTemplateDataBasic( ) .....    | 94 |
| GXTemplateMapBasic class .....  | 97 |
| Put( ) .....                    | 98 |

|  |     |
|--|-----|
| <b>Chapter 3 Interfaces</b> .....        | 101 |
| IGXAppEvent interface (deprecated) ..... | 103 |
| DeleteEvent( ) .....                     | 106 |
| DisableEvent( ) .....                    | 107 |
| EnableEvent( ) .....                     | 107 |
| EnumEvents( ) .....                      | 107 |
| QueryEvent( ) .....                      | 110 |
| RegisterEvent( ) .....                   | 110 |
| SetEvent( ) .....                        | 113 |
| IGXAppEventMgr interface .....           | 117 |
| CreateEvent( ) .....                     | 120 |
| DeleteEvent( ) .....                     | 120 |
| DisableEvent( ) .....                    | 121 |
| EnableEvent( ) .....                     | 121 |
| EnumEvents( ) .....                      | 122 |
| GetEvent( ) .....                        | 124 |
| RegisterEvent( ) .....                   | 125 |
| TriggerEvent( ) .....                    | 127 |
| IGXAppEventObj interface .....           | 129 |
| AddAction( ) .....                       | 129 |
| DeleteActions( ) .....                   | 131 |
| EnumActions( ) .....                     | 131 |
| GetAttributes( ) .....                   | 131 |
| GetName( ) .....                         | 132 |
| SetAttributes( ) .....                   | 132 |
| IGXBuffer interface .....                | 135 |
| Alloc( ) .....                           | 135 |
| GetAddress( ) .....                      | 136 |
| GetSize( ) .....                         | 137 |
| SetData( ) .....                         | 137 |
| IGXCallableStmt interface .....          | 139 |
| Close( ) .....                           | 140 |
| Execute( ) .....                         | 140 |

|                                |     |
|--------------------------------|-----|
| ExecuteMultipleRS( ) .....     | 143 |
| GetMoreResults( ) .....        | 146 |
| GetParams( ) .....             | 147 |
| GetResultSet( ) .....          | 148 |
| SetParams( ) .....             | 149 |
| IGXColumn interface .....      | 151 |
| GetName( ) .....               | 153 |
| GetNullsAllowed( ) .....       | 155 |
| GetPrecision( ) .....          | 156 |
| GetScale( ) .....              | 158 |
| GetSize( ) .....               | 159 |
| GetTable( ) .....              | 160 |
| GetType( ) .....               | 162 |
| IGXDataConn interface .....    | 165 |
| CloseConn( ) .....             | 166 |
| CreateTrigger( ) .....         | 167 |
| DisableTrigger( ) .....        | 169 |
| DropTrigger( ) .....           | 169 |
| EnableTrigger( ) .....         | 170 |
| ExecuteQuery( ) .....          | 170 |
| GetConnInfo( ) .....           | 172 |
| GetConnProps( ) .....          | 173 |
| GetDriver( ) .....             | 174 |
| GetTable( ) .....              | 175 |
| GetTables( ) .....             | 176 |
| PrepareCall( ) .....           | 177 |
| PrepareQuery( ) .....          | 179 |
| SetConnProps( ) .....          | 181 |
| IGXDataConnSet interface ..... | 183 |
| AddConn( ) .....               | 184 |

|                                  |     |
|----------------------------------|-----|
| IGXEnumObject interface .....    | 185 |
| EnumCount( ) .....               | 185 |
| EnumNext( ) .....                | 186 |
| EnumReset( ) .....               | 188 |
| IGXError interface .....         | 189 |
| GetErrorCode( ) .....            | 189 |
| GetErrorCodeNum( ) .....         | 190 |
| GetErrorMessage( ) .....         | 191 |
| GetErrorFacility( ) .....        | 191 |
| IGXHierQuery interface .....     | 193 |
| AddQuery( ) .....                | 194 |
| DelQuery( ) .....                | 196 |
| Execute( ) .....                 | 196 |
| IGXHierResultSet interface ..... | 199 |
| Count( ) .....                   | 202 |
| GetColumn( ) .....               | 203 |
| GetColumnByOrd( ) .....          | 204 |
| GetResultSet( ) .....            | 206 |
| GetRowNumber( ) .....            | 206 |
| GetValueDateString( ) .....      | 207 |
| GetValueDouble( ) .....          | 208 |
| GetValueInt( ) .....             | 209 |
| GetValueString( ) .....          | 209 |
| MoveNext( ) .....                | 210 |
| MoveTo( ) .....                  | 211 |
| IGXLock interface .....          | 213 |
| ChangeMode( ) .....              | 214 |
| Lock( ) .....                    | 215 |
| Unlock( ) .....                  | 216 |
| IGXMailBox interface .....       | 219 |
| Close( ) .....                   | 220 |
| Open( ) .....                    | 221 |
| Retrieve( ) .....                | 222 |

|                                  |     |
|----------------------------------|-----|
| RetrieveCount( ) .....           | 224 |
| RetrieveReset( ) .....           | 225 |
| Send( ) .....                    | 226 |
| IGXObject interface .....        | 229 |
| AddRef( ) .....                  | 230 |
| QueryInterface( ) .....          | 231 |
| Release( ) .....                 | 232 |
| IGXOrder interface .....         | 233 |
| GetState( ) .....                | 233 |
| IGXPreparedQuery interface ..... | 237 |
| Execute( ) .....                 | 238 |
| SetParams( ) .....               | 239 |
| IGXQuery interface .....         | 241 |
| GetFields( ) .....               | 242 |
| GetGroupBy( ) .....              | 243 |
| GetHaving( ) .....               | 244 |
| GetOrderBy( ) .....              | 245 |
| GetSQL( ) .....                  | 246 |
| GetTables( ) .....               | 247 |
| GetWhere( ) .....                | 248 |
| SetFields( ) .....               | 249 |
| SetGroupBy( ) .....              | 250 |
| SetHaving( ) .....               | 251 |
| SetOrderBy( ) .....              | 252 |
| SetSQL( ) .....                  | 253 |
| SetTables( ) .....               | 254 |
| SetWhere( ) .....                | 255 |
| IGXResultSet interface .....     | 257 |
| Close( ) .....                   | 259 |
| EnumColumnReset( ) .....         | 259 |
| EnumColumns( ) .....             | 259 |
| FetchNext( ) .....               | 260 |
| GetColumn( ) .....               | 261 |

|                                |     |
|--------------------------------|-----|
| GetColumnByOrd( ) .....        | 262 |
| GetColumnOrdinal( ) .....      | 263 |
| GetNumColumns( ) .....         | 264 |
| GetOrder( ) .....              | 265 |
| GetRowNumber( ) .....          | 265 |
| GetStatus( ) .....             | 266 |
| GetValueBinary( ) .....        | 267 |
| GetValueBinaryPiece( ) .....   | 267 |
| GetValueDateString( ) .....    | 269 |
| GetValueDouble( ) .....        | 270 |
| GetValueInt( ) .....           | 271 |
| GetValueSize( ) .....          | 271 |
| GetValueString( ) .....        | 272 |
| GetValueText( ) .....          | 273 |
| GetValueTextPiece( ) .....     | 274 |
| MoveTo( ) .....                | 274 |
| RowCount( ) .....              | 275 |
| WasNull( ) .....               | 276 |
| IGXSequence interface .....    | 279 |
| Drop( ) .....                  | 280 |
| GetCurrent( ) .....            | 280 |
| GetNext( ) .....               | 281 |
| IGXSequenceMgr interface ..... | 283 |
| CreateSequence( ) .....        | 283 |
| GetSequence( ) .....           | 286 |
| IGXSession2 interface .....    | 289 |
| GetSessionApp( ) .....         | 290 |
| GetSessionData( ) .....        | 291 |
| GetSessionFlags( ) .....       | 292 |
| GetSessionID( ) .....          | 293 |
| GetSessionTimeout( ) .....     | 294 |
| SaveSession( ) .....           | 295 |
| SetSessionData( ) .....        | 296 |

|                                 |     |
|---------------------------------|-----|
| IGXSessionIDGen interface ..... | 297 |
| GenerateSessID( ) .....         | 298 |
| GenerateVariantID( ) .....      | 299 |
| MapToBaseID( ) .....            | 300 |
| IGXState2 interface .....       | 303 |
| CreateStateChild( ) .....       | 304 |
| DeleteStateChild( ) .....       | 306 |
| GetStateChild( ) .....          | 306 |
| GetStateChildCount( ) .....     | 307 |
| GetStateContents( ) .....       | 307 |
| GetStateFlags( ) .....          | 308 |
| GetStateName( ) .....           | 309 |
| GetStateTimeout( ) .....        | 309 |
| SaveState( ) .....              | 310 |
| SetStateContents( ) .....       | 310 |
| IGXStreamBuffer interface ..... | 313 |
| GetStreamData( ) .....          | 313 |
| IGXTable interface .....        | 315 |
| AddRow( ) .....                 | 316 |
| AllocRow( ) .....               | 318 |
| DeleteRow( ) .....              | 320 |
| EnumColumnReset( ) .....        | 321 |
| EnumColumns( ) .....            | 322 |
| GetColumn( ) .....              | 323 |
| GetColumnByOrd( ) .....         | 323 |
| GetColumnOrdinal( ) .....       | 324 |
| GetDataConn( ) .....            | 325 |
| GetName( ) .....                | 326 |
| GetNumColumns( ) .....          | 326 |
| SetValueBinary( ) .....         | 327 |
| SetValueBinaryPiece( ) .....    | 328 |
| SetValueDateString( ) .....     | 329 |
| SetValueDouble( ) .....         | 330 |

|                                 |     |
|---------------------------------|-----|
| SetValueInt( ) .....            | 330 |
| SetValueString( ) .....         | 331 |
| SetValueText( ) .....           | 332 |
| SetValueTextPiece( ) .....      | 333 |
| UpdateRow( ) .....              | 334 |
| IGXTemplateData interface ..... | 337 |
| GetValue( ) .....               | 338 |
| IsEmpty( ) .....                | 339 |
| MoveNext( ) .....               | 339 |
| SetHint( ) .....                | 340 |
| IGXTemplateMap interface .....  | 341 |
| Get( ) .....                    | 342 |
| IGXTile interface .....         | 345 |
| GetTileChild( ) .....           | 347 |
| GetTileValue( ) .....           | 348 |
| MoveTileNextRecord( ) .....     | 349 |
| MoveTileToRecord( ) .....       | 349 |
| IGXTrans interface .....        | 351 |
| Begin( ) .....                  | 353 |
| Commit( ) .....                 | 354 |
| Rollback( ) .....               | 355 |
| IGXValList interface .....      | 357 |
| Count( ) .....                  | 359 |
| GetNextKey( ) .....             | 360 |
| GetVal( ) .....                 | 361 |
| GetValBLOB( ) .....             | 362 |
| GetValBLOBSize( ) .....         | 362 |
| GetValByRef( ) .....            | 363 |
| GetValInt( ) .....              | 364 |
| GetValString( ) .....           | 365 |
| RemoveVal( ) .....              | 366 |
| ResetPosition( ) .....          | 366 |
| SetVal( ) .....                 | 367 |

|                                      |            |
|--------------------------------------|------------|
| SetValBLOB( ) .....                  | 368        |
| SetValByRef( ) .....                 | 369        |
| SetValInt( ) .....                   | 370        |
| SetValString( ) .....                | 371        |
| <b>Chapter 4 C++ Functions</b> ..... | <b>373</b> |
| GXContextGetAppEventMgr( ) .....     | 374        |
| GXContextGetSessionCount( ) .....    | 374        |
| GXCreateBuffer( ) .....              | 375        |
| GXCreateStreamBuffer( ) .....        | 376        |
| GXCreateTemplateDataBasic( ) .....   | 377        |
| GXCreateTemplateMapBasic( ) .....    | 378        |
| GXCreateValList( ) .....             | 379        |
| GXDeleteCriticalSection( ) .....     | 380        |
| GXEnterCriticalSection( ) .....      | 381        |
| GXGetCurrentDateTime( ) .....        | 381        |
| GXGetValList( ) .....                | 382        |
| GXGetValListBLOB( ) .....            | 383        |
| GXGetValListGUID( ) .....            | 384        |
| GXGetValListString( ) .....          | 384        |
| GXGUIDToString( ) .....              | 386        |
| GXInitCriticalSection( ) .....       | 386        |
| GXLeaveCriticalSection( ) .....      | 387        |
| GXProcessOutput( ) .....             | 388        |
| GXSetValList( ) .....                | 390        |
| GXSetValListBLOB( ) .....            | 391        |
| GXSetValListGUID( ) .....            | 391        |
| GXSetValListString( ) .....          | 392        |
| GXStringToGUID( ) .....              | 393        |
| GXSYNC_DEC( ) .....                  | 394        |
| GXSYNC_DESTROY( ) .....              | 394        |
| GXSYNC_INC( ) .....                  | 395        |
| GXSYNC_INIT( ) .....                 | 396        |
| GXSYNC_LOCK( ) .....                 | 397        |

|  |            |
|--|------------|
| GXSYNC_UNLOCK( ) .....                           | 398        |
| GXVALClear( ) .....                              | 399        |
| GXVALCopy( ) .....                               | 400        |
| GXWaitForOrder( ) .....                          | 400        |
| <b>Chapter 5 C++ Macros and Structures</b> ..... | <b>403</b> |
| Macros .....                                     | 403        |
| Structures .....                                 | 403        |
| GXDLM_DECLARE .....                              | 404        |
| GXDLM_IMPLEMENT .....                            | 404        |
| GXDLM_IMPLEMENT_BEGIN .....                      | 405        |
| GXDLM_IMPLEMENT_END .....                        | 406        |
| GXGUID_EQUAL .....                               | 406        |
| GUID struct .....                                | 407        |
| GXDATETIME struct .....                          | 407        |
| GXVAL struct .....                               | 408        |
| <b>Appendix A Return Codes</b> .....             | <b>411</b> |
| <b>Index</b> .....                               | <b>413</b> |



This preface contains the following topics:

- Using the Documentation
- About This Guide
- Naming Conventions

## Using the Documentation

The following table lists the tasks and concepts that are described in the Netscape Application Server (NAS) and Netscape Application Builder (NAB) printed manuals and online README file. If you are trying to accomplish a specific task or learn more about a specific concept, refer to the appropriate manual.

Note that the printed manuals are also available as online files in PDF and HTML format. In addition, note that NAB 4.0 is for developing Java applications.

| For information about   | See the following  | Shipped with  |
|---|--------------------|---|
| Late-breaking information about the software and the documentation  | readme.htm         | NAS 4.0, NAS 4.0 Developer Edition (Solaris), NAB 4.0 |
| Installing Netscape Application Server and its various components (Web Connector plug-in, Netscape Application Server Administrator), and configuring the sample applications | Installation Guide | NAS 4.0 Developer Edition (Solaris), NAS 4.0          |
| Installing Netscape Application Builder   | install.htm        | NAB 4.0   |
| Basic features of NAS, such as its software components, general capabilities, and system architecture   | Overview           | NAS 4.0, NAS 4.0 Developer Edition (Solaris), NAB 4.0 |

| For information about  | See the following    | Shipped with  |
|--|----------------------|---|
| <p>Deploying Netscape Application Server at your site, by performing the following tasks:</p> <ul style="list-style-type: none"> <li>• Planning your Netscape Application Server environment</li> <li>• Integrating the product within your existing enterprise and network topology</li> <li>• Developing server capacity and performance goals</li> <li>• Running stress tests to measure server performance</li> <li>• Fine-tuning the server to improve performance</li> </ul>   | Deployment Guide     | NAS 4.0   |
| <p>Administering one or more application servers using the Netscape Application Server Administrator tool to perform the following tasks:</p> <ul style="list-style-type: none"> <li>• Deploying applications with the Deployment Manager tool</li> <li>• Monitoring and logging server activity</li> <li>• Setting up users and groups</li> <li>• Administering database connectivity</li> <li>• Administering transactions</li> <li>• Load balancing servers</li> <li>• Managing distributed data synchronization</li> </ul> | Administration Guide | NAS 4.0   |
| <p>Migrating your applications to the new Netscape Application Server 4.0 programming model from version 2.1, including a sample migration of an Online Bank application provided with Netscape Application Server</p>   | Migration Guide      | NAS 4.0, NAS 4.0 Developer Edition (Solaris), NAB 4.0 |

| For information about  | See the following                        | Shipped with                                 |
|--|--|--|
| <p>Creating NAS 4.0 applications within an integrated development environment by performing the following tasks:</p> <ul style="list-style-type: none"> <li>• Creating and managing projects</li> <li>• Using wizards</li> <li>• Creating data-access logic</li> <li>• Creating presentation logic and layout</li> <li>• Creating business logic</li> <li>• Compiling, testing, and debugging applications</li> <li>• Deploying and downloading applications</li> <li>• Working with source control</li> <li>• Using third-party tools</li> </ul>  | User's Guide                             | NAB 4.0                                      |
| <p>Creating NAS 4.0 applications that follow the new open Java standards model (Servlets, EJBs, JSPs, and JDBC), by performing the following tasks:</p> <ul style="list-style-type: none"> <li>• Creating the presentation and execution layers of an application</li> <li>• Placing discrete pieces of business logic and entities into Enterprise Java Beans (EJB) components</li> <li>• Using JDBC to communicate with databases</li> <li>• Using iterative testing, debugging, and application fine-tuning procedures to generate applications that execute correctly and quickly</li> </ul> | Programmer's Guide (Java)                | NAS 4.0 Developer Edition (Solaris), NAB 4.0 |
| <p>Using the public classes and interfaces, and their methods in the Netscape Application Server class library to write Java applications</p>  | Server Foundation Class Reference (Java) | NAS 4.0 Developer Edition (Solaris), NAB 4.0 |

| For information about  | See the following                       | Shipped with     |
|--|---|------------------|
| Creating NAS C++ applications using the NAS class library by performing the following tasks: <ul style="list-style-type: none"> <li>• Designing applications</li> <li>• Writing AppLogics.</li> <li>• Creating HTML templates</li> <li>• Creating queries</li> <li>• Running and debugging applications</li> </ul> | Programmer's Guide (C++)                | Order separately |
| Using the public classes and interfaces, and their methods in the Netscape Application Server class library to write C++ applications  | Server Foundation Class Reference (C++) | Order separately |

## About This Guide

The *Netscape Application Server Foundation Class Reference (C++)* provides specification-level documentation for the public classes and interfaces, and their methods, in the Netscape Application Server Foundation Class Library. Use this book to look up how a particular class or interface method works, what syntax is required, and for examples on how to use it.

For conceptual and task-oriented information on designing and developing Netscape Application Server applications, read the *Programmer's Guide (C++)*.

# Naming Conventions

| Item           | Convention   |
|----------------|--|
| Class name     | “GX” prefix, followed by mixed case with initial uppercase. For example, GXTemplateMapBasic class. |
| Interface name | “IGX” prefix, followed by mixed case with initial uppercase. For example, IGXPreparedQuery.        |
| Method name    | Mixed case with initial uppercase. For example, GetTables( ).                                      |
| Parameters     | Mixed case with initial lowercase. For example, myQuery.   |



# Class Library Components by Programming Task

This chapter provides a list grouped according to general functionality, of the classes, interfaces, and methods that make up the Netscape Application Server Foundation Class Library.

For your convenience, this chapter lists the following commonly used tasks along with the classes, interfaces, and methods you need to accomplish those tasks:

- Running AppLogics
- Securing AppLogics
- Managing User Sessions
- Managing Application States
- Working with Databases
- Creating Reports Using Templates
- Creating and Managing Application Events
- Sending and Receiving Electronic Mail
- Managing Object Lifetime

## Running AppLogics

- `Execute()` in the `GXAppLogic` class
- `NewRequest()` in the `GXAppLogic` class
- `NewRequestAsync()` in the `GXAppLogic` class

## Setting and Retrieving AppLogic Parameters

- `GXCreateValList()`
- `IGXValList` interface

## Returning AppLogic Results

- `Result()` in the `GXAppLogic` class
- `StreamResult()` and `StreamResultHeader()` in the `GXAppLogic` class
- `IGXStreamBuffer` interface

## Caching Results and Managing the Result Cache

- `IsCached()`, `SetCacheCriteria()`, and `SkipCache()` in the `GXAppLogic` class
- `DeleteCache()`, `RemoveAllCachedResults()`, and `RemoveCachedResult()` in the `GXAppLogic` class

## Securing AppLogics

`LoginSession()`, `IsAuthorized()`, and `LogoutSession()` in the `GXAppLogic` class

## Managing User Sessions

- `CreateSession()`, `GetSession()`, and `SaveSession()` in the `GXAppLogic` class
- `GXSession2` class
- `IGXSession2` interface

# Managing Application States

- `GetStateTreeRoot()` in the `GXAppLogic` class
- `IGXState2` interface

# Working with Databases

## Connecting to Databases

- `CreateDataConn()` in the `GXAppLogic` class
- `IGXDataConn` interface

## Managing Asynchronous Operations

- `GX_DA_EXEC_ASYNC` parameter flag for `ExecuteQuery()` in the `IGXDataConn` interface
- `IGXOrder` interface

## Managing Database Sequences

- `IGXSequenceMgr` interface
- `IGXSequence` interface

## Using Stored Procedures

- `PrepareCall()` in the `IGXDataConn` interface
- `IGXCallableStmt` interface

## Using Database Triggers

`CreateTrigger()`, `EnableTrigger()`, `DisableTrigger()`, and `DropTrigger()` in the `IGXDataConn` interface

## Managing Database Transactions

- `CreateTrans()` in the `GXAppLogic` class

- IGXTrans interface

## Creating Queries

### Flat queries

- CreateQuery() in the GXAppLogic class
- IGXQuery interface

### Hierarchical queries

- CreateHierQuery() in the GXAppLogic class
- IGXHierQuery interface

### Prepared queries

- CreateDataConnSet() in the GXAppLogic class
- IGXDataConnSet interface
- PrepareQuery() in the IGXDataConn interface
- IGXPreparedQuery interface
- LoadHierQuery() in the GXAppLogic class

## Working with Result Sets

- IGXResultSet interface
- IGXHierResultSet interface
- IGXTable interface
- IGXColumn interface

## Creating Reports Using Templates

- EvalOutput() and EvalTemplate() in the GXAppLogic class
- GXTemplateDataBasic class

- IGXTemplateData interface
- GXTemplateMapBasic class
- IGXTemplateMap interface

## Creating and Managing Application Events

- GXContextGetAppEventMgr( )
- IGXAppEventMgr interface and IGXAppEventObj interface

## Sending and Receiving Electronic Mail

- CreateMailbox( ) in the GXAppLogic class
- IGXMailBox interface

## Managing Object Lifetime

IGXObject interface



# Classes

This chapter provides reference material on the classes in the Netscape Application Server Foundation Class Library.

The following classes are included in this chapter:

- GXAppLogic class
- GXSession2 class
- GXTemplateDataBasic class
- GXTemplateMapBasic class



# GXAppLogic class

The GXAppLogic class is the base class for all AppLogic code. It provides a suite of useful AppLogic-related helper methods and member variables. You can, for example, use methods in your derived GXAppLogic class to create database connections, queries, transactions, and HTML output.

To derive a class from GXAppLogic, include `gxapplogic.h` and write a class declaration such as the following:

```
#include <gxapplogic.h>
class HelloAppLogic : public GXAppLogic
```

In your derived class, override the `Execute()` method to implement the main task of the AppLogic object, as shown in the following example:

```
STDMETHODIMP
HelloAppLogic::Execute()
{
    return Result("<html><body>Hello, world!</body></html>");
}
```

## Include File

`gxapplogic.h`

## Members

| Variable                | Description   |
|-------------------------|---|
| <code>m_pContext</code> | A pointer to the IGXContext object, which provides access to Netscape Application Server services. Some objects require services from IGXContext.   |
| <code>m_pStream</code>  | A pointer to the IGXStream object containing the AppLogic output stream. The <code>EvalOutput()</code> and <code>EvalTemplate()</code> methods merge the contents of dynamic data with a template file to send to the output stream. If the client of the request is a Web browser, the streamed data is HTML output. |

| Variable  | Description  |
|-----------|--|
| m_pValIn  | A pointer to the IGXValList object containing input parameters and other information. During the Execute() method, an AppLogic can access items in the IGXValList to retrieve the arguments passed into the request. |
| m_pValOut | A pointer to the IGXValList object containing output parameters. During the Execute() method, the AppLogic can add or update items in the IGXValList to specify output values for the request.                       |

## Methods

| Method              | Description  |
|---------------------|--|
| CreateDataConn()    | Creates a new data connection object and opens a connection to a database or data source.  |
| CreateDataConnSet() | Creates a collection used to dynamically assign query name / data connection pairs before loading a query file.                              |
| CreateHierQuery()   | Creates a new query object used for building and running a hierarchical query.   |
| CreateMailbox()     | Creates an electronic mailbox object used for communicating with a user through email.   |
| CreateQuery()       | Creates a new query object used for building and running a flat query.   |
| CreateSession()     | Creates a new session object used for tracking a user session.   |
| CreateTrans()       | Creates a new transaction object used for transaction processing operations on a database.   |
| DeleteCache()       | Deletes the result cache for a specified AppLogic.   |
| DestroySession()    | Deletes a user session.  |
| EvalOutput()        | Creates an output report by merging data with a report template file.  |
| EvalTemplate()      | Creates an output report by merging data with a report template file. The report is an HTML document that can be viewed using a Web browser. |

| Method                    | Description  |
|---------------------------|--|
| Execute( )                | Performs the main task of an AppLogic object, such as accessing a database, generating a report, or other operations. Should be overridden or implemented. |
| GetAppEvent( )            | Retrieves the application event object.  |
| GetSession( )             | Returns an existing user session.  |
| GetStateTreeRoot( )       | Returns an existing root node of a state tree or creates a new one.  |
| IsAuthorized( )           | Checks a user's permission level to a specified action or AppLogic.  |
| IsCached( )               | Returns true if AppLogic results are being saved in the result cache.  |
| LoadHierQuery( )          | Creates a hierarchical query by loading a query file and one or more query names with associated data connections.   |
| Log( )                    | Writes a message to the server log.  |
| LoginSession( )           | Logs an authorized user into a session with a secured application.   |
| LogoutSession( )          | Logs a user out of a session with a secured application.   |
| NewRequest( )             | Calls another AppLogic from within the current AppLogic.   |
| NewRequestAsync( )        | Calls another AppLogic from within the current AppLogic, and runs it asynchronously.   |
| RemoveAllCachedResults( ) | Clears an AppLogic's result cache.   |
| RemoveCachedResult( )     | Clears specific results from an AppLogic's result cache.   |
| Result( )                 | Specifies the return value of an AppLogic.   |
| SaveSession( )            | Saves changes to a session.  |
| SetCacheCriteria( )       | Stores AppLogic results, such as HTML, data values, and streamed results, in a result cache.   |
| SetSessionVisibility( )   | Sets the session visibility.   |
| SetVariable( )            | Sets a value that is passed to later AppLogic requests that are called by the same client session.   |

| Method               | Description  |
|----------------------|--|
| SkipCache()          | Skips result caching for the current AppLogic execution. |
| StreamResult()       | Streams output results as a string.                      |
| StreamResultBinary() | Streams output binary data, such as a GIF file.          |
| StreamResultHeader() | Streams output header data.                              |

## Related Topics

Chapter 4, “Writing Server-Side Application Code,” and Chapter 11, “Running and Debugging Applications,” in *Programmer’s Guide*.

## CreateDataConn()

---

Creates a new data connection object and opens a connection to a database or data source.

```
HRESULT CreateDataConn(
    DWORD flags,
    DWORD driver,
    IGXVallist *props,
    IGXContext *context,
    IGXDataConn **ppConn);
```

**flags** One or more optional flags used for connecting to the specified data source.

- To try to use a cached connection, if one is available, specify `GX_DA_CACHED`. If no cached connections are currently available, a new one is created.
- To always create a new connection (instead of using a cached connection), specify `GX_DA_NEW`.
- To retry if a connection is not available, specify `GX_DA_CONN_BLOCK`.
- To return a failure immediately after the first attempt if a connection is not available, specify `GX_DA_CONN_NOBLOCK`.

The AppLogic can pass one parameter from both mutually exclusive pairs, as shown in the following example:

```
(GX_DA_CACHED | GX_DA_CONN_BLOCK)
```

Specify 0 (zero) to use the system's default settings: GX\_DA\_CACHED and GX\_DA\_CONN\_BLOCK

**driver** Specify one of the following:

|                              |                            |
|------------------------------|----------------------------|
| GX_DA_DRIVER_ODBC            | GX_DA_DRIVER_SYBASE_CTLIB  |
| GX_DA_DRIVER_MICROSOFT_JET   | GX_DA_DRIVER_MICROSOFT_SQL |
| GX_DA_DRIVER_INFORMIX_SQLNET | GX_DA_DRIVER_INFORMIX_CLI  |
| GX_DA_DRIVER_INFORMIX_CORBA  | GX_DA_DRIVER_DB2_CLI       |
| GX_DA_DRIVER_ORACLE_OCI      | GX_DA_DRIVER_DEFAULT       |

If GX\_DA\_DRIVER\_DEFAULT is specified, the Netscape Application Server evaluates the drivers and their associated priorities set in the registry to determine the driver to use. Specify GX\_DA\_DRIVER\_DEFAULT if your system uses ODBC and native drivers, and if you want the Netscape Application Server to choose between an ODBC driver and a native driver at connection time.

**props** IGXVallist of connection-specific information required to log in to the data source. Use the following keys for the connection parameters:

- "DSN" for the data source name.
- "DB" for the database name.
- "USER" for the user name.
- "PSWD" for the password.

**context** A pointer to the IGXContext object, which provides access to Netscape Application Server services. Specify NULL.

**ppConn** A pointer to the created IGXDataConn object. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** A data connection is a communication link or session with a database or other data source. Before interacting with a data source, an AppLogic must first establish a connection with it. Each connection is represented by a data

connection object, which contains all the information needed to communicate with a database or data source, such as the name of the database, database driver, user name, password, and so on. A data connection object is an instance of the IGXDataConn interface.

Use `CreateDataConn()` to set up a separate connection for each database or data source you want to access. AppLogic objects refer to the data connection object in their methods that perform subsequent operations on the database.

- Rules**
- Call `CreateDataConn()` before running any other database operations requiring a data connection object.
  - Your network and the database server must be correctly configured and running so that the AppLogic on your application server can log into the database management system with which it will communicate.
  - The data source name, database name, user name, and password must be valid for the database management system to which you want to connect.
  - The AppLogic must log in with sufficient access rights to perform all operations it attempts on the data source.
- Tips**
- Before logging in to the database, the AppLogic should check the user's security level to verify sufficient access rights to perform intended operations on the database.
  - The Data Access Engine (DAE) manages database connections and related housekeeping tasks, such as shutdown and cleanup. While the DAE performs these tasks automatically and intermittently, an AppLogic can also explicitly close data connections using `CloseConn()` in the IGXDataConn interface.
  - Before using an ODBC connection, you must use the ODBC administration utility supplied with your database software to define and name a data source. For more information about how to do this, refer to your ODBC documentation.
  - To connect to a Sybase database, specify NULL for the datasource, and specify the database in the form of *server.database\_name*. For example:

```
devds003:dnet00a
```

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** `// Method to open a connection to a database`

```

STDMETHODIMP
OBBaseAppLogic::GetOBDataConn(IGXDataConn **ppConn)
{
    HRESULT hr=GXE_SUCCESS;

    // Create a vallist for the connection parameters
    IGXValList *pList=GXCreateValList();
    if(pList) {
        // Set up the connection parameters
        GXSetValListString(pList, "DSN", OB_DSN);
        GXSetValListString(pList, "DB", "");
        GXSetValListString(pList, "USER", OB_USER);
        GXSetValListString(pList, "PSWD", OB_PASSWORD);

        // Attempt to create the connection
        hr = CreateDataConn(0, GX_DA_DRIVER_DEFAULT, pList, m_pContext,
            ppConn);

        // Release pList when it's no longer needed
        pList->Release();
    }
    return hr;
}

```

**Related Topics** IGXDataConn interface IGXValList interface

“About Database Connections” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

---

## CreateDataConnSet()

Creates a collection used to dynamically assign query name/data connection pairs before loading a query file.

**Syntax** HRESULT CreateDataConnSet(  
     DWORD flags,  
     IGXDataConnSet \*\*ppDataConnSet);

**flags** Specify 0. Internal use only.

**ppDataConnSet** Pointer to the created IGXDataConnSet object. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use `CreateDataConnSet()` only if you are loading a query file using `LoadHierQuery()`. To use a query file, an AppLogic first establishes a data connection with each database on which any queries will be run.

Next, the AppLogic calls `CreateDataConnSet()` to create an `IGXDataConnSet` object, then populates this collection with query name / data connection pairs. Each query name in the collection matches a named query in the query file. `IDataConnSet` provides a method for adding query name / data connection pairs to the collection. In this way, AppLogic can use standardized queries and select and assign data connections dynamically at runtime.

Finally, the AppLogic calls `LoadHierQuery()` to create the hierarchical query object.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `LoadHierQuery()`,  
`IGXDataConn` interface

“About Database Connections” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

## CreateHierQuery()

---

Creates a new query object used for building and running a hierarchical query.

**Syntax** `HRESULT CreateHierQuery(  
IGXHierQuery **pHQ);`

**pHQ** A pointer to the created `IGXHierQuery` object. When AppLogic is finished using the object, call the `Release()` method to release the interface instance.

**Usage** Use `CreateHierQuery()` for nested output or for merging query results with a template using `EvalOutput()` or `EvalTemplate()`.

A hierarchical query can be more complex than a flat query. A hierarchical query combines one or more flat queries which, when run on the database server, returns a result set with multiple nested levels of data. The number of nested levels is limited only by system resources.

The hierarchical query is not necessarily a single query. In fact, a hierarchical query is a collection of one or more flat queries arranged in a series of cascading parent-child, one-to-many relationships. The parent query obtains

the outer level of information, or summary, and the child query obtains the inner level of information, or detail. The parent level of information determines the grouping of information in its child levels. The child query is run multiple times, once for each row in the parent query's result set.

- Tips**
- Use `CreateQuery()` instead for simple, flat queries requiring tabular, non-nested output that is merged with HTML templates.
  - To use a hierarchical query, an AppLogic first creates each individual flat query and defines its selection criteria. Next, it creates the `IGXHierQuery` object with `CreateHierQuery()`, then calls `AddQuery()` repeatedly to add a child query to a parent query for each level of detail in the hierarchical query.
  - Alternatively, an AppLogic can create a hierarchical query by loading a query file using `LoadHierQuery()`. With this technique, the Netscape Application Server can cache query objects to service requests for identical queries more quickly.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

```
// Create the hierarchical query
IGXHierQuery *pHq=NULL;

if(((hr=CreateHierQuery(&pHq))==GXE_SUCCESS)&&pHq) {
    // Add a query
    pHq->AddQuery(pQuery, pConn, "SelCusts", "", "");
}
```

**Related Topics** `AddQuery()` in the `IGXHierQuery` interface,  
`CreateDataConn()`,  
`CreateQuery()`,  
`Execute()` in the `IGXHierQuery` interface,  
`IGXHierQuery` interface ,  
`IGXHierResultSet` interface

“Writing Hierarchical Queries” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## CreateMailbox()

---

Creates an electronic mailbox object used for communicating with a user's mailbox.

**Syntax** `HRESULT CreateMailbox(`

```

LPSTR pHost ,
LPSTR pUser ,
LPSTR pPassword ,
LPSTR pUserAddr ,
IGXMailbox **ppMailbox);

```

**pHost** Address of POP and SMTP server, such as mail.myOrg.com. If the POP and SMTP servers are running on different hosts, you must use two separate CreateMailbox() calls.

**pUser** Name of user's POP account, such as jdoe.

**pPassword** Password for POP server.

**pUserAddr** Return address for outgoing mail, such as john@myOrg.com. Usually the electronic mail address of the user sending the message.

**ppMailbox** A pointer to the created IGXMailbox object. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use CreateMailbox() to set up a mail session for sending and receiving electronic mail messages.

In the Internet electronic mail architecture, different servers are used for incoming and outgoing messages.

- POP (post-office protocol) servers process incoming mail and forward messages to the recipient's mailbox.
- SMTP (simple mail transport protocol) servers forward outgoing mail to the addressee's mail server.

**Rules**

- The specified user account and password must be valid for the specified POP host name.
- The user address must be valid for the specified SMTP server.

**Tip** Once instantiated, use the methods in the IGXMailBox interface to open and close a mailbox, as well as send and receive mail messages.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** IGXMailBox interface

“Introduction to Email in Netscape Application Server Applications” in Chapter 10, “Integrating Applications with Email” in *Programmer’s Guide*.

## CreateQuery()

Creates a new query object used for building and running a flat query.

**Syntax** HRESULT CreateQuery(  
IGXQuery \*\*ppQuery);

**ppQuery** A pointer to the created IGXQuery object. When AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** A flat query is the simplest type of query. It retrieves data in a tabular, non-hierarchical result set. Unlike a hierarchical query, a flat query returns a result set that is *not* divided into levels or groups.

An AppLogic can also use CreateQuery() to create a query object to perform SELECT, INSERT, DELETE, or UPDATE operations on a database.

- Tips**
- To query a database, the AppLogic first uses CreateQuery() to create the query object, then constructs the query selection criteria using methods in the IGXQuery interface, and finally runs the query on a database server. The AppLogic can process results using methods in the IGXResultSet interface.
  - Alternatively, AppLogic can pass a SQL SELECT statement directly to the database server using SetSQL() in the IGXQuery interface.
  - To retrieve data with nested levels of information, use CreateHierQuery() instead.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
// Create a query to insert data into a table
IGXQuery *pUserQuery=NULL;

if(((hr=CreateQuery(&pUserQuery))==GXE_SUCCESS)&&pUserQuery) {
    pUserQuery->SetSQL("INSERT INTO OBUser(userName, password, userType,
    eMail) VALUES (:userName, :password, :userType, :eMail)");
```

**Related Topics** CreateDataConn()

CreateQuery( ),  
 IGXHierQuery interface ,  
 IGXHierResultSet interface ,  
 ExecuteQuery( ) in the IGXDataConn interface

## CreateSession()

---

Creates a new session object used for tracking a user session.

**Syntax** HRESULT CreateSession(  
     DWORD dwFlags,  
     ULONG dwTimeout,  
     LPSTR pAppName,  
     LPSTR pSessionID,  
     IGXSessionIDGen \*pIDGen,  
     IGXSession2 \*\*ppSession);

**dwFlags** Specify one of the following flags, or 0 to use the default system settings:

- GXSESSION\_LOCAL to make the session visible to AppLogics in the local process only.
- GXSESSION\_CLUSTER to make the session visible to all AppLogics within the cluster.
- GXSESSION\_DISTRIB to make the session visible to all AppLogics on all Netscape Application Servers.
- GXSESSION\_TIMEOUT\_ABSOLUTE to specify that the session expires at a specific date and time. Do not use this flag. It is currently unimplemented but reserved for future use.
- GXSESSION\_TIMEOUT\_CREATE to specify that the session expires *n* seconds from the time the session was created.

The default scope is distributed and the default timeout is 60 seconds from the time the session was last accessed.

**dwTimeout** Session timeout, in number of seconds, or zero for no timeout. The meaning of timeout depends on the timeout flag specified in dwFlags. A value of 0 means the session is deleted when the AppLogic calls DestroySession( ).

**pAppName** Name of the application associated with the session. The application name enables the Netscape Application Server to determine which AppLogics have access to the session data. Specify NULL to use the application name assigned to the AppLogic during kreg registration.

**pSessionID** The session ID to use. Specify NULL to use the default ID generated by the system.

**pIDGen** The session ID generation object used to generate session IDs. Specify NULL to use the default IGXSessionIDGen object, or specify a custom session ID generation object.

**ppSession** A pointer to the created IGXSession2 object. When AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use CreateSession() to create a new session between a user and your application. AppLogics use sessions to store information about each user's interaction with an application. For example, a login AppLogic might create a session object to store the user's login name and password. This session data is then available to other AppLogics in the application.

**Rule** If you implement a custom session class, you must override CreateSession().

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** In the following code, GetSession() checks if a session exists. If there isn't an existing session, CreateSession() creates a new session.

```
hr = GetSession(0, "Catalog", NULL, &m_pSession);
if (hr != GXE_SUCCESS)
{
    Log("Could not get session, creating a new one");
    hr = CreateSession(GXSESSION_DISTRIB, 0, NULL,
        NULL, NULL, &m_pSession);
}
```

**Related Topics** GetSession(),  
SaveSession(),  
GXSession2 class,  
IGXSession2 interface

"Starting a Session" in Chapter 8, "Managing Session and State Information" in *Programmer's Guide*.

“Writing Hierarchical Queries” in Chapter 6, “Querying a Database” in *Programmer’s Guide*.

## CreateTrans()

---

Creates a new transaction object used for transaction processing operations on a database.

**Syntax** HRESULT CreateTrans(  
    IGXTrans \*\*ppTrans);

**ppTrans** A pointer to the created IGXTrans object. When AppLogic is finished using the object, after a call to either Commit() or Rollback(), call the Release() method to release the interface instance.

**Usage** Transaction processing allows the AppLogic to define a series of operations that succeed or fail as a group. If all operations in the group succeed, then the system commits, or saves, all of the modifications from the operations. If any operation in the group fails for any reason, then the AppLogic can roll back, or abandon, any proposed changes to the target table(s).

If your application requires transaction processing, use CreateTrans() to create a transaction object. Pass this transaction object to subsequent methods, such as AddRow() or ExecuteQuery(), that make up a transaction.

- Tips**
- Use this method in conjunction with AddRow(), UpdateRow(), and DeleteRow() methods in the IGXTable interface and ExecuteQuery() in the IGXDataConn interface.
  - To manage transaction processing operations, use CreateTrans() to create an instance of the IGXTrans interface, then use Begin(), Commit(), and Rollback() in the IGXTrans interface to begin, commit, and rollback the transaction, respectively.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
// Create a transaction for several insert operations
IGXTrans *pTx=NULL;

if(((hr=CreateTrans(&pTx))==GXE_SUCCESS)&&pTx) {
    // Begin the transaction
    pTx->Begin();
    IGXResultSet *pRset=NULL;
```

```

// Update User
if(((hr=pUserPQuery->Execute(0, pUserValList, pTx, NULL,
&pRset))==GXE_SUCCESS)&&pRset) {

    // The result set is not needed; release it
    pRset->Release();

    // Update Customer
    if(((hr=pCustPQuery->Execute(0, pCustValList, pTx, NULL,
&pRset))==GXE_SUCCESS)&&pRset) {

        // All updates succeeded. Commit the transaction
        pTx->Commit(0, NULL);
        GXSetValListString(m_pValIn, "ssn", m_pSsn);
        GXSetValListString(m_pValIn, "OUTPUTMESSAGE", "Successfully
updated customer record");

        if(NewRequest("AppLogic CShowCustPage", m_pValIn, m_pValOut,
0)!=GXE_SUCCESS)
            HandleOBSYSTEMError("Could not chain to CShowCustPage
applogic");
    }
    else {
        pTx->Rollback();
        HandleOBSYSTEMError("Could not insert checking account record
for new customer");
    }
}
else {
    pTx->Rollback();
    HandleOBSYSTEMError("Could not insert checking account record for
new customer");
}
pTx->Release();
}
else
    HandleOBSYSTEMError("Could not start transaction");

```

**Related Topics** IGXTrans interface

“Managing Database Transactions” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

## DeleteCache()

---

Deletes the result cache for a specified AppLogic.

**Syntax** HRESULT DeleteCache(  
LPSTR guid);

**guid** The guid that identifies the AppLogic whose result cache to delete. Specify NULL to delete the current AppLogic’s cache.

**Usage** To free system resources, use DeleteCache() to clear all results from an AppLogic’s cache when the results are no longer needed. This method also stops further caching of results.

- Tips**
- To clear an AppLogic’s result cache, but continue caching, use RemoveAllCachedResults().
  - To clear a specific result from the cache, use RemoveCachedResult().

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```

HRESULT hr;
LPSTR guid;

guid = GXGetValListString(m_pValIn, "applogic");

hr = DeleteCache(guid);

if (hr == GXE_SUCCESS)
    sprintf(msg, "Successfully deleted cache");
else
    sprintf(msg, "Failed to delete cache");

```

**Related Topics** RemoveAllCachedResults(),  
RemoveCachedResult(),  
SetCacheCriteria()

“Caching AppLogic Results to Improve Performance” in Chapter 4, “Writing Server-Side Application Code” in *Programmer’s Guide*.

## DestroySession()

---

Deletes a user session.

**Syntax** `HRESULT DestroySession(  
IGXSessionIDGen *pIDGen);`

**pIDGen** The session ID generation object used to generate session IDs. Specify NULL to use the default IGXSessionIDGen object, or specify a custom session ID generation object.

**Usage** To increase security and conserve system resources, use DestroySession() to delete a session between a user and the application when the session is no longer required. An AppLogic typically calls DestroySession() when the user logs out of an application.

**Tip** If the AppLogic set a timeout value for the session when it was created, you need not delete the session explicitly with DestroySession(). The session is deleted automatically when the timeout expires.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateSession(),  
GetSession()

“Removing a Session and Its Related Data” in Chapter 8, “Managing Session and State Information” in *Programmer’s Guide*.

## EvalOutput()

---

Creates an output report by merging data with a report template file. Depending on the client—AppLogic or web browser—EvalOutput() returns either a self-describing data stream or HTML output.

**Syntax 1** Merges a template with data from a hierarchical query object.

```
HRESULT EvalOutput(  
    LPSTR templatePath,  
    IGXHierQuery *query,  
    IGXTemplateMap *map,  
    IGXStream *stream,  
    IGXValList *props);
```

**Syntax 2** Merges a template with data from an IGXTemplateData object or IGXHierResultSet object. IGXHierResultSet objects implement the IGXTemplateData interface.

```
HRESULT EvalOutput(
    LPSTR templatePath,
    IGXTemplateData *data,
    IGXTemplateMap *map,
    IGXStream *stream,
    IGXValList *props);
```

**templatePath** Path to the template file used to create the report. At a minimum, specify the file name. Do not specify the filename extension; for example, specify “report” instead of “report.html”. The EvalOutput() method automatically uses the correct filename extension depending on the client type. Use a relative path whenever possible. The Netscape Application Server first searches for the template using the specified path. If the template is not found, the Netscape Application Server uses the configured TEMPLATE\PATH search path to find it. For more information on configuring the search path, see *Administration Guide*.

**query** Pointer to the hierarchical query object from which CreateTrans() derives the hierarchical result set to merge with the template. The Template Engine runs the query on the database server. To specify this parameter, the AppLogic must first create the specified hierarchical query, using CreateHierQuery() in the GXAppLogic class, and then define it using methods in the IGXHierQuery interface or calling LoadHierQuery().

**map** Pointer to the field map that links template fields to calculated values. Fields in the template are expressed with the cell type gx tags. Additionally, the map can be used to map source data with a non-matching field name but identically-formatted data. To specify this parameter, the AppLogic should instantiate the GXTemplateMapBasic class, add template / field mappings using Put() in the IGXTemplateMap interface, then pass the populated IGXTemplateMap object to EvalOutput() for template processing.

**stream** Pointer to the output stream where results will be captured for subsequent retrieval and processing. Specify NULL to use the default stream, which sends results back to the client. To specify this parameter, an AppLogic creates a stream buffer object from IGXStreamBuffer, which it passes to EvalOutput(). After EvalOutput() returns, the AppLogic calls GetStreamData() in the IGXStreamBuffer interface to retrieve the contents of the buffer as an array of byte values.

**data** Pointer to the IGXTemplateData object containing data. This can be a hierarchical result set from executing a hierarchical query or it can be data programmatically organized in memory. To specify this data in memory, an AppLogic must first instantiate the GXTemplateDataBasic class (or implement your own version of the IGXTemplateData interface), populate the IGXTemplateData object with rows of hierarchical data, then pass it to EvalOutput() for template processing.

**props** Specify NULL.

**Usage** Use EvalOutput() in an AppLogic that returns output to different types of clients. The EvalOutput() method detects the client type, selects the appropriate template file to merge with the data, and generates the appropriate output, as described in the following table:

| Client  | Template File Used by EvalOutput() | Output Returned by EvalOutput()   |
|---|------------------------------------|---|
| Web browser   | HTML                               | HTML page   |
| AppLogic that passed to its NewRequest() call the following key and value in the input IGXValList parameter:<br>key: gx_client_type<br>value: "ocl"   | GXML                               | Self-describing data stream, which contains the names of the fields in the result set and their values. |
| AppLogic that does not specify a client type explicitly, or that passed to its NewRequest() call the following key and value in the input IGXValList parameter:<br>key: gx_client_type<br>value: "http" | HTML                               | HTML page   |

Both the GXML and HTML template files contain embedded tags, called GX tags, that specify how the Template Engine merges dynamic data with the template to produce the output report. In addition, the HTML template file can contain graphics, static text, and other components, just like any HTML-formatted document.

The data that the Template Engine merges with the template can come from several sources. Most commonly, it comes from the result set of a hierarchical query. However, it can also come from an IGXTemplateData object containing data organized hierarchically in memory.

- Tips**
- If possible, write queries so that field names in the result set match the field names in the template. Otherwise, you must use an IGXTemplateMap object to map field names.
  - To create an GXML file, you can convert an HTML template file with the khtml2gxml utility. This utility strips HTML tags from the template file and saves the file as a GXML file. The following is an example of how to run the utility from the command line:

```
khtml2gxml mytemplate.html
```

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
// Create a hierarchical query used for template processing
IGXHierQuery *pHq=NULL;

if(((hr=CreateHierQuery(&pHq))==GXE_SUCCESS)&&pHq) {
    // Add a query that has already been defined
    pHq->AddQuery(pQuery, pConn, "SelCusts", "", "");

    // Pass the hierarchical query to EvalOutput()
if(EvalOutput("apps/template/customer", pHq, NULL, NULL,
    NULL)!=GXE_SUCCESS)
    Result("<HTML><BODY>Unable to evaluate template.</BODY></HTML>");
```

**Related Topics** EvalTemplate(),  
Result(),  
IGXHierQuery interface,  
GXTemplateDataBasic class and the IGXTemplateData interface,  
GXTemplateMapBasic class and the IGXTemplateMap interface

“Returning Results From an AppLogic Object” in Chapter 4, “Writing Server-Side Application Code” in *Programmer’s Guide*.

## EvalTemplate()

---

Creates an output report by merging data with a report template file. The report is an HTML document that can be viewed using a Web browser.

**Syntax 1** Merges an HTML report template with data from a hierarchical query object.

```
HRESULT EvalTemplate(
    LPSTR path,
    IGXHierQuery *query,
    IGXTemplateMap *map,
    IGXStream *stream,
    IGXVallList *props);
```

**Syntax 2** Merges an HTML report template with data from an IGXTemplateData object or IGXHierResultSet object. IGXHierResultSet objects implement the IGXTemplateData interface.

```
HRESULT EvalTemplate(
    LPSTR path,
    IGXTemplateData *data,
    IGXTemplateMap *map,
    IGXStream *stream,
    IGXVallList *props);
```

**path** Path to the HTML template file used to create the report. At a minimum, specify the file name. Use a relative path whenever possible. The Netscape Application Server first searches for the template using the specified path. If the template is not found, the Netscape Application Server uses the configured TEMPLATE\PATH search path to find it. For more information on configuring the search path, see the *Administration Guide*.

**query** Pointer to the hierarchical query object from which EvalTemplate() derives the hierarchical result set to merge with the HTML template. The Template Engine runs the query on the database server. To specify this parameter, the AppLogic must first create the specified hierarchical query, using CreateHierQuery() in the GXAppLogic class, and then define it using methods in the IGXHierQuery interface or calling LoadHierQuery().

**map** Pointer to the field map that links template fields to calculated values. Fields in the template are expressed with the cell type gx tags. Additionally, the map can be used to map source data with a non-matching field name but identically-formatted data. To specify this parameter, the AppLogic should instantiate the GXTemplateMapBasic class, add template / field mappings using Put() in the IGXTemplateMap interface, then pass the populated IGXTemplateMap object to EvalTemplate() for template processing.

**stream** Pointer to the output stream where results will be captured for subsequent retrieval and processing. Specify NULL to use the default stream, which sends results back to the client. To specify this parameter, an AppLogic

creates a stream buffer object from `IGXStreamBuffer`, which it passes to `EvalOutput()`. After `EvalTemplate()` returns, the `AppLogic` calls `GetStreamData()` in the `IGSXstreamBuffer` interface to retrieve the contents of the buffer as an array of byte values.

**data** Pointer to the `IGXTemplateData` object containing data. This can be a hierarchical result set from executing a hierarchical query or it can be data programmatically organized in memory. To specify this data in memory, an `AppLogic` must first instantiate the `GXTemplateDataBasic` class (or implement your own version of the `IGXTemplateData` interface), populate the `IGXTemplateData` object with rows of hierarchical data, then pass it to `EvalTemplate()` for template processing.

**props** Specify `NULL`.

**Usage** Use `EvalTemplate()` to create an HTML report by merging data with an HTML template file. An HTML template is an HTML document with the addition of special embedded tags, called GX tags, that specify how the Template Engine merges dynamic data with the template to produce the output report or HTML page. In addition to these dynamic links, a template can contain static text, graphics, and other components, just like any HTML-formatted document.

The data that the Template Engine merges with the template can come from several sources. Most commonly, it comes from the result set of a hierarchical query. However, it can also come from an `IGXTemplateData` object containing data organized hierarchically in memory.

- Tips**
- If your `AppLogic` requires the flexibility of returning different output depending on the client—a Web browser or another `AppLogic`—use `EvalOutput()` instead.
  - If possible, write queries so that field names in the result set match the field names in the template. Otherwise, you must use an `IGXTemplateMap` object to map field names.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```
// Create a flat query
IGXQuery *pQuery=NULL;
pQuery->SetTables("OBCustomer, OBAccount");
pQuery->SetFields("lastName, firstName, userName, ssn");
pQuery->SetWhere(whereClause);
pQuery->SetOrderBy("lastName, firstName");
```

```

// Create the hier query used for template processing
IGXHierQuery *pHq=NULL;

if((hr=CreateHierQuery(&pHq))==GXE_SUCCESS)&&pHq) {
    // Add a query
    pHq->AddQuery(pQuery, pConn, "SelCusts", "", "");

// Pass the hierarchical query to EvalTemplate()
if(EvalTemplate("Customer.html", pHq, NULL, NULL, NULL)!=GXE_SUCCESS)
    Result("<HTML><BODY>Unable to evaluate template.</BODY></HTML>");

```

**Related Topics** EvalOutput(),  
IGXHierQuery interface,  
GXTemplateDataBasic class and the IGXTemplateData interface,  
GXTemplateMapBasic class and the IGXTemplateMap interface

“Returning Results From an AppLogic Object” in Chapter 4, “Writing Server-Side Application Code” in *Programmer’s Guide*.

---

## Execute()

Performs the main task of an AppLogic, such as accessing a database, generating a report, or other operations. It should be overridden in your AppLogic subclass.

**Syntax** HRESULT Execute()

**Usage** Netscape Application Server calls the AppLogic’s Execute() method automatically whenever a request is received for an AppLogic, such as when a user submits a form or an information request.

**Rule** By default, Execute() does nothing except return a value of zero (0). You should always write code to override this method in your GXAppLogic derived class.

- Tips**
- In general, your AppLogic class will inherit from the GXAppLogic class and override the default behavior of the Execute() method, such as retrieving an orders report from a database.
  - The AppLogic can analyze the m\_pValIn member variable for input arguments using methods in the IGXValList interface.

- The AppLogic can modify the m\_pValIn member variable using methods in the IGXValList interface.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** In the following example, Execute() displays an HTML page:

```
OBSHOWNewCustPage::Execute()
{
    if(EvalTemplate("GXApp/OnlineBank/templates/NewCust.html",
        (IGXHierQuery*)NULL, NULL, NULL, NULL)!=GXE_SUCCESS)
        Result("<HTML><BODY>Unable to evaluate template.</BODY></HTML>");
    return GXE_SUCCESS;
}
```

**Related Topics** Result(),  
IGXValList interface

“Performing the Main Task in an AppLogic Object” in Chapter 4, “Writing Server-Side Application Code” in *Programmer’s Guide*.

## GetAppEvent()

Retrieves the application event object.

**GetAppEvent() is deprecated.** See New Usage section for more information.

**Syntax** HRESULT GetAppEvent(  
IGXAppEvent \*\*ppAppEvent);

**ppAppEvent** A pointer to the retrieved IGXAppEvent object. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**New Usage** This method is deprecated and is provided for backward compatibility only.

New applications should use the IGXAppEventManager interface and IGXAppEventObj interface, along with the helper function GXContextGetAppEventManager().

**Old Usage** Use GetAppEvent() to retrieve an IGXAppEvent object. Through the IGXAppEvent interface, you can create and manage application events. An AppLogic uses application event objects to define events that are triggered at a specified time or times or when triggered explicitly.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** IGXAppEvent interface,  
RegisterEvent() in the IGXAppEvent interface

“Using Events” in Chapter 3, “Application Development Techniques” in *Programmer’s Guide*.

## GetSession()

---

Returns an existing user session.

**Syntax**

```
HRESULT GetSession(
    DWORD dwFlags,
    LPSTR pAppName,
    IGXSessionIDGen *pIDGen,
    IGXSession2 **ppSession);
```

**dwFlags** Specify 0 (zero).

**pAppName** Name of the application associated with the session. The application name enables the Netscape Application Server to determine which AppLogics have access to the session data. Specify NULL to use the application name assigned to the AppLogic during kreg registration.

**pIDGen** The session ID generation object used to generate session IDs. Specify NULL to use the default IGXSessionIDGen object, or specify a custom session ID generation object.

**ppSession** A pointer to the created or retrieved IGXSession2 object. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use GetSession() to obtain an existing session. Use it also to determine if a user session exists before calling CreateSession() to create one.

**Rule** If you implement a custom session class, you must implement your own method to get a session, which in turn, can call the GetSession() method.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example 1** In the following code, GetSession() checks if a session exists. If there isn’t an existing session, CreateSession() creates a new session.

```
hr = GetSession(0, "Catalog", NULL, &m_pSession);
```

```

if (hr != GXE_SUCCESS)
{
    Log("Could not get session, creating a new one");
    hr = CreateSession(GXSESSION_DISTRIB, 0, NULL,
        NULL, NULL, &m_pSession);
}

```

**Example 2** In the following code, `GetSession()` gets an existing session, then checks if the user is authorized to perform a secured task:

**Related Topics** `CreateSession()`,  
`LoginSession()`,  
`SaveSession()`,  
GXSession2 class,  
IGXSession2 interface

“Using an Existing Session” in Chapter 8, “Managing Session and State Information” in *Programmer’s Guide*.

---

## GetStateTreeRoot()

Returns an existing root node of a state tree or creates a new one.

**Syntax** HRESULT GetStateTreeRoot(  
 DWORD dwFlags,  
 LPSTR pName,  
 IGXState2 \*\*ppStateTree)

**dwFlags** Specify one of the following flags or zero to use the default settings:

- GXSTATE\_LOCAL to make the node visible to the local process only.
- GXSTATE\_CLUSTER to make the node visible to all AppLogics within the cluster.
- GXSTATE\_DISTRIB, the default, to make the node visible to all AppLogics on all servers.

**pName** The name of the root node. If a node with this name doesn’t already exist, a new node is created.

**ppStateTree** A pointer to the created IGXState2 object. When the AppLogic is finished using the object, call the `Release()` method to release the interface instance.

**Usage** Use `GetStateTreeRoot()` to create a state tree. A state tree is a hierarchical data storage mechanism. It is used primarily for storing application data that needs to be distributed across server processes and clusters.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Example** The following code shows how to create a state tree and a child node:

```
HRESULT hr;

hr = GetStateTreeRoot(GXSTATE_DISTRIB, "Grammy", &m_pStateRoot);

if (hr == NOERROR && m_pStateRoot)
{
    IGXState2 *pState = NOERROR;
    hr = m_pStateRoot->GetStateChild("Best Female Vocal",
        &pState);
    if (hr != NOERROR || !pState)
    {
        hr = m_pStateRoot->CreateStateChild("Best Female Vocal",
            0, GXSTATE_DISTRIB, &pState);
    }
}
```

**Related Topics** `IGXState2` interface

“Using the State Layer” in Chapter 8, “Managing Session and State Information” in *Programmer’s Guide*.

## IsAuthorized()

---

Checks a user’s permission level to a specified action or AppLogic.

**Syntax 1** Use in most cases.

```
HRESULT IsAuthorized(
    LPSTR pTarget,
    LPSTR pPermission,
    DWORD *pResult);
```

**Syntax 2** Contains several parameters that are placeholders for future functionality.

```
HRESULT IsAuthorized(
    LPSTR pDomain,
    LPSTR pTarget,
    LPSTR pPermission,
    DWORD method,
```

```

    DWORD flags,
    IGXCred *pCred,
    IGXObject *pEnv,
    DWORD *pResult);

```

**pDomain** The type of Access Control Lists (ACL). An ACL (created by the server administrator) defines the type of operations, such as Read or Write, that a user or group can perform. There are two types of ACLs: AppLogic and general. For this parameter, specify one of the following strings, which specifies the type of ACL to check for this user:

"kiva:acl,logic"

"kiva:acl,general"

**pTarget** The name of the ACL, if the ACL is a general type. If the ACL is an AppLogic ACL, specify the AppLogic name or GUID string.

**pPermission** The type of permission, for example, "EXECUTE."

**method** Specify 0.

**flags** Specify 0.

**pCred** Specify NULL.

**pEnv** Specify NULL.

**pResult** Pointer to the client-allocated variable that contains the returned permission status. The variable is set to one of the following enum constants:

| Constant         | Description   |
|------------------|---|
| GXACL_ALLOWED    | The specified permission is granted to the user.                          |
| GXACL_NOTALLOWED | The specified permission is not granted to the user.                      |
| GXACL_DONTKNOW   | The specified permission is unlisted or there is conflicting information. |

**Usage** Use `IsAuthorized()` in portions of the code where application security is enforced through Access Control Lists (ACL). This method lets an application check if a user has permission to execute an AppLogic or perform a particular

action. The application can use the result of `IsAuthorized()` as a condition in an `If` statement. It can, for example, return a message to users who are denied access to an AppLogic.

Application developers should obtain the list of registered ACLs, users and groups from the server administrator who created these items. ACLs are created through the Enterprise Administrator tool or through the kreg tool.

**Rule** Before calling `IsAuthorized()`, the application must create a session with `CreateSession()` and a user must be logged in with `LoginSession()`.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Example** `DWORD auth_result = 0;`

```
if (IsAuthorized("Shop_Inventory", "WRITE", &auth_result) != NOERROR ||
    auth_result != (DWORD) GXACL_ALLOWED)
{
    Log("Unauthorized access: Shop_Inventory");

    EvalOutput("kivaapp/shop/unauthorized_access",
               (IGXTemplateData *) NULL,
               (IGXTemplateMap *) NULL, NULL, NULL);
}
else
// Update inventory
```

**Related Topics** `LoginSession()`

“Secure Sessions” in Chapter 9, “Writing Secure Applications” in *Programmer’s Guide*.

## IsCached()

Returns true if AppLogic results are being saved in the result cache.

**Syntax** `BOOL IsCached()`

**Usage** Call `IsCached()` to determine whether caching is enabled for the current AppLogic. You should, for example, call `IsCached()` before calling `SetCacheCriteria()` to avoid inadvertently overwriting the current contents of the result cache.

**Return Value** A `BOOL` true if caching is enabled, or a `BOOL` false if not.

**Related Topics** SkipCache()

“Caching AppLogic Results to Improve Performance” in Chapter 4, “Writing Server-Side Application Code” in *Programmer’s Guide*.

## LoadHierQuery()

---

Creates a hierarchical query by loading a query file containing one or more query names and associated data connections.

**Syntax**

```
HRESULT LoadHierQuery
    LPSTR pFileName,
    IGXDataConnSet *pDataConnSet,
    DWORD flags,
    IGXVallist *pParams,
    IGXHierQuery **ppHierQuery);
```

**pFileName** Name of the query (.GXQ) file, including the path. Use a relative path when possible.

A query file is an ASCII text file containing one or more SQL statements. You can create the file using any ASCII text editor. Use the following syntactical guidelines:

- The file for a hierarchical query contains several SQL SELECT statements (compliant with ANSI SQL89) with the following additions:

- Each query is preceded by the following line:

```
query queryName using (driverCode, DSN, UserName) is
```

- For a child query, append the following line after the SQL SELECT statement:

```
join currentQueryName to parent parentName where
currentQueryName.table.column = parentName.colorAlias
```

- In the query file, do not use any semicolons (;) or other vendor-specific SQL statement terminators.

**pDataConnSet** Collection of query name/data connection pairs. The query names in the collection must match the named queries in the query file. The associated IDataConn object identifies the data connection for the query.

**flags** Specify 0 (zero). Internal use only.

**pParams** IGXVallist of query file parameters, or NULL. A collection of placeholders for the WHERE clause. A placeholder may be a name or a number. It is prefixed by a colon (:) character. The placeholders can be replaced by specifying replacement values in the ValList parameter.

**ppHierQuery** Pointer to the created IGXHierQuery object. When the AppLogic is finished using the object, call the Release( ) method to release the interface instance.

**Usage** Use LoadHierQuery( ) to create a hierarchical query object. An AppLogic can retrieve standardized queries stored in a data file and, at runtime, can dynamically select and assign the data sources on which the query is run. You create the query file separately using the Query Designer or an ASCII text editor, ANSI 89 standard SQL SELECT statements, and specialized syntax. A query file can define both flat and hierarchical queries.

To use a query file, the AppLogic first establishes a data connection with each database on which any queries will be run. Next, the AppLogic calls CreateDataConnSet( ) in the AppLogic class to create an IGXDataConnSet collection, then populates this collection with query name / data connection pairs. Each query name in the collection matches a named query in the query file.

IGXDataConnSet provides a method for adding query name / data connection pairs to the collection. In this way, AppLogic can use standardized queries and assign data connections dynamically at runtime. Finally, the AppLogic calls LoadHierQuery( ) to create the hierarchical query object.

- Rules**
- AppLogic must first call CreateDataConnSet( ) to create an IGXDataConnSet, then add query name / data connection pairs using AddConn( ) in the IGXDataConnSet interface.
  - The query names in the collection must match the query names in the query file.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** The following example shows a query (GXQ) file and a section of an AppLogic that loads the hierarchical query file and creates an HTML report:

Query file:

```
/* STATES */
query STATES using (ODBC, kstates, kuser) is
select STATES.STATE as STATES_STATE
```

```

from STATES
where (STATES.REGION = ':REGION')
order by STATES.STATE asc

/* DETAILS */
query DETAILS using (ODBC, kdetails, kuser) is
select COUNTIES.COUNTYNAM as COUNTIES_COUNTYNAM,
       COUNTIES.POP as COUNTIES_POP,
       COUNTIES.STATE as COUNTIES_STATE
from COUNTIES
order by COUNTIES.COUNTYNAM asc

join DETAILS to parent STATES
where DETAILS.COUNTIES.STATE = 'STATES.STATES_STATE'

```

### AppLogic code snippet:

```

IGXDataConnSet *connSet = NULL;
HRESULT hr;
hr = CreateDataConnSet(0, &connSet);
if (hr == GXE_SUCCESS)
{
    // Create database connections
    IGXDataConn *conn_detailDB = NULL;
    IGXDataConn *conn_statesDB = NULL;

    IGXValList *pList=GXCreateValList();
    pList->SetValString("DSN", "kdetails");
    pList->SetValString("DB", "");
    pList->SetValString("USER", "kuser");
    pList->SetValString("PSWD", "kpassword");

    // Create first connection
    hr = CreateDataConn(0, GX_DA_DRIVER_DEFAULT, pList,
                       NULL, &conn_detailDB);
    if (hr == GXE_SUCCESS)
    {
        pList->SetValString("DSN", "dstates");
        pList->SetValString("DB", "");
        pList->SetValString("USER", "kuser");
        pList->SetValString("PSWD", "kpassword");

        // Create second connection
    }
}

```

```

hr = CreateDataConn(0, GX_DA_DRIVER_DEFAULT, pList,
                   NULL, &conn_statesDB);

pList->Release();

if (hr == GXE_SUCCESS)
{
    // Specify query / db connection pairs
    connSet->AddConn("DETAILS", conn_detailDB);
    connSet->AddConn("STATES", conn_statesDB);

    // Create IGXValList that contains the
    // REGION parameter value to pass to the
    // hierarchical query
    IGXValList param = GXCreateValList();
    param->SetValString("REGION", "WEST");

    IGXHierQuery *hqry;
    // Load the GXQ file with the db connection set
    // and parameter value

    hr = LoadHierQuery("state.gxq", connSet, 0,
                      param, &hqry);

    if (hr == GXE_SUCCESS)
    {
        // Run the report
        EvalOutput("state", hqry, NULL,
                  NULL, NULL);
    }
    else
        ....
}

```

**Related Topics** CreateDataConnSet(),  
IGXDataConnSet interface,  
IGXHierQuery interface

“Working with Query Files” in Chapter 6, “Querying a Database” in *Programmer’s Guide*.

---

## LoadQuery()

Creates a flat query by loading a query file.

**Syntax**

```
HRESULT LoadQuery(
    LPSTR pFileName,
    LPSTR pQueryName,
    DWORD flags,
    IGXValList *pParams,
    IGXQuery **ppQuery);
```

**pFileName** Name of the query (.GXQ) file, including the path. Use a relative path when possible.

A query file is an ASCII text file containing one or more SQL statements. You can create the file using any ASCII text editor. Use the following syntactical guidelines:

- The query file for a flat query contains a SQL SELECT statement (compliant with ANSI SQL89) preceded by the following line:

```
/* optional comments */
query queryName using (driverCode, DSN, UserName) is
```

where *queryName* is the name of the flat query. Do not use any semicolons (;) in the query file.

- In the query file, do not use any semicolons (;) or other vendor-specific SQL statement terminators. The SQL statement may contain placeholders in the WHERE clause.

**pQueryName** Name of the query in the query file.

**flags** Specify 0 (zero). Internal use only.

**pParams** IGXValList of query file parameters, or null. A collection of placeholders for the WHERE clause. A placeholder may be a name or a number. It is prefixed by a colon (:) character. The placeholders can be replaced by specifying replacement values in the IGXValList parameter.

**ppQuery** Pointer to the created IGXQuery object. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use LoadQuery() to create a flat query object by loading a query (.GXQ) file. An AppLogic can retrieve standardized queries stored in a data file and, at runtime, can dynamically select and assign the data source on which the query is run.

You create the query file separately using the Query Designer or an ASCII text editor, ANSI 89 standard SQL SELECT statements, and special syntax.

To run the flat query, call `ExecuteQuery()` in the `IGXDataConn` interface.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Example** The following example shows a query (GXQ) file and a section of an AppLogic that loads and executes the query:

Query file:

```
/* STATES */
query STATES using (ODBC, kstates, kuser) is
select STATES.STATE as STATES_STATE
from STATES
where (STATES.REGION = ':REGION')
order by STATES.STATE asc
```

AppLogic code snippet:

```
HRESULT hr;

// Create database connection
IGXDataConn *conn = NULL;

IGXValList *pList=GXCreateValList();
pList->SetValString("DSN", "kstates");
pList->SetValString("DB", "");
pList->SetValString("USER", "kuser");
pList->SetValString("PSWD", "kpassword");

hr = CreateDataConn(0, GX_DA_DRIVER_DEFAULT, pList,
                    NULL, &conn);
if (hr == GXE_SUCCESS)
{
    // Create IGXValList that contains the REGION
    // parameter value to pass to the
    // hierarchical query
    IGXValList param = GXCreateValList();
    param->SetValString("REGION", "WEST");

    IGXQuery *qry;
```

```

// Load the GXQ file with the parameter value
hr = LoadQuery("state.gxq", "STATES", 0,
               param, &qry);

// Execute the query
IGXResultSet *rs = NULL;
hr = conn->ExecuteQuery(GX_DA_RS_BUFERRING, qry, NULL,
                       NULL, &rs);

```

**Related Topics** IGXQuery interface

“Working with Query Files” in Chapter 6, “Querying a Database” in *Programmer’s Guide*.

---

## Log()

Writes a message to the server log.

**Syntax 1** Logs a message (type = GXEVENTTYPE\_INFORMATION and category = 0).

```

HRESULT Log(
    LPSTR msg);

```

**Syntax 2** Logs an event with a message, specifying the type and category of event.

```

HRESULT Log(
    DWORD type,
    DWORD category,
    LPSTR msg);

```

**msg** Message text to log.

**type** Message type. Use one of the following variables:

- GXEVENTTYPE\_INFORMATION
- GXEVENTTYPE\_ERROR
- GXEVENTTYPE\_SYSTEM
- GXEVENTTYPE\_WARNING

**category** User-defined message category. Do not use the range of values reserved for the Netscape Application Server systems, which is 0 to 65535, inclusive.

**Usage** Use Log() for displaying or storing simple messages or for debugging. The output can be directed to the console, to a text file, or to a database table. To direct output, use the Netscape Application Server Administrator. For more information, see the *Administration Guide*.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

```
hr = GetSession(0, "Catalog", NULL, &m_pSession);
if (hr != GXE_SUCCESS)
{
    Log("Could not get session; creating a new one");
    hr = CreateSession(GXSESSION_DISTRIB, 0, NULL,
        NULL, NULL, &m_pSession);
}
```

---

## LoginSession()

Logs an authorized user into a session with a secured application.

**Syntax 1** Use in most cases.

```
HRESULT LoginSession(
    LPSTR pName,
    LPSTR pPassword);
```

**Syntax 2** Contains several parameters that are placeholders for future functionality.

```
HRESULT LoginSession(
    LPSTR pDomain,
    DWORD dwMethod,
    DWORD dwFlags,
    LPSTR pName,
    LPBYTE pAuthData,
    ULONG nAuthData);
```

**name** The login user name.

**password** The user password.

**pDomain** Specify NULL.

**dwMethod** Specify 0.

**dwFlags** Specify 0.

**pName** The login user name.

**pAuthData** The user password.

**nAuthData** The size of the password.

**Usage** Call `LoginSession()` after creating a user session with `CreateSession()` or after retrieving a user session with `GetSession()`. `LoginSession()` checks the passed in login name and password against the user names and passwords stored in the Netscape Application Server (the administrator sets up and manages this information) and logs the user into the session if the login name and password are valid.

If login is successful, a security credential object is created and associated with the session. The server checks this security credential object each time it receives an `AppLogic` request, and verifies if the user has execute permission for the `AppLogic`.

Using `LoginSession()` in conjunction with `IsAuthorized()`, an application can ensure that only authorized users can execute certain `AppLogics` or take certain actions.

**Tip** The server administrator creates users and passwords and manages access to `AppLogics` and specified resources, such as sales or forecast reports. During the development and debugging phases, application developers can use the kreg tool to create users, groups, and ACLs in the GXR file. These tasks cannot be done programmatically.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```
STDMETHODIMP
ShopWelcome::Execute()
{
    char buffer[256];
    buffer[0] = '\0';

    // Verify user login
    if (m_pValIn->GetValString("NAME", buffer, sizeof(buffer)) != NOERROR
        ||
        m_pValIn->GetValString("PASSWORD", buffer, sizeof(buffer)) !=
        NOERROR)
    {
        Log("missing login NAME/PASSWORD");
        return EvalOutput("kivaapp/shop/please_login_again",
            (IGXTemplateData *) NULL,
            (IGXTemplateMap *) NULL, NULL, NULL);
    }
}
```

```

}
// If login is successful, create a session
IGXSession2 *mySess = NULL;
HRESULT hr;
hr = GetSession(0, NULL, NULL, &mySess);
if (hr != NOERROR ||
    !mySess)
{
    hr=CreateSession(0, 60000, NULL, NULL, NULL, &mySess);
    if (hr == NOERROR)
        Log("created session: success");
    else
        Log("created session: fail");
} else
    Log("got session");

// Now, look up user NAME/PASSWORD in database
// and see what role the user has. The database
// should have a user table which tracks all the
// users of the online shop application.
//
LPSTR role;
role = /* Database lookup here. */ "Shop_Customer";

// Call LoginSession() to set up the session with that
// role. Future requests to AppLogics in this session
// will now operate under the right role.
//
LoginSession(role, "");
SaveSession(NULL);

if (mySess)
    mySess->Release();

// Check to see if the current role is authorized
// against some of the more advanced operations, and
// choose the appropriate main menu page to return to
// the user.
DWORD auth_result = 0;

if ((IsAuthorized("Shop_Inventory", "READ", &auth_result) == NOERROR
    &&

```

```

        auth_result == (DWORD) GXACL_ALLOWED) ||
        (IsAuthorized("Shop_Daily_Forecast", "READ", &auth_result) ==
NOERROR &&
        auth_result == (DWORD) GXACL_ALLOWED) ||
        (IsAuthorized("Shop_Weekly_Forecast", "READ", &auth_result) ==
NOERROR &&
        auth_result == (DWORD) GXACL_ALLOWED))
return EvalOutput("kivaapp/shop/mainmenu_advanced",
        (IGXTemplateData *) NULL,
        (IGXTemplateMap *) NULL, NULL, NULL);
return EvalOutput("kivaapp/shop/mainmenu_regular",
        (IGXTemplateData *) NULL,
        (IGXTemplateMap *) NULL, NULL, NULL);
}

```

**Related Topics** IsAuthorized(),  
LogoutSession()

“Secure Sessions” in Chapter 9, “Writing Secure Applications” in *Programmer’s Guide*.

## LogoutSession()

Logs a user out of a session with a secured application.

**Syntax** HRESULT LogoutSession(  
        DWORD dwFlags);

**dwFlags** Specify 0.

**Usage** If the AppLogic called LoginSession() to log into a session with a secured application, call LogoutSession() when the user exits the application, or the secured portion of it.

**Rule** Call GetSession() before calling LogoutSession().

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** GetSession(),  
IsAuthorized(),  
LoginSession()

“Secure Sessions” in Chapter 9, “Writing Secure Applications” in *Programmer’s Guide*.

## NewRequest()

---

Calls another AppLogic from within the current AppLogic.

**Syntax 1** Passes in the specified IGXVallist of input parameters and result values. The location of the AppLogic execution depends on partitioning and load balancing criteria.

```
HRESULT NewRequest(
    LPSTR guid,
    IGXObject *vIn,
    IGXObject *vOut,
    DWORD flag);
```

**Syntax 2** Same as Syntax 1, but explicitly specifies the location of AppLogic execution.

```
HRESULT NewRequest(
    LPSTR guid,
    IGXObject *vIn,
    IGXObject *vOut,
    DWORD host,
    DWORD port,
    DWORD flag);
```

**guid** String GUID or name of the AppLogic to execute.

**vIn** IGXVallist object containing input parameters to pass to the called AppLogic.

**vOut** IGXVallist object containing result values of the called AppLogic.

**host** IP address of the Internet host of the Netscape Application Server where the AppLogic is to be executed. Specify 0 to execute the AppLogic locally.

**port** Internet port of the Netscape Application Server where the AppLogic is to be executed. Specify 0 to execute the AppLogic locally.

**flag** Specify zero.

**Usage** Use NewRequest( ) to call another AppLogic from within the current AppLogic. When it calls NewRequest( ), the AppLogic passes to the Netscape Application Server the GUID or name of the AppLogic to execute and, optionally, any input and output parameters.

Netscape Application Server constructs a request using the parameters specified and processes it like any other request, by instantiating the AppLogic and passing in its parameters. The results from the called AppLogic module are returned to the calling AppLogic.

The AppLogic that `NewRequest()` invokes can do one of the following tasks:

- Process application logic and return result values in the `vOut` parameter.
- Process application logic and return the resulting data form (such as a report) by streaming the output or by calling `Result()`.
- Process application logic and return result values in the `vOut` parameter as well as return the resulting data form (such as a report) by streaming the output or by calling `Result()`.

If the called AppLogic uses `EvalOutput()` to stream results, `EvalOutput()` returns HTML results by default. The current AppLogic can, however, specify that `EvalOutput()` return a non-HTML data stream by setting the `gx_client_type` key to "ocl" in the input `IGXValList` of `NewRequest()`. For example:

```
vallist.SetValString("gx_client_type", "ocl");
```

**Rule** The specified GUID string, input parameters, and output parameters must be valid for the specified AppLogic.

- Tips**
- The calling AppLogic can create new input and output `IGXValList`s so as to avoid changing its own input and output `IGXValList`s.
  - The AppLogic can call another AppLogic, passing its own input and output `IGXValList`s. In this case, the called AppLogic accesses the same stream destinations as the calling AppLogic.
  - Use `NewRequestAsync()` instead of `NewRequest()` to execute asynchronous request.
  - Called AppLogics might reside on different servers, depending on partitioning and load balancing configurations, might be written in a different language, or might have cached results. The calling AppLogic can be unaware or independent of these conditions.
  - Using `NewRequest()`, you can modularize parts of the application, build dynamic header/footer information and smart reporting templates, and hide complex or confidential business logic in secure submodules or even separate servers.

- Use `NewRequest()` judiciously. Each invoked AppLogic uses a certain amount of communications and server resources.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** IGXValList interface

“Passing Parameters to AppLogic From Code” in Chapter 4, “Writing Server-Side Application Code” in *Programmer’s Guide*.

---

## NewRequestAsync()

Calls another AppLogic from within the current AppLogic, and runs it asynchronously.

**Syntax 1** Passes in the specified IGXValList of input parameters and result values. The location of the AppLogic execution depends on partitioning and load balancing criteria.

```
HRESULT NewRequestAsync(
    LPSTR guid,
    IGXObject *vIn,
    IGXObject *vOut,
    DWORD flag,
    IGXOrder **ppOrder);
```

**Syntax 2** Same as Syntax 1, but explicitly specifies the location of AppLogic execution.

```
HRESULT NewRequestAsync(
    LPSTR guid,
    IGXObject *vIn,
    IGXObject *vOut,
    DWORD host,
    DWORD port,
    DWORD flag,
    IGXOrder **ppOrder);
```

**guid** String GUID or name of the AppLogic to execute.

**vIn** IGXValList object containing input parameters to pass to the called AppLogic.

**vOut** EvalOutput() object containing result values of the called AppLogic.

**host** IP address of the Internet host of the Netscape Application Server where the AppLogic is to be executed. Specify 0 to execute the AppLogic locally.

**port** Internet port of the Netscape Application Server where the AppLogic is to be executed. Specify 0 to execute the AppLogic locally.

**flag** Specify 0.

**ppOrder** Pointer to the returned IGXOrder object, which the AppLogic can use to obtain the status of the request. When the calling AppLogic is finished using the order object, call the `Release()` method to release the interface instance.

**Usage** Use `NewRequestAsync()` to call another AppLogic from within the current AppLogic, and run it asynchronously. Executing an AppLogic asynchronously is useful if the AppLogic performs a lengthy operation, or if the AppLogic acts as a monitor or remains persistent. For example, an asynchronous AppLogic may perform a lengthy database query to produce a complex result set that it sends an e-mail to a destination address. Another AppLogic module may run continuously and re-index HTML pages every 24 hours.

When an AppLogic calls `NewRequestAsync()`, it passes to the Netscape Application Server the GUID of the AppLogic module to execute and, optionally, any input and output parameters.

The Netscape Application Server constructs a request using the parameters specified and processes it like any other request, by instantiating the AppLogic and passing in its parameters. The results from the called AppLogic module are returned to the calling AppLogic.

The AppLogic that `NewRequestAsync()` invokes can do one of the following tasks:

- Process application logic and return result values in the `vOut` parameter.
- Process application logic and return the resulting data form (such as a report) by streaming the output or by calling `Result()`.
- Process application logic and return result values in the `vOut` parameter as well as return the resulting data form (such as a report) by streaming the output or by calling `Result()`.

**Rules**

- The specified AppLogic must be accessible to the Netscape Application Server.

- The specified GUID string, input parameters, and output parameters must be valid for the specified AppLogic module.
- Tips**
- To get the current status of the request, use the `GetState()` method in the returned `IGXOrder` object.
  - The calling AppLogic can use `GXWaitForOrder()` to wait for one or multiple asynchronous requests to return.
  - The calling AppLogic can create new input and output `IGXValLists` so as to avoid changing its own input and output `IGXValLists`.
  - The AppLogic can call another AppLogic, passing its own input and output `IGXValLists`. In this case, the called AppLogic accesses the same stream destinations as the calling AppLogic. To prevent conflicts in streaming, the calling AppLogic can use `GXWaitForOrder()` to wait until the called AppLogic is finished.
  - Using `NewRequestAsync()`, you can modularize parts of the application, build dynamic header/footer information and smart reporting templates, and hide complex or confidential business logic in secure submodules or even separate servers.
  - Use `NewRequestAsync()` judiciously. Each invoked AppLogic uses a certain amount of communications and server resources.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```

IGXOrder *pOrder;
ULONG      nOrder;
HRESULT hr, ReqResult;

if (NewRequestAsync(asyncGUIDStr, m_pValIn,
                   m_pValOut, 0, &pOrder) == GXE_SUCCESS)
{
    Log("Successfully invoked async AppLogic\n");

    // wait for async applogic to finish (max 100 seconds)
    hr = GXWaitForOrder(&pOrder, 1, &nOrder, m_pContext, 100);
    if (hr != NOERROR)
    {
        return Result("Error in executing async request:
order wait returned an error");
    }
}

```

```

        else
        {
            pOrder->GetState(NULL, &ReqResult, NULL);
            if (ReqResult != NOERROR)
                return Result("Error in executing async request");
        }
    }
    else
    {
        Log("Failed to invoke async AppLogic\n");
    }
}

```

**Related Topics** IGXOrder interface,  
IGXValList interface

“Passing Parameters to AppLogic From Code” in Chapter 4, “Writing Server-Side Application Code” in *Programmer’s Guide*.

---

## RemoveAllCachedResults()

Clears an AppLogic’s result cache.

**Syntax** HRESULT RemoveAllCachedResults(  
LPSTR guid);

**guid** The guid that identifies the AppLogic whose result cache to clear. Specify NULL to clear the current AppLogic’s cache.

**Usage** To free system resources, use RemoveAllCachedResults() to clear an AppLogic’s result cache when the results are no longer needed. This method clears the cache, but does not disable caching.

- Tips**
- To clear an AppLogic’s entire result cache and discontinue caching, use DeleteCache().
  - To clear a specific result from the cache, use RemoveCachedResult().

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```

HRESULT hr;
LPSTR guid;

guid = GXGetValListString(m_pValIn, "applogic");

```

```

hr = RemoveAllCachedResults(guid);

if (hr == GXE_SUCCESS)
    sprintf(msg, "Successfully cleared cached results");
else
    sprintf(msg, "Failed to clear cached results");

```

**Related Topics** DeleteCache(),  
RemoveCachedResult(),  
SetCacheCriteria()

“Caching AppLogic Results to Improve Performance” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

---

## RemoveCachedResult()

Clears a specific result from an AppLogic’s result cache.

**Syntax** HRESULT RemoveCachedResult(  
LPSTR guid  
IGXValList \*criteria);

**guid** The guid that identifies the AppLogic whose cached result to clear. Specify NULL to clear the current AppLogic’s cached result.

**criteria** An IGXValList object that contains the criteria for selecting the result to remove. In the IGXValList object, set a specific value that matches the cache criteria passed to SetCacheCriteria(). For example, if the cache criteria passed to SetCacheCriteria() was "Salary=40000-60000", you can remove results where salary is 50000 by setting in the IGXValList object a "Salary" key to a value of "50000".

**Usage** Use RemoveCachedResult() to clear a specific result from an AppLogic’s cache when the result is no longer needed.

- Tips**
- To clear an AppLogic’s entire result cache and discontinue caching, use DeleteCache().
  - To clear an AppLogic’s entire result cache, but continue caching, use RemoveAllCachedResults().

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** HRESULT hr;

```

LPSTR guid;
guid = GXGetValListString(m_pValIn, "applogic");

resultList = GXCreateValList();
hr = resultList->SetValString("Salary", "50000");

hr = RemovedCachedResult(guid, resultList);

if (hr == GXE_SUCCESS)
    sprintf(msg, "Successfully deleted specified result");
else
    sprintf(msg, "Failed to delete specified result");

```

**Related Topics** DeleteCache(),  
RemoveAllCachedResults(),  
SetCacheCriteria()

“Caching AppLogic Results to Improve Performance” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

---

## Result()

Specifies the return value of an AppLogic.

**Syntax** HRESULT Result(  
LPSTR result);

**result** Text representing the result value of the current AppLogic.

**Usage** Use Result() in conjunction with the Execute() method to define a return value for an AppLogic. In general, use Result() in an AppLogic that services HTTP or HTML requests and returns a simple HTML string that does not require streaming.

In the Execute() method, the AppLogic can call Result() in conjunction with the return statement to send data results directly back to the entity that called the AppLogic.

**Rule** An AppLogic can stream results using StreamResultHeader() or StreamResult(). If the AppLogic streams results, call Result() only *after* finishing streaming.

**Tips** • An AppLogic can cache results for reuse using SetCacheCriteria().

- Alternatively, the AppLogic can return results using a template. The AppLogic can call EvalOutput( ) to merge a dynamically created result set from a hierarchical query with a template to produce formatted results. The result from EvalOutput( ) is streamed automatically.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

```
if (EvalTemplate("GXApp/COnlineBank/templates/NewCust.html",
(IGXHierQuery*)NULL, NULL, NULL, NULL)!=GXE_SUCCESS)
    Result("<HTML><BODY>Unable to evaluate template.</BODY></HTML>");
return GXE_SUCCESS;
```

**Related Topics** Execute( ),  
StreamResult( ),  
StreamResultHeader( )

“Returning HTML Results” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

## SaveSession( )

---

Saves changes to a session.

**Syntax** HRESULT SaveSession(  
IGXSessionIDGen \*pIDGen);

**pIDGen** The session ID generation object used to generate session IDs. Specify NULL to use the default IGXSessionIDGen object, or specify a custom session ID generation object.

**Usage** Use SaveSession( ) to ensure that changes are saved in the distributed state storage area, which stores the session information for subsequent use if any other AppLogics are invoked within the same session.

The SaveSession( ) method uses a cookie—if the Web browser supports cookies—to pass the session ID back and forth between the Web browser and Netscape Application Server. It transfers only the session ID, not the session information itself, to provide better information security.

HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

Because SaveSession( ) uses StreamResultHeader( ) to register the cookie, be sure to call SaveSession( ) before calling StreamResult( ), EvalTemplate( ), or any other HTTP body streaming methods.

- Tip**
- The AppLogic needs to call the SaveSession() method in the GXAppLogic class at least once to set a cookie. The SaveSession() method in the IGXSession2 interface only saves data to the distributed state store, whereas SaveSession() in the GXAppLogic class saves data to the distributed state store *and* sets a cookie.
  - The AppLogic should call SaveSession() to save changes after updating session data.
  - To improve performance, keep smaller amounts of information in the session.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateSession(),  
GetSession(),  
GXSession2 class,  
IGXSession2 interface

“Starting a Session” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

---

## SetCacheCriteria()

Stores AppLogic results, such as HTML, data values, and streamed data, in a result cache.

**Syntax**

```
HRESULT SetCacheCriteria(
    ULONG timeout,
    ULONG cachesize,
    LPSTR criteria)
```

**timeout** Number of seconds the AppLogic result remains in the result cache after the last access. To clear the result cache after a specified time from its creation, use the GXREPOSIT\_TIMEOUT\_CREATE flag, as shown in the following example: SetCacheCriteria(GXREPOSIT\_TIMEOUT\_CREATE | 300, ...). In this example, the cache is cleared 300 seconds after it is created. Set timeout to zero to clear the result cache and disable caching for this AppLogic.

**cachesize** Maximum number of results to be cached for the AppLogic at any time. The result cache stores distinct AppLogic output up to the cachesize limit. If the AppLogic generates another output to cache, the least accessed member of the cache is dropped. Setting cachesize to zero clears the result cache and disables caching for this AppLogic.

**criteria** Criteria expression containing a string of comma-delimited descriptors. Each descriptor defines a match with one of the input parameters to the AppLogic. Use the following syntax:

| Syntax                 | Description  |
|------------------------|--|
| <code>arg</code>       | Test succeeds for any value of <i>arg</i> in the input parameter list. For example:<br><code>SetCacheCriteria(3600,1,"EmployeeCode");</code>   |
| <code>arg=v</code>     | Test whether <i>arg</i> matches <i>v</i> (a string or numeric expression). For example:<br><code>"stock=NSCP"</code><br>Assign an asterisk (*) to the argument to cache a new set of results every time the AppLogic module runs with a different value. For example:<br><code>SetCacheCriteria(3600,1,"EmployeeCode=*");</code> |
| <code>arg=v1 v2</code> | Test whether <i>arg</i> matches any values in the list ( <i>v1</i> , <i>v2</i> , and so on). For example:<br><code>"dept=sales marketing support"</code>   |
| <code>arg=n1-n2</code> | Test whether <i>arg</i> is a number that falls within the range. For example:<br><code>"salary=40000-60000"</code>   |

**Usage** Use **ppSession** A pointer to the created or retrieved IGXSession2 object. When the AppLogic is finished using the object, call the `Release()` method to release the interface instance.

`SetCacheCriteria()` to specify caching for the results from an AppLogic. An AppLogic can cache any type of result. Caching improves performance for time-consuming operations such as queries and report generation.

When caching is enabled for an AppLogic, the Netscape Application Server stores its input parameter values and its results in the cache so that, if the AppLogic is called again with the same parameters (matching the cache criteria), the Netscape Application Server retrieves its results directly from the

cache instead of running the AppLogic again. If the AppLogic is called with different parameters, the Netscape Application Server runs the AppLogic again and saves its result in the cache as well.

Each AppLogic has only one cache but it can contain multiple sets of results if the AppLogic was run multiple times with different parameters for each call.

- Tips**
- Do not use caching if real-time results are needed. For example, to ensure current data, caching is not recommended for query operations on highly volatile data.
  - Use `SkipCache()` to bypass result caching if an error occurred during AppLogic execution.
  - Use `IsCached()` to test whether caching is currently enabled. Calling `IsCached()` is important because it prevents calling `SetCacheCriteria()` too many times.
  - To change the caching criteria for AppLogic, call again, this time specifying different caching criteria. Each subsequent call supersedes the previous call, discarding the current contents of the result cache, and its criteria remain in effect until the next `SetCacheCriteria()` call, if applicable.
  - To stop caching results, call `DeleteCache()`. A subsequent call to `SetCacheCriteria()` can reactivate caching.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Example 1**

```
// Verify AppLogic caching before setting cache criteria
if(!IsCached()) {
    Log ("Set criteria to save output from 3 deptcodes");
    if(SetCacheCriteria(60, 3, "deptcode")!=GXE_SUCCESS)
        Log("Could not set criteria");
    else
        Log("Succeeded in setting criteria");
}
else
    Log("Not setting Criteria");
```

**Example 2**

```
// Cache multiple results for up to 100 values of Department
SetCacheCriteria(3600,100,"Department");
```

**Example 3**

```
// Cache single result for given matching value of Department
SetCacheCriteria(3600,1,"Department=Operations");
```

**Example 4** `// Cache multiple results for two matching values of dept  
SetCacheCriteria(3600,2,"Department=Research | Engineering");`

**Example 5** `// Cache one result for salary in a range  
SetCacheCriteria(3600,1,"Salary=40000-60000");`

**Example 6** `// Cache two results for several parameters  
SetCacheCriteria(3600,2,  
"Department=Sales, Salary=40000-60000");`

**Related Topics** `DeleteCache( ),  
RemoveAllCachedResults( ),  
RemoveCachedResult( ),  
IsCached( ),  
SkipCache( )`

“Caching AppLogic Results to Improve Performance” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

## SetSessionVisibility()

---

Sets the session visibility.

**Syntax** `HRESULT SetSessionVisibility(  
LPSTR domain,  
LPSTR path,  
BOOL isSecure)`

**domain** The domain in which the session is visible.

**path** The path to which this session must be visible.

**isSecure** If TRUE, the session is visible only to secure servers (HTTPS).

**Usage** Because of the way cookies are used to identify sessions, NAS sessions are, by default, accessible only within the same URL name space where they were created. As a result, if you call only the `SaveSession()` method, then your session is not visible to any other domain or URL.

However, if you call `SetSessionVisibility()` before calling `SaveSession()`, you can control the visibility of the session. The `SetSessionVisibility()` method internally controls the attributes of the cookie used in transmitting the session ID.

You must be part of the domain to set the domain attribute. For example, if the domain is set to netscape.com, then the session is visible to foo.netscape.com, bar.netscape.com, and so on. Domains must have at least two periods (.) in them. For example, .net is an invalid domain attribute.

By default, the session is visible only to the URL that created the session cookie. Use the path parameter to specify different URLs that will be visible. For example, the path /phoenix would match "/phoenixbird" and "/phoenix/bird.html". To make the entire server root visible, specify a path of "/", the most general value possible.

Both the domain and path parameters are null-terminated character strings. They are not modified within the SetSessionVisibility() method.

**Rule** For the session visibility to take effect, you must invoke SetSessionVisibility() before a call to SaveSession(). The SaveSession() method uses the visibility attributes set from SetSessionVisibility().

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** SaveSession()

## SetVariable()

---

Sets a value that is passed to later AppLogic requests that are called by the same client. If the client is a browser, cookies are used to transfer variable values.

**Syntax 1** HRESULT SetVariable(  
     LPSTR name,  
     LPSTR value);

**Syntax 2** HRESULT SetVariable(  
     LPSTR name,  
     LPSTR value,  
     ULONG timeout,  
     LPSTR urlPath,  
     LPSTR urlDomain,  
     BOOL secure);

**name** The name of the value to record for this browser session. The value will appear on any future AppLogic's input IGXValList under this name.

**value** The string value to record.

**timeout** Number of seconds before the cookie expires. Applies to HTTP clients only.

**urlPath** The subset of URLs in a domain for which the cookie is valid. Applies to HTTP clients only.

**urlDomain** The domain for which the cookie is valid. Applies to HTTP clients only.

**secure** If a cookie is marked secure, it will be sent only if the communications channel with the host is a secure one. Currently, this means that secure cookies will be sent only to HTTPS (HTTP over SSL) servers. Applies to HTTP clients only.

**Usage** Use `SetVariable()` to store information specific to a client that you want to pass to other AppLogics invoked by the same client. The values set with `SetVariable()` are passed to the input `IGXValList (m_pValIn)` of the called AppLogics.

In the case of an HTTP client, `SetVariable()` streams the variable out in an HTTP header. The HTTP header registers a cookie, which is the mechanism used to pass data back and forth between the browser and the Netscape Application Server.

**Rule** Because `SetVariable()` streams information in an HTTP header, call it before calling any HTTP body streaming methods, such as `StreamResult()`, `EvalOutput()`, and `EvalTemplate()`.

**Tip** If your application requires more security, you should use Netscape Application Server's session mechanism instead of cookies to maintain session information. With a Netscape Application Server session, data is stored on the server and only a session ID is passed between the client and the server. For more information about the session mechanism, see `IGXSession2` interface.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** "Using Cookies" in Chapter 3, "Application Development Techniques," in *Programmer's Guide*.

---

## SkipCache()

Skips result caching for the current AppLogic execution.

**Syntax** `HRESULT SkipCache()`

**Usage** Use `SkipCache()` to prevent results from the current request from being saved in the results cache if an error occurs during AppLogic execution.

**Rule** For `SkipCache()` to have any effect, you must first enable caching by calling `SetCacheCriteria()`.

#### Return Value

**Related Topics** `DeleteCache()`,  
`RemoveAllCachedResults()`,  
`RemoveCachedResult()`,  
`IsCached()`,  
`SetCacheCriteria()`,  
IGXValList interface

“Caching AppLogic Results to Improve Performance” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

## StreamResult()

---

Streams results as a string.

**Syntax**

```
HRESULT StreamResult(
    LPSTR res);
```

**res** The body data to stream. If returning HTML body data, you can use HTML formatting following HTTP body conventions. See your HTTP documentation for more information.

**Usage** Use `StreamResult()` to stream data as soon as it is available. With streaming, an AppLogic can make the first portion of the data available for use immediately, even if the remainder of the stream has not yet been processed. This is especially useful with large volumes of data, such as a query that takes a while for the database server to process completely. An AppLogic can process and display those rows in the result set that have been returned. Without streaming, AppLogic must prepare the entire result first before returning any data.

The `StreamResult()` method is typically used to stream HTTP body content. Before calling `StreamResult()`, the AppLogic must call `StreamResultHeader()` to return the HTTP header data first. The HTTP protocol separates data streams into header and body data, and specifies that the header data and body data are returned in that order. For details about HTTP header and body data, see your HTTP documentation.

- Tips**
- Alternatively, use `EvalTemplate()` to stream HTTP body output. It merges data with an HTML template. As soon as a segment of the output page is finished, `EvalTemplate()` streams it out to the Web browser.
  - An AppLogic can call `StreamResultHeader()` and `StreamResult()` repeatedly to stream more results.
  - To stream binary data, use `StreamResultBinary()`.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `StreamResultBinary()`,  
`StreamResultHeader()`

“Streaming Results” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

---

## StreamResultBinary()

Streams binary data, such as a GIF file.

**Syntax**

```
HRESULT StreamResultBinary(
    LPBYTE buf,
    ULONG offset,
    ULONG length);
```

**buf** The array from which binary data is streamed.

**offset** Index in the array. The starting position in the array to start streaming binary body data.

**length** Number of bytes to stream from the array, starting at the specified offset position.

**Usage** Use `StreamResultBinary()` to stream binary data as soon as it is available. With streaming, an AppLogic can make the first portion of the data available for use immediately, even if the remainder of the stream has not yet been processed. This is especially useful with large volumes of data, such as a query that takes a while for the database server to process completely. An AppLogic can process and display those rows in the result set that have been returned. Without streaming, AppLogic must prepare the entire result first before returning any data.

The `StreamResultBinary()` method is used to stream HTTP body data of binary type, such as an image (GIF) file. Before calling `StreamResultBinary()`, the `AppLogic` should call `StreamResultHeader()` to return the HTTP header data first. The HTTP protocol separates data streams into header and body data, and specifies that the header data and body data are returned in that order. For details about HTTP header and body data, see your HTTP documentation.

- Tips**
- Alternatively, use `EvalTemplate()` to stream HTTP body output. It merges data with an HTML template. As soon as a segment of the output page is finished, `EvalTemplate()` streams it out to the waiting Web browser.
  - To stream non-binary data, use `StreamResult()`.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `StreamResult()`,  
`StreamResultHeader()`

“Streaming Results” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

## StreamResultHeader()

---

Streams header data.

**Syntax** HRESULT StreamResultHeader(  
LPSTR hdr)

**hdr** The header data to stream. If returning HTTP header data, use the HTTP header conventions, such as the following:

```
"Content-Type: text/html"  
"Location: <redirect url>"
```

See your HTTP documentation for more information.

**Usage** Use `StreamResultHeader()` to return header data before streaming body data. With streaming, an `AppLogic` can make the first portion of the data available for use immediately, even if the remainder of the stream has not yet been processed. This is especially useful with large volumes of data, such as a query that takes a while for the database server to process completely. An `AppLogic` can process and display those rows in the result set that have been returned. Without streaming, `AppLogic` must prepare the entire result first before returning any data.

The `StreamResultHeader()` method is typically used in conjunction with `StreamResult()` to stream HTTP data. Before calling `StreamResult()`, the `AppLogic` should call `StreamResultHeader()` to return the HTTP header data first. The HTTP protocol separates data streams into header and body data, and specifies that the header data and body data are returned in that order. For details about HTTP header and body data, see your HTTP documentation.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `StreamResult()`,  
`StreamResultBinary()`

“Streaming Results” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.



## GXSession2 class

The GXSession2 class is designed to help you implement a custom session class if your application requires additional session functionality. To create a custom session class, subclass the GXSession2 class, then define new methods. Your subclass can, for example, define accessor methods to read and write information specific to your session. An online shopping application, for example, might require specialized methods, such as AddItemToCart(), to track shopping items per user session.

When you subclass the GXSession2 class, you must do the following:

- Override the CreateSession() and GetSession() methods in the GXAppLogic class. In these methods, you can invoke the base GXAppLogic methods to obtain an IGXSession2 object, and construct your own session class by passing in this object, as shown in the following example:

```
HRESULT hr;
IGXSession2 *pSession = Null;
hr = GXAgent::GetSession(dwFlags, pAppName, pIDGen, &pSession);

if (hr == GXE_SUCCESS)
{
    m_pSession = new MySession(pSession);
    if (!m_pSession)
        hr = GXE_ALLOC_FAILED;
    pSession->Release();
}
return hr;
```

- Pass in the IGXSession2 interface in the subclass constructor, as shown in the following example:

```
class MySession : public GXSession2
{
public:
    MySession(IGXSession2 *pSess);
```

Because the GXSession2 class delegates the implementation of methods in the IGXSession2 interface to the object passed to its constructor, you don't have to implement every method of that interface in your subclass. You need only define the methods you want to add.

## Include File

gxapplogic.h

## Related Topics

IGXSession2 interface

“Using Custom Sessions” in Chapter 8, “Managing Session and State Information” in *Programmer’s Guide*.

# GXTemplateDataBasic class

The GXTemplateDataBasic class represents a memory-based, hierarchical source of data used for HTML template processing. It implements the IGXTemplateData interface, and provides methods for creating and managing this hierarchical data.

The most common sources of data used for template processing are result sets obtained from queries on supported relational database management systems. However, an AppLogic might need to obtain data from non-RDBMS sources. For example, an AppLogic might display a list of numbers generated from a formula, or it might display a list of processors available on the server machine and their CPU loads. To display such information, the AppLogic can create an instance of the GXTemplateDataBasic class, populate that instance with rows of hierarchical data, and then pass the GXTemplateDataBasic object to the Template Engine for processing by calling EvalTemplate() or EvalOutput() in the GXAppLogic class.

Alternatively, to provide application-specific special processing and to hook into the template generation process, AppLogic can subclass the GXTemplateDataBasic class and override the member methods in the IGXTemplateData interface.

An AppLogic can create a flat or hierarchical data structure.

- For a flat data structure, create the data structure using GXTemplateDataBasic(), then call RowAppend() for each row of data to be added, specifying the column name and data in each row.
- For a hierarchical data structure, proceed in the following sequence:
  1. Create the parent GXTemplateDataBasic() instance.
  2. Create the child GXTemplateDataBasic() instance.
  3. Add one or more rows to the child data structure using RowAppend() on the child instance.
  4. Define the start of a new parent row by using RowAppend() on the parent instance.
  5. Join the child data structure to the parent data structure using the parent's GroupAppend().

- Repeat steps 2 through 5 for each subsequent group of data.

The number of nesting levels is limited only by system resources. One parent row can contain many joined child instances, in which case the AppLogic calls the parent's GroupAppend() more than once after calling the parent's RowAppend().

## Include File

gxtmplbasic.h

## Methods

| Method                | Description   |
|-----------------------|---|
| GroupAppend()         | Links the specified child group to the current parent group.            |
| RowAppend()           | Appends a new row of data to the current template data object or group. |
| GXTemplateDataBasic() | Creates an empty template data object with the specified name.          |

## Implements

IGXTemplateData interface

## Related Topics

EvalTemplate() and EvalOutput() in the GXAppLogic class

IGXTemplateData interface

“Constructing a Hierarchical Result Set with GXTemplateDataBasic” in Chapter 7, “Working with Templates” in *Programmer's Guide*.

## GroupAppend()

Links the specified child group to the current parent group.

**Syntax** HRESULT GroupAppend(  
GXTemplateDataBasic \*pChild);

**pChild** Pointer to the child GXTemplateDataBasic object to link. When AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use GroupAppend() to define the hierarchical relationship from a parent row to child GXTemplateDataBasic objects.

- Rules**
- Call GroupAppend() only after calling RowAppend(). The child instance is associated with the last row from the last call to RowAppend() on the parent.
  - The AppLogic must first create the parent and child objects using new GXTemplateDataBasic(), then populate the child object with rows of data using RowAppend().

**Tip** Use GroupAppend() for hierarchical data objects only.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** EvalTemplate() and EvalOutput() in the GXAppLogic class

IGXTemplateData interface

“Constructing a Hierarchical Result Set with GXTemplateDataBasic” in Chapter 7, “Working with Templates” in *Programmer’s Guide*.

## RowAppend()

---

Appends a new row of data to the current template data object or group.

**Syntax** HRESULT RowAppend(  
LPSTR szRow);

**szRow** String containing a series of column name and value pairs, separated by semi-colons, using the following format:

```
"column1=value1[:column2=value2[...]]"
```

The columns must be identical for each RowAppend() call within the same GXTemplateDataBasic object.

**Usage** Use RowAppend() to populate the template data object with rows of data.

**Rule** AppLogic must first create the template data object using GXTemplateDataBasic( ).

**Tip** Add rows in the sequence in which you want the Template Engine to process them. The template data object is processed in physical order only. AppLogic can only append rows to the data object. It cannot subsequently insert, delete, or sort records in the template data object.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** EvalTemplate( ) and EvalOutput( ) in the GXAppLogic class

IGXTemplateData interface

“Constructing a Hierarchical Result Set with GXTemplateDataBasic” in Chapter 7, “Working with Templates” in *Programmer’s Guide*.

---

## GXTemplateDataBasic( )

Creates an empty template data object with the specified name.

**Syntax** HRESULT GXTemplateDataBasic(  
LPSTR pName);

**pName** Name of the parent or child data object referred to in the template.

**Usage** Use new GXTemplateDataBasic( ) to create parent and child data objects.

**Rule** The specified data object name must be unique within this template data object.

- Tips**
- Use RowAppend( ) to populate this data object with rows of data.
  - For hierarchical template data objects, use GroupAppend( ) to define the hierarchy among GXTemplateDataBasic objects.
  - The specified data object name must be unique within the hierarchical result set.
  - Create parent and child groups in the sequence in which you want the Template Engine to process them. The template data object is processed in physical order only.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** EvalTemplate( ) and EvalOutput( ) in the GXAppLogic class

IGXTemplateData interface

“Constructing a Hierarchical Result Set with GXTemplateDataBasic” in Chapter 7, “Working with Templates” in *Programmer’s Guide*.



# GXTemplateMapBasic class

The GXTemplateMapBasic class represents an object that contains one or more mappings between fields in an HTML template and the data used to replace those fields. It provides a method for defining these mappings before processing the template using EvalTemplate() or EvalOutput() in the GXAppLogic class.

Fields in the HTML template are defined using special GX markup tags. The data, which the Template Engine uses to replace those fields dynamically at runtime, can come from any of the following sources: a calculated value, a column in a result set, a field in an IGXTemplateData template data object, or a field from a map object.

Before calling EvalTemplate() or EvalOutput() in the GXAppLogic class, an AppLogic uses the Put() method in the GXTemplateMapBasic class to link the field name in the GX markup tag with a precomputed value or a named column or field in the data source. After defining the mappings, the AppLogic passes the populated IGXTemplateMapBasic object as the map parameter in EvalTemplate() or EvalOutput(). The Template Engine uses these mappings during template processing to dynamically transfer data values from the data source to the HTML output report.

Mapping allows the AppLogic to use the same template for multiple data sources with different column names, for a data source whose schema changes over time, or for memory-based data sources defined using a TemplateDataBasic object.

While it is not necessary, you may derive a class from GXTemplateMapBasic, by writing a class declaration such as the following:

```
class MyTemplateMapBasic : public TemplateMapBasic
```

To provide application-specific special processing, the AppLogic can subclass GXTemplateMapBasic and override its Get() method in the IGXTemplateMap interface to hook into the Template Engine generation process. For example, AppLogic can intercept and filter data from a database before the Template Engine processes it.

## Include File

gxtmplbasic.h

## Methods

|       |                                     |
|-------|-------------------------------------|
| Put() | Adds a mapping to the template map. |
|-------|-------------------------------------|

## Implements

IGXTemplateMap interface

## Related Topics

EvalTemplate() and EvalOutput() in the GXAppLogic class,  
GXTemplateDataBasic class,  
IGXTemplateData interface,  
IGXTemplateMap interface

“TagAttributes” in Chapter 7, “Working with Templates” in *Programmer’s Guide*.

## Put()

Maps the value assigned to the id attribute in the HTML template to another value.

**Syntax**

```
HRESULT Put(
    LPSTR szKey,
    IGXBuffer *pBuff);
```

**szKey** In the GX markup tag in the HTML template, the name of the field, or placeholder, assigned to the id attribute. Must be an identical match (case-sensitive).

**pBuff** Pointer to the IGXBuffer object that contains the expression to substitute for the specified template field name, such as:

- Calculated value, such as a number or date.
- Name of the column in the hierarchical result set that the Template Engine uses to process the template. In your template, the column name must begin with a “\$” character.

- Name of a field in the GXTemplateDataBasic object that the Template Engine uses to process the template. In your template, the field name must begin with a “\$” character.

**Usage** Use Put() to add template field/data source pairs to the template map before calling EvalTemplate() or EvalOutput() in the GXAppLogic class.

**Rule** Use the GXCreateBuffer() function to create the IGXBuffer object. Thereafter, use methods in the IGXBuffer interface to manipulate the memory block.

**Tip** The AppLogic can place the Put() method call inside a loop to construct the field map iteratively. For example, the AppLogic could use this technique to populate a map from a file, line by line.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
STDMETHODIMP
OBBaseAppLogic::HandleOBValidationError(LPSTR pMessage)
{
    HRESULT hr=GXE_SUCCESS;
    GXTemplateMapBasic map;
    IGXBuffer *pBuff=GXCreateBuffer();
    if(pBuff)
    {
        pBuff->Alloc(strlen(pMessage)+1);
        strcpy((char*)pBuff->GetAddress(), pMessage);
        map.Put("OUTPUTMESSAGE", pBuff);
        // Send it to the template
        hr=EvalTemplate("GXApp/COOnlineBank/templates/
            ValidationError.html", (IGXTemplateData*) NULL, &map, NULL,
            NULL);
        pBuff->Release();
    }
    return hr;
}
```

**Related Topics** EvalTemplate() and EvalOutput() in the GXAppLogic class, GXTemplateDataBasic class, IGXTemplateData interface, IGXTemplateMap interface

“TagAttributes” in Chapter 7, “Working with Templates” in *Programmer’s Guide*.



# Interfaces

This chapter provides reference material on the interfaces in the Netscape Application Server Foundation Class Library.

The following interfaces are included in this chapter:

|                                    |                            |
|------------------------------------|----------------------------|
| IGXAppEvent interface (deprecated) | IGXPreparedQuery interface |
| IGXAppEventMgr interface           | IGXQuery interface         |
| IGXAppEventObj interface           | IGXResultSet interface     |
| IGXBuffer interface                | IGXSequence interface      |
| IGXCallableStmt interface          | IGXSequenceMgr interface   |
| IGXColumn interface                | IGXSession2 interface      |
| IGXDataConn interface              | IGXSessionIDGen interface  |
| IGXDataConnSet interface           | IGXState2 interface        |
| IGXEnumObject interface            | IGXStreamBuffer interface  |
| IGXError interface                 | IGXTable interface         |
| IGXHierQuery interface             | IGXTemplateData interface  |
| IGXHierResultSet interface         | IGXTemplateMap interface   |
| IGXLock interface                  | IGXTile interface          |
| IGXMailBox interface               | IGXTrans interface         |
| IGXObject interface                | IGXValList interface       |
| IGXOrder interface                 |                            |



# IGXAppEvent interface (*deprecated*)

IGXAppEvent is deprecated and is provided for backward compatibility only. New applications should use the NAS API's two replacement interfaces: IGXAppEventMgr and IGXAppEventObj.

The IGXAppEvent interface represents the defined events an application supports. An AppLogic can define events that are triggered at a specified time or times or when triggered explicitly.

Currently, an AppLogic can execute two actions when an event is triggered:

- Run a specified AppLogic
- Send an email

Events are stored persistently in the Netscape Application Server, and are removed only when your application explicitly deletes them. They are typically used to schedule routine administrative tasks, such as making back-ups or getting statistics.

The IGXAppEvent interface defines methods for registering, triggering, enabling, disabling and deleting events. To create an instance of the IGXAppEvent interface, use the GetAppEvent() method in the GXAppLogic class.

## Include File

gxiappevent.h

## Methods

| Method         | Description  |
|----------------|--|
| DeleteEvent()  | Removes a registered event from Netscape Application Server. |
| DisableEvent() | Disables a registered event.                                 |
| EnableEvent()  | Enables a registered event.                                  |
| EnumEvents()   | Enumerates through the list of registered events.            |
| QueryEvent()   | Returns the properties of a registered event.                |

| Method          | Description                                      |
|-----------------|--|
| RegisterEvent() | Registers a named event for use in applications. |
| SetEvent()      | Triggers a registered event.                     |

## Example

The following example shows AppLogic code that registers two application events:

- The first event runs the RepGenAgent AppLogic at 5 AM everyday
- The second event emails a report generated by RepGenAgent at 6 AM everyday

```

STDMETHODIMP
ReportAgent::Execute()
{
    HRESULT hr = NOERROR;
    IGXAppEvent *pAppEvent = NULL;
    IGXValList *pValList = NULL;
    LPSTR pReportEvName = "ReportEvent";
    LPSTR pRepGenEvName = "RepGenEvent";

    // Get a reference to the AppEvent Manager
    hr = GetAppEvent(&pAppEvent);
    if ((hr != NOERROR) || (pAppEvent == NULL))
        return StreamResult("AppEvent not found!<br>");

    // Create a vallist to pass information
    // for appevent registration of the first event
    pValList = GXCreateValList();
    if (pValList == NULL) {
        pAppEvent->Release();
        return Result(m_pValOut, "Can't create ValList<br>");
    }
    // Add the appevent name to the vallist
    GXSetValListString(pValList, GX_AE_RE_KEY_NAME, pRepGenEvName);

    // Set the appevent state to be enabled
    GXSetValListInt(pValList, GX_AE_RE_KEY_STATE,
GX_AE_RE_EVENT_ENABLED);

```

```

// Set the appevent time to be 05:00:00 hrs everyday
GXSetValListString(pValList, GX_AE_RE_KEY_TIME, "5:0:0 */*/*");

// Set the appevent action to run the RepGenAgent applogic
GXSetValListString(pValList, GX_AE_RE_KEY_NREQ, "GUIDGX-{630CB09B-
1A1D-1315-AD23-0800207B918B}");

// Register the appevent
hr = pAppEvent->RegisterEvent(pRepGenEvName, pValList);
pValList->Release();

// Create a vallist to pass information
// for appevent registration of the second event
//
pValList = GXCreateValList();
if (pValList == NULL) {
    pAppEvent->Release();
    return Result(m_pValOut, "Can't create ValList<br>");
}
// Add the appevent name to the vallist
GXSetValListString(pValList, GX_AE_RE_KEY_NAME, pReportEvName);

// Set the appevent state to be enabled
GXSetValListInt(pValList, GX_AE_RE_KEY_STATE,
GX_AE_RE_EVENT_ENABLED);

// Set the appevent time to be 06:00:00 hrs everyday
GXSetValListString(pValList, GX_AE_RE_KEY_TIME, "6:0:0 */*/*");

// Set the appevent action to send
// e-mail to report@acme.com
GXSetValListString(pValList, GX_AE_RE_KEY_MTO, "report@acme.com");

// The content of the e-mail is in /tmp/report-file
GXSetValListString(pValList, GX_AE_RE_KEY_MFILE, "/tmp/report-
file");

// The e-mail host running the SMTP server is mailsvr
GXSetValListString(pValList, GX_AE_RE_KEY_MHOST, "mailsvr");

// The sender's e-mail address is admin@acme.com

```

```
GXSetValListString(pValList, GX_AE_RE_KEY_SADDR, "admin@acme.com");

// Register the appevent
hr = pAppEvent->RegisterEvent(pReportEvName, pValList);

// Clean-up resources and return
//
pValList->Release();
pAppEvent->Release();
return StreamResult("Successfully Registered RepGenEvent and
ReportEvent<br>");
}
```

## Related Topics

GetAppEvent() method in the GXAppLogic class

IGXAppEventMgr interface

IGXAppEventObj interface

IGXValList interface

“Using Events” in Chapter 3, “Application Development Techniques” in *Programmer’s Guide*.

---

## DeleteEvent()

Removes a registered event from the Netscape Application Server.

**Syntax** HRESULT DeleteEvent(  
LPSTR pEventName);

**pEventName** The name of the registered event to delete.

**Usage** Use DeleteEvent() to remove an event that is no longer required. To temporarily stop a event from being triggered, use DisableEvent().

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** DisableEvent()

RegisterEvent()

## DisableEvent()

---

Disables a registered event.

**Syntax** `HRESULT DisableEvent(  
LPSTR pEventName);`

**pEventName** The name of the registered event to disable.

**Usage** Use `DisableEvent()` to temporarily stop an event from being triggered. The event is disabled until it is enabled with `EnableEvent()`. To remove an event from the Netscape Application Server permanently, use `DeleteEvent()`.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `DeleteEvent()`

`EnableEvent()`

`RegisterEvent()`

## EnableEvent()

---

Enables a registered event.

**Syntax** `HRESULT EnableEvent(  
LPSTR pEventName);`

**pEventName** The name of the registered event to enable.

**Usage** Use `EnableEvent()` to prepare a specified event for activation. Call `EnableEvent()` after you register an event with `RegisterEvent()`, or to enable a trigger that was disabled with `DisableEvent()`.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `DisableEvent()`

`RegisterEvent()`

## EnumEvents()

---

Returns the list of registered events.

**Syntax** HRESULT EnumEvent(  
           IGXEnumObject \*\*ppEvents);

**ppEvent** Pointer to the IGXEnumObject object that contains the list of registered events. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use EnumEvents() to get information on all the registered events. The IGXEnumObject object returned by EnumEvents() contains a set of IGXVallist objects, one per event. Each IGXVallist contains the properties assigned to the event when it was registered with RegisterEvent().

**Tip** Use the methods in the IGXEnumObject interface to iterate through the contents of the returned IGXEnumObject object.

**Example** The following AppLogic code shows how to use EnumEvents() to get information on all the registered events and save it to a report:

```

HRESULT          hr = NOERROR;
IGXEnumObject   *pEObjs = NULL;
IGXAppEvent     *pAppEvent = NULL;
IGXVallist      *pVallist = NULL;
CHAR            pBuf[128];
ULONG           ulCount = 0;
FILE            *fp;

// Open /tmp/report-file for writing the report
fp = fopen("/tmp/report-file", "w");

// Get a reference to the AppEvent Manager
hr = GetAppEvent(&pAppEvent);

// Get the Enumeration object for all registered appevents
hr = pAppEvent->EnumEvents(&pEObjs);

// Retrieve the count of registered appevents
hr = pEObjs->EnumCount(&ulCount);

    fprintf(fp, "Number of Registered Events: %d\n", ulCount);

// Reset the next enumeration object to be the first instance
hr = pEObjs->EnumReset(0);

// Iterate through all the enumeration instances

```

```

while (ulCount--) {
    CHAR    pKey[256];
    GXVAL   val;

    // Get the next instance
    hr = pEObjs->EnumNext((IGXObject **) &pValList);

    // Retrieve and print the name of the appevent
    pValList->GetValByRef(GX_AE_RE_KEY_NAME, &val);
    fprintf(fp, "\nDefinitions for AppEvent named %s\n", val.u.pstrVal);

    // Reset to the first GXVAL in the ValList
    pValList->ResetPosition();

    // Iterate through all the GXVALs in the
    // vallist and print them to a file
    while (pValList->GetNextKey(pKey, 256) == NOERROR) {
        pValList->GetValByRef(pKey, &val);

        if (GXVT_TYPE(val.vt) == GXVT_LPSTR)
            fprintf(fp, "\t%s=%s (LPSTR)\n", pKey, val.u.pstrVal);
        else
            fprintf(fp, "\t%s=%d (DWORD)\n", pKey, val.u.ulVal);
        }
        pValList->Release();
    }
    // Save the file
    fclose(fp);

    // Release all resources used and return
    pEObjs->Release();
    pAppEvent->Release();
    return StreamResult("Successfully generated report<br>");

```

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** GetAppEvent() method in the GXAppLogic class

IGXValList interface

---

## QueryEvent()

Returns the properties of a registered event.

**Syntax** HRESULT QueryEvent(  
LPSTR pEventName,  
IGXValList \*\*ppValList);

**pEventName** The name of the registered event to enable.

**ppValList** Pointer to the IGXValList object that contains the returned event information. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** When an AppLogic calls RegisterEvent(), it can specify any of the following:

- The initial state—enable or disabled—of the event
- The time the event is to be triggered
- The AppLogic to execute when the event is triggered
- The email to send when the event is triggered

Use QueryEvent() to get the properties that were specified for a specific event.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** RegisterEvent()

---

## RegisterEvent()

Registers a named event for use in applications.

**Syntax** HRESULT RegisterEvent(  
LPSTR pEventName,  
IGXValList \*pValList);

**pEventName** The name of the event to register.

**pValList** The IGXValList object that specifies the properties of the event. The following table lists the keys and values you can specify:

| Key                 | Value   |
|---------------------|---|
| GX_AE_RE_KEY_NAME   | A string representing the name of the event. If specified, the name must be the same one specified as the pEventName parameter.   |
| GX_AE_RE_KEY_STATE  | An enum that specifies the initial state of the event:<br>GX_AE_RE_EVENT_DISABLED<br>GX_AE_RE_EVENT_ENABLED   |
| GX_AE_RE_KEY_TIME   | The time at which the event will be triggered. Use the following format:<br>hh:mm:ss W/DD/MM<br>hh: 0 -23<br>mm: 0 - 59<br>ss: 0 - 59<br>W (day of the week): 0 - 6 with 0 = Sunday.<br>DD (day of the month): 1 - 31<br>MM (month): 1 - 12<br>Each of these fields may be either an asterisk (meaning all legal values) or a list of elements separated by commas. An element is either a number or two numbers separated by a minus sign indicating an inclusive range. For example, 2, 5 - 7:0:0 5/*/* means the event is triggered at 2 AM, 5AM, 6 AM and 7 AM every Friday.<br>The specification of days can be made by two fields: day of the month (DD) and day of the week (W). If both are specified, both take effect. For example, 1:0:0 1/15/* means the event is triggered at 1 AM every Monday, as well as on the fifteenth of each month. To specify days by only one field, set the other field to *. |
| GX_AE_RE_KEY_NREQ   | The AppLogic to execute when the event is triggered. Use the following format:<br>GUIDGX-{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}?Param1=ABC&Param2=123   |
| GX_AE_RE_KEY_MFILE* | The name of the file that contains the body of an email message.  |
| GX_AE_RE_KEY_MTO*   | A comma separated list of users to send the email to.   |
| GX_AE_RE_KEY_MHOST* | The name of the SMTP mail server.   |
| GX_AE_RE_KEY_SADDR* | The sender's email address.   |

\* You must specify all of these items if sending email when the event is triggered.

**Usage** Use RegisterEvent() to define each event your application will use. You can specify that a triggered event sends an email, or runs an AppLogic, or both.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** The following example shows how to define and register an application event:

```

HRESULT hr = NOERROR;
IGXAppEvent *pAppEvent = NULL;
IGXValList *pValList = NULL;
LPSTR pReportEvName = "ReportEvent";
LPSTR pRepGenEvName = "RepGenEvent";

// Get a reference to the AppEvent Manager
hr = GetAppEvent(&pAppEvent);
if ((hr != NOERROR) || (pAppEvent == NULL))
    return StreamResult("AppEvent not found!<br>");

// Create a vallist to pass information
// for appevent registration of the first event
pValList = GXCreateValList();
if (pValList == NULL) {
    pAppEvent->Release();
    return Result(m_pValOut, "Can't create ValList<br>");
}
// Add the appevent name to the vallist
GXSetValListString(pValList, GX_AE_RE_KEY_NAME, pRepGenEvName);

// Set the appevent state to be enabled
GXSetValListInt(pValList, GX_AE_RE_KEY_STATE, GX_AE_RE_EVENT_ENABLED);

// Set the appevent time to be 05:00:00 hrs everyday
GXSetValListString(pValList, GX_AE_RE_KEY_TIME, "5:0:0 */*/*");

// Set the appevent action to run the RepGenAgent applogic
GXSetValListString(pValList, GX_AE_RE_KEY_NREQ, "GUIDGX-{630CB09B-1A1D-1315-AD23-0800207B918B}");

// Register the appevent
hr = pAppEvent->RegisterEvent(pRepGenEvName, pValList);

```

```
pValList->Release();
```

**Related Topics** EnableEvent( )

SetEvent( )

GetAppEvent( ) method in the GXAppLogic class

IGXValList interface

## SetEvent()

---

Triggers a registered event.

**Syntax**

```
HRESULT SetEvent(
    LPSTR pEventName,
    DWORD dwOverrideFlag,
    IGXValList *pValList);
```

**pEventName** The name of the event to trigger.

**dwOverrideFlag** Specify 0 (zero) to trigger the event with the previously specified actions. To override the defined actions, you can specify the following:

- GX\_AE\_SE\_ACTION\_NOMAIL if you don't want to send email when the event is triggered.
- GX\_AE\_SE\_ACTION\_NOREQ if you don't want to run an AppLogic when the event is triggered.

**pValList** The IGXValList object that specifies the event properties you want to override. Specify NULL to use the properties already defined for the event. The following table lists the keys and values you can specify:

| Key                | Value   |
|--------------------|---|
| GX_AE_RE_KEY_NAME  | A string representing the name of the event. If specified, the name must be the same one specified as the pEventName parameter. |
| GX_AE_RE_KEY_STATE | An enum that specifies the initial state of the event:<br>GX_AE_RE_EVENT_DISABLED<br>GX_AE_RE_EVENT_ENABLED                     |

| Key                 | Value   |
|---------------------|---|
| GX_AE_RE_KEY_TIME   | <p>The time at which the event will be triggered. Use the following format:</p> <p>hh:mm:ss W/DD/MM</p> <p>hh: 0 - 23</p> <p>mm: 0 - 59</p> <p>ss: 0 - 59</p> <p>W (day of the week): 0 - 6 with 0 = Sunday.</p> <p>DD (day of the month): 1 - 31</p> <p>MM (month): 1 - 12</p> <p>Each of these fields may be either an asterisk (meaning all legal values) or a list of elements separated by commas. An element is either a number or two numbers separated by a minus sign indicating an inclusive range. For example, 2, 5 - 7:0:0 5/*/* means the event is triggered at 2 AM, 5AM, 6 AM and 7 AM every Friday.</p> <p>The specification of days can be made by two fields: day of the month (DD) and day of the week (W). If both are specified, both take effect. For example, 1:0:0 1/15/* means the event is triggered at 1 AM every Monday, as well as on the fifteenth of each month. To specify days by only one field, set the other field to *.</p> |
| GX_AE_RE_KEY_NREQ   | <p>The AppLogic to execute when the event is triggered. Use the following format:</p> <p>GUIDGX-{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}?Param1=ABC&amp;Param2=123</p>  |
| GX_AE_RE_KEY_MFILE* | The name of the file that contains the body of an email message.  |
| GX_AE_RE_KEY_MTO*   | A comma separated list of users to send the email to.   |
| GX_AE_RE_KEY_MHOST* | The name of the SMTP mail server.   |
| GX_AE_RE_KEY_SADDR* | The sender's email address.   |

\* You must specify all of these items if sending email when the event is triggered.

**Usage** Use SetEvent( ) to trigger a registered event immediately. This is useful for testing purposes.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics**   RegisterEvent()  
                          GetAppEvent() method in the GXAppLogic class  
                          IGXValList interface



# IGXAppEventMgr interface

Application components can define events that are either triggered at a specified time or triggered explicitly. Events are stored persistently in the Netscape Application Server, and are removed only when your application explicitly deletes them. Events are typically used to schedule routine administrative tasks, such as making back-ups or getting statistics.

NAS uses two new interfaces to support events:

- The `IGXAppEventMgr` interface manages application events. This interface defines methods for creating, registering, triggering, enabling, disabling, enumerating, and deleting events.
- The `IGXAppEventObj` interface represents the defined events an application supports. This interface defines methods not only for getting or setting attributes of an event, but also for adding, deleting, or enumerating actions of the event.

`IGXAppEventMgr` and `IGXAppEventObj` should be used in new or revised applications. Existing NAS applications can continue using the deprecated `IGXAppEvent` interface, which supports the previous model for application events.

## Attributes and Actions

For each event, you define associated attributes and actions. Attributes determine the following characteristics:

- the event's state (enabled or disabled)
- the execution mode (concurrent or serial) of the event's actions
- the time at which to trigger the event

When an event is triggered, it performs one or more of the following types of actions:

- executes a specified servlet.
- executes a specified `AppLogic`.
- sends an email message.

## Features of Application Event Support

Support for application events includes the following:

- Added functionality
  - Execution of multiple actions of any type. (IGXAppEvent supports only one action of each type.)
  - In addition to executing an AppLogic and sending email, a triggered event can now execute a servlet as one of the supported action types.
  - Synchronous or asynchronous triggering of events. (IGXAppEvent supports only synchronous triggering.)
  - Execution of actions in the same order they are registered.
  - Execution of actions either concurrently or serially.
  - Support for passing an input IGXValList object to triggered events.

- Different interfaces

Previously, application events were represented by an IGXValList object, and you used the IGXAppEvent interface to manage the events. Now an application event is represented by an IGXAppEventObj, and you use IGXAppEventMgr to manage and control the events.

- Separate actions and attributes

Previously, attributes and actions were not distinguished. They were all treated as event properties and specified within a single IGXValList object. Now attributes are described by entries in one IGXValList object, and each action is represented by its own additional IGXValList object.

IGXAppEventObj has methods for getting or setting attributes and for adding, deleting, or enumerating actions.

## Accessing and Creating Application Events

To access an IGXAppEventMgr object, use the C++ helper function `GXContextGetAppEventMgr()`:

```
HRESULT GXContextGetAppEventMgr (
    IGXContext *pContext
    IGXAppEventMgr **ppAppEventMgr);
```

The `pContext` parameter is a pointer to an `IGXContext` object, which provides access to Netscape Application Server services. Specify a value of `m_pContext`, a member variable in the `GXAppLogic` class.

The ppAppEventMgr parameter is a pointer to the returned IGXAppEventMgr object.

After creating the IGXAppEventMgr object, you can create an application event (an instance of IGXAppEventObj) by calling CreateEvent( ) on the IGXAppEventMgr object.

## Registering Events

After creating an application event, you can set its attributes and add actions using methods on the IGXAppEventObj. Then, register the application event by calling registerEvent( ) on the manager object.

## Include File

gxiappevent.h

## Methods

| Method           | Description  |
|------------------|--|
| CreateEvent( )   | Creates an empty application event object.                   |
| DeleteEvent( )   | Removes a registered event from Netscape Application Server. |
| DisableEvent( )  | Disables a registered event.                                 |
| EnableEvent( )   | Enables a registered event.                                  |
| EnumEvents( )    | Enumerates through the list of registered events.            |
| GetEvent( )      | Retrieves the IGXAppEventObj for a registered event.         |
| RegisterEvent( ) | Registers a named event for use in applications.             |
| TriggerEvent( )  | Triggers a registered event.                                 |

## Related Topics

GXContextGetAppEventMgr( ),  
 IGXValList interface,  
 IGXAppEventObj interface

“Using Events” in Chapter 3, “Application Development Techniques” in *Programmer’s Guide*.

---

## CreateEvent()

Creates an empty application event object.

**Syntax** HRESULT CreateEvent(  
     LPSTR pEventName  
     IGXAppEventObj \*\*appeventObj);

**pEventName** The name of the event to create.

**appeventObj** A pointer to the returned IGXAppEventObj.

**Usage** Use CreateEvent() to create an empty IGXAppEventObj object. You can use methods of the IGXAppEventObj interface to set attributes and actions on the returned object.

Changes to the event object do not take effect until it is registered with the manager object, through a call to RegisterEvent().

Call the Release() method (defined in the IGXObject interface) when you are done.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** GetEvent()

RegisterEvent()

---

## DeleteEvent()

Removes a registered event from Netscape Application Server.

**Syntax** HRESULT DeleteEvent(  
     LPSTR pEventName);

**pEventName** The name of the registered event to delete.

**Usage** Use DeleteEvent() to remove an event that is no longer required. The specified event is removed from NAS.

To temporarily stop an event from being triggered, use DisableEvent().

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** DisableEvent()  
RegisterEvent()

---

## DisableEvent()

Disables a registered event.

**Syntax** HRESULT DisableEvent(  
LPSTR pEventName);

**pEventName** The name of the registered event to disable.

**Usage** Use DisableEvent() to temporarily stop an event from being triggered. The event is disabled until it is enabled with EnableEvent(). To permanently remove an event from the registry, use DeleteEvent().

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** DeleteEvent()  
EnableEvent()  
RegisterEvent()

---

## EnableEvent()

Enables a registered event.

**Syntax** HRESULT EnableEvent(  
LPSTR pEventName);

**pEventName** The name of the registered event to enable.

**Usage** Use EnableEvent() to enable an event. A given event could have been disabled in either of two ways: by a previous call to DisableEvent() or by initially registering the event using a disabled state attribute.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** DisableEvent()  
RegisterEvent()

## EnumEvents()

---

Enumerates through the list of registered events.

**Syntax** HRESULT EnumEvents(  
IGXEnumObject \*\*ppEvents);

**ppEvents** Pointer to the IGXEnumObject object that contains the list of registered events. When an application component is finished using the object, call the Release() method to release the interface instance.

**Usage** Use EnumEvents() to get information on all the registered events. The IGXEnumObject object returned by EnumEvents() contains a set of IGXAppEventObj objects, one per event. Each IGXAppEventObj contains the attributes and actions that were assigned to the event when it was registered with RegisterEvent().

**Tip** Use the methods in the IGXEnumObject interface to iterate through the contents of the returned IGXEnumObject object.

**Example** The following AppLogic code shows how to use EnumEvents() to get information on all the registered events and save it to a report:

```

HRESULT hr = NOERROR;
IGXEnumObject *pEObjs = NULL;
IGXAppEventMgr *pAppEventMgr = NULL;
CHAR pBuf[128];
ULONG ulCount = 0;
FILE *fp;

// Open /tmp/report-file for writing the report
fp = fopen("/tmp/report-file", "w");

// Get a reference to the AppEvent Manager
hr = GXContextGetAppEventMgr(m_context, &pAppEventMgr);

// Get the Enumeration object for all registered appevents
hr = pAppEventMgr->EnumEvents(&pEObjs);

// Retrieve the count of registered appevents
hr = pEObjs->EnumCount(&ulCount);
fprintf(fp, "Number of Registered Events: %d\n", ulCount);

// Reset the next enumeration object to the first instance

```

```

hr = pEObjs->EnumReset(0);

// Iterate through all the enumeration instances
while (ulCount-->0) {
    CHAR pKey[256];
    CHAR name[256];
    GXVAL val;
    IGXValList *pValList = NULL;
    IGXAppEventObj *pAppEventObj = NULL;

    // Get the next instance
    hr = pEObjs->EnumNext((IGXObject **)&pAppEventObj);

    // Retrieve and print the name of the appevent
    hr = pAppEventObj->GetName(name, 256);
    fprintf(fp, "\nDefinitions for AppEvent named %s\n", name);

    // Retrieve attributes
    hr = pAppEventObj->GetAttributes(&pValList);

    // Reset to the first GXVAL in the ValList
    pValList->ResetPosition();

    fprintf(fp, "\nAttributes for AppEvent\n");
    // Iterate through all the GXVALs in the
    // vallist and print them to a file
    while (pValList->GetNextKey(pKey, 256) == NOERROR) {
        pValList->GetValByRef(pKey, &val);
        if (GXVT_TYPE(val.vt) == GXVT_LPSTR)
            fprintf(fp, "\t%s=%s (LPSTR)\n", pKey, val.u.pstrVal);
        else
            fprintf(fp, "\t%s=%d (DWORD)\n", pKey, val.u.ulVal);
    }
    pValList->Release();

    // Retrieve and print Actions
    fprintf(fp, "\nActions for AppEvent\n");
    hr = pAppEventObj->EnumActions(&pEActionObjs);

    // Retrieve the count of registered appevents
    hr = pEActionObjs->EnumCount(&ulActionCount);
    fprintf(fp, "Number of Actions for event: %d\n", ulActionCount);
}

```

```

// Reset the next enumeration object to be the first instance
hr = pEActionObjs->EnumReset(0);

// Iterate through all the enumeration instances
while (ulActionCount--) {
    // Get the next action
    hr = pEActionObjs->EnumNext((IGXObject **) &pValList);

    // Reset to the first GXVAL in the ValList
    pValList->ResetPosition();

    // Iterate through all the GXVALs that describe the action
    while (pValList->GetNextKey(pKey, 256) == NOERROR) {
        pValList->GetValByRef(pKey, &val);
        if (GXVT_TYPE(val.vt) == GXVT_LPSTR)
            fprintf(fp, "\t%s=%s (LPSTR)\n", pKey, val.u.pstrVal);
        else
            fprintf(fp, "\t%s=%d (DWORD)\n", pKey, val.u.ulVal);
    }
    pValList->Release();
}

pEActionObjs->Release();
pAppEventObj->Release();
}

// Save the file
fclose(fp);

// Release all resources used and return
pEObjs->Release();
pAppEventMgr->Release();
return StreamResult("Successfully generated report<br>");

```

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** IGXEnumObject interface

## GetEvent()

Retrieves the IGXAppEventObj for a registered event.

**Syntax** HRESULT GetEvent(  
LPSTR pEventName,  
IGXAppEventObj \*\*ppAppEvent);

**pEventName** The name of the registered event.

**ppAppEvent** Pointer to the IGXAppEventObj object for the given event. When the application component is finished using the object, call the Release() method to release the interface instance.

**Usage** After calling GetEvent(), you can call methods on the returned IGXAppEventObj. For example, you can query the object by calling GetAttributes() or EnumActions(), or you can modify the object by calling SetAttributes().

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** RegisterEvent()

---

## RegisterEvent()

Registers a named event for use in applications.

**Syntax** HRESULT RegisterEvent(  
IGXAppEventObj \*appEventObj);

**appEventObj** The event object whose attributes and actions have been set.

**Usage** After an application event object is created with CreateEvent(), you define its attributes and actions using methods of the IGXAppEventObj interface. Then you use RegisterEvent() to register the specified event object. Registration commits the changed attributes and actions to the server and to the registry. If an event object already exists for the given name, the existing object is deleted and replaced with the specified object.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** The following example shows how to define and register an application event:

```
HRESULT hr = NOERROR;
IGXAppEventMgr *pAppEventMgr = NULL;
IGXValList *pValList = NULL;

LPSTR pRepGenEvName = "RepGenEvent";
```

```

// Get a reference to the AppEvent Manager
hr = GXContextGetAppEventMgr(m_context, &pAppEvent);

if ((hr != NOERROR) || (pAppEventMgr == NULL))
    return StreamResult("AppEventMgr not found!<br>");

// Create an empty Event Object
hr = pAppEventMgr->CreateEvent(pRepGenEvName, &pAppEventObj);
if ((hr != NOERROR) || (pAppEventObj == NULL))
    return StreamResult("CreateEvent failed!<br>");

// Prepare and set the Attributes.
// Create a vallist for the Attributes
pValList = GXCreateValList();

// Set the appevent time to be 05:00:00 hrs everyday
GXSetValListString(pValList, GX_AE2_RE_KEY_TIME, "5:0:0 */*/*");

// Set the attributes
hr = pAppEventObj->SetAttributes(pValList);
if (hr != NOERROR)
    return StreamResult("Can't set Attributes<br>");

pValList->Release();

// Add 4 Actions in the order we want them to be executed

// Set action 1 to run the SummaryRepGenAgent1 applogic
pValList = GXCreateValList();
GXSetValListString(pValList, GX_AE2_RE_KEY_NREQ,
    "GUIDGX-{630CB09B-1A1D-1315-AD23-0800207B918B}");
hr = pAppEventObj->AddAction(pValList);
pValList->Release();

// Set action 2 to run the SummaryRepGenAgent2 applogic
pValList = GXCreateValList();
GXSetValListString(pValList, GX_AE2_RE_KEY_NREQ,
    "GUIDGX-{414643FA-B74A-1544-C25E-0800207B8777}");
hr = pAppEventObj->AddAction(pValList);
pValList->Release();

// Set action 3 to run the DetailRepGenServlet1 servlet

```

```

pValList = GXCreateValList();
GXSetValListString(pValList, GX_AE2_RE_KEY_SERVLET,
    "DetailRepGenServlet1");
hr = pAppEventObj->AddAction(pValList);
pValList->Release();

// Set action 4 to run the DetailRepGenServlet2 servlet
pValList = GXCreateValList();
GXSetValListString(pValList, GX_AE2_RE_KEY_SERVLET,
    "DetailRepGenServlet2");
hr = pAppEventObj->AddAction(pValList);
pValList->Release();

// Register the appevent
hr = pAppEventMgr->RegisterEvent(pAppEventObj);

pAppEventObj->Release();
pAppEventMgr->Release();

```

**Related Topics** EnableEvent()

TriggerEvent()

---

## TriggerEvent()

Triggers a registered event.

**Syntax** HRESULT TriggerEvent(  
LPSTR pEventName,  
IGXValList \*pValList,  
BOOL syncFlag);

**pEventName** The name of the event to trigger.

**pValList** The IGXValList object that specifies the input that is passed to the triggered event and its actions.

**syncFlag** The boolean flag that indicates whether the event is to be triggered synchronously (value is true) or asynchronously (value is false).

**Usage** Use TriggerEvent() to trigger a registered event. When you specify the pValList parameter, a copy of this IGXValList object is passed as input to all actions registered with the application event.

| If the action is ... | Then pValList is ...   |
|----------------------|--|
| an AppLogic.         | passed as input to that AppLogic.                              |
| an email message.    | simply ignored.  |
| a servlet.           | passed to the servlet as the valIn of the underlying AppLogic. |

Use the syncFlag parameter to determine synchronous or asynchronous execution. Typical usage is to set syncFlag to false, which provides asynchronous execution and better application performance. When syncFlag is false, the event is triggered, and the method call returns immediately, without waiting for the actions to finish executing.

If syncFlag is true, then the method call does not return immediately. Instead, the call blocks until the event is triggered and all actions have executed. In some cases, it may be desirable for actions to finish executing before returning control to the application.

Actions are triggered in the same order in which they were added to the application event object.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** RegisterEvent( )

# IGXAppEventObj interface

See the IGXAppEventMgr interface for details on IGXAppEventObj.

## Include File

gxiappevent.h

## Methods

| Method           | Description   |
|------------------|---|
| AddAction( )     | Appends an action to an ordered list of actions.        |
| DeleteActions( ) | Deletes all actions added to this IGXAppEventObj.       |
| EnumActions( )   | Enumerates the actions added to this IGXAppEventObj.    |
| GetAttributes( ) | Retrieves the list of attributes of an IGXAppEventObj.  |
| GetName( )       | Retrieves the name of the IGXAppEventObj.               |
| SetAttributes( ) | Sets a list of attribute values for the IGXAppEventObj. |

## Related Topics

IGXAppEventMgr interface

## AddAction( )

---

Appends an action to an ordered list of actions.

**Syntax** HRESULT AddAction(  
                   IGXValList \*action);

**action** The input IGXValList object that defines the action to add. When an event is triggered, actions are executed in the same order in which they were added. The entries in this IGXValList object vary from one action type to another.

The keys and values you can specify are as follows.

For AppLogics:

| Key                | Value  |
|--------------------|--|
| GX_AE2_RE_KEY_NREQ | The AppLogic to execute when the event is triggered. Use the following format: GUIDGX-{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}?Param1=ABC&Param2=123. The parameters and their values are passed as input to the events. |

For email:

To send email when the event is triggered, all of the following items must be specified.

| Key                 | Value  |
|---------------------|--|
| GX_AE2_RE_KEY_MFILE | The name of the file that contains the body of an email message. |
| GX_AE2_RE_KEY_MTO   | A comma separated list of users to send the email to.            |
| GX_AE2_RE_KEY_MHOST | The name of the SMTP mail server.                                |
| GX_AE2_RE_KEY_SADDR | The sender's email address.                                      |

For servlets:

| Key                   | Value   |
|-----------------------|---|
| GX_AE2_RE_KEY_SERVLET | The name of the servlet to be executed when the event is triggered. Use the following format: <i>appName/ServletName?Param1=ABC&amp;Param2=123</i> . Parameters and their values are passed as input to the events. The only required item is the servlet name. The application name and parameters are optional. |

**Usage** Use the `AddAction()` method after creating an application event object (by calling `CreateEvent()` on the `IGXAppEventMgr` object). After you change an event (for example, by adding or deleting actions or by setting attributes), you must register the event in order for the changes to take effect.

To list the added actions, use `EnumActions()`. To delete all actions, use `DeleteActions()`.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

---

## DeleteActions()

Deletes all actions added to this `IGXAppEventObj`.

**Syntax** `HRESULT DeleteActions();`

**Usage** Use this method to remove all actions associated with this `IGXAppEventObj`.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

---

## EnumActions()

Enumerates the actions added to this `IGXAppEventObj`.

**Syntax** `HRESULT EnumActions(  
IGXEnumObject **actions);`

**actions** A pointer to the returned `IGXEnumObject`.

**Usage** Use this method to obtain a list of actions that have been added to this `IGXAppEventObj`. Each entry in the returned `IGXEnumObject` is an `IGXVallist` object representing an action.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

---

## GetAttributes()

Retrieves the list of attributes of an `IGXAppEventObj`.

**Syntax** `HRESULT GetAttributes(  
IGXVallist **attrList);`

**attrList** A pointer to the returned `IGXVallist` object.

**Usage** Call this method after calling `SetAttributes()`.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `SetAttributes()`

## GetName()

---

Retrieves the name of the IGXAppEventObj.

**Syntax** HRESULT GetName(  
LPSTR pName,  
unsigned long nName);

**pName** A pointer to an input buffer.

**nName** The size of the input buffer.

**Usage** The name of an IGXAppEventObj is set by calling CreateEvent( ) on the IGXAppEventMgr object. After creating an application event object, use the GetName( ) method to retrieve the name.

**Return Value** A string representing the name of the application event object, or null for failure.

## SetAttributes()

---

Sets a list of attribute values for the IGXAppEventObj.

**Syntax** HRESULT SetAttributes(  
IGXValList \*attrList);

**attrList** The input IGXValList object that specifies the attributes. The keys and values you can specify are as follows.

- GX\_AE2\_RE\_KEY\_STATE  
An enum that specifies the initial state of the event. This key is optional and has the following possible values:
  - GX\_AE2\_RE\_EVENT\_DISABLED
  - GX\_AE2\_RE\_EVENT\_ENABLED (the default)
- GX\_AE2\_RE\_KEY\_TIME  
An optional key that specifies the time at which the event will be triggered. Use the following format:
  - hh:mm:ss W/DD/MM
  - hh: 0 -23
  - mm: 0 - 59

- ss: 0 - 59
- W (day of the week): 0 - 6 with 0 = Sunday.
- DD (day of the month): 1 - 31
- MM (month): 1 - 12

Each of these fields may be either an asterisk (meaning all legal values) or a list of elements separated by commas. An element is either a number or two numbers separated by a minus sign indicating an inclusive range. For example, 2, 5 - 7:0:0 5/\*/\* means the event is triggered at 2 AM, 5AM, 6 AM and 7 AM every Friday.

The specification of days can be made by two fields: day of the month (DD) and day of the week (W). If both are specified, both take effect. For example, 1:0:0 1/15/\* means the event is triggered at 1 AM every Monday, as well as on the fifteenth of each month. To specify days by only one field, set the other field to \*.

- GX\_AE2\_RE\_KEY\_ACTION\_MODE

An optional key that specifies whether actions are to be executed concurrently (at the same time) or in series (one after another). In serial execution, each action finishes executing before the next one starts, and execution occurs in the same order in which the actions were added.

This key has the following possible values:

- GX\_AE2\_RE\_ACTION\_SERIAL
- GX\_AE2\_RE\_ACTION\_CONCURRENT (the default)

**Usage** Use the SetAttributes( ) method after creating an application event object (by calling CreateEvent( ) on the IGXAppEventMgr object). After you change an event (for example, by adding or deleting actions or by changing attributes), you must register the event in order for the changes to take effect.

**Tip** None of the attributes are required to be set. The default state is enabled, and the default action mode is concurrent.

To retrieve the list of attributes that are set, use GetAttributes( ).

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** GetAttributes( )



# IGXBuffer interface

The IGXBuffer interface represents a block of memory. Input arguments and output value(s) of methods are sometimes stored in IGXBuffer objects. For example, the Get\*\*() methods in the IGXQuery interface return values in an IGXBuffer object.

IGXBuffer provides methods for specifying and obtaining the size of the memory block, obtaining its starting address, and copying data to it.

To create an instance of the IGXBuffer interface, use the GXCreateBuffer() function.

## Include File

gxibuff.h

## Methods

| Method       | Description                                       |
|--------------|---|
| Alloc()      | Specifies the size of the memory block, in bytes. |
| GetAddress() | Returns the address of the memory block.          |
| GetSize()    | Returns the size of the memory block, in bytes.   |
| SetData()    | Copies data to a memory block.                    |

## Alloc()

Specifies the size of the memory block, in bytes.

**Syntax** HRESULT Alloc(  
    ULONG nSize);

**nSize** Size of the memory block, in bytes.

**Usage** After creating a memory buffer with the GXCreateBuffer() function, use Alloc() to specify its size.

Subsequent calls to `GetSize()` return the size that `AppLogic` specified when it called `Alloc()`.

- Rules**
- If the `AppLogic` creates its own new `IGXBuffer` object, it must first specify the size of the memory block by calling `Alloc()` before using other methods in the interface.
  - `AppLogic` can call `Alloc()` only once.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```
LPSTR str = "Hello World";
IGXBuffer buff;
buff = GXCreateBuffer();
buff->Alloc(128);
buff->SetData((LPBYTE) str, 12);
```

**Related Topics** `GetAddress()`

---

## GetAddress()

Returns the address of the memory block.

**Syntax** `LPBYTE GetAddress();`

**Usage** Use `GetAddress()` to obtain the starting address of the buffer that was allocated by `Alloc()`. The starting address of the buffer is needed when copying data to and from the buffer.

**Rule** Before calling `GetAddress()`, the memory buffer must be allocated first by calling `Alloc()`. When the system returns an `IGXBuffer` object (for example, when the `AppLogic` calls a `Get**()` method in the `IGXQuery` interface), it automatically allocates the memory buffer.

**Return Value**

```
HRESULT hr;
IGXBuffer *buff;
buff = NULL;
hr = query->GetTables(&buff);
if (hr == NOERROR && buff)
{
    // Use IGXBuffer interface here. The memory held by
    // the IGXBuffer object should be treated as read-only.
```

```

//
StreamResult("The tables accessed by the query are ");
StreamResult((LPSTR) buff->GetAddress());
StreamResult(".<br>");

// Release buff when done with it.
//
buff->Release();
}

```

**Related Topics** Alloc()

---

## GetSize()

Returns the size of the memory block, in bytes.

**Syntax** ULONG GetSize();

**Usage** Use GetSize() to determine the length of the memory buffer that the AppLogic specified when it called Alloc().

**Rule** Before calling GetSize(), AppLogic must first specify the size of the memory block by calling Alloc().

**Return Value** Size of the memory block, in bytes.

**Related Topics** GetAddress()

SetData()

---

## SetData()

Copies data to a memory block.

**Syntax** HRESULT SetData(  
LPBYTE pData,  
ULONG nDataLen);

**pData** The data to copy to the memory buffer.

**nDataLen** The length, in bytes, of the data to copy to the memory buffer.

**Usage** Use `SetData()` to copy data to a memory buffer. The buffer can then be passed to a method, such as the `SetValPieceByOrd()` method in the `IGXTable` interface, that accepts data values in a buffer object.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```
LPSTR str = "Hello World";
IGXBuffer buff;
buff = GXCreateBuffer();
buff->Alloc(128);
buff->SetData((LPBYTE) str, 12);

table->SetValPieceByOrd(1, buff, 12);

buff->Release();
```

**Related Topics** `GetAddress()`  
`GetSize()`

# IGXCallableStmt interface

The IGXCallableStmt interface provides a standard way to call stored procedures in any database server. A stored procedure is a block of SQL statements stored in a database. Stored procedures provide centralized code for manipulating data and reduce the amount of data that needs to be sent to the client side of an application. They are typically used to execute database operations, for example, modify, insert, or delete records.

To call a stored procedure from an AppLogic, use the IGXCallableStmt object. The IGXCallableStmt interface defines methods for executing a stored procedure or function, and setting and getting parameter values to and from a stored procedure.

To create an instance of the IGXCallableStmt interface, use PrepareCall() in the IGXDataConn interface.

## Include File

gxidata.h

## Methods

| Method              | Description  |
|---------------------|--|
| Close()             | Releases the callable statement.   |
| Execute()           | Executes the stored procedure called by the IGXCallableStatement object.   |
| ExecuteMultipleRS() | Executes a stored procedure, called by the IGXCallableStmt object, that can return multiple result sets.   |
| GetMoreResults()    | Checks if there is a result set to retrieve. This method is valid only if you used ExecuteMultipleRS(), not Execute(), to execute a stored procedure called by the IGXCallableStmt object. |
| GetParams()         | Returns the value of the stored procedure's output parameter or parameters.  |

| Method          | Description  |
|-----------------|--|
| GetResultSet( ) | Retrieves a result set. This method is valid only if you used ExecuteMultipleRS( ) (instead of Execute( )) to execute a stored procedure called by the IGXCallableStmt object. |
| SetParams( )    | Specifies the parameter values to pass to the stored procedure.  |

## Related Topics

PrepareCall( ) in the IGXDataConn interface

“Using Stored Procedures” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

## Close( )

Releases the callable statement.

**Syntax** HRESULT Close( )

**Usage** Use Close( ) to release a callable statement object after the AppLogic has finished processing the results returned by the stored procedure.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

## Execute( )

Executes the stored procedure called by the IGXCallableStmt object.

**Syntax** HRESULT Execute(  
 DWORD dwFlags,  
 IGXValList \*pParams,  
 IGXTrans \*pTrans,  
 IGXValList \*pProps,  
 IGXResultSet \*\*ppResultSet);

### dwFlags

- For synchronous operations, the default, specify 0 (zero) or GX\_DA\_EXEC\_SYNC.

- For asynchronous operations, specify `GX_DA_EXEC_ASYNC`.

**pParams** Pointer to an `IGXVallist` object that contains parameters to pass to the callable statement. If you use `SetParams()` instead to specify the parameters, specify `NULL` here.

**pTrans** Pointer to an `IGXTrans` object that contains the transaction associated with this callable statement, or `NULL` for no transaction.

**pProps** Pointer to the `IGXVallist` object that contains properties, or `NULL` for no properties. This parameter applies only if the callable statement returns a result set. Informix stored procedures, for example, return out parameter values only as a result set. Sybase, DB2, and MS SQL Server stored procedures also support the return of a result set. Multiple result sets, however, is not supported.

After instantiating an object of the `IGXVallist` interface, set any of the following properties:

- `RS_BUFFERING` turns on result set buffering when set to “TRUE” or “YES.”
- `RS_INIT_ROWS` specifies the initial size of the buffer, in number of rows. If the result set size exceeds this setting, a `FetchNext()` call will return the error `GX_DA_BUFFER_EXCEEDED` and result set buffering will be turned off.
- `RS_MAX_ROWS` specifies the maximum number of rows for the buffer. If the result set size exceeds this setting, a `FetchNext()` call will return the error `GX_DA_BUFFER_EXCEEDED` and result set buffering will be turned off.
- `RS_MAX_SIZE` specifies the maximum number of bytes for the buffer.

If `RS_BUFFERING` is enabled and if the optional parameters are not specified, the global values in the registry are used instead.

**ppResultSet** Pointer to the `IGXResultSet` object that contains the returned result set from the stored procedure, if the database supports this feature. Informix, DB2, MS SQL Server and Sybase support it. When the `AppLogic` is finished using the object, call the `Close()` method in the `IGXResultSet` interface, then call the `Release()` method to release the interface instance.

**Usage** Use `Execute()` to run a callable statement that has been created with `PrepareCall()` in the `IGXDataConn` interface. If the stored procedure called by the `IGXCallableStmt` object can return multiple result sets, use `ExecuteMultipleRS()` instead.

If the stored procedure called by the `IGXCallableStmt` object contains parameters, instantiate an `IGXValList` object and use `SetVal()` or `SetValByRef()` in the `IGXValList` interface to specify the parameter values to pass to the stored procedure.

After creating and setting up the `IGXValList` object, pass it to `Execute()` or `SetParams()`. If you use `SetParams()` to pass parameters to the stored procedure, specify `NULL` for the `params` parameter in `Execute()`.

**Rule** When accessing a stored procedure on Sybase or MS SQL Server, input parameter names specified in the call must be prefixed with the ampersand (&) character, for example, `&param1`. Other database drivers accept the ampersand, as well, as the colon (:) character. For all database drivers, input/output and output parameter names are prefixed with the colon (:) character, for example, `:param2`.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```
// Write the command to call the stored procedure
IGXQuery *qry = NULL;
hr = CreateQuery(&qry);
if (hr == NOERROR &&
    qry)
{
    qry->SetSQL("{:ret = call myFunction(&param1)}");

    //Prepare the callable statement for execution
    IGXCallableStmt *s = NULL;
    hr = conn->PrepareCall(0, qry, NULL, NULL, &s);
    if (hr == NOERROR &&
        s)
    {
        // Set the in parameter values
        IGXValList *params;
        params = GXCreateValList();
        params->SetValInt(":ret", 9999);
        params->SetValInt("&param1", 20);
    }
}
```

```

IGXResultSet *rs = NULL;

// Run the callable statement
hr = s->Execute(0, params, NULL, NULL, &rs);
if (hr == NOERROR &&
    rs)
{
    // Get the stored procedure's output value
    IGXValList *paramsOut = NULL;
    hr = s->GetParams(0, &paramsOut);
    if (hr == NOERROR &&
        paramsOut)
    {

```

**Related Topics** PrepareCall() in the IGXDataConn interface

GetParams()

SetParams()

“Using Stored Procedures” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

---

## ExecuteMultipleRS()

Executes a stored procedure, called by the IGXCallableStmt object, that can return multiple result sets.

**Syntax** HRESULT ExecuteMultipleRS(  
     DWORD dwFlags,  
     IGXValList \*pParams,  
     IGXTrans \*pTrans,  
     IGXValList \*pProps)

### dwFlags

Specify 0.

**pParams** Pointer to an IGXValList object that contains parameters to pass to the callable statement. If no parameters are required, pass in an empty IGXValList. If you use SetParams() instead to specify the parameters, specify NULL here.

**pTrans** Pointer to an IGXTrans object that contains the transaction associated with this callable statement, or NULL for no transaction.

**pProps** Pointer to the IGXValList object that contains properties, or NULL for no properties.

After instantiating an object of the IGXValList interface, set any of the following properties:

- **RS\_BUFFERING** turns on result set buffering when set to “TRUE” or “YES.”
- **RS\_INIT\_ROWS** specifies the initial size of the buffer, in number of rows. If the result set size exceeds this setting, a `FetchNext()` call will return the error `GX_DA_BUFFER_EXCEEDED` and result set buffering will be turned off.
- **RS\_MAX\_ROWS** specifies the maximum number of rows for the buffer. If the result set size exceeds this setting, a `FetchNext()` call will return the error `GX_DA_BUFFER_EXCEEDED` and result set buffering will be turned off.
- **RS\_MAX\_SIZE** specifies the maximum number of bytes for the buffer.

If **RS\_BUFFERING** is enabled and if the optional parameters are not specified, the global values in the registry are used instead.

**Usage** Use `ExecuteMultipleRS()` to run a callable statement that returns multiple result sets. The callable statement should already have been created with `PrepareCall()` in the `IGXDataConn` interface.

If the stored procedure called by the `IGXCallableStmt` object contains parameters, instantiate an `IGXValList` object and use `SetVal()` or `SetValByRef()` in the `IGXValList` interface to specify the parameter values to pass to the stored procedure.

After creating and setting up the `IGXValList` object, pass it to `ExecuteMultipleRS()` or `SetParams()`. If you use `SetParams()` to pass parameters to the stored procedure, specify NULL for the `pParams` parameter in `ExecuteMultipleRS()`.

**Rule** When accessing a stored procedure on Sybase or MS SQL Server, input parameter names specified in the call must be prefixed with the ampersand (&) character, for example, `&param1`. Other database drivers accept the ampersand,

as well, as the colon (:) character. For all database drivers, input/output and output parameter names are prefixed with the colon (:) character, for example, :param2.

**Tip** The difference between `Execute()` and `ExecuteMultipleRS()` is that `Execute()` can return only a single result set. If you're not sure how many results sets, if any, a stored procedure returns, use `ExecuteMultipleRS()`.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```
hr = stmt->ExecuteMultipleRS(0, params, NULL, NULL);
DWORD moreResult = TRUE;
do {
    hr = stmt->GetMoreResults(&moreResult);
    if (moreResult == FALSE)
    {
        StreamResult("No more Results to process<BR>");
        break;
    }
    else
    {
        IGXResultSet *pResultSet;
        hr = stmt->GetResultSet(&pResultSet);
        if (pResultSet)
        {
            DisplayResult(pResultSet);
            pResultSet->Release();
        }
    }
} while(TRUE);
```

**Related Topics** `PrepareCall()` in the `IGXDataConn` interface

`GetMoreResults()`

`GetResultSet()`

“Using Stored Procedures” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

## GetMoreResults()

---

Checks if there is a result set to retrieve. This method is valid only if you used `ExecuteMultipleRS()` (instead of `Execute()`) to execute a stored procedure called by the `IGXCallableStmt` object.

**Syntax** `HRESULT GetMoreResults(  
    BOOL *pMoreResult)`

**pMoreResult** Pointer to the client-allocated `BOOL` variable that contains the returned information.

**Usage** If you used `ExecuteMultipleRS()` to execute a stored procedure that returns multiple results sets, use `GetMoreResults()` in conjunction with `GetResultSet()` to check if there is a result set before retrieving it.

If there is a current result set with unretrieved rows, `GetMoreResults()` discards the current result set and makes the next result set available.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```
hr = stmt->ExecuteMultipleRS(0, params, NULL, NULL);
DWORD moreResult = TRUE;
do {
    hr = stmt->GetMoreResults(&moreResult);
    if (moreResult == FALSE)
    {
        StreamResult("No more Results to process<BR>");
        break;
    }
    else
    {
        IGXResultSet *pResultSet;
        hr = stmt->GetResultSet(&pResultSet);
        if (pResultSet)
        {
            DisplayResult(pResultSet);
            pResultSet->Release();
        }
    }
} while(TRUE);
```

**Related Topics** `PrepareCall()` in the `IGXDataConn` interface

ExecuteMultipleRS()

GetResultSet()

“Using Stored Procedures” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

## GetParams()

Returns the value of the stored procedure’s output parameter or parameters.

**Syntax** `HRESULT GetParams(  
    DWORD dwFlags  
    IGXValList **ppParams);`

**dwFlags** Specify 0 (zero).

**ppParams** Pointer to the IGXValList object that contains the stored procedure’s output parameters. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Some stored procedures return output parameters. If the stored procedure your callable statement executes returns output parameters, use GetParams() to get the values.

The GetParams() method returns the values in an IGXValList object. The key names associated with the values are the parameter names as specified in the query that was passed to the PrepareCall() method.

**Tip** Informix stored procedures return output parameters in a result set. This result set is returned by Execute() or ExecuteMultipleRS(). The GetParams() method, therefore, does not apply to Informix stored procedures.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
// Write the command to call the stored procedure
IGXQuery *qry = NULL;
hr = CreateQuery(&qry);
if (hr == NOERROR &&
    qry)
{
    qry->SetSQL("{:ret = call myFunction(&param1)}");

    //Prepare the callable statement for execution
```

```

IGXCallableStmt *s = NULL;
hr = conn->PrepareCall(0, qry, NULL, NULL, &s);
if (hr == NOERROR &&
    s)
{
    // Set the in parameter values
    IGXValList *params;
    params = GXCreateValList();
    params->SetValInt(":ret", 9999);
    params->SetValInt("&param1", 20);

    IGXResultSet *rs = NULL;

    // Run the callable statement
    hr = s->Execute(0, params, NULL, NULL, &rs);
    if (hr == NOERROR &&
        rs)
    {
        // Get the stored procedure's output value
        IGXValList *paramsOut = NULL;
        hr = s->GetParams(0, &paramsOut);
        if (hr == NOERROR &&
            paramsOut)
        {

```

**Related Topics** PrepareCall() in the IGXDataConn interface

Execute()

SetParams()

“Using Stored Procedures” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

## GetResultSet()

---

Retrieves a result set. This method is valid only if you used ExecuteMultipleRS() (instead of Execute()) to execute a stored procedure called by the IGXCallableStmt object.

**Syntax** HRESULT GetResultSet(  
 IGXResultSet \*\*ppResultSet)

**ppResultSet** Pointer to the IGXResultSet object that contains the returned result set. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** If you used ExecuteMultipleRS() to execute a stored procedure that returns multiple results sets, use GetResultSet() in conjunction with GetMoreResults() to retrieve the results sets.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
hr = stmt->ExecuteMultipleRS(0, params, NULL, NULL);
DWORD moreResult = TRUE;
do {
    hr = stmt->GetMoreResults(&moreResult);
    if (moreResult == FALSE)
    {
        StreamResult("No more Results to process<BR>");
        break;
    }
    else
    {
        IGXResultSet *pResultSet;
        hr = stmt->GetResultSet(&pResultSet);
        if (pResultSet)
        {
            DisplayResult(pResultSet);
            pResultSet->Release();
        }
    }
} while(TRUE);
```

**Related Topics** PrepareCall() in the IGXDataConn interface

ExecuteMultipleRS()

GetMoreResults()

“Using Stored Procedures” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

## SetParams()

Specifies the parameter values to pass to the stored procedure.

**Syntax** HRESULT SetParams(  
    DWORD dwFlags  
    IGXValList \*pParams);

**dwFlags** Specify 0 (zero). For internal use only.

**pParams** Pointer to the IGXValList object that contains the parameters to pass to the stored procedure. You must set all parameters required by the stored procedure. If you don't, a runtime error will occur when Execute() is called. If you use SetParams(), specify NULL for the pParams parameter in Execute().

**Usage** If the stored procedure the callable statement executes accepts input parameters, use SetParams() to pass the parameter or parameter values. The alternative is to pass the parameter values with the Execute() method. Parameters passed to Execute() supersede parameters specified with SetParams().

For both SetParams() and Execute(), you pass the parameter values in an IGXValList object.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** PrepareCall() in the IGXDataConn interface

Execute()

GetParams()

“Using Stored Procedures” in Chapter 5, “Working with Databases” in *Programmer's Guide*.

# IGXColumn interface

The IGXColumn interface represents a column definition in a table. IGXColumn provides methods for obtaining descriptive information about a table column from the database catalog, which contains the column definition. Column attributes include the column name, precision, scale, size, table, and data type.

IGXColumn is part of the Data Access Engine (DAE) service.

To create an instance of this interface, use one of the following methods:

- `GetColumn()` or `GetColumnByOrd()` in the `IGXHierResultSet` interface
- `GetColumn()`, `GetColumnByOrd()`, or `EnumColumns()` in the `IGXTable` interface
- `GetColumn()`, `GetColumnByOrd()`, or `EnumColumns()` in the `IGXResultSet` interface

## Include File

`gxidata.h`

## Methods

| Method                         | Description   |
|--------------------------------|---|
| <code>GetName()</code>         | Returns the name of the column or alias.  |
| <code>GetNullsAllowed()</code> | Returns true if NULL values are allowed in the column.  |
| <code>GetPrecision()</code>    | Returns the precision, which is the maximum length or maximum number of digits, of the column.                    |
| <code>GetScale()</code>        | Returns the scale, which is the number of digits to the right of the decimal point, of the column of type double. |
| <code>GetSize()</code>         | Returns the maximum length, in number of bytes, allowed for a value in this column.                               |
| <code>GetTable()</code>        | Returns the table object in which this column exists.   |
| <code>GetType()</code>         | Returns the data type of the column.  |

The following example shows how to iterate through a table to get the names and types of the columns:

```

HRESULT hr;
IGXDataConn *conn;
// Retrieve connection with CreateDataConn().
// Not shown here.

IGXTable *table = NULL;
hr = conn->GetTable("Products", &table);
if (hr == NOERROR &&
    table)
{
    // Stream back column information.
    //
    StreamResult("<h2>Products Table:</h2>");
    hr = table->EnumColumnReset();
    if (hr == NOERROR)
    {
        while (TRUE)
        {
            IGXColumn *column = NULL;
            hr = table->EnumColumns(&column);
            if (hr == NOERROR &&
                column)
            {
                char buffer[256];
                buffer[0] = '\0';

                column->GetName(buffer, sizeof(buffer));
                StreamResult("Column Name = ");
                StreamResult(buffer);
                StreamResult(", ");

                DWORD type;
                type = 0;
                column->GetType(&type);
                sprintf(buffer, "Column Type = %d", type);
                StreamResult(buffer);
                StreamResult("<br>");

                column->Release();
            }
        }
    }
}

```

```

        }
        else
        {
            // No more columns, exit loop.
            break;
        }
    }
}
table->Release();
}

```

## Related Topics

GetColumn() or GetColumnByOrd() in the IGXHierResultSet interface

GetColumn(), GetColumnByOrd(), or EnumColumns() in the IGXTable interface

GetColumn(), GetColumnByOrd(), or EnumColumns() in the IGXResultSet interface

---

## GetName()

Returns the name of the column or alias.

**Syntax** HRESULT GetName(  
     LPSTR pBuff,  
     ULONG nBuff);

**pBuff** Buffer allocated by the client to hold the zero-terminated string that contains the returned column name or alias.

**nBuff** Length of the buffer allocated by the client for the returned column name or alias.

**Usage** Use GetName() when the name of the column is unknown and is required for subsequent operations.

**Tips**

- For computed columns in a query, specify aliases so that using GetName() returns the alias name. Otherwise, the column can be identified only by ordinal position.

- Do not rely on the case of the returned name. It might be all uppercase or mixed case, depending on the database.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** The following example shows how to iterate through a table to get the names of columns:

```

HRESULT hr;
IGXDataConn *conn;

// Retrieve connection with CreateDataConn().
// Not shown here.

IGXTable *table = NULL;
hr = conn->GetTable("Products", &table);
if (hr == NOERROR &&
    table)
{
    // Stream back column names.
    //
    StreamResult("<h2>Products Table:</h2>");
    hr = table->EnumColumnReset();
    if (hr == NOERROR)
    {
        while (TRUE)
        {
            IGXColumn *column = NULL;
            hr = table->EnumColumns(&column);
            if (hr == NOERROR &&
                column)
            {
                char buffer[256];
                buffer[0] = '\0';

                column->GetName(buffer, sizeof(buffer));
                StreamResult("Column Name = ");
                StreamResult(buffer);
                StreamResult(", ");

                column->Release();
            }
            else

```

```

        {
            break;
        }
    }
}

```

**Related Topics** “Getting Information About Columns or Fields” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

---

## GetNullsAllowed()

Determines whether NULL values are allowed in the column.

**Syntax** HRESULT GetNullsAllowed(  
    BOOL \*pNullsAllowed);

**pNullsAllowed** Pointer to the variable that contains the returned boolean result.

**Usage** A column may require data values. Use GetNullsAllowed() if this information is unknown to determine, for subsequent operations, whether nulls are allowed or not.

**Tip** For numeric columns that allow NULLs, the value is usually zero (0) in the column if a NULL is inserted. For more information, see your database vendor’s documentation.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** The following example shows how to iterate through a result set and return the column names, as well as whether null values are allowed in each column:

```

HRESULT hr;
IGXResultSet *resultset;

// Perform query here to retrieve resultset, not shown,
// with IGXDataConn::ExecuteQuery.

StreamResult("<h2>ResultSet column information:</h2>");
hr = resultset->EnumColumnReset();
if (hr == NOERROR)
{
    while (TRUE)

```

```

{
    IGXColumn *column = NULL;
    hr = resultset->EnumColumns(&column);
    if (hr == NOERROR &&
        column)
    {
        char buffer[256];
        buffer[0] = '\0';

        column->GetName(buffer, sizeof(buffer));
        StreamResult("Column Name = ");
        StreamResult(buffer);
        StreamResult(", ");

        BOOL nullsAllowed;
        nullsAllowed = FALSE;
        column->GetNullsAllowed(&nullsAllowed);
        sprintf(buffer, "Nulls Allowed = %s", (nullsAllowed
            ? "TRUE" : "FALSE"));
        StreamResult(buffer);
        StreamResult(", ");
    }
    else
    {
        // No more columns; exit loop.
        break;
    }
}
}

```

**Related Topics** “Getting Information About Columns or Fields” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

## GetPrecision()

---

Returns the precision, which is the maximum length or maximum number of digits, of the column.

**Syntax** HRESULT GetPrecision(  
 ULONG \*pPrecision);

**pPrecision** Pointer to the variable that contains the returned precision, which represents the maximum length or maximum number of digits of the column.

**Usage** Use GetPrecision() when the precision of the column is unknown and is required for subsequent operations.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** The following example shows how to iterate through a result set and return the column names, as well as the precision value of each column:

```

HRESULT hr;
IGXResultSet *resultset;

// Perform query here to retrieve resultset, not shown,
// with IGXDataConn::ExecuteQuery.

StreamResult("<h2>ResultSet column information:</h2>");
hr = resultset->EnumColumnReset();
if (hr == NOERROR)
{
    while (TRUE)
    {
        IGXColumn *column = NULL;
        hr = resultset->EnumColumns(&column);
        if (hr == NOERROR &&
            column)
        {
            char buffer[256];
            buffer[0] = '\0';

            column->GetName(buffer, sizeof(buffer));
            StreamResult("Column Name = ");
            StreamResult(buffer);
            StreamResult(", ");

            ULONG precision;
            precision = 0;
            column->GetPrecision(&precision);
            sprintf(buffer, "Column precision = %d",precision);
            StreamResult(buffer);
            StreamResult("<br>");
        }
    }
}

```

```

        else
        {
            // No more columns; exit loop.
            break;
        }
    }
}

```

**Related Topics** “Getting Information About Columns or Fields” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

---

## GetScale()

Returns the scale, which is the number of digits to the right of the decimal point, of a column of type double.

**Syntax** HRESULT GetScale(  
           ULONG \*pScale);

**pScale** Pointer to the variable that contains the returned scale, which represents the fixed number of digits to the right of the decimal point.

**Usage** Use GetScale() when the scale of the column is unknown and is required for subsequent operations.

- Rules**
- Use GetScale() with numeric columns, including SQL DECIMAL, NUMERIC, and FLOAT data types.
  - The value returned from GetScale() depends on the data type of the column. For example, it returns zero (0) for integers. For more information, see your database server documentation.
  - For computed columns in a result set, the value returned from GetScale() depends on the data type of the evaluated expression.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Getting Information About Columns or Fields” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

## GetSize()

---

Returns the maximum length, in number of bytes, allowed for a value in this column.

**Syntax** HRESULT GetSize(  
    ULONG \*pSize);

**pSize** Pointer to the variable that contains the returned size, which represents the maximum length of the column.

**Usage** Use GetSize() when the maximum allowable length of the column is unknown and is required for subsequent operations. Note that GetSize() does not return the actual size of data in the column.

- Rules**
- The value returned from GetSize() depends on the data type of the column. For more information, see your database server documentation.
  - For computed columns in a result set, the value returned from GetSize() depends on the data type of the evaluated expression.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** The following example shows how to iterate through a result set and return the column names, as well as the maximum allowable length of each column:

```

HRESULT hr;
IGXResultSet *resultset;

// Perform query here to retrieve resultset, not shown,
// with IGXDataConn::ExecuteQuery.

StreamResult("<h2>ResultSet column information:</h2>");
hr = resultset->EnumColumnReset();
if (hr == NOERROR)
{
    while (TRUE)
    {
        IGXColumn *column = NULL;
        hr = resultset->EnumColumns(&column);
        if (hr == NOERROR &&
            column)
        {
            char buffer[256];

```

```

        buffer[0] = '\0';

        column->GetName(buffer, sizeof(buffer));
        StreamResult("Column Name = ");
        StreamResult(buffer);
        StreamResult(", ");

        ULONG size;
        size = 0;
        column->GetSize(&size);
        sprintf(buffer, "Max Size = %d", size);
        StreamResult(buffer);
        StreamResult(", ");
    }
    else
    {
        // No more columns; exit loop.
        break;
    }
}
}
}

```

**Related Topics** “Getting Information About Columns or Fields” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

## GetTable()

---

Returns the table object in which this column exists.

**Syntax** HRESULT GetTable(  
           IGXTable \*\*ppTable);

**ppTable** Pointer to the returned IGXTable object that contains the table definition associated with this column. When AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use GetTable() when the table definition of the column is unknown and is required for subsequent operations. For result set columns, this method returns a table object, which is a description of the columns in the result set.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** // Walk through all columns in a resultset and stream

```

// back column information.

HRESULT hr;
IGXResultSet *resultset;

// Perform query here to retrieve resultset, not shown,
// with IGXDataConn::ExecuteQuery.

StreamResult("<h2>ResultSet column information:</h2>");
hr = resultset->EnumColumnReset();
if (hr == NOERROR)
{
    while (TRUE)
    {
        IGXColumn *column = NULL;
        hr = resultset->EnumColumns(&column);
        if (hr == NOERROR &&
            {
                char buffer[256];
                buffer[0] = '\0';

                column->GetName(buffer, sizeof(buffer));
                StreamResult("Column Name = ");
                StreamResult(buffer);
                StreamResult(", ");

                // Get the table object in which this column exists
                IGXTable *table;
                table = NULL;
                if (column->GetTable(&table) == NOERROR &&
                    table)
                {
                    buffer[0] = '\0';
                    table->GetName(buffer, sizeof(buffer));
                    StreamResult("Column Table = ");
                    StreamResult(buffer);
                    StreamResult(", ");
                    table->Release();
                }
            }
        // Process other column information

```

**Related Topics**   IGXTable interface

“Getting Information About Columns or Fields” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

## GetType()

Returns the data type of the column.

**Syntax** HRESULT GetType(  
    DWORD \*pdwType);

**pdwType** Pointer to the variable that contains one of the following macro-defined constants (defined in `gxidata.h`), which represent SQL data types.

**Note:** Some SQL data types are combined under a single category of data types. For example, `GX_DA_TYPE_LONG` includes short and integer data types, as well as tiny, small, and big integers.

### Usage

| Variable                         | Description  |
|----------------------------------|--|
| <code>GX_DA_TYPE_ERROR</code>    | Error data type. See Appendix A, “Return Codes” for more information.  |
| <code>GX_DA_TYPE_BINARY</code>   | All binary data types, including binary large objects (BLOBs).   |
| <code>GX_DA_TYPE_DATETIME</code> | Timestamp (date and time) data type. See the <code>GXDATETIME</code> struct in Chapter 5, “C++ Macros and Structures.” |
| <code>GX_DA_TYPE_DATE</code>     | Date data type.  |
| <code>GX_DA_TYPE_TIME</code>     | Time data type.  |
| <code>GX_DA_TYPE_DOUBLE</code>   | Double and related data types, including real, float, and decimal data types.  |
| <code>GX_DA_TYPE_LONG</code>     | Long and related data types, including int.  |
| <code>GX_DA_TYPE_STRING</code>   | String and related data types, including char and variable-length strings.   |

Use `GetType()` when the data type of the column is unknown and is required for subsequent operations.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** “Getting Information About Columns or Fields” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.



# IGXDataConn interface

The IGXDataConn interface represents a connection to a relational data source. IGXDataConn provides methods for preparing a query, executing a query, identifying table(s) to work with, and closing the connection explicitly. In addition, the data connection object is used in other operations for interacting with a data source.

IGXDataConn is part of the Data Access Engine (DAE) service. To create an instance of the IGXDataConn interface, use `CreateDataConn()` in the `GXAppLogic` class.

## Include File

`gxidata.h`

## Methods

| Method                        | Description   |
|-------------------------------|---|
| <code>CloseConn()</code>      | Explicitly closes a database connection.  |
| <code>CreateTrigger()</code>  | Creates a new trigger object in the specified table.  |
| <code>DisableTrigger()</code> | Disables a trigger associated with a specified table. This feature is supported by Oracle databases only. |
| <code>DropTrigger()</code>    | Removes a trigger from a specified table.   |
| <code>EnableTrigger()</code>  | Enables a trigger for a specified table. This feature is supported by Oracle databases only.              |
| <code>ExecuteQuery()</code>   | Executes a flat query on the data connection.   |
| <code>GetConnInfo()</code>    | Returns database and user information about the current database connection.                              |
| <code>GetConnProps()</code>   | Returns registry information about the current database connection.                                       |
| <code>GetDriver()</code>      | Returns the identifier of the data source driver that the current database connection is using.           |
| <code>GetTable()</code>       | Returns the table definition object for the specified table.  |

| Method         | Description   |
|----------------|---|
| GetTables()    | Returns an IGXVallist of database tables or views that are available to the specified user. |
| PrepareCall()  | Creates an IGXCallableStmt object that contains a call to a stored procedure.               |
| PrepareQuery() | Prepares a flat query object for subsequent execution.                                      |
| SetConnProps() | Specifies registry values for the current database connection.                              |

## Related Topics

CreateDataConn() in the GXAppLogic class

GetDataConn() in the IGXTable interface

AddRow(), DeleteRow(), and UpdateRow() in the IGXTable interface

IGXSequence interface

“Running Hierarchical Queries” in Chapter 6, “Querying a Database” in *Programmer’s Guide*.

“Inserting Records in a Database,” “Updating Records in a Database,” and “Deleting Records From a Database” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

## CloseConn()

Explicitly closes the database connection.

**Syntax** HRESULT CloseConn(  
    DWORD dwFlags);

**dwFlags** Specify 0, or GX\_DA\_UNBIND\_TRANS, which explicitly unbinds a physical connection from a transaction.

**Usage** The Data Access Engine performs certain housekeeping tasks, such as shutdown and cleanup, automatically and intermittently. Use CloseConn() to explicitly close a database connection and release system resources, such as

when memory is low. Calling `CloseConn()` breaks the virtual connection to the database and puts the physical connection back into the database connection cache.

- Rules**
- Closing the database connection changes the state of the `IGXDataConn` object to closed.
  - Close a database connection only after the `AppLogic` no longer needs it. A run-time error will occur if subsequent operations attempt to use a data connection object that has already been closed.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `CreateDataConn()` in the `GXAppLogic` class

“About Database Connections” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

---

## CreateTrigger()

Creates a new trigger object in the specified table.

**Syntax**

```
HRESULT CreateTrigger(
    LPSTR pTable,
    LPSTR pName,
    LPSTR pCondition,
    LPSTR pOptions,
    LPSTR pSQLBlock);
```

**pTable** The table on which the trigger is defined. You can specify the name of the owner as a prefix to the table name, for example, "jim.myTable".

**pName** The name of the trigger object to create.

**pCondition** The condition that determines whether or not the SQL procedure (defined in the `pSQLBlock` parameter) executes. For example, you can specify that the SQL procedure executes only if a column contains a specific value:

```
"FOR EACH ROW WHEN(city = 'San Francisco')"
```

**pOptions** The row operations that determine when the trigger executes. For example, you can specify that the trigger be activated BEFORE or AFTER an INSERT, UPDATE, and/or DELETE operation:

```
"AFTER INSERT, UPDATE"
```

**pSQLBlock** The definition of the SQL block to execute when the trigger goes into effect. Refer to your database documentation for information on the SQL block format.

**Usage** A trigger is a SQL procedure associated with a table. It is automatically activated when a specified row operation, such as INSERT, UPDATE, and DELETE, is issued against the table. Use `CreateTrigger()` to specify the table and the data modification command that should activate the trigger, and the action or actions the trigger is to take.

- Tips**
- For specific information on supported trigger options and conditions, refer to the description of triggers in your database documentation.
  - After creating a trigger, enable it by calling `EnableTrigger()`. The following are exceptions to the rule:
    - Sybase does not support the enabling or disabling of triggers.
    - Oracle automatically enables a trigger when the trigger is created; you can optionally call `EnableTrigger()`, but it will have no effect.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
IGXDataConn *conn = NULL;
HRESULT hr;

hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL, &conn);
if (hr == NOERROR &&
    conn)
{
    hr = conn->CreateTrigger("employees", "ProcessNew",
        "FOR EACH ROW WHEN(title='Director')",
        "AFTER INSERT",
        "[SQL instruction here]");
    if (hr == NOERROR)
    {
        conn->EnableTrigger("employees", "ProcessNew");
    }
    conn->Release();
}
```

**Related Topics** `DisableTrigger()`

`DropTrigger()`

EnableTrigger( )

---

## DisableTrigger( )

Disables a trigger associated with a specified table. This feature is supported by Oracle databases only.

**Syntax** HRESULT DisableTrigger(  
     LPSTR pTable,  
     LPSTR pName);

**pTable** The table in which the trigger is located.

**pName** The name of the trigger to disable.

**Usage** Use DisableTrigger( ) to temporarily stop a trigger from being activated. The trigger is disabled until it is enabled with EnableTrigger( ). To remove a trigger from a table permanently, use DropTrigger( ).

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateTrigger( )  
 DropTrigger( )  
 EnableTrigger( )

---

## DropTrigger( )

Removes a trigger from a specified table.

**Syntax** HRESULT DropTrigger(  
     LPSTR pTable,  
     LPSTR pName);

**pTable** The table on which the trigger is defined.

**pName** The name of the trigger to remove.

**Usage** Use DropTrigger( ) to delete a trigger that is no longer required. To temporarily stop a trigger from being activated, use DisableTrigger( ).

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateTrigger( )

DisableTrigger()

EnableTrigger()

---

## EnableTrigger()

Enables a trigger for a specified table. This feature is supported by Oracle databases only.

**Syntax** HRESULT EnableTrigger(  
     LPSTR pTable,  
     LPSTR pName);

**pTable** The table on which the trigger is defined.

**pName** The name of the trigger to enable.

**Usage** Use EnableTrigger() to prepare a specified trigger for activation. Call EnableTrigger() after you create a trigger with CreateTrigger(), or to enable a trigger that was disabled with DisableTrigger().

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** CloseConn()

**Related Topics** CreateTrigger(); DisableTrigger()

DropTrigger()

---

## ExecuteQuery()

Executes a flat query on the data connection.

**Syntax** HRESULT ExecuteQuery(  
     DWORD dwFlags,  
     IGXQuery \*pQuery,  
     IGXTrans \*pTrans,  
     IGXValList \*pProps,  
     IGXResultSet \*\*ppResultSet);

**dwFlags** Specifies flags used to execute this query.

- For synchronous operations, the default, specify zero or GX\_DA\_EXEC\_SYNC.

- For asynchronous operations, specify `GX_DA_EXEC_ASYNC`.
- To activate result set buffering, specify `GX_DA_RS_BUFFERING`.

The `AppLogic` can pass both result set buffering and either synchronous or asynchronous queries as the `flags` parameter, as shown in the following example:

```
(GX_DA_EXEC_ASYNC | GX_DA_RS_BUFFERING)
```

**pQuery** Pointer to the `IGXQuery` object that contains the flat query object to execute.

**pTrans** Pointer to the `IGXTrans` object that contains the transaction to which this query applies, or `NULL`.

**pProps** Pointer to the `IGXValList` object that contains query properties, or `NULL` for no properties. After instantiating an object of the `IGXValList` interface, set any of the following properties:

- `RS_BUFFERING` turns on result set buffering when set to “TRUE”.
- `RS_INIT_ROWS` specifies the initial size of the buffer, in number of rows. If the result set size exceeds this setting, a `FetchNext()` call will return the error `GX_DA_BUFFER_EXCEEDED`.
- `RS_MAX_ROWS` specifies the maximum number of rows for the buffer. If the result set size exceeds this setting, a `FetchNext()` call will return the error `GX_DA_BUFFER_EXCEEDED`.
- `RS_MAX_SIZE` specifies the maximum number of bytes for the buffer.

If `RS_BUFFERING` is enabled and if the optional parameters are not specified, the global values in the registry are used instead.

**ppResultSet** Pointer to the `IGXResultSet` object that contains the returned result of the query. When the `AppLogic` is finished using the object, and *after* calling the `CloseConn()` method, call the `Release()` method to release the interface instance.

- Rules**
- Before calling `ExecuteQuery()`, `AppLogic` must create a query by first calling `CreateQuery()` in the `GXAppLogic` class to create the `IGXQuery` object, then using methods in the `IGXQuery` interface to define the query.

- If the query is part of a transaction, before calling `ExecuteQuery()`, the `AppLogic` must first create the `IGXTrans` transaction object using `CreateTrans()` in the `GXAppLogic` class, then begin the transaction using `Begin()` in the `IGXTrans` interface, and then specify the `IGXTrans` object as a parameter when calling `ExecuteQuery()`.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

```
// Create a vallist for loadQuery() parameters
IGXValList *pList=GXCreateValList();

if(pList) {
    GXSetValListString(pList, "ssn", pSsn);

    // Load the query from the query file
    IGXQuery *pQuery=NULL;
    if(((hr=LoadQuery(SelCustAccts.gxq", "SelCustAccts", 0, pList,
        &pQuery))==GXE_SUCCESS)&&pQuery) {

        // Execute the query
        IGXResultSet *pRset=NULL;
        if(((hr=pConn->ExecuteQuery(0, pQuery, NULL, NULL,
            &pRset))==GXE_SUCCESS)&&pRset) {

            // Process the result set
```

**Related Topics** `CreateDataConn()` in the `GXAppLogic` class

`IGXQuery` interface

`IGXResultSet` interface

`IGXTrans` interface

`IGXValList` interface

“About Database Connections” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

## GetConnInfo()

Returns database and user information about the current database connection.

**Syntax** `HRESULT GetConnInfo(`

```
IGXValList **ppConnInfo);
```

**ppConnInfo** A pointer to the IGXValList object that contains the returned connection information. When the client code is finished using the object, call the Release() method to release the interface instance.

**Usage** When the client code calls the CreateDataConn() method in the GXAppLogic class to create a connection between the client and the specified database, it passes the following parameters: flags, driver, datasource, database, username, and password. Once a data connection has been established, you can call GetConnInfo() to return the datasource, database, user, and password values.

**Tip** To return the driver value, use GetDriver().

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

---

## GetConnProps()

Returns registry information about the current database connection.

**Syntax** HRESULT GetConnProps(  
IGXValList \*\*ppProps);

**ppProps** A pointer to the IGXValList object that contains the returned connection information. When the client code is finished using the object, call the Release() method to release the interface instance.

**Usage** Use GetConnProps() to get database connection information that the Netscape Application Server administrator set through the Enterprise Administrator. The information is returned in an IGXValList object that contains the following keys and values:

| Key                  | Value  |
|----------------------|--|
| "cache_free_entries" | An integer indicating the number of slots set for free connections.              |
| "cache_alloc_size"   | An integer indicating the initial number of slots in the connection cache.       |
| "conn_db_vendor"     | A string that identifies the database vendor, for example, "Oracle" or "Sybase." |

The `GetConnProps()` method might return other information depending on the database being used.

Applications typically use the database vendor information in conditional code that executes differently depending on the type of database.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** `SetConnProps()`

---

## GetDriver()

Returns the identifier of the data source driver that the current database connection is using.

**Syntax** `public int getDriver()`

```
HRESULT GetDriver(
    DWORD *pdwDriver);
```

**pdwDriver** Pointer to the variable that contains the returned driver information, which can be one of the following:

|                              |                            |
|------------------------------|----------------------------|
| GX_DA_DRIVER_ODBC            | GX_DA_DRIVER_SYBASE_CTLIB  |
| GX_DA_DRIVER_MICROSOFT_JET   | GX_DA_DRIVER_MICROSOFT_SQL |
| GX_DA_DRIVER_INFORMIX_SQLNET | GX_DA_DRIVER_INFORMIX_CLI  |
| GX_DA_DRIVER_INFORMIX_CORBA  | GX_DA_DRIVER_DB2_CLI       |
| GX_DA_DRIVER_ORACLE_OCI      | GX_DA_DRIVER_DEFAULT       |

**Usage** When the client code calls the `CreateDataConn()` method in the `GXAppLogic` class to create a connection between the client and the specified database, it passes the following parameters: flags, driver, datasource, database, username, and password. Once a data connection has been established, you can call various methods in the `IGXDataConn` interface to return the values that were passed to `CreateDataConn()`.

Call `GetDriver()` to return the driver information.

**Tip** To return the datasource, database, user, and password values, use `GetConnInfo()`.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

## GetTable()

---

Returns the table definition object for the specified table.

**Syntax** HRESULT GetTable(  
LPSTR szTable,  
IGXTable \*\*ppTable);

**szTable** Name of the table to request. This can include the schema name, for example, "jim.myTable." Do not use patterns or wildcards.

**ppTable** Pointer to the IGXTable object that contains the returned result of the query. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use GetTable() for the following reasons:

- To change data in the table using methods in the IGXTable interface to insert, update, and delete rows.
- When the schema of a table is unknown, to obtain information about the table definition from the database catalog, such as table name, table columns, data connection, and so on.

**Rule** The AppLogic usually calls GetTable() only once to obtain a table definition. Subsequent calls return a separate IGXTable object that represents the same table. Each AppLogic can call GetTable() and operate on its own copy of the table definition.

- Tips**
- If the table name is unknown, use GetTables() to retrieve an IGXValList of tables in the data source, then use methods in the IGXValList interface and the GXVAL struct to iterate through the table names obtained and determine which table to retrieve.
  - To obtain additional information about individual columns, use the IGXColumn interface.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

```
// Create the data connection
IGXDataConn *pConn=NULL;

if(((hr=GetOBDataConn(&pConn))==GXE_SUCCESS)&&pConn) {
    IGXTable *pTable=NULL;
```

```

// Get the table
if(((hr=pConn->GetTable("OBTransaction",
&pTable))==GXE_SUCCESS)&&pTable) {

    // Look up the column ordinals for the table
    ULONG transTypeCol=0;
    pTable->GetColumnOrdinal("transType", &transTypeCol);
    ULONG postDateCol=0;
    pTable->GetColumnOrdinal("postDate", &postDateCol);
    ULONG acctNumCol=0;
    pTable->GetColumnOrdinal("acctNum", &acctNumCol);
    ULONG amountCol=0;
    pTable->GetColumnOrdinal("amount", &amountCol);
}

```

**Related Topics** IGXTable interface

CreateDataConn() in the GXAppLogic class

GXVAL struct

IGXValList interface

“About Database Connections” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

## GetTables()

---

Returns an IGXValList of database tables or views that are available to the specified user.

**Syntax**

```

HRESULT GetTables(
    LPSTR szQualifier,
    LPSTR szOwner,
    LPSTR szTable,
    IGXValList **ppTableList);

```

**szQualifier** Specify NULL. Driver-dependent.

**szOwner** Specify NULL, or a schema name, which returns tables for that schema.

**szTable** Table or view name with wildcards, or NULL for all tables. Wildcards must be in the format supported by the data source. For example, you can use search patterns using the following characters:

- underscore (\_) for single characters
- percent sign (%) for any sequence of zero or more characters

**ppTableList** Pointer to the IGXValList object that contains the returned list of table names. When AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use GetTables() when the list of available tables on the data source is unknown. The AppLogic can obtain a subset of available tables by specifying wildcards in the table name.

- Rules**
- The AppLogic must be logged in with sufficient privileges to obtain a list of tables from the database. For more information, see your database server documentation.
  - The AppLogic must specify a valid table name, view name, or name pattern. Aliases and synonyms are not supported for security reasons.

**Tip** Use methods in the IGXValList interface and the GXVAL struct to iterate through the table names obtained and determine which table(s) to work with. Thereafter, use CreateDataConn() to access each table.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** IGXTable interface

CreateDataConn() in the GXAppLogic class

IGXValList interface

“About Database Connections” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

## PrepareCall()

---

Creates an IGXCallableStmt object that contains a call to a stored procedure.

**Syntax**

```
HRESULT PrepareCall(
    DWORD dwFlags,
    IGXQuery *pQuery,
    IGXTrans *pTrans,
    IGXValList *pProps,
    IGXCallableStmt **ppCall);
```

**dwFlags** Specify 0.

**pQuery** Pointer to the IGXQuery object that contains the call to a stored procedure. The stored procedure call should have been specified with the SetSQL( ) method in the IGXQuery interface.

**pTrans** Pointer to an IGXTrans object that contains the transaction associated with this callable statement, or NULL for no transaction. This same IGXTrans object must then be passed to the Execute( ) method of the IGXCallableStmt interface.

**pProps** Specify NULL.

**ppCall** Pointer to the returned IGXCallableStmt object. When the AppLogic is finished using the object, call the Release( ) method to release the interface instance.

**Usage** Use PrepareCall( ) to create a IGXCallableStmt object that contains a call to a stored procedure. After creating the callable statement, run it by calling Execute( ) in the IGXCallableStmt interface.

- Rules**
- Before calling PrepareCall( ), the AppLogic must create a query by first calling CreateQuery( ) in the GXAppLogic class to create the IGXQuery object, then using the SetSQL( ) method in the IGXQuery interface to define the call to a stored procedure.
  - When accessing a stored procedure on Sybase or MS SQL Server, input parameter names must be prefixed with the ampersand (&) character, for example, &param1. Other database drivers accept the ampersand, as well as, the colon (:) character. For all database drivers, input/output and output parameter names are prefixed with the colon (:) character, for example, :param2.

**Example**

```
IGXValList *conn_params;

// Set connection parameters
conn_params = GXCreateValList();
conn_params->SetValString("DSN", "salesDB");
conn_params->SetValString("DB", "salesDB");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");

IGXDataConn *conn = NULL;
HRESULT hr;
```

```

// Create the data connection
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL, &conn);
if (hr == NOERROR &&
    conn)
{
    // Create query that contains the call to the
    // stored procedure
    IGXQuery *qry = NULL;

    hr = CreateQuery(&qry);
    if (hr == NOERROR &&
        qry)
    {
        qry->SetSQL("{:ret = call myFunction(:param1)}");
        IGXCallableStmt *s = NULL;

        // Prepare the callable statement for execution
        hr = conn->PrepareCall(0, qry, NULL, NULL, &s);
        if (hr == NOERROR &&
            s)

            // Set parameters and run callable statement

```

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** IGXCallableStmt interface

---

## PrepareQuery()

Prepares a flat query object for subsequent execution.

**Syntax** HRESULT PrepareQuery(  
 DWORD dwFlags,  
 IGXQuery \*pQuery,  
 IGXTrans \*pTrans,  
 IGXValList \*pProps,  
 IGXPreparedQuery \*\*ppPQuery);

**dwFlags** Specify 0.

**pQuery** Pointer to the IGXQuery object that contains the query or statement to execute.

**pTrans** Pointer to the IGXTrans object that contains the transaction to which this query applies, or NULL. This same **Include File** object must then be passed to the Execute() method of the IGXPreparedQuery interface.

**pProps** Specify NULL.

**ppPQuery** Pointer to the IGXPreparedQuery object that contains the returned prepared query. When AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use PrepareQuery() to prepare the query, then execute the prepared query using Execute() in the IGXPreparedQuery interface. An application can also use PrepareQuery() with result set buffering to pre-fetch result set data efficiently from a back-end database.

**Rule** Before calling PrepareQuery(), AppLogic must create a query by first calling CreateQuery() in the GXAppLogic class to create the IGXQuery object, then using methods in the IGXQuery interface to define the query.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

```
// Create an Insert query
IGXQuery *pUserQuery=NULL;
if((hr>CreateQuery(&pUserQuery))==GXE_SUCCESS)&&pUserQuery) {
    pUserQuery->SetSQL("INSERT INTO OBUser(userName, password, userType,
        eMail) VALUES (:userName, :password, :userType, :eMail)");

// Create another Insert query
IGXQuery *pAcctQuery=NULL;
if((hr>CreateQuery(&pAcctQuery))==GXE_SUCCESS)&&pAcctQuery) {
    pAcctQuery->SetSQL("INSERT INTO OBAccount VALUES (:acctNum, :ssn,
        :acctType, :balance)");

// Create the data connection and prepared query objects
IGXDataConn *pConn=NULL;

if((hr=GetOBDataConn(&pConn))==GXE_SUCCESS)&&pConn) {
    IGXPreparedQuery *pUserPQuery=NULL;
    IGXPreparedQuery *pAcctPQuery=NULL;

// Create prepared queries
pConn->PrepareQuery(0, pUserQuery, NULL, NULL, &pUserPQuery);
pConn->PrepareQuery(0, pAcctQuery, NULL, NULL, &pAcctPQuery);
```

**Related Topics** IGXPreparedQuery interface  
 IGXQuery interface  
 IGXTrans interface  
 IGXValList interface  
 CreateDataConn() in the GXAppLogic class  
 “About Database Connections” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

---

## SetConnProps()

Specifies registry values for the current database connection.

**Syntax** HRESULT SetConnProps(  
           IGXValList \*pProps);

**pProps** A pointer to the IGXValList object that contains the connection properties to set in the registry. Use the following defined key names for the connection properties:

| Key                  | Value  |
|----------------------|--|
| "cache_free_entries" | An integer indicating the number of slots set for free connections.        |
| "cache_alloc_size"   | An integer indicating the initial number of slots in the connection cache. |

**Usage** Use SetConnProps() to override database connection properties that the Netscape Application Server administrator set through the Enterprise Administrator. To get the current connection properties programmatically, call GetConnProps().

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** GetConnProps()



# IGXDataConnSet interface

The IGXDataConnSet interface represents a collection of data connections and associated query names. It is used in conjunction with loading a query file.

Use IGXDataConnSet when loading a hierarchical query from a file. The AppLogic first establishes a data connection with each database on which any queries will be run. Next, the AppLogic calls CreateDataConnSet( ) in the GXAppLogic class to create an empty IGXDataConnSet object, then populates this object with query name / data connection pairs.

In this way, the AppLogic can use parameterized queries and select and assign data connections dynamically at runtime. Finally, the AppLogic calls LoadHierQuery( ) in the GXAppLogic class to create the hierarchical query object.

IGXDataConnSet is part of the Data Access Engine (DAE) service.

To create an instance of the IGXDataConnSet interface, use CreateDataConnSet( ) in the GXAppLogic class.

## Include File

gxidata.h

## Methods

|            |   |
|------------|---|
| AddConn( ) | Associates a query name with a data connection object and adds it to the IGXDataConnSet object. |
|------------|---|

## Related Topics

CreateDataConnSet( ) in the GXAppLogic class

“About Database Connections” in Chapter 5, “Working with Databases” in the *Programmer’s Guide*.

## AddConn()

---

Associates a query name with a data connection object and adds it to the IGXDataConnSet object.

**Syntax** HRESULT AddConn(  
LPSTR pQueryName,  
IGXDataConn \*pConn);

**pQueryName** Name of a query in the query file.

**pConn** Name of the data connection object representing an active connection with the data source on which the query will be run.

- Rules**
- Every named query in the query file must have a corresponding named query in the IGXDataConnSet object.
  - The AppLogic must first create the data connection object using CreateDataConn() in the GXAppLogic class.
  - Duplicate query names are not permitted.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateDataConnSet() in the GXAppLogic class

“About Database Connections” in Chapter 5, “Working with Databases” in the *Programmer’s Guide*.

# IGXEnumObject interface

The IGXEnumObject interface represents an enumeration object that contains IGXObject instances. Some methods that return a list of objects, such as EnumEvents() in the IGXAppEventMgr interface, return an IGXEnumObject object.

The IGXEnumObject interface defines methods for counting and accessing the IGXObject instances in an IGXEnumObject.

## Include File

gxienum.h

## Methods

| Method      | Description  |
|-------------|--|
| EnumCount() | Returns the number of IGXObject instances in an IGXEnumObject. |
| EnumNext()  | Returns the next IGXObject instance in an IGXEnumObject.       |
| EnumReset() | Resets to the first IGXObject instance in an IGXEnumObject.    |

## Related Topics

EnumEvents() in the IGXAppEventMgr interface

## EnumCount()

Returns the number of IGXObject instances in an IGXEnumObject.

**Syntax** HRESULT EnumCount(  
    ULONG \*pCount);

**pCount** Pointer to the variable that contains the returned number of IGXObject instances in the IGXEnumObject.

**Usage** Use EnumCount() to determine the number of objects to process before iterating through the IGXObject instances in the IGXEnumObject.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** In the following example, EnumEvents() returns all the application events registered with the Netscape Application Server in an IGXEnumObject. The EnumCount() method is used in conjunction with EnumNext() and EnumReset() to access objects in the IGXEnumObject.

```
IGXEnumObject *pEObjs = NULL;
ULONG ulCount = 0;

// suppose pAppEventMgr has a valid reference to IGXAppEventMgr object

// Get the Enumeration object for all registered appevents
hr = pAppEventMgr->EnumEvents(&pEObjs);

// Retrieve the count of registered appevents
hr = pEObjs->EnumCount(&ulCount);

fprintf(fp, "Number of Registered Events: %d\n", ulCount);

// Reset the next enumeration object to be the first instance
hr = pEObjs->EnumReset(0);

// Iterate through all the enumeration instances
while (ulCount--) {
// Process the objects
}
```

**Related Topics** EnumEvents() in the IGXAppEventMgr interface

EnumNext()

EnumReset()

## EnumNext()

Returns the next IGXObject instance in an IGXEnumObject.

**Syntax** HRESULT EnumNext(  
IGXObject \*\*ppNext);

**ppNext** Pointer to the returned IGXObject object. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use EnumNext() in conjunction with EnumCount() and EnumReset() to iterate through an IGXEnumObject.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** In the following example, EnumEvents() returns all the application events registered with the Netscape Application Server in an IGXEnumObject. The EnumNext() method is used in conjunction with EnumCount() and EnumReset() to access objects in the IGXEnumObject.

```
// Retrieve the count of registered appevents
hr = pEObjs->EnumCount(&ulCount);

// Reset to the first object
hr = pEObjs->EnumReset(0);

// Iterate through all the enumeration instances
while (ulCount-->0) {
    IGXObject *pObj = NULL;

    // Get the next instance
    hr = pEObjs->EnumNext(&pObj);
    if ((hr != NOERROR) || (pObj == NULL)) {
        pEObjs->Release();
        return StreamResult("EnumNext failed!<br>");
    }

    // Make sure the object supports the IGXAppEventObj
    // interface (it should)
    IGXAppEventObj* pAEObj = NULL;
    hr = pObj->QueryInterface(IID_IGXAppEventObj, (LPVOID *)&pAEObj);
    pObj->Release();
    if ((hr != NOERROR) || (pAEObj == NULL)) {
        pAEObj->Release();
        return StreamResult("QueryInterface on EnumNext Obj
failed!<br>");
    }

    // Process the objects
```

```

        // Release when done.
        pAEObj->Release();
    }

```

**Related Topics** EnumEvents() in the IGXAppEventManager interface

EnumCount()

EnumReset()

---

## EnumReset()

Resets to the first IGXObject instance in an IGXEnumObject.

**Syntax** HRESULT EnumReset(  
    DWORD dwFlags);

**dwFlags** Specify 0.

**Usage** Use EnumReset() before iterating through an IGXEnumObject. Doing so ensures that iteration begins at the first IGXObject instance in the IGXEnumObject.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

### Example

```

// Retrieve the count of objects in the IGXEnumObject
hr = pEObjs->EnumCount(&ulCount);

// Reset the next enumeration object to be the first instance
hr = pEObjs->EnumReset(0);

// Iterate through all the enumeration instances
while (ulCount-- > 0) {
    // Process the objects
}

```

**Related Topics** EnumEvents() in the IGXAppEventManager interface

EnumCount()

EnumNext()

# IGXError interface

The IGXError interface represents an error code object that consists of a code and a corresponding error message that originates from a facility, such as an operating system or a database. In this release, IGXError handles database errors only.

Use the methods in the IGXError interface to get error codes and messages returned by a database.

The IGXError interface is implemented by the IGXDataConn object. To use it, cast IGXDataConn to the IGXError interface, as shown in the following example:

```
IGXDataConn *conn;
IGXError *error;
conn->QueryInterface(IID_IGXError, (LPVOID *) &error);
```

## Include File

gxierror.h

## Methods

| Method             | Description   |
|--------------------|---|
| GetErrorCode()     | Returns the current error code as a string.                         |
| GetErrorCodeNum()  | Returns the current error code as a number.                         |
| GetErrorMessage()  | Returns the message associated with the current error code.         |
| GetErrorFacility() | Returns a description of the facility that generated an error code. |

## GetErrorCode()

Returns the current error code as a string.

**Syntax**

```
HRESULT getErrorCode(
    LPSTR pCode,
    ULONG nSize);
```

**pCode** Pointer to the buffer allocated by the client to store the returned error code.

**nSize** The size of the buffer to store the error code. 256 bytes is usually sufficient. If the error code string exceeds the specified size, it is truncated.

**Usage** Use `GetErrorCode()` after a database operation, such as running a stored procedure or executing a query, to retrieve the error code for debugging or error-handling purposes. The following is an example of a returned error code: "ORA-03130".

**Tip** For ODBC, the error codes usually consist of the ODBC error code and the database error code separated by a space, for example, "S1000 1017". Sometimes just the ODBC error code, such as "S1000", is returned.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** `GetErrorCodeNum()`  
`GetErrorMessage()`  
`GetErrorFacility()`

---

## GetErrorCodeNum()

Returns the current error code as a number.

**Syntax** `HRESULT getErrorCodeNum(  
DWORD *nCode);`

**nCode** Pointer to the variable allocated by the client to store the returned error code.

**Usage** Use `GetErrorCodeNum()` after a database operation, such as running a stored procedure or executing a query, to retrieve the error code for debugging or error-handling purposes.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** `GetErrorCode()`  
`GetErrorMessage()`  
`GetErrorFacility()`

## GetErrorMessage()

---

Returns the message associated with the current error code.

**Syntax** `HRESULT getErrorCode(  
LPSTR pCode,  
LPSTR pMessage,  
ULONG nSize);`

**pCode** Specify NULL.

**pMessage** Pointer to the buffer allocated by the client to store the returned message.

**nSize** The size of the buffer to store the error message.

**Usage** Use `GetErrorMessage()` after a database operation, such as running a stored procedure or executing a query, to retrieve the message associated with the current error code. The AppLogic can then display the message to users. The following is an example of a returned error message: "[ODBC][Visigenic driver][S1000]Connection attempt failed".

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** `GetErrorCode()`  
`GetErrorCodeNum()`  
`GetErrorFacility()`

## GetErrorFacility()

---

Returns a description of the facility that generated an error code.

**Syntax** `HRESULT getErrorFacility(  
LPSTR pDescription,  
ULONG nSize);`

**pDescription** Pointer to the buffer allocated by the client to store the returned string description.

**nSize** The size of the buffer to store the string description. If the string exceeds the specified size, it is truncated.

**Usage** Use `GetErrorFacility()` after a database operation, such as running a stored procedure or executing a query, to get information on which driver generated the current error code. The following is an example of a description returned by `GetErrorFacility()`: "ODBC DAD".

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `GetErrorCode()`  
`GetErrorCodeNum()`  
`GetErrorMessage()`

# IGXHierQuery interface

The IGXHierQuery interface represents a hierarchical query. IGXHierQuery provides methods for retrieving hierarchical information organized in nested levels of detail, as in the following example:

```
Asia           170
  China        110
  Japan         60
Europe         80
  France        70
  Portugal     10
```

A hierarchical query combines multiple flat queries organized in cascading, parent-child relationships. Each query is an IGXQuery object containing data selection criteria. The IGXHierQuery object contains the definition of the hierarchical structure of parent-child relationships among IGXQuery objects.

To use a hierarchical query, the AppLogic first creates each individual query and defines its selection criteria. Next, it creates the IGXHierQuery object and calls AddQuery( ) repeatedly to add a child query to a parent query for each level of detail in the hierarchical query.

After the hierarchical query is constructed, the AppLogic calls its Execute( ) method to run the hierarchical query on the target data source and retrieve a hierarchical result set in an IGXHierResultSet object.

Alternatively, the AppLogic can load a hierarchical query stored in a file. For more information, see LoadHierQuery( ) and CreateDataConnSet( ) in the GXAppLogic class.

To create an instance of the IGXHierQuery interface, use CreateHierQuery( ) in the GXAppLogic class.

## Include File

gxidatap.h

## Methods

| Method     | Description   |
|------------|---|
| AddQuery() | Adds a child query to a parent query, defining an additional level of detail in the hierarchical query. |
| DelQuery() | Removes a child query from its parent query.  |
| Execute()  | Executes a hierarchical query and returns a hierarchical result set.                                    |

## Related Topics

CreateHierQuery() in the GXAppLogic class

“Writing Hierarchical Queries” in Chapter 6, “Querying a Database” in *Programmer’s Guide*.

## AddQuery()

---

Adds a child query to a parent query, defining an additional level of detail in the hierarchical query.

**Syntax**

```
HRESULT AddQuery(
    IGXQuery *pQuery,
    IGXDataConn *pConn,
    LPSTR szAlias,
    LPSTR szParent,
    LPSTR szJoin);
```

**pQuery** Pointer to the IGXQuery object that contains the flat query object to append as a child to the parent query.

**pConn** Pointer to the IGXDataConn object that contains the data connection where the child query will be executed. Each flat query in the hierarchical query can retrieve data from a different data source.

**szAlias** Name used to uniquely identify this child query in the query hierarchy. AppLogic must specify a child name that is unique within the hierarchical query.

**szParent** Name of the parent query to contain this child query. Use an empty string ("" ) for the highest level in the hierarchical query. When adding a child query to an existing parent query, the specified parent name must have already been specified in a previous AddQuery( ) call.

**szJoin** Join clause used to specify a join for this query, defining the relationship between a field in the child query and a field in the parent query. Use an empty string for the highest level in the hierarchical query. Use the following Netscape-compliant syntax for the join clause:

```
"ParentQuery.table.column='childQuery.table.column' "
```

Optionally, you can specify the schema:

```
"ParentQuery.schema.table.column='childQuery.schema.table.column' "
```

**Note** The only difference between the Netscape Application Server and SQL join syntax is that, with Netscape, you prepend the clause with the query name.

To refer to a field name in the parent query, include the parent query name before the field name, as shown in the following example, in which CITY is the name of the parent query:

```
HRESULT hr = hqr->AddQuery(pQryEMP, pConn, "EMP", "CITY",  
"EMP.employee.city = 'CITY.city'");
```

Use the AND and OR operators to specify additional join conditions. Use parentheses to specify the order of precedence in complex join criteria.

**Usage** Use AddQuery( ) when constructing the hierarchical query to define the hierarchical relationships among child and parent queries. The number of nested levels, and thus the number of AddQuery( ) calls, is limited only by system resources.

- Rules**
- The AppLogic must first create the data connection using CreateDataConn( ) in the GXAppLogic class.
  - The AppLogic must then create the specified child query using CreateQuery( ) in the GXAppLogic class. A separate child query must exist for every level of data.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

```
// Create the hier query  
IGXHierQuery *pHq=NULL;  
  
if(((hr>CreateHierQuery(&pHq))==GXE_SUCCESS)&&pHq) {
```

```
// Add a query
pHq->AddQuery(pQuery, pConn, "SelCusts", "", "");
```

**Related Topics** CreateHierQuery() in the GXAppLogic class

IGXQuery interface

IGXDataConn interface

“Writing Hierarchical Queries” in Chapter 6, “Querying a Database” in *Programmer’s Guide*.

## DelQuery()

---

Removes a child query from its parent query.

**Syntax** HRESULT DelQuery(  
LPSTR szName);

**szName** Name of the child query to remove.

**Usage** Use DelQuery() to remove a child query that is no longer needed. Any children of the deleted child query are also removed.

**Rule** The specified child query must exist in the hierarchical query.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateHierQuery() in the GXAppLogic class

“Writing Hierarchical Queries” in Chapter 6, “Querying a Database” in *Programmer’s Guide*.

## Execute()

---

Executes a hierarchical query and returns a hierarchical result set.

**Syntax** HRESULT Execute(  
DWORD dwFlags,  
DWORD dwTimeout,  
IGXValList \*pProps,  
IGXHierResultSet \*\*ppHierResultSet);

**dwFlags** Specifies flags used to execute this hierarchical query.

- For synchronous operations, the default, specify zero or `GX_DA_EXEC_SYNC`.
- For asynchronous operations, specify `GX_DA_EXEC_ASYNC`.
- To activate result set buffering, specify `GX_DA_RS_BUFFERING`.

The `AppLogic` can pass both result set buffering and either synchronous or asynchronous queries as the `flags` parameter, as shown in the following example:

```
(GX_DA_EXEC_ASYNC | GX_DA_RS_BUFFERING).
```

**dwTimeout** Specify 0 (zero).

**pProps** Pointer to the `IGXValList` object that contains query properties, or `NULL` for no properties. After instantiating an object of the `IGXValList` interface, set any of the following properties:

- `RS_BUFFERING` turns on result set buffering when set to “TRUE”.
- `RS_INIT_ROWS` specifies the initial size of the buffer, in number of rows. If the result set size exceeds this setting, a `FetchNext()` call will return the error `GX_DA_BUFFER_EXCEEDED`.
- `RS_MAX_ROWS` specifies the maximum number of rows for the buffer. If the result set size exceeds this setting, a `FetchNext()` call will return the error `GX_DA_BUFFER_EXCEEDED`.
- `RS_MAX_SIZE` specifies the maximum number of bytes for the buffer.

If `RS_BUFFERING` is enabled and if the optional parameters are not specified, the global values in the registry are used instead..

**ppHierResultSet** Pointer to the `IGXHierResultSet` object that contains the returned result of the hierarchical query. When the `AppLogic` is finished using the object, call the `Release()` method to release the interface instance.

**Usage** After constructing a hierarchical query using `AddQuery()`, the `AppLogic` uses `Execute()` to execute the query on the database server. Results are returned in a hierarchical result set.

**Rule** `AppLogic` must first construct the hierarchical query using `AddQuery()`.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** CreateHierQuery( ) in the GXAppLogic class

IGXHierResultSet interface

IGXValList interface

“Writing Hierarchical Queries” in Chapter 6, “Querying a Database” in *Programmer’s Guide*.

# IGXHierResultSet interface

The IGXHierResultSet interface represents a hierarchical result set retrieved by a hierarchical query. IGXHierResultSet provides methods to iterate through rows in the hierarchical result set and retrieve information about each row.

Alternatively, an AppLogic can process hierarchical result sets by passing them directly to the Template Engine using EvalOutput() in the GXAppLogic class.

IGXHierResultSet is part of the Data Processing Engine (DPE) service. To create an instance of IGXHierResultSet, use Execute() in the IGXHierQuery interface, as shown in the following example:

```
IGXHierResultSet *hrs = NULL;
HRESULT hr;
hr = hqry->Execute(0, 0, NULL, &hrs);
```

## Include File

gxidatap.h

## Methods

| Method               | Description   |
|----------------------|---|
| Count()              | Returns the total number of rows retrieved so far from the data source for the specified child query.         |
| GetColumn()          | Returns the column definition for the column with the specified name in the specified child query.            |
| GetColumnByOrd()     | Returns the column definition for the column in the specified ordinal position for the specified child query. |
| GetResultSet()       | Returns the result set for a specified child query.   |
| GetRowNumber()       | Returns the number of the current row for the specified child query in the hierarchical result set.           |
| GetValueDateString() | Returns the value of a Date type column, as a string, from the specified child query in the result set.       |
| GetValueDouble()     | Returns the value of a double type column from the specified child query in the result set.                   |

| Method           | Description   |
|------------------|---|
| GetValueInt()    | Returns the value of an int type column from the specified child query in the result set.   |
| GetValueString() | Returns the value of a string type column from the specified child query in the result set. |
| MoveNext()       | Moves to the next row for the specified child query in the result set.                      |
| MoveTo()         | Moves to the specified row for the specified child query in the result set.                 |

## Example

The following code runs a hierarchical query and with the returned hierarchical result set, checks a user's access level to determine which listbox options to display:

```
LPSTR templateName;
IGXDataConn *conn;
IGXQuery *qry;
LPSTR wantedUser;

// Not shown here, creation of data connection and creation
// of query of users.

IGXHierQuery *hqry = NULL;
CreateHierQuery(&hqry);
hqry->AddQuery(qry, conn, "USERS", "", "");

// Execute the hierarchical query.
IGXHierResultSet *hrs = NULL;
HRESULT hr;
hr = hqry->Execute(0, 0, NULL, &hrs);
if (hr == NOERROR && hrs)
{
    ULONG i;
    if (hrs->GetRowNumber("USERS", &i) == NOERROR &&
        i > 0)
    {
        // The current row is row 1, so there is at least
        // one user returned in the USERS sub-query.
    }
}
```

```

//
// The business logic here is to check the user's
// access level, and show different listbox options
// depending on the level.
//
LPSTR selAdmin;
LPSTR selNormal;
char access[64];
access[0] = '\0';
char buffer[1024];
buffer[0] = '\0';

hr = hrs->GetValueString("USERS", "AccessLevel", access,
    sizeof(access));
if (hr == NOERROR &&
    strcmp(access, "AccessAdmin") == 0)
    {
    selAdmin = "<option selected>AccessAdmin</option>";
    selNormal = "<option>Normal</option>";
    }
else
    {
    selAdmin = "<option>AccessAdmin</option>";
    selNormal = "<option selected>Normal</option>";
    }
sprintf(buffer,
"<select name=accessControlLevel>\n%s%s</select>",
    selAdmin,
    selNormal);

// We have a template map which we fill
// with dynamic values. The template should
// refer to these values in gx cell
// placeholders.
//
GXTemplateMapBasic *map = new GXTemplateMapBasic();
IGXBuffer *b;
b = GXCreateBufferFromString(buffer);
map->Put("ACCESS", b);
b->Release();
b = GXCreateBufferFromString(access);
map->Put("ACCESS_LEVEL", b);

```

```

        b->Release();
        hr = EvalOutput(templateName,
            (IGXTemplateData *) hrs,
            (IGXTemplateMap *) map,
            NULL, NULL);
        map->Release();
    }
    else
    {
        // No users returned in the USERS sub-query.
        //
        StreamResult("No user matches the login name: ");
        StreamResult(wantedUser);
    }
    hrs->Release();
}

```

## Related Topics

CreateHierQuery() in the GXAppLogic class

IGXHierQuery interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database” in *Programmer’s Guide*.

## Count()

---

Returns the total number of rows retrieved so far from the data source for the specified child query.

**Syntax** HRESULT Count(  
     LPSTR qryName,  
     ULONG \*nRows);

**qryName** Name of the child query that generated the result set.

**nRows** Pointer to the variable that contains the returned number of rows in the result set.

**Usage** Use Count() to return the current number of rows processed so far in the result set. If iterating through rows in a result set that has been completely returned, use Count() to determine the current maximum number of rows to process.

**Tip** If result set buffering is enabled, the AppLogic can use Count() to find the current number of rows in the buffer.

**Rule** The specified child query must exist in the result set.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateHierQuery() in the GXAppLogic class

IGXHierQuery interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database” in *Programmer’s Guide*.

---

## GetColumn()

Returns the column definition for the column with the specified name in the specified child query.

**Syntax**

```
HRESULT GetColumn(
    LPSTR qryName,
    LPSTR colName,
    IGXColumn **ppCol);
```

**qryName** Name of the child query that generated the result set.

**colName** Name of the column. Must *not* be qualified with the schema name or table name (if necessary, use column alias to ensure that the colName is unambiguous).

**ppCol** Pointer to the IGXColumn object that contains the returned column definition. When AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use GetColumn() when the data definition of the column is unknown and is required for subsequent operations. The AppLogic can then use methods in the IGXColumn interface to obtain descriptive information about a table column from the database catalog, such as the column name, precision, scale, size, table, and data type.

- Rules**
- The specified child query must exist in the result set.
  - The specified column name must exist in the result set.

- Tips**
- Use `GetColumnByOrd()` instead when the column position is known but its name is unknown.
  - Columns that are the result of query expressions or formulas, such as `invoice.count * product.price`, should have a column alias in the result set. `AppLogic` can call `SetFields()` in the `IGXQuery` interface to specify field aliases using the "as" keyword.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```
IGXHierResultSet *hrs = NULL;

// Not shown here, execution of hierarchical query
// that retrieves the hierarchical resultset.

IGXColumn *col = NULL;
HRESULT hr;
hr = hrs->GetColumn("INVOICES", "Date", &col);
if (hr == NOERROR &&col)
{
    // Call column methods, such as IGXColumn::GetName() here.

    col->Release();
}
```

**Related Topics** `CreateHierQuery()` in the `GXAppLogic` class

`IGXHierQuery` interface

`IGXColumn` interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database” in *Programmer’s Guide*.

## GetColumnByOrd()

---

Returns the column definition for the column in the specified ordinal position for the specified child query.

**Syntax**

```
HRESULT GetColumnByOrd(
    LPSTR qryName,
    ULONG colIndex,
    IGXColumn **ppCol);
```

**qryName** Name of the child query that generated the result set.

**collIndex** Ordinal position of a column in the result set. The ordinal position of the first column in the result set is 1, the second column is 2, and so on.

**ppCol** Pointer to the IGXColumn object that contains the returned column definition. When AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use GetColumnByOrd() when the data definition of the column is unknown and is required for subsequent operations. AppLogic can then use methods in the IGXColumn interface to obtain descriptive information about a table column from the database catalog, such as the column name, precision, scale, size, table, and data type.

**Rules**

- The specified child query must exist in the result set.
- The specified column position must exist in the result set.

**Tip** Use GetColumn() instead when the column name is known but its ordinal position is unknown.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** `IGXHierResultSet *hrs = NULL;`

```
// Not shown here, execution of hierarchical query
// that retrieves the hierarchical resultset.

IGXColumn *col = NULL;
HRESULT hr;
hr = hrs->GetColumnByOrd("INVOICES", 1, &col);
if (hr == NOERROR && col)
{
    // Call column methods, such as IGXColumn::GetName() here.

    col->Release();
}
```

**Related Topics** CreateHierQuery() in the GXAppLogic class

IGXHierQuery interface

IGXColumn interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database” in *Programmer’s Guide*.

---

## GetResultSet()

Returns the result set for a specified child query.

**Syntax** HRESULT GetResultSet(  
     LPSTR qryName,  
     IGXResultSet \*\*ppResultSet);

**qryName** Name of the child query that generated the result set to retrieve.

**ppResultSet** Pointer to the IGXResultSet object that contains the returned result set. When AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use GetResultSet() to retrieve and manipulate a particular child result set in the hierarchical result set. The AppLogic can then use methods in the IGXResultSet interface to get data from the result set columns.

**Rule** The specified child query must exist in the result set.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
// Look up list of customers matching criteria from database
IGXHierResultSet *pHRset=NULL;

if(((hr=LookupCustomer(pSsn, pLastName, pFirstName, pAcctNum,
    &pHRset))==GXE_SUCCESS)&&pHRset) {

    // Check the result set to see if any customers are found
    IGXResultSet *pRset=NULL;
    if(((hr=pHRset->GetResultSet("SelCusts",
        &pRset))==GXE_SUCCESS)&&pRset) {
```

**Related Topics** “Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database” in *Programmer’s Guide*.

---

## GetRowNumber()

Returns the number of the current row for the specified child query in the hierarchical result set.

**Syntax** HRESULT GetRowNumber(  
LPSTR qryName,  
ULONG \*pOrd);

**qryName** Name of the child query that generated the result set.

**pOrd** Pointer to the variable that contains the returned row number for the current row.

**Usage** When iterating through rows in a child set, use GetRowNumber() to keep track of the number of rows processed.

**Rule** The specified child query must exist in the result set.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateHierQuery() in the GXAppLogic class

IGXHierQuery interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database” in *Programmer’s Guide*.

---

## GetValueDateString()

Returns the value of a Date type column, as a string, from the specified child query in the result set.

**Syntax** HRESULT GetValueDateString(  
LPSTR qryName,  
LPSTR colName,  
LPSTR pVal,  
ULONG nVal);

**qryName** Name of the child query that generated the result set.

**colName** Name of the column from which to retrieve the date.

**pVal** Pointer to the variable that contains the returned column value.

**nVal** Length of the variable.

**Usage** Use GetValueDateString() to retrieve date values from the result set for subsequent processing. The following is an example of the format in which GetValueDateString() returns a date:

Jan 26 1998 12:35:00

**Rule** The specified column must be a Date, Date Time, or Time data type.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** `IGXHierResultSet *hrs = NULL;`

```
// Not shown here, execution of hierarchical query
// that retrieves the hierarchical resultset.
```

```
char buffer[256];
buffer[0] = '\0';
HRESULT hr;
hr = hrs->GetValueDateString("INVOICES", "ShipDate", buffer,
sizeof(buffer));
```

**Related Topics** `GetValueDouble()`

`GetValueInt()`

`GetValueString()`

## GetValueDouble()

---

Returns the value of a double type column from the specified child query in the result set.

**Syntax** `HRESULT GetValueDouble(
 LPSTR qryName,
 LPSTR colName,
 double *pVal);`

**qryname** Name of the child query that generated the result set.

**colName** Name of the column from which to retrieve the double value.

**pVal** Pointer to the variable that contains the returned column value.

**Usage** Use `GetValueDouble()` to retrieve decimal, floats, real, numeric, and double values from the result set for subsequent processing.

**Rule** The specified column must be a double data type.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** GetValueDateString()  
 GetValueInt()  
 GetValueString()

---

## GetValueInt()

Returns the value of an int type column from the specified child query in the result set.

**Syntax** HRESULT GetValueInt(  
     LPSTR qryName,  
     LPSTR colName,  
     ULONG \*pVal);

**qryname** Name of the child query that generated the result set.

**colName** Name of the column from which to retrieve the value.

**pVal** Pointer to the variable that contains the returned column value.

**Usage** Use GetValueInt() to retrieve int or long values from the result set for subsequent processing.

**Rule** The specified column must be an int or long data type.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** GetValueDateString()  
 GetValueDouble()  
 GetValueString()

---

## GetValueString()

Returns the value of a string type column from the specified child query in the result set.

**Syntax** HRESULT GetValueString(  
     LPSTR qryName,  
     LPSTR colName,  
     LPSTR pVal,

```
ULONG nVal);
```

**qryname** Name of the child query that generated the result set.

**colName** Name of the column from which to retrieve the value.

**pVal** Pointer to the variable that contains the returned column value.

**nVal** Length of the variable.

**Usage** Use GetValueString( ) to retrieve string values from the result set for subsequent processing.

**Rule** The specified column must be a String data type.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
IGXHierResultSet *hrs = NULL;
```

```
// Not shown here, execution of hierarchical query
// that retrieves the hierarchical resultset.
```

```
char buffer[256];
buffer[0] = '\0';
HRESULT hr;
```

```
hr = hrs->GetValueString("CUSTOMERS", "Country", buffer,
sizeof(buffer));
if (hr == NOERROR)
{
    StreamResult("The customer lives in the country of ");
    StreamResult(buffer);
    StreamResult("<br>");
}
```

**Related Topics** GetValueDateString( )

GetValueDouble( )

GetValueInt( )

## MoveNext( )

---

Moves to the next row for the specified child query in the result set.

**Syntax** HRESULT MoveNext(  
LPSTR qryName);

**qryName** Name of the child query that generated the result set.

**Usage** Use MoveNext() when iterating through rows in the result set to retrieve the contents of the next sequential row.

**Rule** The specified child query must exist in the result set.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds. If the target row is out of range, HRESULT is set to -1.

**Related Topics** CreateHierQuery() in the GXAppLogic class

IGXHierQuery interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database” in *Programmer’s Guide*.

---

## MoveTo()

Moves to the specified row for the specified child query in the result set.

**Syntax** HRESULT MoveTo(  
LPSTR qryName,  
ULONG nRow);

**qryName** Name of the child query that generated the result set.

**nRow** Number of the row in the result set to move to. The number of the first row in the result set is 1, the second row is 2, and so on.

**Usage** Use MoveTo() to move the internal cursor to a specific row in the result set, skipping over rows to be excluded from processing.

- Rules**
- The specified child query must exist in the result set.
  - The specified row number must exist in the result set.
  - If RS\_BUFFERING is turned on, AppLogic can move forward and backwards in the result set. However, if RS\_BUFFERING is not turned on, AppLogic can move forward to subsequent rows only. AppLogic cannot return to rows that have been processed previously.

**Tip** For certain database drivers, this operation may be very slow and should be avoided if possible.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds. If the target row is out of range, HRESULT is set to -1.

**Related Topics** CreateHierQuery( ) in the GXAppLogic class

IGXHierQuery interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database” in *Programmer’s Guide*.

# IGXLock interface

The IGXLock interface provides concurrency control for objects operating in a multithreaded environment (for example, in applications that use distributed state).

AppLogics use locks to protect objects during concurrent operations. For example, state and session nodes implement this interface. Applications that access state or session data concurrently must synchronize using the methods in this interface.

A lock has the following attributes:

- A lock mode. You can specify an exclusive or shared lock. An exclusive lock prevents other threads from accessing a locked object. You can also use the lock mode to specify that an operation may continue even if the desired locking mode is not available.
- A caller ID. This setting provides a unique identifier for the caller that places or removes a lock. The identifier is an array of bytes.

The IGXLock interface defines methods for locking and unlocking objects. It also defines a method for changing the lock mode.

## Include File

gxilock.h

## Methods

| Method       | Description   |
|--------------|---|
| ChangeMode() | Changes the lock mode of a currently locked object. This method is not available for the lock interface implemented by state and session objects. |
| Lock()       | Locks an object.  |
| Unlock()     | Unlocks a previously locked object.   |

### ChangeMode()

Changes the lock on an object.

**Note** This method is not supported for locks on state and session nodes. State and session support only one lock mode, GXLOCK\_EXCL, which cannot be changed.

**Syntax**

```
HRESULT ChangeMode(
    DWORD dwOldMode,
    int dwNewMode,
    LPBYTE pID
    ULONG nSize);
```

**dwOldMode** Current lock mode applied to an object. The mode is one of GXLOCK\_EXCL (exclusive lock) or GXLOCK\_SHARE (shared lock).

**dwNewMode** New locking mode, one of GXLOCK\_EXCL (exclusive lock) or GXLOCK\_SHARE (shared lock). Optionally, the mode may also include GXLOCK\_NOBLOCK if the operation should be allowed to continue if the desired locking mode is not available. If GXLOCK\_NOBLOCK is not specified, then a thread is blocked if the desired locking mode is not available.

**pID** ID of the caller requesting the change to the lock. This value is read only.

**nSize** Size of the identifier.

**Usage** Use ChangeMode() to change a lock on an object.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

## Lock()

---

Locks an object.

**Syntax** HRESULT Lock(  
     DWORD dwFlags,  
     LPBYTE pID  
     ULONG nSize);

**dwFlags** Locking mode, one of GXLOCK\_EXCL (exclusive lock) or GXLOCK\_SHARE (shared lock). Optionally, the mode may also include GXLOCK\_NOBLOCK if the operation should be allowed to continue if the desired locking mode is not available. If GXLOCK\_NOBLOCK is not specified, then a thread is blocked if the desired locking mode is not available.

GXLOCK\_EXCL is the only mode currently supported for locking a state or session node. You cannot specify GXLOCK\_NOBLOCK for state and session nodes.

**pID** ID of the caller requesting the lock. This value is a byte array. For state and session objects that implement the locking interface, you can pass in a null value for pID because these implementations automatically use the ID of the calling thread for pID.

**nSize** Size of the identifier.

**Usage** Use Lock() to lock an object.

**Rules** When you lock certain kinds of nodes, the following rules apply:

- After locking a parent state node, do not create or delete a child node under it.
- After locking a state or session node, do not delete the node.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds, or an error code, such as GXE\_FAIL on failure.

**Example** The following code shows how to lock and unlock a state node:

```
IGXState2 *marketnews = NULL;
HRESULT hr = cacheroot->GetStateChild("mktnews", &marketnews);
if (hr != GXE_SUCCESS || !marketnews)
    return;

// we expect marketnews state node to be accessed concurrently
```

```

IGXLock *l = null;
hr = marketnews->QueryInterface(IID_IGXLock, (LPVOID *)&l);
if (hr != GXE_SUCCESS || !l)
{
    marketnews->Release();
    return;
}

hr = l->Lock(GXLOCK_EXCL, NULL, 0);
if (hr != GXE_SUCCESS)
{
    marketnews->Release();
    l->Release();
    Log("lock error");
    return;
}

// we now have the node locked in exclusive mode
// ..... do work .....
// and unlock the node

hr = l->Unlock(GXLOCK.GXLOCK_EXCL, NULL, 0);
if (hr != GXE_SUCCESS)
{
    marketnews->Release();
    l->Release();
    Log("unlock error");
    return;
}

```

**Related Topics** GXAppLogic or GXSession2 classes

“Starting a Session” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

## Unlock()

---

Unlocks a previously locked object.

**Syntax** HRESULT Unlock(  
     DWORD dwFlags,  
     LPBYTE pID  
     ULONG nSize);

**dwFlags** The locking mode previously used to lock the object, either GXLOCK\_EXCL (exclusive lock), or GXLOCK\_SHARE (shared lock).

**GXLOCK\_EXCL** is the only mode currently supported for unlocking a state or session node.

**pID** The ID of the caller that requests lock removal. This value is a byte array. The ID must match the ID with which you set the lock.

Usually you pass in the ID of the executing thread that requests the lock. For state and session objects that implement the locking interface, you can pass in a null value for pID because these implementations automatically use the ID of the calling thread for pID.

**nSize** Size of the identifier.

**Usage** Use `Unlock()` to remove a lock on an object.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Example** The following code shows how to lock and unlock a state node:

```
IGXState2 *marketnews = NULL;
HRESULT hr = cacheroot->GetStateChild("mktnews", &marketnews);
if (hr != GXE_SUCCESS || !marketnews)
    return;

// we expect marketnews state node to be accessed concurrently

IGXLock *l = null;
hr = marketnews->QueryInterface(IID_IGXLock, (LPVOID *)&l);
if (hr != GXE_SUCCESS || !l)
{
    marketnews->Release();
    return;
}

hr = l->Lock(GXLOCK_EXCL, NULL, 0);
if (hr != GXE_SUCCESS)
{
    marketnews->Release();
    l->Release();
    Log("lock error");
    return;
}

// we now have the node locked in exclusive mode
// ..... do work .....
// and unlock the node

hr = l->Unlock(GXLOCK_EXCL, NULL, 0);
if (hr != GXE_SUCCESS)
{
    marketnews->Release();
```

```
l->Release();  
Log("unlock error");  
return;  
}
```

**Related Topics** GXAppLogic or GXSession2 classes

“Starting a Session” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

# IGXMailBox interface

The IGXMailBox interface represents an electronic mailbox used for communicating with incoming and outgoing electronic mail. IGXMailBox provides methods for opening and closing a mailbox, as well as for receiving and sending mail messages. You must have access to either an SMTP or POP mail server.

To create an instance of the IGXMailbox interface, use `CreateMailbox( )` in the `GXAppLogic` class, as shown in the following example:

```
IGXMailbox *pSendMBox = NULL;
CreateMailbox(pSendHost, pUser, pPswd, pUserAddr,
              &pSendMBox)
```

## Include File

`gxmailbox.h`

## Methods

| Method                        | Description   |
|-------------------------------|---|
| <code>Close( )</code>         | Closes an open electronic mailbox session.  |
| <code>Open( )</code>          | Opens a session with the mail server.   |
| <code>Retrieve( )</code>      | Retrieves unread electronic mail messages from the inbox.   |
| <code>RetrieveCount( )</code> | Counts the number of available unread electronic mail messages in the inbox.  |
| <code>RetrieveReset( )</code> | Resets the status of retrieved messages in the mailbox from read to unread and abandons (rolls back) any message deletions. |
| <code>Send( )</code>          | Sends an electronic mail message to one or more mail addresses.   |

## Related Topics

`CreateMailbox( )` in the `GXAppLogic` class

Chapter 10, “Integrating Applications with Email,” in *Programmer’s Guide*.

## Close()

---

Closes an open electronic mailbox session.

**Syntax** HRESULT Close()

**Usage** Use Close() to close a mailbox session and commit changes on the mail server, if applicable. If sessions are open on both the POP and SMTP server, Close() terminates both sessions.

Closing a session does not terminate the IGXMailbox object. The AppLogic can later reopen a session using Open().

**Rule** The AppLogic can only close a mailbox session that is open.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
// Define the string parameters that will be passed
// to IGXMailbox methods
LPSTR SendHost = "smtp.kivasoft.com";
LPSTR RecvHost = "pop.kivasoft.com";
LPSTR pUser = "eugene";
LPSTR pPswd = "eugenesSecretPassword";
LPSTR pUserAddr = "eugene@kivasoft.com";
LPSTR pSendTo[] = {"friend@otherhost.net", NULL};
LPSTR pMesg = "Hi Friend, How are you?";

HRESULT hr = NULL;

public void SendMail()
{
    // Create an IGXMailbox instance
    IGXMailbox *pSendMBox = NULL;
    if ((hr = CreateMailbox(pSendHost, pUser, pPswd, pUserAddr,
        &pSendMBox)) == NOERROR && pSendMBox != NULL)
    {
        // Open the mailbox to send the message
        if ((hr = pSendMBox->Open(OPEN_SEND)) == NOERROR)
        {
            pSendMBox->Send(pSendTo, pSendMesg);

            // Close the mailbox
            pSendMBox->Close();
        }
    }
}
```

```

    }
}
pSendMBox->Release();
}

```

**Related Topics** Open( )

CreateMailbox( ) in the GXAppLogic class

Chapter 10, “Integrating Applications with Email,” in *Programmer’s Guide*.

## Open( )

---

Opens a session with the mail server.

**Syntax** HRESULT Open(  
    DWORD dwFlag);

**dwFlag** Access level used to open the mailbox. Specify one of the following options:

- OPEN\_RECV to receive emails. Sets up a session with the POP server only.
- OPEN\_SEND to send emails. Sets up a session with the SMTP server only.
- OPEN\_SEND | OPEN\_RECV to send and receive emails.

**Usage** Use Open( ) to explicitly open a session with the mail server after instantiating the IGXMailbox object. Alternatively, the AppLogic can open a session after having closed a previous session using Close( ).

Depending on the setting of the dwFlag parameter, Open( ) starts a session on the SMTP server only, on the POP server only, or on both servers at once (two separate sessions).

**Rule** The AppLogic must call Open( ) before calling other methods.

**Tip** To conserve system resources, use only the access level you need. For example, if the AppLogic will only be sending electronic mail messages, specify OPEN\_SEND, not OPEN\_SEND | OPEN\_RECV.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** // Define the string parameters that will be passed  
// to IGXMailbox methods

```

LPSTR SendHost = "smtp.kivasoft.com";
LPSTR RecvHost = "pop.kivasoft.com";
LPSTR pUser = "eugene";
LPSTR pPswd = "eugenesSecretPassword";
LPSTR pUserAddr = "eugene@kivasoft.com";
LPSTR pSendTo[] = {"friend@otherhost.net", NULL};
LPSTR pMesg = "Hi Friend, How are you?";

HRESULT hr = NULL;

public void SendMail()
{
    // Create an IGXMailbox instance
    IGXMailbox *pSendMBox = NULL;
    if ((hr = CreateMailbox(pSendHost,pUser,pPswd,pUserAddr,
        &pSendMBox)) == NOERROR && pSendMBox != NULL)
    {
        // Open the mailbox to send the message
        if ((hr = pSendMBox->Open(OPEN_SEND)) == NOERROR)
        {
            pSendMBox->Send(pSendTo, pSendMesg);

            // Close the mailbox
            pSendMBox->Close();
        }
    }
    pSendMBox->Release();
}

```

**Related Topics** Send()

CreateMailbox() in the GXAppLogic class

Chapter 10, "Integrating Applications with Email," in *Programmer's Guide*.

## Retrieve()

Retrieves electronic mail messages from the inbox.

**Syntax** HRESULT Retrieve(  
 BOOL bLatest,  
 BOOL bDelete

```
IGXValList **ppMsgs);
```

**bLatest** Specify true to retrieve the latest unread messages. Specify false to retrieve all messages in the inbox.

**bDelete** Specify true to delete retrieved messages when the mailbox session is closed. Specify false to leave the retrieved messages on the mail server.

**ppMsgs** Pointer to the IGXValList object that contains the message strings. The keys are the message numbers. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use Retrieve() to get unread messages from the inbox. Once retrieved, messages are marked as READ.

**Rule** To use Retrieve(), the AppLogic must have first opened the mailbox session using Open() and have specified either OPEN\_RECV or OPEN\_SEND |OPEN\_RECV as the dwFlag parameter.

**Tip** AppLogic can use RetrieveReset() to undo changes (deletes, read flags) to messages in the inbox.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
public void RecvMail()
{
    IGXMailbox *pRecvMBox = NULL;
    IGXValList *pRecvMsg = NULL;
    int NumMsgs = 0;

    // Only check messages received after the last open
    BOOL Latest = true;
    // Remove retrieved messages from the mail server
    BOOL Delete = true;

    // Create a mailbox instance
    if ((hr = CreateMailbox(host,user,passwd,useraddr,
        &pRecvMBox)) == NOERROR && pRecvMBox != NULL)
    {
        // Open a mailbox to receive new messages
        if ((hr = pRecvMBox->Open(OPEN_RECV)) == NOERROR)
        {
            // Count the number of unretrieved messages
            if((NumMsgs = pRecvMBox->RetrieveCount()) > 0)
```

```

    {
        // Get the messages
        if((pRecvMBox->Retrieve(Latest, Delete,
            &pRecvMsg) == NOERROR)
        {
            // Use IGXValList methods to iterate through
            // the returned IGXValList. The keys in the
            // IGXValList are the message numbers. The
            // values are the email messages as strings

```

**Related Topics** Open( )

CreateMailbox( ) in the GXAppLogic class

Chapter 10, “Integrating Applications with Email,” in *Programmer’s Guide*.

## RetrieveCount( )

---

Counts the number of unread electronic mail messages in the inbox.

**Syntax** LONG RetrieveCount( );

**Usage** Before calling Retrieve( ), use RetrieveCount( ) to count the number of retrievable messages in the inbox. The AppLogic might do this to avoid retrieving an empty inbox. If the AppLogic iterates through the messages after they have been retrieved, the AppLogic can call RetrieveCount( ) to determine the maximum number of iterations required to process all available inbox messages.

**Rule** To use RetrieveCount( ), the AppLogic must have first opened the mailbox session using Open( ) and have specified either OPEN\_RECV or OPEN\_SEND | OPEN\_RECV as the dwFlag parameter.

**Return Value** The number of available unread electronic mail messages in the inbox. The RetrieveCount( ) method returns 0 for no messages and a negative number if an error occurred.

**Example**

```

public void RecvMail()
{
    IGXMailbox *pRecvMBox = NULL;
    IGXValList *pRecvMsg = NULL;
    int NumMsgs = 0;

    // Only check messages received after the last open

```

```

BOOL Latest = true;
// Remove retrieved messages from the mail server
BOOL Delete = true;

// Create a mailbox instance
if ((hr = CreateMailbox(host,user,passwd,useraddr,
    &pRecvMBox)) == NOERROR && pRecvMBox != NULL)
{
    // Open a mailbox to receive new messages
    if ((hr = pRecvMBox->Open(OPEN_RECV)) == NOERROR)
    {
        // Count the number of unretrieved messages
        if((NumMsgs = pRecvMBox->RetrieveCount()) > 0)
        {
            // Get the messages
            if((pRecvMBox->Retrieve(Latest, Delete,
                &pRecvMsg)) == NOERROR)
            {
                // Use IGXValList methods to iterate through
                // the returned IGXValList. The keys in the
                // IGXValList are the message numbers. The
                // values are the email messages as strings

```

**Related Topics** Open()

Retrieve()

CreateMailbox() in the GXAppLogic class

Chapter 10, “Integrating Applications with Email,” in *Programmer’s Guide*.

## RetrieveReset()

---

Resets the status of retrieved messages in the mailbox from read to unread and abandons (rolls back) any message deletions.

**Syntax** HRESULT RetrieveReset();

**Usage** Use RetrieveReset() to undo any changes made as a result of retrieving inbox messages with Retrieve().

- Rules**
- To use `RetrieveReset()`, the `AppLogic` must have first opened the mailbox session using `Open()` and have specified either `OPEN_RECV` or `OPEN_SEND|OPEN_RECV` as the `dwFlag` parameter.
  - Before calling `RetrieveReset()`, the `AppLogic` must first call `Retrieve()`.
  - To abandon changes made with `Retrieve()`, `AppLogic` must call `RetrieveReset()` before calling `Close()` or terminating the session.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `Open()`

`Retrieve()`

`CreateMailbox()` in the `GXAppLogic` class

Chapter 10, “Integrating Applications with Email,” in *Programmer’s Guide*.

## Send()

---

Sends an electronic mail message to one or more mail addresses.

**Syntax**

```
HRESULT Send(
    LPSTR *ppTo,
    LPSTR pMsg);
```

**ppTo** A list of email addresses, to which you want to send e-mail. The address or addresses must be supplied in a null-terminated array.

**pMsg** Text of the electronic mail message. Use Internet mail formatting conventions for specifying advanced features in the message text, such as `CC:` or `BCC:` addresses, the Subject header, uuencode, MIME attachments, receipt notification, and so on. For syntax specifications, see your POP and SMTP protocol documentation.

- Rules**
- To use `Send()`, the `AppLogic` must have first opened the mailbox session using `Open()` and have specified either `OPEN_SEND` or `OPEN_SEND|OPEN_RECV` as the `dwFlag` parameter.
  - The specified addresses must be valid Internet mail addresses.
  - The specified message text must follow POP and SMTP protocol conventions.

**Tip** The `Send()` method automatically includes the FROM: address that the `AppLogic` specified in the `pUserAddr` parameter of `CreateMailbox()` in the `GXAppLogic` class.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```
// Define the string parameters that will be passed
// to IGXMailbox methods
LPSTR SendHost = "smtp.kivasoft.com";
LPSTR RecvHost = "pop.kivasoft.com";
LPSTR pUser = "eugene";
LPSTR pPswd = "eugenesSecretPassword";
LPSTR pUserAddr = "eugene@kivasoft.com";
LPSTR pSendTo[] = {"friend@otherhost.net", NULL};
LPSTR pMsg = "Hi Friend, How are you?";

HRESULT hr = NULL;

public void SendMail()
{
    // Create an IGXMailbox instance
    IGXMailbox *pSendMBox = NULL;
    if ((hr = CreateMailbox(pSendHost,pUser,pPswd,pUserAddr,
        &pSendMBox)) == NOERROR && pSendMBox != NULL)
    {
        // Open the mailbox to send the message
        if ((hr = pSendMBox->Open(OPEN_SEND)) == NOERROR)
        {
            pSendMBox->Send(pSendTo, pSendMsg);

            // Close the mailbox
            pSendMBox->Close();
        }
    }
    pSendMBox->Release();
}
```

**Related Topics** `Open()`,  
`Retrieve()`

`CreateMailbox()` in the `GXAppLogic` class

Chapter 10, "Integrating Applications with Email," in *Programmer's Guide*.



# IGXObject interface

The IGSXObject interface provides methods for managing object lifecycles, memory, and interface access. The IGSXObject interface is the base interface for all Netscape C++ classes.

It is derived from the IUnknown interface in the Component Object Module (COM) specification published by Microsoft Corporation. For more information, see your Microsoft documentation or the Microsoft Web Site ([www.microsoft.com](http://www.microsoft.com)).

In the IGSXObject interface, use the `QueryInterface()` method to access and to navigate multiple interfaces on an object. In addition, use the `AddRef()` and `Release()` methods to manage the reference count for object instances.

Reference counting provides lifecycle control over objects. Each object contains a 32-bit reference counter that internally tracks the number of clients connected to it at runtime (the number of current pointers to any interfaces in any client). When the reference count is decremented to zero (0), the object is destroyed.

Unless otherwise noted, methods in the *Netscape Application Server Foundation Class Reference* that return objects automatically increment the reference count on these objects on behalf of your calling client code. However, developers must explicitly decrement the reference count in their `AppLogic`, using `Release()`, when a pointer to the instance is no longer needed.

To use the IGSXObject interface, the `AppLogic` must include the `gxicore.h` file, as shown in the following example:

```
#include <gxicore.h>
```

## Include File

`gxicore.h`

## Methods

| Method           | Description   |
|------------------|---|
| AddRef()         | Adds a reference to an interface instance and increments the object's reference count by one (1). |
| QueryInterface() | Determines whether an object supports a particular interface.                                     |
| Release()        | Decrements the object's reference count by one (1).   |

## Related Topics

“Introduction to Interfaces and COM” in Chapter 1, “Introduction to Applications” in *Programmer's Guide*.

## AddRef()

---

Adds a reference to an interface instance and increments the object's reference count by one (1).

**Syntax** `ULONG AddRef();`

**Usage** Use AddRef() to explicitly increase the reference count on the object by one (1).

Unless otherwise noted in the *Netscape Application Server Foundation Class Reference*, the Netscape Application Server system calls AddRef() implicitly for methods in the Netscape Application Builder. For example, when AppLogic calls CreateQuery() in the GXAppLogic class, the Netscape Application Server calls AddRef() automatically, thereby incrementing the reference count for the created IGXQuery object by one (1). Similarly, when AppLogic calls ExecuteQuery() in the IGXDataConn interface, the Netscape Application Server calls AddRef() automatically for the created IGXResultSet object. It is the responsibility of the AppLogic developer to call Release() on these objects for each corresponding implicit AddRef() call.

In more advanced applications, developers may need to explicitly increment the reference count in their AppLogic using AddRef(). You might want to do this in order to ensure that the object is valid when using the object for a long time.

- Return Value** ULONG, which represents the returned value of the incremented reference count. Its value is meaningful for debugging purposes only.
- Rules**
- For every `AddRef()` call (whether explicit or implicit via Netscape Application Server Foundation Class Library helper functions), `AppLogic` must make a corresponding single `Release()` call, through the pointer, to explicitly decrement the reference count on the object by one (1).
  - Calling `AddRef()` applies to any object that inherits from the `IGXObject` interface. It does not apply to structures, however.
- Tip** In general, explicit calls to `AddRef()` in `AppLogic` are required only for more advanced programming tasks, such as for implementing dynamically loadable modules (DLMs).
- Related Topics** “Introduction to Interfaces and COM” in Chapter 1, “Introduction to Netscape Application Server Applications.”

---

## QueryInterface()

Determines whether an object supports a particular interface.

**Syntax**

```
HRESULT QueryInterface(
    REFIID riid,
    void **ppvObject);
```

**riid** Reference identifier to the requested interface, which is defined in the header file associated with that interface. Examples include `IID_IGXValllist`, `IID_IGXResultSet`, and `IID_IGXQuery`.

**ppvObject** Pointer to a pointer, allocated by the client, that will contain the returned interface pointer if the query succeeds. If an error occurs (such as the interface is not supported), `ppvObject` is set to `NULL`.

**Usage** Use `QueryInterface()` to obtain a pointer to an interface on an object. You can use `QueryInterface()` to determine the interfaces that an object supports.

If `QueryInterface()` successfully obtains the interface pointer, it implicitly calls `AddRef()` to increment the reference count on that interface instance.

**Rule** For every successful `QueryInterface()` call, the `AppLogic` must make a corresponding single `Release()` call, through the returned pointer, in order to match the implicit `AddRef()` call made by `QueryInterface()`.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Introduction to Interfaces and COM” in Chapter 1, “Introduction to Applications.”

---

## Release()

Decrements the object’s reference count by one (1).

**Syntax** ULONG Release();

**Usage** Use Release() in conjunction with AddRef() or QueryInterface() to explicitly release an object reference when the AppLogic is finished using it.

When the reference count is decremented to zero, the object is deleted, its memory deallocated, and all interfaces to it become invalid.

- Rules**
- For every AddRef() or successful QueryInterface() call, whether made implicitly by the system or explicitly in an AppLogic module, the AppLogic must make a corresponding Release() call, through the pointer.
  - After calling Release() for an object, all subsequent references to that object through that pointer are invalid.
  - Calling Release() applies to any object that inherits from the IGXObject interface. It does not apply to structures, however.

**Tip** The AppLogic should call Release() on an object immediately after it is done using the object and has no further use for the pointer to it.

**Return Value** ULONG, which represents the returned value of the decremented reference count. Its value is useful for debugging purposes only.

**Example**

```
// The following command releases the m_pQuery pointer:  
m_pQuery->Release();
```

**Related Topics** “Introduction to Interfaces and COM” in Chapter 1, “Introduction to Applications.”

# IGXOrder interface

The IGXOrder interface represents the current processing status of an asynchronous operation. IGXOrder provides methods for obtaining the status and return code of an asynchronous operation.

To run an asynchronous database operation, the AppLogic must specify `GX_DA_EXEC_ASYNC` as the `dwFlags` parameter in any of the following methods:

- `ExecuteQuery()` in the `IGXDataConn` interface
- `AddRow()`, `DeleteRow()`, or `UpdateRow()` in the `IGXTable` interface

To create an instance of the IGXOrder interface for an asynchronous query, use `GetOrder()` in the `IGXResultSet` interface.

## Include File

`gxiorder.h`

## Methods

|                         |   |
|-------------------------|---|
| <code>GetState()</code> | Returns the processing status of the asynchronous operation on the database server: active, done, canceled, or unknown. |
|-------------------------|---|

## Related Topics

`ExecuteQuery()` in the `IGXDataConn` interface

`GetOrder()` in the `IGXResultSet` interface

`GXWaitForOrder()` helper function

## GetState()

---

Returns the processing status of the asynchronous operation.

**Syntax** `HRESULT GetState(  
    DWORD *pdwState,`

```
DWORD *pdwResult,
ULONG *pGuess);
```

**pdwState** Pointer to the variable that contains the returned status code. The variable is set to one of the following:

| Constant              | Description  |
|-----------------------|--|
| GXORDER_STATE_ACTIVE  | The asynchronous operation is still being processed.   |
| GXORDER_STATE_CANCEL  | The asynchronous operation has been cancelled.   |
| GXORDER_STATE_DONE    | The asynchronous operation has been completely processed. Check the <code>pdwResult</code> variable to see if the operation completed with a result of success or failure. |
| GXORDER_STATE_UNKNOWN | The status of the asynchronous operation is unknown.   |

**pdwResult** Pointer to the variable that contains the returned result, which is the HRESULT return value of the operation (which is what is obtained if the operation were called synchronously.)

**pGuess** Pointer to the variable that contains the returned estimate about the current completion percentage of the operation. Netscape Application Server internal use only.

**Usage** Use `GetState()` to return status information to use in error-handling code.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```
IGXOrder *pOrder;
ULONG     nOrder;
HRESULT hr, ReqResult;

if (NewRequestAsync(asyncGUIDStr, m_pValIn,
                    m_pValOut, 0, &pOrder) == GXE_SUCCESS)
{
    Log("Successfully invoked async AppLogic\n");

    // wait for async applogic to finish (max 100 seconds)
    hr = GXWaitForOrder(&pOrder, 1, &nOrder, m_pContext, 100);
```

```

if (hr != NOERROR)
{
    return Result("Error in executing async request:
order wait returned an error");
}
else
{
    pOrder->GetState(NULL, &ReqResult, NULL);
    if (ReqResult != NOERROR)
        return Result("Error in executing async
request");
}
}
else
{
    Log("Failed to invoke async AppLogic\n");
}
}

```

**Related Topics** [ExecuteQuery\(\) in the IGXDataConn interface](#)  
[GetOrder\(\) in the IGXResultSet interface](#)  
[GXWaitForOrder\(\) helper function](#)



# IGXPreparedQuery interface

The IGXPreparedQuery interface represents a prepared flat query. An IGXPreparedQuery object contains a SQL statement that has been compiled. This is what makes a statement “prepared.” An AppLogic uses a prepared query when it needs to execute a SQL statement multiple time with different parameters.

For example, if an AppLogic runs an INSERT statement several times, each time with a different set of values to insert into the table, using a prepared query involves the following steps:

1. Prepare (compile) the INSERT statement with placeholder parameters whose values will be specified later.
2. Specify a set of parameter values.
3. Execute the prepared query.
4. Specify another set of parameter values.
5. Execute the prepared query.

By preparing the SQL statement, the database needs to compile the statement only once. Without prepared statements, the database must recompile each statement every time it is executed, which is less efficient.

To create an instance of the IGXPreparedQuery interface, use PrepareQuery() in the IGXDataConn interface.

## Include File

gxidata.h

## Methods

| Name         | Description  |
|--------------|--|
| Execute( )   | Executes a prepared query.                               |
| SetParams( ) | Specifies the parameters and flags for a prepared query. |

## Related Topics

PrepareQuery( ) in the IGXDataConn interface

“Using Prepared Database Commands” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

## Execute()

---

Executes a prepared query.

**Syntax**

```
HRESULT Execute(
    DWORD dwFlags,
    IGXValList *pParams,
    IGXTrans *pTrans,
    IGXValList *pProps,
    IGXResultSet **ppResultSet);
```

**dwFlags** Specifies flags used to execute this prepared query. To activate result set buffering, specify `GX_DA_RS_BUFFERING`. Otherwise, specify zero.

**pParams** Pointer to an `IGXValList` object that contains parameters to pass to the prepared query. Parameters are used to execute the query.

**pTrans** Pointer to an `IGXTrans` object that contains the transaction associated with this query, or `NULL` for no transaction.

**pProps** Pointer to the `IGXValList` object that contains query properties, or `NULL` for no properties. After instantiating an object of the `IGXValList` interface, set any of the following properties:

- `RS_BUFFERING` turns on result set buffering when set to “TRUE”.
- `RS_INIT_ROWS` specifies the initial size of the buffer, in number of rows. If the result set size exceeds this setting, a `FetchNext( )` call will return the error `GX_DA_BUFFER_EXCEEDED` and result set buffering will be turned off.
- `RS_MAX_ROWS` specifies the maximum number of rows for the buffer. If the result set size exceeds this setting, a `FetchNext( )` call will return the error `GX_DA_BUFFER_EXCEEDED` and result set buffering will be turned off.
- `RS_MAX_SIZE` specifies the maximum number of bytes for the buffer.

If RS\_BUFFERING is enabled and if the optional parameters are not specified, the global values in the registry are used instead.

**ppResultSet** Pointer to the IGXResultSet object that contains the returned result set from the callable statement, if the database supports this feature. When AppLogic is finished using the object, call the Close() method in the IGXResultSet interface, then call the Release() method to release the interface instance.

**Usage** Use Execute() to run a prepared query. If the command contains parameters, instantiate an IGXValList object and use SetVal() or SetValByRef() in the IGXValList interface to specify the parameter values to pass to the command.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

```
// Create the prepared query
IGXPreparedQuery *pPQuery=NULL;

if(((hr=pConn->PrepareQuery(0, pQuery, NULL, NULL,
&pPQuery))!=GXE_SUCCESS)&&pPQuery) {

    IGXResultSet *pRset=NULL;

    // Execute the prepared query
    if(((hr=pPQuery->Execute(0, pList, NULL, NULL,
&pRset))!=GXE_SUCCESS)&&pRset) {
```

**Related Topics** IGXValList interface

IGXTrans interface

PrepareQuery() in the IGXDataConn interface

“Using Prepared Database Commands” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

## SetParams()

Specifies the parameters for a prepared query.

**Syntax** HRESULT SetParams(  
     DWORD dwFlags,  
     IGXValList \*pParams);

**dwFlags** Specify zero (0).

**pParams** Pointer to an IGXValList object that contains parameters to pass to the prepared query.

**Usage** To pass parameters to the prepared query using SetParams( ), you must pass NULL for the pParams parameter in Execute( ).

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** IGXValList interface

IGXTrans interface

PrepareQuery( ) in the IGXDataConn interface

“Using Prepared Database Commands” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

# IGXQuery interface

The IGXQuery interface represents a flat query. IGXQuery provides methods for specifying and obtaining the criteria used to select data from a data source. The AppLogic uses IGXQuery member methods to specify all parts of the SQL SELECT statement, including the SELECT, FROM, GROUP BY, HAVING, ORDER BY, and WHERE clauses.

To run a flat query, the AppLogic performs the following steps:

1. Creates an IGXQuery object using CreateQuery() in the GXAppLogic class.
2. Specifies query criteria using methods in the IGXQuery interface.
3. Executes the query, passing the loaded IGXQuery object to ExecuteQuery() in the IGXDataConn interface.
4. Processes the result set using methods in the IGXResultSet interface.

The AppLogic can also use IGXQuery methods to obtain information about query criteria when the criteria are unknown. Before executing the query on the data source, the AppLogic can evaluate and, if necessary, dynamically change the query criteria.

To create an instance of the IGXQuery interface, use the CreateQuery() method in the GXAppLogic class.

## Include File

gxidata.h

## Methods

| Method       | Description   |
|--------------|---|
| GetFields()  | Returns a comma-separated list of arbitrary SQL expressions or columns to be included in the result set of the query. |
| GetGroupBy() | Returns the GROUP BY clause of the query.   |
| GetHaving()  | Returns the HAVING clause of the query.   |
| GetOrderBy() | Returns the ORDER BY clause of the query.   |

| Method       | Description  |
|--------------|--|
| GetSQL()     | Returns the SQL pass-through statement associated with the query.  |
| GetTables()  | Returns a comma-separated list of tables in the FROM clause of the query.  |
| GetWhere()   | Returns the WHERE clause of the query.   |
| SetFields()  | Specifies the list of columns and computed fields to be included in the result set of the query. Required method when writing a query. |
| SetGroupBy() | Specifies the GROUP BY clause of the query, determining how rows are grouped and calculated.   |
| SetHaving()  | Specifies the HAVING clause of the query, determining which aggregate rows qualify for inclusion in the result set.                    |
| SetOrderBy() | Specifies the ORDER BY clause of the query, determining how rows are sorted in the result set.   |
| SetSQL()     | Specifies the SQL statement to be passed directly to the data source.  |
| SetTables()  | Specifies the FROM clause of the query, identifying one or more tables to be queried. Required method when writing a query.            |
| SetWhere()   | Specifies the WHERE clause of the query, determining which rows qualify for inclusion in the result set.                               |

## Related Topics

CreateQuery() in the GXAppLogic class

ExecuteQuery() in the IGXDataConn interface

IGXResultSet interface

“Writing Flat Queries” in Chapter 6, “Querying a Database,” in *Programmer's Guide*.

## GetFields()

Returns a comma-separated list of arbitrary SQL expressions or columns to be included in the result set of the query.

**Syntax** HRESULT GetFields(  
     IGXBuffer \*\*ppBuff);

**ppBuff** Pointer to the IGXBuffer object that contains the returned text, a comma-separated list of columns that the query defines for the result set, starting with the first column and proceeding sequentially, left to right. This method allocates the IGXBuffer object automatically. When the AppLogic is finished using the object, call the Release( ) method to release the interface instance.

**Usage** In a SQL SELECT statement, the first clause specifies the SELECT keyword as well as the list of columns to be retrieved in the result set.

Use GetFields( ) when the requested columns in a query are unknown, such as when using a query from another source. The AppLogic can analyze this list to determine the names of the columns as well as the order in which they will appear in the result set. Before executing or re-executing the query, the AppLogic can evaluate and, if necessary, dynamically change columns and column order in the query by calling SetFields( ).

- Tips**
- To use a query obtained from another source such as a file, the AppLogic can call GetFields( ) and other GetXXXX( ) member methods to test the query statement before submitting it to the server for processing. The AppLogic can then use the SetXXXX( ) member methods to change the statement and avoid lengthy queries or syntax errors.
  - Use methods in the IGXBuffer interface to manipulate the returned memory block.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateQuery( ) in the GXAppLogic class

“Writing Flat Queries” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

---

## GetGroupBy( )

Returns the GROUP BY clause of the query.

**Syntax** HRESULT GetGroupBy(  
IGXBuffer \*\*ppBuff);

**ppBuff** Pointer to the IGXBuffer object that contains the returned text, the GROUP BY clause of the query. This method allocates the IGXBuffer object automatically. When the AppLogic is finished using the object, call the Release( ) method to release the interface instance.

**Usage** In a SQL SELECT statement, the GROUP BY clause specifies rows to summarize into aggregate rows using column functions (such as SUM or MAX) or column names.

Use GetGroupBy( ) when the GROUP BY clause of the query is unknown, such as when using a query from another source. Before executing the query, the AppLogic can evaluate and, if necessary, dynamically change the GROUP BY clause by calling SetGroupBy( ).

- Tips**
- To use a query obtained from another source such as a file, the AppLogic can call GetGroupBy( ) and other GetXXXX( ) member methods to test the query statement before submitting it to the server for processing. The AppLogic can then use the SetXXXX( ) member methods to change the statement and avoid lengthy queries or syntax errors.
  - Use methods in the IGXBuffer interface to manipulate the returned memory block.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateQuery( ) in the GXAppLogic class

“Writing Flat Queries” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

---

## GetHaving( )

Returns the HAVING clause of the query.

**Syntax** HRESULT GetHaving(  
IGXBuffer \*\*ppBuff);

**ppBuff** Pointer to the IGXBuffer object that contains the returned text, the HAVING clause of the query. This method allocates the IGXBuffer object automatically. When the AppLogic is finished using the object, call the Release( ) method to release the interface instance.

**Usage** In a SQL SELECT statement, the HAVING clause specifies which of the aggregate rows returned by the GROUP BY clause are selected for the result set.

Use `GetHaving()` when the HAVING clause of the query is unknown, such as when using a query from another source. Before executing the query, the `AppLogic` can evaluate and, if necessary, dynamically change the HAVING clause by calling `SetHaving()`.

- Tips**
- To use a query obtained from another source such as a file, the `AppLogic` can call `GetHaving()` and other `GetXXXX()` member methods to test the query statement before submitting it to the server for processing. The `AppLogic` can then use the `SetXXXX()` member methods to change the statement and avoid lengthy queries or syntax errors.
  - Use methods in the `IGXBuffer` interface to manipulate the returned memory block.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `CreateQuery()` in the `GXAppLogic` class

“Writing Flat Queries” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

---

## GetOrderBy()

Returns the ORDER BY clause of the query.

**Syntax** `HRESULT GetOrderBy(  
IGXBuffer **ppBuff);`

**ppBuff** Pointer to the `IGXBuffer` object that contains the returned text, the ORDER BY clause of the query. This method allocates the `IGXBuffer` object automatically. When the `AppLogic` is finished using the object, call the `Release()` method to release the interface instance.

**Usage** In a SQL SELECT statement, the ORDER BY clause specifies one or more columns by which rows in the result set are sorted, as well as whether they appear in ascending or descending ASCII order.

Use `GetOrderBy()` when the ORDER BY clause of the query is unknown, such as when using a query from another source. Before executing the query, the `AppLogic` can evaluate and, if necessary, dynamically change the ORDER BY clause by calling `SetOrderBy()`.

**Rule** Some database vendors have restrictions on the ordering and usage of ORDER BY clauses. Read your database vendor's documentation carefully and test queries to ensure that they return the desired results.

- Tips**
- To use a query obtained from another source such as a file, the AppLogic can call GetOrderBy() and other GetXXXX() member methods to test the query statement before submitting it to the server for processing. The AppLogic can then use the SetXXXX() member methods to change the statement and avoid lengthy queries or syntax errors.
  - Use methods in the IGXBuffer interface to manipulate the returned memory block.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateQuery() in the GXAppLogic class

"Writing Flat Queries" in Chapter 6, "Querying a Database," in *Programmer's Guide*.

---

## GetSQL()

Returns the SQL pass-through statement associated with the query.

**Syntax** HRESULT GetSQL(  
IGXBuffer \*\*ppBuff);

**ppBuff** Pointer to the IGXBuffer object that contains the returned text, the SQL pass-through statement of the query, in a single concatenated string. This method allocates the IGXBuffer object automatically. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use GetSQL() when the query string is unknown, such as when using a query from another source. Before executing the query, the AppLogic can dynamically change the SQL statement by calling SetSQL().

**Rule** If a query is set using SetSQL() as well as the SetXXXX() methods, the SetSQL() string will be executed, not the string specified by SetXXXX().

- Tips**
- To use a query obtained from another source such as a file, the AppLogic can call GetSQL( ) and other GetXXXX( ) member methods to test the query statement before submitting it to the server for processing. The AppLogic can then use the SetXXXX( ) member methods to change the statement and avoid lengthy queries or syntax errors.
  - Use methods in the IGXBuffer interface to manipulate the returned memory block.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateQuery( ) in the GXAppLogic class

“Writing Flat Queries” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## GetTables()

---

Returns a comma-separated list of tables in the FROM clause of the query.

**Syntax** HRESULT GetTables(  
IGXBuffer \*\*ppBuff);

**ppBuff** Pointer to the IGXBuffer object that contains the returned text, the FROM clause of the query. This method allocates the IGXBuffer object automatically. When the AppLogic is finished using the object, call the Release( ) method to release the interface instance.

**Usage** In a SQL SELECT statement, the FROM clause specifies one or more source tables, views, or table aliases to search in the query. In Netscape Application Builder, the AppLogic can obtain table names only.

Use getTables( ) when the FROM clause of the query is unknown, such as when using a query from another source. Before executing the query, the AppLogic can evaluate and, if necessary, dynamically change the FROM clause by calling SetTables( ).

- Tips**
- To use a query obtained from another source such as a file, the AppLogic can call getTables( ) and other GetXXXX( ) member methods to test the query statement before submitting it to the server for processing. The AppLogic can then use the SetXXXX( ) member methods to change the statement and avoid lengthy queries or syntax errors.

- Use methods in the IGXBuffer interface to manipulate the returned memory block.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateQuery( ) in the GXAppLogic class

“Writing Flat Queries” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

---

## GetWhere()

Returns the WHERE clause of the query.

**Syntax** HRESULT GetWhere(  
IGXBuffer \*\*ppBuff);

**ppBuff** Pointer to the IGXBuffer object that contains the returned text, the WHERE clause of the query. This method allocates the IGXBuffer object automatically. When the AppLogic is finished using the object, call the Release( ) method to release the interface instance.

**Usage** In a SQL SELECT statement, the WHERE clause specifies the search condition and determines which rows in the table are selected for the result set.

Use GetWhere( ) when the WHERE clause of the query is unknown, such as when using a query from another source. Before executing the query, the AppLogic can evaluate and, if necessary, dynamically change the WHERE clause by calling SetWhere( ).

- Tips**
- To use a query obtained from another source such as a file, the AppLogic can call GetWhere( ) and other GetXXXX( ) member methods to test the query statement before submitting it to the server for processing. The AppLogic can then use the SetXXXX( ) member methods to change the statement and avoid lengthy queries or syntax errors.
  - Use methods in the IGXBuffer interface to manipulate the returned memory block.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateQuery( ) in the GXAppLogic class

“Writing Flat Queries” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## SetFields()

---

Specifies the list of columns and computed fields to be included in the result set of the query. Required method when writing a query.

**Syntax** HRESULT SetFields(  
LPSTR szFields);

**szFields** List of field names, separated by commas, or an asterisk (\*) to include all fields. Extra whitespace characters are ignored. Use the AS keyword to specify field aliases. Defaults to all fields (\*).

**Usage** In a SQL SELECT statement, the first clause specifies the SELECT keyword as well as the list of columns and computed fields to be retrieved in the result set. The AppLogic can specify field aliases using the AS keyword in the SetFields() parameter list.

A computed field is the result of an expression using either of the following kinds of expressions:

- **Mathematical functions**, including SQL string, numeric, time, date, system, and data type conversion functions and mathematical operators
- **Aggregate functions**, including SUM, COUNT, MIN, MAX, AVG, to summarize values per column across a group of rows. These functions are commonly used in conjunction with the GROUP BY clause, which the AppLogic can specify using SetGroupBy().

- Rules**
- Use ANSI 92 SQL-compliant syntax for the field list.
  - Use implementation-specific SQL syntax extensions only on data sources that support them. Using extensions may compromise portability across platforms.
  - Any specified column names must appear in one of the tables specified in SetTables(). Table qualified names are permitted, such as "prod.name, emp.name".

**Tip** For computed fields, use the AS keyword so that the AppLogic can process the column in the result set by alias name.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

```
IGXQuery *pQuery=NULL;

if(pAcctNum)
    pQuery->SetTables("OBCustomer, OBAccount");
else
    pQuery->SetTables("OBCustomer");

pQuery->SetFields("lastName, firstName, userName, ssn");
pQuery->SetWhere(whereClause);
pQuery->SetOrderBy("lastName, firstName");
```

**Related Topics** CreateQuery( ) in the GXAppLogic class

“Specifying Columns and Computed Fields” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

---

## SetGroupBy( )

Specifies the GROUP BY clause of the query, determining how rows are grouped and calculated.

**Syntax** HRESULT SetGroupBy(  
LPSTR szGroupBy);

**szGroupBy** GROUP BY clause of the query, using standard SQL syntax.

**Usage** In a SQL SELECT statement, the GROUP BY clause specifies rows to combine using column functions (such as SUM or MAX) or column names. Such groupings are called aggregate rows, which are single rows in a result set that combine data from a group of database rows with one or more column values in common.

- Rules**
- Use ANSI 92 SQL-compliant syntax for the GROUP BY clause.
  - Use implementation-specific SQL syntax extensions only on data sources that support them. Using extensions may compromise portability across platforms.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateQuery( ) in the GXAppLogic class

“Summarizing Data” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## SetHaving()

---

Specifies the HAVING clause of the query, determining which aggregate rows qualify for inclusion in the result set.

**Syntax** HRESULT SetHaving(  
LPSTR szGroupBy);

**szGroupBy** HAVING clause of the query, using standard SQL syntax.

**Usage** The HAVING clause is used in conjunction with the aggregate functions (SUM, AVG, and so on) and the GROUP BY clause. In a SQL SELECT statement, the HAVING clause specifies a condition that determines which aggregate rows are selected for the result set. The HAVING clause restricts the number of aggregate rows retrieved in the result set. If unspecified, all aggregate rows will be retrieved.

- Rules**
- Use ANSI 92 SQL-compliant syntax for the HAVING clause.
  - Use implementation-specific SQL syntax extensions only on data sources that support them. Using extensions may compromise portability across platforms.
- Tips**
- The order in which you specify a HAVING clause, in relation to other query clauses, may affect which records are retrieved in the result set. See your RDBMS server documentation for more information.
  - To improve the AppLogic performance, be sure to specify a HAVING or WHERE clause to avoid retrieving rows unnecessarily, especially for large tables.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateQuery() in the GXAppLogic class

“Summarizing Data” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## SetOrderBy()

---

Specifies the ORDER BY clause of the query, determining how rows are sorted in the result set.

**Syntax** HRESULT SetOrderBy(  
LPSTR szOrderBy);

**szOrderBy** ORDER BY clause of the query, using standard SQL syntax. Supports the ASC and DESC keywords for sorting.

**Usage** In a SQL SELECT statement, the ORDER BY clause specifies one or more columns by which rows in the result set are sorted. The AppLogic can also specify whether records appear in ascending (the default) or descending ASCII order using the ASC and DESC keywords, respectively.

- Rules**
- Use ANSI 92 SQL-compliant syntax for the ORDER BY clause.
  - Use implementation-specific SQL syntax extensions only on data sources that support them. Using extensions may compromise portability across platforms.
  - Any specified column names must appear in one of the columns specified in SetFields().
  - Some database vendors have restrictions on the ordering and usage of ORDER BY clauses. Read your database vendor's documentation carefully and test queries to ensure that they return the desired results.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** IGXQuery \*pQuery=NULL;

```
if(pAcctNum)
    pQuery->SetTables("OBCustomer, OBAccount");
else
    pQuery->SetTables("OBCustomer");

pQuery->SetFields("lastName, firstName, userName, ssn");
pQuery->SetWhere(whereClause);
pQuery->SetOrderBy("lastName, firstName");
```

**Related Topics** CreateQuery() in the GXAppLogic class

“Sorting Data” in Chapter 6, “Querying a Database” of *Programmer's Guide*.

## SetSQL()

---

Specifies the SQL statement to be passed directly to the data source.

**Syntax** `HRESULT SetSQL(  
LPSTR szSQL);`

**szSQL** SQL statement, using standard SQL syntax, to execute on the target data source. Specify a single, concatenated string. Do not use semicolon (;) characters or other vendor-specific statement delimiters.

**Usage** The AppLogic can use SetSQL() as an alternative to using other Netscape Application Builder methods, such as constructing queries, inserting, updating, and deleting rows, and managing transactions. The AppLogic can also use SetSQL() to run specialized SQL statements, such Data Definition Language (DDL) commands, Data Control Language (DCL) commands, and so on.

- Rules**
- Use ANSI 92 SQL-compliant syntax for the SQL statement.
  - Use implementation-specific SQL syntax extensions only on data sources that support them. Using extensions may compromise portability across platforms.
  - The AppLogic must be logged in with sufficient privileges to permit any operations requested in the passed-through SQL statement.
  - If inserting or updating rows in a table, the AppLogic must specify values that are valid. For example, the AppLogic cannot omit specifying a value for any column defined as NOT NULL and without a DEFAULT value, such as keys.
  - Using SetSQL() overrides all previous calls to SetXXXX() member methods for this query object. If a query is set using SetSQL() as well as the SetXXXX() methods, the SetSQL() string will be executed, not the string specified by SetXXXX().
  - If the statement is part of a transaction, the AppLogic must first create an instance of the IGXTrans interface using CreateTrans() in the GXAppLogic class. The AppLogic must then call Begin() before executing the statement and, after executing the statement, call Commit() or Rollback() as appropriate.

**Tip** To determine whether a column is defined as NOT NULL, use GetNullsAllowed() in the IGXColumn interface.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

```
// Create a query to update a table
IGXQuery *pUserQuery=NULL;

if(((hr=CreateQuery(&pUserQuery))==GXE_SUCCESS)&&pUserQuery) {
    pUserQuery->SetSQL("UPDATE OBUser SET password = :password, eMail =
        :eMail WHERE userName = :userName");
}
```

**Related Topics** CreateQuery( ) in the GXAppLogic class

“Using Pass-Through Database Commands” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

“Writing Flat Queries” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

Vendor documentation regarding SQL programming for the specific data source that is the target of the SQL statement.

---

## SetTables( )

Specifies the FROM clause of the query, identifying one or more tables to be queried. Required method when writing a query.

**Syntax** HRESULT SetTables(  
LPSTR szTables);

**szTables** List of table names separated by commas. Whitespace characters are ignored.

**Usage** In a SQL SELECT statement, the FROM clause specifies one or more source tables, views, or table aliases to search in the query. In Netscape Application Builder, the AppLogic can specify table names only.

- Rules**
- Use ANSI 92 SQL-compliant syntax for the FROM clause.
  - Use implementation-specific SQL syntax extensions only on data sources that support them. Using extensions may compromise portability across platforms.
  - The AppLogic can specify table names but not table aliases or view names.
  - The AppLogic can use the same table several times in a query. To do so, specify a different alias name each time the table is used.

**Return Value** SetWhere( )

```
IGXQuery *pQuery=NULL;

if(pAcctNum)
    pQuery->SetTables("OBCustomer, OBAccount");
else
    pQuery->SetTables("OBCustomer");

pQuery->SetFields("lastName, firstName, userName, ssn");
pQuery->SetWhere(whereClause);
pQuery->SetOrderBy("lastName, firstName");
```

**Related Topics** CreateQuery( ) in the GXAppLogic class

“Specifying Tables” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

---

## SetWhere( )

Specifies the WHERE clause of the query, determining which rows qualify for inclusion in the result set.

**Syntax** HRESULT SetWhere(  
LPSTR szWhere);

**szWhere** WHERE clause of the query, using standard SQL syntax.

**Usage** In a SQL SELECT statement, the WHERE clause specifies the search condition and determines which rows in the table are selected for the result set. The WHERE clause restricts the number of rows retrieved in the result set. If unspecified, all rows in the source table will be retrieved.

- Rules**
- Use ANSI 92 SQL-compliant syntax for the WHERE clause.
  - Use implementation-specific SQL syntax extensions only on data sources that support them. Using extensions may compromise portability across platforms.

**Tip** To improve AppLogic performance, be sure to specify a HAVING or WHERE clause to avoid retrieving rows unnecessarily, especially for large tables.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

```
IGXQuery *pQuery=NULL;
```

```
if(pAcctNum)
    pQuery->SetTables("OBCustomer, OBAccount");
else
    pQuery->SetTables("OBCustomer");

pQuery->SetFields("lastName, firstName, userName, ssn");
pQuery->SetWhere(whereClause);
pQuery->SetOrderBy("lastName, firstName");
```

**Related Topics** CreateQuery( ) in the GXAppLogic class

“Specifying Conditions on Row Retrieval” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

# IGXResultSet interface

The IGXResultSet interface represents the results of a flat query. IGXResultSet provides methods to iterate through rows in the result set and retrieve data from each row. To retrieve data from the result set, the AppLogic uses methods tailored for specific column types. For example, if retrieving data from a string column, use GetValueString(). If retrieving binary data, use GetValueBinary().

To process hierarchical result sets, use methods in the IGXHierResultSet interface or EvalTemplate() in the GXAppLogic class instead.

IGXResultSet is part of the Data Access Engine (DAE) service.

To create an instance of the IGXResultSet interface, use ExecuteQuery() in the IGXDataConn interface or Execute() in the IGXPreparedQuery interface.

## Include File

gxidata.h

## Methods

| Method             | Description  |
|--------------------|--|
| Close()            | Releases the connection used by the result set.  |
| EnumColumnReset()  | Resets the column enumeration to the first column in the result set.                                     |
| EnumColumns()      | Returns the definition of the next column in the result set.   |
| FetchNext()        | Retrieves the next row in the result set.  |
| GetColumn()        | Returns the column definition of the column with the specified name.                                     |
| GetColumnByOrd()   | Returns the column definition for the column in the specified ordinal position.                          |
| GetColumnOrdinal() | Returns the ordinal position of the column with the specified name.                                      |
| GetNumColumns()    | Returns the number of columns in the result set.   |
| GetOrder()         | For asynchronous queries, returns an IGXOrder object used for obtaining the current status of the query. |

| Method                 | Description   |
|------------------------|---|
| GetRowNumber( )        | Returns the number of the current row in the result set.  |
| GetStatus( )           | Returns the processing status of the asynchronous database operation on the database server.                      |
| GetValueBinary( )      | Returns the value of a BINARY column from the current row in the result set.                                      |
| GetValueBinaryPiece( ) | Returns the value of a LONGBINARY column from the current row in the result set.                                  |
| GetValueDateString( )  | Returns the value of a Date type column from the current row in the result set.                                   |
| GetValueDouble( )      | Returns the value of a double type column from the current row in the result set.                                 |
| GetValueInt( )         | Returns the value of an int type column from the current row in the result set.                                   |
| GetValueSize( )        | Returns the cumulative number of bytes that have been fetched from a column in the current row of the result set. |
| GetValueString( )      | Returns the value of a String type column from the current row in the result set.                                 |
| GetValueText( )        | Returns the value of a TEXT column from the current row in the result set.  |
| GetValueTextPiece( )   | Returns the value of a LONGTEXT column from the current row in the result set.                                    |
| MoveTo( )              | Moves to the specified row in the result set.   |
| RowCount( )            | Returns the total number of rows retrieved thus far from the data source.   |
| WasNull( )             | Checks if the value of a column is null or not.   |

## Related Topics

ExecuteQuery( ) in the IGXDataConn interface

Execute( ) in the IGXPreparedQuery interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## Close()

---

Releases the connection used by the result set.

**Syntax** `HRESULT FetchNext(  
    DWORD dwFlags);`

**dwFlags** Specify 0 (zero). Internal use only.

**Usage** Call Close() to release a connection used by a result set object when the connection is no longer required. An AppLogic should release unused connections to prevent bottlenecks, especially for applications that support many concurrent users, or that access heavily-used databases.

**Tip** After calling Close(), release the result set object by calling its Release() method.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

## EnumColumnReset()

---

Resets the column enumeration to the first column in the result set.

**Syntax** `HRESULT EnumColumnReset();`

**Usage** Use EnumColumnReset() before iterating through and retrieving columns in a result set. The EnumColumnReset() method ensures that column retrieval starts from the first column.

Thereafter, use EnumColumns() to retrieve each column sequentially. Each EnumColumns() call returns an IGXColumn object for the next column.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## EnumColumns()

---

Returns the definition of the next column in the result set.

**Syntax** `HRESULT EnumColumns(  
    IGXColumn **ppColumn);`

**ppColumn** Pointer to the IGXColumn object that contains the returned column of data. When the client code is finished using the object, call the `Release()` method to release the interface instance.

**Usage** Use `EnumColumns()` when the column definition is unknown and required for subsequent operations. The AppLogic can use the returned IGXColumn object to determine characteristics of the column, such as its name, data type, size, whether nulls are allowed, and so on.

Before iterating through columns, the AppLogic should call `EnumColumnReset()` to ensure that `EnumColumns()` starts with the first column in the table. Each subsequent `EnumColumns()` call moves to the next sequential column in the result set and retrieves its column definition in an IGXColumn object.

- Tips**
- The columns might not be returned in the order in which they are defined in the database catalog.
  - Test for NULL to determine when the last column has been retrieved.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** IGXColumn interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

---

## FetchNext()

Retrieves the next row in the result set.

**Syntax** `HRESULT FetchNext();`

**Usage** Use `FetchNext()` when iterating through rows in the result set to retrieve the contents of the next sequential row and put them in the row buffer for subsequent processing (if `RS_BUFFERING` has been turned ON).

If result set buffering was activated, `FetchNext()` checks the buffer first before fetching the result set from the actual data source. For more information about result set buffering, see the description of the `props` parameter of `ExecuteQuery()` in the IGXDataConn interface.

- Tips**
- If the AppLogic needs to iterate through the result set more than once, be sure to start with the first row again by calling MoveTo() and specifying row number 1. This works only when buffering is enabled.
  - If result set buffering is enabled, the AppLogic can use MoveTo() to go to any row in the buffer.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

- If the end of the result set has been reached, HRESULT is set to GX\_DA\_END\_OF\_FETCH, a macro-based constant (defined in gxidata.h).
- If the length of the buffer has been exceeded, HRESULT is set to GX\_DA\_BUFFER\_EXCEEDED, a macro-based constant (defined in gxidata.h).

**Related Topics** ExecuteQuery() in the IGXDataConn interface

Execute() in the IGXPreparedQuery interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

---

## GetColumn()

Returns the column definition of the column with the specified name.

**Syntax**

```
HRESULT GetColumn(
    LPSTR colName,
    IGXColumn **ppCol);
```

**colName** Name of a column or column alias (such as computed columns) in the result set, or an empty string if no alias is specified for the computed column.

**ppCol** Pointer to the IGXColumn object that contains the returned column definition. Calling GetColumn() creates the IGXColumn object automatically. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use `GetColumn()` when the data definition of the column is unknown and is required for subsequent operations. The `AppLogic` can then use methods in the `IGXColumn` interface to obtain descriptive information about a table column from the database catalog, such as the column name, precision, scale, size, table, and data type.

- Tips**
- Use `GetColumnByOrd()` instead when the column position is known but its name is unknown.
  - Columns that are the result of query expressions or formulas, such as `invoice.count * product.price`, should have a field alias for the column in the result set. Otherwise, the `AppLogic` can refer to the column only by its ordinal position. The `AppLogic` calls `SetFields()` in the `IGXQuery` interface to specify field aliases.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `IGXColumn` interface

`ExecuteQuery()` in the `IGXDataConn` interface

`Execute()` in the `IGXPreparedQuery` interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

---

## GetColumnByOrd()

Returns the column definition for the column in the specified ordinal position.

**Syntax**

```
HRESULT GetColumnByOrd(
    ULONG colIndex,
    IGXColumn **ppCol);
```

**colIndex** Ordinal position of a column in the result set. The ordinal position of the first column in the result set is 1, the second column is 2, and so on. The ODBC maximum is 255 columns.

**ppCol** Pointer to the `IGXColumn` object that contains the returned column definition. Calling `GetColumnByOrd()` creates the `IGXColumn` object automatically. When the `AppLogic` is finished using the object, call the `Release()` method to release the interface instance.

**Usage** Use `GetColumnByOrd()` when the name of the column is unknown and is required for subsequent operations. The AppLogic can then use methods in the `IGXColumn` interface to obtain descriptive information about a table column from the database catalog, such as the column name, precision, scale, size, table, and data type.

**Tip** Use `GetColumn()` instead when the column name is known but its ordinal position is unknown.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `IGXColumn` interface

`ExecuteQuery()` in the `IGXDataConn` interface

`Execute()` in the `IGXPreparedQuery` interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

---

## GetColumnOrdinal()

Returns the ordinal position of the column with the specified name.

**Syntax** `HRESULT GetColumnOrdinal(  
LPSTR szColumn,  
ULONG *pOrdinal);`

**szColumn** Name of a column in the result set.

**pOrdinal** Pointer to the variable that contains the returned ordinal position of the specified column. The ordinal position of the first column in the result set is 1, the second column is 2, and so on.

**Usage** Use `GetColumnOrdinal()` when the ordinal position of the column is unknown but is required for subsequent operations. For example, the ordinal position of a column is a required parameter value for the `GetValue**()` methods, such as `GetValueString()` and `GetValueInt()`.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```
IGXHierResultSet *pHRset=NULL;

// Execute a hierarchical query
if(((hr=pHq->Execute(0, 0, NULL, &pHRset))== GXE_SUCCESS)&&pHRset) {
```

```

IGXResultSet *pRset=NULL;

// Get a result set from the hierarchical result set
if(((hr=pHRset->GetResultSet("SelCust", &pRset))!= GXE_SUCCESS)&&pRset)
{

// Retrieve a value from the result set
// First, get the ordinal position of the column
ULONG ssnIndex=0;
pRset->GetColumnOrdinal("ssn", &ssnIndex);

char tmpStr[200];

// Next, get the value of the specified column
pRset->GetValueString(ssnIndex, tmpStr, 200);

```

**Related Topics** ExecuteQuery() in the IGXDataConn interface

Execute() in the IGXPreparedQuery interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## GetNumColumns()

---

Returns the number of columns in the result set.

**Syntax** HRESULT GetNumColumns(  
    ULONG \*pnCols);

**pnCols** Pointer to the variable that contains the returned number of columns in the result set.

**Usage** Use GetNumColumns() if the number of columns in the result set is unknown and required for subsequent operations. For example, when iterating through columns in the result set, the AppLogic can use this information to specify the maximum number of iterations.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** ExecuteQuery() in the IGXDataConn interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## GetOrder()

---

For asynchronous queries, returns an IGXOrder object used for obtaining the current status and return value of the query.

**Syntax** HRESULT GetOrder(  
IGXOrder \*\*ppOrder);

**ppOrder** Pointer to the IGXOrder object that contains the returned IGXOrder object. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use GetOrder() to create an IGXOrder object that the AppLogic can use to return status information about an asynchronous query.

**Rule** The query must be run asynchronously. To run an asynchronous query, the AppLogic must specify GX\_DA\_EXEC\_ASYNC as the dwFlags parameter in ExecuteQuery() in the IGXDataConn interface.

**Tips**

- The AppLogic can determine the status of the query (active, done, cancelled, or unknown) using GetState() in the IGXOrder interface.
- Alternatively, use the GXWaitForOrder() function, which waits until the asynchronous operation is done, to determine the processing status of an asynchronous query.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** IGXOrder interface

ExecuteQuery() in the IGXDataConn interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## GetRowNumber()

---

Returns the number of the current row in the result set.

**Syntax** HRESULT GetRowNumber(  
ULONG \*pOrd);

**pOrd** Pointer to the variable that contains the returned row number. The number of the first row in the result set is 1, the second row is 2, and so on. If zero is returned the first time the AppLogic calls `GetRowNumber()`, that means the result set is empty.

**Usage** When iterating through rows in the result set, use `GetRowNumber()` to keep track of the number of rows processed.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `ExecuteQuery()` in the `IGXDataConn` interface

`Execute()` in the `IGXPreparedQuery` interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

---

## GetStatus()

Returns the processing status of the asynchronous database operation on the database server.

**Syntax**

```
HRESULT GetStatus(  
    DWORD *pStatus);
```

**pStatus** Pointer to the variable that contains the returned status code. The variable is set to one of the following macro-based constants (defined in `gxioder.h`):

| Constant                           | Description  |
|------------------------------------|--|
| <code>GXORDER_STATE_ACTIVE</code>  | The asynchronous database operation is still being processed.      |
| <code>GXORDER_STATE_CANCEL</code>  | The asynchronous database operation has been cancelled.            |
| <code>GXORDER_STATE_DONE</code>    | The asynchronous database operation has been completely processed. |
| <code>GXORDER_STATE_UNKNOWN</code> | The status of the asynchronous database operation is unknown.      |

**Usage** Use `GetStatus()` to return status information to use in error-handling code.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

---

## GetValueBinary()

Returns the value of a BINARY column in the current row of the result set.

**Syntax** `HRESULT GetValueBinary(  
    ULONG Ordinal,  
    LPBYTE pValue,  
    ULONG nSize);`

**Ordinal** Ordinal number (position) of the column in the table definition. The first column is 1, the second column is 2, and so on.

**pValue** Pointer to the buffer that contains the returned column value.

**nSize** Size of the buffer to contain the returned column value.

**Usage** Use GetValueBinary() to retrieve binary data of which the total size is equal to or smaller than 64Kb. If the value of the data is larger than 64Kb, use GetValueBinaryPiece().

**Rule** The data type of the column must be BINARY, VARBINARY, or equivalent database type.

**Tip** If the value of the data is of type LONGBINARY, use GetValueBinaryPiece().

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

---

## GetValueBinaryPiece()

Returns the value of a LONGBINARY column in the current row from the result set.

**Syntax** `HRESULT GetValueBinaryPiece(  
    ULONG Ordinal,  
    ULONG nLength,  
    LPBYTE pValue,`

```
ULONG nSize);
```

**Ordinal** Ordinal number (position) of the column in the table definition. The first column is 1, the second column is 2, and so on.

**nLength** The requested length of the data, in bytes. Up to 64Kb.

**pValue** Pointer to the buffer that contains the returned column value.

**nSize** Size of the client-allocated buffer to contain the returned column value.

**Usage** Use `GetValueBinaryPiece()` to retrieve binary data of which the total size is larger than 64K. Such binary data must be retrieved in 64K increments. Therefore, you might use `GetValueBinaryPiece()` several times to retrieve large amounts of data.

**Rules**

- The data type of the column must be longvarbinary or equivalent database vendor binary type.
- You cannot call `GetValueBinaryPiece()` for a row after you call `FetchNext()`.

**Tips**

- To determine the total size of the binary data that has been retrieved, use `GetValueSize()`.
- To retrieve binary data of which the total size is less than 64Kb, use `GetValueBinary()`.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** The following example shows how to retrieve BLOBs from a database:

```
HRESULT hr;
IGXQuery      *pQuery = NULL;
IGXResultSet  *pRS    = NULL;

CreateQuery(&pQuery);

pQuery->SetTables("blobtable");
pQuery->SetFields("blobcol");

hr = pConn->ExecuteQuery(0, pQuery, NULL, NULL, &pRS);
if (hr == GXE_SUCCESS && pRS != NULL)
{
    ULONG nRows;
```

```

hr = pRS->GetRowNumber(&nRows);

if (hr == GXE_SUCCESS && nRows)
{
    LPBYTE pBlobChunk = NULL;
    ULONG expectSize, gotSize;
    expectSize = 65535;

    pBlobChunk = new LPBYTE[65536];
    if (!pBlobChunk)
        return -1;

    hr = pRS->GetValueBinaryPiece(1, expectSize, &pBlobChunk, 65536);

    if (hr == GXE_SUCCESS)
    {
        pRS->GetValueSize(1, &gotSize);
        if (gotSize == expectSize)
            fprintf(stderr, "got a full chunk, size = %d\n",
                    gotSize);
        else
            fprintf(stderr, "got a partial chunk, size = %d\n",
                    gotSize);
    }
}
pRS->Release();
}

```

**Related Topics** “Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

---

## GetValueDateString()

Returns the value of a Date type column, as a string, from the current row in the result set.

**Syntax** HRESULT GetValueDateString(  
 ULONG colIndex,  
 LPSTR pVal,  
 ULONG nVal);

**colIndex** Ordinal position of a column in the result set. The ordinal position of the first column in the result set is 1, the second column is 2, and so on.

**pVal** Pointer to the variable that contains the returned column value.

**nVal** Length of the variable.

**Usage** Use `GetValueDateString()` to retrieve date values from the result set for subsequent processing. The following is an example of the format in which `GetValueDateString()` returns a date:

```
Jan 26 1998 12:35:00
```

**Rule** The specified column must be a Date, Date Time, or Time data type.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## GetValueDouble()

---

Returns the value of a double type column from the current row in the result set.

**Syntax**

```
HRESULT GetValueDouble(
    ULONG colIndex,
    double *pVal);
```

**colIndex** Ordinal position of a column in the result set. The ordinal position of the first column in the result set is 1, the second column is 2, and so on.

**pVal** Pointer to the variable that contains the returned column value.

**Usage** Use `GetValueDouble()` to retrieve decimal, floats, real, numeric, and double values from the result set for subsequent processing.

**Rule** The specified column must be a double data type.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## GetValueInt()

---

Returns the value of an int type column from the current row in the result set.

**Syntax** `HRESULT GetValueInt(  
    ULONG colIndex,  
    ULONG *pVal);`

**colIndex** Ordinal position of a column. The ordinal position of the first column in the result set is 1, the second column is 2, and so on.

**pVal** Pointer to the variable that contains the returned column value.

**Usage** Use `GetValueInt()` to retrieve int or long values from the result set for subsequent processing.

**Rule** The specified column must be an int or long data type.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** “Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## GetValueSize()

---

Returns the cumulative number of bytes that have been fetched from a column in the current row of the result set.

**Syntax** `HRESULT GetValueSize(  
    ULONG colIndex,  
    ULONG *pSize);`

**colIndex** Ordinal number (position) of the column in the table definition. The first column is 1, the second column is 2, and so on.

**pSize** Pointer to the buffer that contains the returned number of bytes that have been fetched.

**Usage** Use `GetValueSize()` during data retrieval to check the size of the BLOB column that has been retrieved. When the `AppLogic` first calls `GetValueSize()` before calling `GetValueBinaryPiece()` to retrieve the value of a `LONGBINARY` column, `GetValueSize()` returns 0.

Each subsequent `GetValueSize()` call during data retrieval returns the cumulative size of the data that has been retrieved.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `GetValueBinaryPiece()`

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

---

## GetValueString()

Returns the value of a String type column from the current row in the result set.

**Syntax** HRESULT GetValueString(  
     ULONG colIndex,  
     LPSTR pVal,  
     ULONG nVal);

**colIndex** Ordinal position of a column in the result set. The ordinal position of the first column in the result set is 1, the second column is 2, and so on.

**pVal** Pointer to the variable that contains the returned column value.

**nVal** Length of the variable.

**Usage** Use `GetValueString()` to retrieve String values from the result set for subsequent processing.

**Rule** The specified column must be a String data type.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

```
IGXHierResultSet *pHRset=NULL;

// Execute a hierarchical query
if(((hr=pHq->Execute(0, 0, NULL, &pHRset))== GXE_SUCCESS)&&pHRset) {

IGXResultSet *pRset=NULL;

// Get a result set from the hierarchical result set
if(((hr=pHRset->GetResultSet("SelCust", &pRset))== GXE_SUCCESS)&&pRset)
{

// Retrieve a value from the result set
```

```
// First, get the ordinal position of the column
ULONG ssnIndex=0;
pRset->GetColumnOrdinal("ssn", &ssnIndex);

char tmpStr[200];

// Next, get the value of the specified column
pRset->GetValueString(ssnIndex, tmpStr, 200);
```

**Related Topics** “Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## GetValueText()

---

Returns the value of a TEXT column in the current row from the result set.

**Syntax** HRESULT GetValueText(  
     ULONG Ordinal,  
     LPSTR pValue,  
     ULONG nSize);

**Ordinal** Ordinal number (position) of the column in the table definition. The first column is 1, the second column is 2, and so on.

**pValue** Pointer to the buffer that contains the returned column value.

**nSize** Size of the client-allocated buffer to contain the returned column value.

**Usage** Use GetValueText() to retrieve TEXT data of which the total size is equal to or smaller than 64K.

**Rule** The data type of the column must be TEXT or database equivalent.

**Tips**

- To determine the actual size of the TEXT data, use GetValueSize().
- If the value of the data is of type LONGTEXT, use GetValueTextPiece().

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## GetValueTextPiece()

---

Returns the value of a LONGTEXT column in the current row from the result set.

**Syntax** HRESULT GetValueTextPiece(  
    ULONG Ordinal,  
    ULONG nLength,  
    LPSTR pValue,  
    ULONG nSize);

**Ordinal** Ordinal number (position) of the column in the table definition. The first column is 1, the second column is 2, and so on.

**nLength** The requested length of the data, in bytes. Up to 64Kb.

**pValue** Pointer to the buffer that contains the returned column value.

**nSize** Size of the client-allocated buffer to contain the returned column value.

**Usage** Use GetValueTextPiece() to retrieve LONGTEXT data. LONGTEXT values must be retrieved in 64K increments, therefore, you must use GetValueTextPiece() repeatedly to retrieve the data.

**Rules**

- The data type of the column must be LONGTEXT or database equivalent.
- Call GetValueTextPiece() until you get all the data before calling FetchNext() again.

**Tips**

- To determine the actual size of the LONGTEXT data, use GetValueSize(). The actual size of the data determines the number of times you need to call GetValueTextPiece().
- For data of type TEXT, use GetValueText().

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## MoveTo()

---

Moves to the specified row in the result set.

**Syntax** HRESULT MoveTo(

```
ULONG nRow);
```

**nRow** Number of the row in the result set to move to. The number of the first row in the result set is 1, the second row is 2, and so on.

**Usage** Use `MoveTo()` to move the internal cursor to a specific row in the result set, skipping over rows to be excluded from processing. In addition, if `RS_BUFFERING` is `ON`, after iterating through all rows in a result set, the `AppLogic` can return to the first row in the result set in preparation for the next iteration.

**Rules**

- The specified row number must exist in the result set.
- If row buffering is not enabled for the result set, the `AppLogic` can move forward to subsequent rows only. The `AppLogic` cannot return to rows that have been processed previously.

**Tip** Use `RowCount()`, if the database driver supports it, to obtain the maximum number of rows in the result set.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds. If the end of the result set has been reached, `HRESULT` is set to `GXE_EOF`.

**Related Topics** `ExecuteQuery()` in the `IGXDataConn` interface

`Execute()` in the `IGXPreparedQuery` interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## RowCount()

---

Returns the total number of rows retrieved thus far from the data source.

**Syntax** `HRESULT RowCount(  
    ULONG *nRows);`

**nRows** Pointer to the variable that contains the returned number of rows in the result set.

**Usage** Use `RowCount()` to return the current number of rows processed so far in the result set. This method is useful for checking that data exists in the result set before processing the result set.

If iterating through rows in a result set that has been completely returned, use `RowCount()` to determine the current maximum number of rows to process.

**Tip** If result set buffering is enabled, the AppLogic can use `RowCount()` to find the current number of rows in the buffer.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
// Execute the query
IGXResultSet *pRset=NULL;

if(((hr=pConn->ExecuteQuery(0, pQuery, NULL, NULL,
&pRset))==GXE_SUCCESS)&&pRset) {

    // Check if there is data in the result set
    ULONG numRows=0;
    if(((hr=pRset->RowCount(&numRows))==GXE_SUCCESS)&&numRows)
    {
        // Process result set
    }
}
```

**Related Topics** ExecuteQuery() in the IGXDataConn interface

Execute() in the IGXPreparedQuery interface

“Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.

## WasNull()

Checks if the value of a column is null or not.

**Syntax** HRESULT WasNull(  
    ULONG Ordinal,  
    BOOL \*bNull);

**Ordinal** Ordinal number (position) of the column in the table definition. The first column is 1, the second column is 2, and so on.

**bNull** Pointer to the client-allocated BOOL variable that contains the returned information.

**Usage** Use WasNull() to check if a column value is null or not. This method is useful for determining if a null return value is an error condition or if the column contained no value.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Getting Data From a Flat Query’s Result Set” in Chapter 6, “Querying a Database,” in *Programmer’s Guide*.



# IGXSequence interface

The IGXSequence interface represents a sequence in an underlying database. Sequences are implemented in the database server to provide unique, incremental numbers assigned to records in a database. For example, the AppLogic can create a customer ID sequence to generate customer IDs, or create a purchase order sequence to generate purchase order numbers.

The IGXSequence interface provides methods to determine the current sequence value or to increment to the next sequence value. Sequences are useful for many types of applications, such as order entry applications.

The IGXSequence interface is part of the Data Access Engine (DAE) service.

To create an instance of the IGXSequence interface, use CreateSequence() in the IGXSequenceMgr interface, as shown in the following example:

```
IGXDataConn *conn
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL, &conn);

//Cast the connection to the ISequenceMgr interface
hr = conn->QueryInterface(IID_IGXSequenceMgr, (LPVOID *)
    &seqmgr);

IGXSequence *seq = NULL;
hr = seqmgr->CreateSequence("mySeq", "orders.ID", 100,
    1, NULL, &seq);
```

## Include File

gxisequence.h

## Methods

| Method       | Description  |
|--------------|--|
| Drop()       | Deletes the sequence from the database.                    |
| GetCurrent() | Returns the current value in the sequence.                 |
| GetNext()    | Increments the sequence and returns its incremented value. |

## Related Topics

CreateSequence( ) in the IGXSequenceMgr interface

“Using Sequences” in Chapter 5, “Working with Databases,” in *Programmer's Guide*.

---

## Drop()

Deletes the sequence from the database.

**Syntax** HRESULT Drop( );

**Usage** Use Drop( ) to remove a sequence from the database. Be careful when using this method. If the database implements the sequence as a field in a table, Drop( ) will delete the entire table, not just the sequence field. If the database implements the sequence as an object, as does Oracle for example, Drop( ) deletes only the sequence object.

Typically, once you start a sequence there is no reason to delete it. The sequence is normally used to create a permanent, unique numbering system for data in a database. However, you might use Drop( ) if you are using the sequence mechanism to generate unique sequential numbers for a temporary programmatic purpose.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateSequence( ) in the IGXSequenceMgr interface

“Using Sequences” in Chapter 5, “Working with Databases,” in *Programmer's Guide*.

---

## GetCurrent()

Returns the current value in the sequence.

**Syntax** HRESULT GetCurrent(  
    DWORD \*dwCurrVal);

**dwCurrVal** Pointer to the variable that contains the returned current value of the sequence.

**Usage** Use `GetCurrent()` to obtain the current value of the sequence without actually incrementing the sequence value.

Alternatively, use `GetNext()` to increment the sequence and obtain its incremented value.

**Rule** For Oracle databases, the session must first call `GetNext()` before it can call `GetCurrent()`.

**Tip** Unlike `GetNext()`, calling `GetCurrent()` does not change the value of the sequence.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `CreateSequence()` in the `IGXSequenceMgr` interface

“Using Sequences” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

---

## GetNext()

Increments the sequence and returns its incremented value.

**Syntax**

```
HRESULT GetNext(
    DWORD *dwCurrVal);
```

**dwCurrVal** Pointer to the variable that contains the returned incremented value of the sequence.

**Usage** Use `GetNext()` to increment and return the value of the sequence by the amount specified in the `dwIncrement` parameter in the `CreateSequence()` method in the `IGXSequenceMgr` interface. The incrementation value is always a positive integer.

Alternatively, use `GetCurrent()` to obtain the current value of the sequence without incrementing the sequence.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

- Rules**
- For Informix and Sybase databases, the session that creates the sequence must call `GetNext()` at least once before any other session can call `GetSequence()` in the `IGXSequenceMgr` interface.
  - For Oracle databases, the session must first call `GetNext()` before it can call `GetCurrent()`.

**Tip** Successive calls to `GetNext()` return successive integers.

**Example**

```

hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL, &conn);
if (hr == NOERROR &&
    conn)
{
    IGXSequenceMgr *seqmgr;

    // Cast the connection to an ISequenceMgr interface
    // and set up the sequence
    hr = conn->QueryInterface(IID_IGXSequenceMgr, (LPVOID *)
        &seqmgr);
    if (hr == NOERROR)
    {
        IGXSequence *seq = NULL;
        hr = seqmgr->CreateSequence("mySeq", "orders.ID", 100,
            1, NULL, &seq);
        if (hr == NOERROR &&
            seq)
        {
            DWORD seqVal = 0;
            // To start the sequence, call GetNext()
            hr = seq->GetNext(&seqVal);
            if (hr == NOERROR)
            {
                // Use the sequence number.
                //
                IGXQuery *qry;
                CreateQuery(&qry);
                char tmp[512];
                sprintf(tmp, "INSERT into orders (ID) values
                    (%d), (cust) values (%s)", seqVal,
                    custName);

                qry->SetSQL(tmp);

                // ... Execute insert command.
            }
        }
    }
}

```

**Related Topics** `CreateSequence()` in the `IGXSequenceMgr` interface

“Using Sequences” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

# IGXSequenceMgr interface

The IGXSequenceMgr interface provides methods for creating and retrieving an IGXSequence object, which represents a sequence in an underlying database. Sequences provide unique, incremental numbers assigned to records in a database. After creating a sequence by calling CreateSequence( ), the AppLogic can use methods in the IGXSequence interface to retrieve sequence values.

The IGXSequenceMgr interface is part of the Data Access Engine (DAE) service.

The IGXSequenceMgr interface is implemented by the IGXDataConn object. To use it, cast IGXDataConn to the IGXSequenceMgr interface, as shown in the following example:

```
IGXDataConn *dc;
IGXSequenceMgr *sm;
dc->QueryInterface(IID_IGXSequenceMgr, (LPVOID *) &sm);
```

## Include File

gxisequence.h

## Methods

| Method           | Description   |
|------------------|---|
| CreateSequence() | Creates a new sequence object in the underlying database.                                       |
| GetSequence()    | Returns an existing sequence object for the specified sequence name in the underlying database. |

## Related Topics

IGXSequence interface

## CreateSequence()

Creates a new sequence object in the underlying database.

**Syntax** HRESULT CreateSequence(

```

LPSTR  szName,
LPSTR  szCol,
DWORD  dwStart,
DWORD  dwIncrement,
LPSTR  szOptions,
IGXSequence **ppSequence);

```

**szName** Name of the sequence. The name can be simple (such as "mySeq") or qualified with the name of the database owner (such as "mary.mySeq").

**szCol** Name of the column in the database table to use if the database supports sequence column types. For more information, see your database vendor's documentation. If NULL, defaults to "SEQVAL".

**dwStart** Starting value of the sequence. Must be a positive integer.

**dwIncrement** Value by which to increment the sequence with each call to `GetNext()`. Must be a positive integer. Defaults to one (1). Not all databases support this feature. For more information, see your database vendor's documentation.

**szOptions** Additional sequence creation options that are database vendor-specific:

- For Oracle, these are options to the "CREATE Sequence" command.
- For SQL Server (Sybase and Microsoft) databases, these are column options for the "CREATE Table" command.
- For Informix, no options exist.

For more information, see your database vendor's documentation.

**ppSequence** Pointer to the returned `IGXSequence` object. When the `AppLogic` is finished using the object, call the `Release()` method to release the interface instance.

**Usage** Use `CreateSequence()` to create a new `IGXSequence` object, representing an incremental number generator, with the specified starting value. The `AppLogic` can then use methods in the `IGXSequence` interface to obtain the current or next value of this sequence object.

Sequences provide unique, incremental numbers assigned to records in a database. For example, you can create a customer ID sequence to generate customer IDs, or create a purchase order sequence to generate purchase order numbers.

**Tip** For Oracle databases, `CreateSequence()` creates a sequence object. For Sybase, Informix, and Microsoft SQL Server databases, `CreateSequence()` creates a table object with a sequence column.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL, &conn);
if (hr == NOERROR &&
    conn)
{
    IGXSequenceMgr *seqmgr;

    // Cast the connection to an ISequenceMgr interface
    // and set up the sequence
    hr = conn->QueryInterface(IID_IGXSequenceMgr, (LPVOID *)
        &seqmgr);
    if (hr == NOERROR)
    {
        IGXSequence *seq = NULL;
        hr = seqmgr->CreateSequence("mySeq", "orders.ID", 100,
            1, NULL, &seq);
        if (hr == NOERROR &&
            seq)
        {
            DWORD seqVal = 0;
            // To start the sequence, call GetNext()
            hr = seq->GetNext(&seqVal);
            if (hr == NOERROR)
            {
                // Use the sequence number.
                //
                IGXQuery *qry;
                CreateQuery(&qry);
                char tmp[512];
                sprintf(tmp, "INSERT into orders (ID) values
                    (%d), (cust) values (%s)", seqVal,
                    custName);
            }
        }
    }
}
```

```

gry->SetSQL(tmp);

// ... Execute insert command.

```

**Related Topics** IGXSequence interface

## GetSequence()

Returns an existing sequence object, for the specified sequence name, from the underlying database.

**Syntax**

```

IGXDataConn *dc;
IGXSequenceMgr *sm;
dc->QueryInterface(IID_IGXSequenceMgr, (LPVOID *) &sm);

HRESULT GetSequence(
    LPSTR szName,
    LPSTR szCol,
    IGXSequence **ppSequence);

```

**szName** Name of the sequence. The name can be simple (such as "mySeq") or qualified with the name of the database owner (such as "mary.mySeq").

**szCol** Name of the column in the database table to use if the database supports sequence column types. For more information, see your database vendor's documentation. If NULL, defaults to "SEQVAL".

**ppSequence** Pointer to the returned IGXSequence object. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use GetSequence() to obtain the IGXSequence object with the specified name in the underlying database. The AppLogic can then use methods in the IGXSequence interface to obtain the current or next value of this sequence object.

Sequences provide unique, incremental numbers assigned to records in a database. For example, you can create a customer ID sequence to generate customer IDs, or create a purchase order sequence to generate purchase order numbers.

- Rules**
- Use CreateSequence() to create the IGXSequence object.
  - The specified sequence name must be valid.

- For Informix and Sybase databases, the session that creates the sequence must call `GetNext()` at least once before any other session can call `GetSequence()`.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** IGXSequence interface



# IGXSession2 interface

The IGXSession2 interface represents a session between a user and an application. AppLogics use sessions to store information about each user's interaction with the application. For example, a login AppLogic might create a session object to store the user's login name and password. This session data is then available to other AppLogics in the application.

Session data is stored in a distributed state layer in the Netscape Application Server, so that the data is available even when the server destroys the AppLogic when it has finished executing. Storing the session data in the distributed state layer also enables AppLogics running in different clusters or servers to access the data.

A session has the following attributes, which are set when the AppLogic creates a session:

- A unique ID. You can specify an ID, or use the default ID the system generates.
- An association with an application. This setting enables the Netscape Application Server to determine which AppLogics have access to the session data.
- A timeout value. You can specify if the session is automatically destroyed after a specified time. If you don't specify a timeout value (timeout = 0), the session is destroyed when you call the DestroySession( ) method in the GXAppLogic class.
- Scope. You can specify if the session data is available at the local, cluster, or enterprise-wide level.

The IGXSession2 interface defines methods for setting and retrieving data in a session. It also defines methods for retrieving the attributes—ID, associated application, timeout value, and scope—of a session.

To create an instance of the IGXSession2 interface, use the CreateSession( ) method in the GXAppLogic class.

If your application requires a custom session object, for example, to support additional methods, you can subclass the GXSession2 class and define your own methods.

## Include File

gxapplogic.h

## Methods

| Method               | Description  |
|----------------------|--|
| GetSessionApp( )     | Returns the name of the application associated with the session.   |
| GetSessionData( )    | Returns session data.  |
| GetSessionFlags( )   | Returns the flags associated with the session when it was created. |
| GetSessionID( )      | Returns the session ID.  |
| GetSessionTimeout( ) | Returns the session's timeout value in seconds.                    |
| SaveSession( )       | Saves changes to a session.  |
| SetSessionData( )    | Sets session data.   |

## GetSessionApp( )

Returns the name of the application associated with the session.

**Syntax** HRESULT GetSessionApp(  
LPSTR pAppName  
ULONG nAppName);

**pAppName** Pointer to the buffer allocated by the client to store the returned application name.

**nAppName** The size of the buffer to store the application name.

**Usage** Use GetSessionApp( ) to retrieve the name of the application associated with the session. All AppLogics in an application have access to the same session data.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** The following code shows how to create a session and get the name of the application associated with the session:

```

HRESULT hr;
CHAR    AppName[128];
IGXSession2 *m_pSession

//Create a session and associate it with myApp application
hr = CreateSession(GXSESSION_DISTRIB, 0, "myApp",
                  NULL, NULL, &m_pSession);

//Get the application name associated with the session
//GetSessionApp() should return "myApp"
hr = m_pSession->GetSessionApp(AppName, 128);
if (hr != GXE_SUCCESS)
    return Result("GetSessionApp returned error");
sprintf(msg, "Session application name:%s\n\n", AppName);
Log(msg);

```

**Related Topics** CreateSession() in the GXAppLogic class

“Starting a Session” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

## GetSessionData()

Returns session data.

**Syntax** HRESULT GetSessionData(  
           IGXValList \*\*ppSessionData);

**ppSessionData** Pointer to the IGXValList object that contains the returned session data. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use GetSessionData() to retrieve session data for processing. Data is returned in an IGXValList object. This method retrieves the contents that were last saved in the distributed store with SaveSession(). Use methods in the IGXValList interface to iterate through and access items in the IGXValList object.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** // Method for retrieving the user name from session data  
 STDMETHODCALLTYPE LPSTR  
           OBSession::GetUserName()  
 {

```

LPSTR pRet=NULL;

// Pull the username from the session data
IGXValList *pData=NULL;
if((GetSessionData(&pData)==GXE_SUCCESS)&&pData) {
    LPSTR pTmp=GXGetValListString(pData, "userName");

    if(pTmp) {
        pRet=new char[strlen(pTmp)+1];
        strcpy(pRet, pTmp);
    }
    pData->Release();
}
return pRet;
}

```

**Related Topics** CreateSession() in the GXAppLogic class

“Starting a Session” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

## GetSessionFlags()

Returns the flags associated with the session when it was created.

**Syntax** HRESULT GetSessionFlags(  
    DWORD \*pdwFlags);

**pdwFlags** Pointer to the client-allocated variable that contains the returned session flag.

**Usage** Use GetSessionFlags() to retrieve the flags that were specified when the session was created with CreateSession(). The following table describes the valid session flags:

| Flag              | Description   |
|-------------------|---|
| GXSESSION_LOCAL   | The session is visible to the local process only.           |
| GXSESSION_CLUSTER | The session is visible to all AppLogics within the cluster. |

| Flag                     | Description  |
|--------------------------|--|
| GXSESSION_DISTRIB        | The session is visible to all AppLogics in the enterprise environment.   |
| GXSESSION_TIMEOUT_CREATE | The session expires <i>n</i> seconds from the time the node was created. |

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** The following code shows how to create a session and get the associated flags:

```

HRESULT hr;
DWORD   Flag;
IGXSession2 *m_pSession

//Create a session with distributed scope
hr = CreateSession(GXSESSION_DISTRIB, 0, "myApp",
                  NULL, NULL, &m_pSession);

//Get the flag associated with the session
//GetSessionFlags() should return GXSESSION_DISTRIB
hr = m_pSession->GetSessionFlags(&Flag);
if (hr != GXE_SUCCESS)
    return Result("GetSessionFlags returned error");
sprintf(msg, "Session flag:0x%x\n", Flag);
Log(msg);

```

**Related Topics** CreateSession( ) in the GXAppLogic class

“Starting a Session” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

## GetSessionID()

Returns the session ID.

**Syntax** HRESULT GetSessionID(  
LPCSTR pSessID  
ULONG nSessID);

**pSessID** Pointer to the buffer allocated by the client to store the returned session ID.

**nSessID** The size of the buffer to store the session ID.

**Usage** Use `GetSessionID()` to retrieve the unique ID associated with a session. The `GetSessionID()` method returns the base or intrinsic ID, not the transformed IDs generated by a custom ID generator.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Example** The following code shows how to create a session and get the session ID:

```
HRESULT hr;
DWORD   SessID[128];
IGXSession2 *m_pSession

//Create a session using the default ID generator
hr = CreateSession(GXSESSION_DISTRIB, 0, "myApp",
                  NULL, NULL, &m_pSession);

//Get the session ID
hr = m_pSession->GetSessionID(SessID, 128);
if (hr != GXE_SUCCESS)
    return Result("GetSessionID returned error");
sprintf(msg, "Session ID:%s\n", SessID);
Log(msg);
```

**Related Topics** `CreateSession()` in the `GXAppLogic` class

“Starting a Session” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

## GetSessionTimeout()

Returns the session’s timeout value in seconds.

**Syntax** `HRESULT GetSessionData(`  
`ULONG *pTimeout);`

**pTimeout** Pointer to the buffer allocated by the client to store the returned timeout value.

**Usage** Use `GetSessionTimeout()` to find out if a session is terminated after a specified time, or if it needs to be terminated explicitly. A timeout value of 0 means the session ends when it is explicitly terminated with the `DestroySession()` method.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** The following code shows how to create a session and get session's timeout value:

```

HRESULT hr;
ULONG   Timeout;
IGXSession2 *m_pSession

//Create a session with no timeout value
hr = CreateSession(GXSESSION_DISTRIB, 0, "myApp",
                  NULL, NULL, &m_pSession);

//Get the timeout value
//getSessionTimeout() should return 0
hr = m_pSession->GetSessionTimeout(&Timeout);
if (hr != GXE_SUCCESS)
    return Result("GetSessionTimeout returned error");
sprintf(msg, "Session timeout value:%d\n", Timeout);
Log(msg);

```

**Related Topics** CreateSession() in the GXAppLogic class

“Starting a Session” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

## SaveSession()

---

Saves changes to a session.

**Syntax** HRESULT saveSession(  
          DWORD dwFlags);

**dwFlags** Specify 0 (zero).

**Usage** Use SaveSession() to ensure changes are saved in the distributed state storage area, which stores the session information for subsequent use if any other AppLogic objects are invoked within the same session.

**Tips**

- The AppLogic needs to call the SaveSession() method in the GXAppLogic class at least once to set a cookie, which passes the session ID between the Web browser and Netscape Application Server. The SaveSession() method

in the IGXSession2 interface only saves data to the distributed state store, whereas SaveSession() in the GXAppLogic class saves data to the distributed state store *and* sets a cookie.

- The AppLogic should call SaveSession() to save changes after updating session data with SetSessionData() or after modifying the IGXValList returned by GetSessionData().
- To improve performance, keep smaller amounts of information in the session.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateSession() and SaveSession() in the GXAppLogic class

“Starting a Session” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

## SetSessionData()

---

Updates session with specified data.

**Syntax**

```
HRESULT SetSessionData(
    IGXValList *pSessionData);
```

**pSessionData** The IGXValList object containing the session data to set.

**Usage** Use SetSessionData() to write or update session data. Session data is stored in a distributed state layer in the Netscape Application Server, making session data accessible to distributed server processes.

- Tips**
- The AppLogic should call SaveSession() to save changes after updating session data with SetSessionData().
  - To improve performance, keep smaller amounts of information in the session.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** CreateSession() in the GXAppLogic class

“Starting a Session” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

# IGXSessionIDGen interface

The IGXSessionIDGen interface represents a session ID generator. The session-related methods in the GXAppLogic class take an IGXSessionIDGen object as a parameter. By default, Netscape Application Server uses the IGXSessionIDGen object to generate a session ID when an AppLogic creates a new session with the CreateSession() method in the GXAppLogic class.

The session ID—based on a 64-bit number—uniquely identifies a session between a user and an application. In a Web-based application, session IDs are passed between the Web browser and Netscape Application Server to verify user sessions as users traverse the application. For non-browser clients, session IDs are tracked on the server.

If you want to use your own technique for generating session IDs, you can create a class that implements the IGXSessionIDGen interface and add your own code.

If your application requires additional security, you can implement a custom session ID generator that continually changes the session ID that is passed between the Web browser and Netscape Application Server. Internally, however, there must be a constant or base ID that remains unchanged for the Netscape Application Server to identify sessions correctly. Therefore, your custom code needs to implement an algorithm for creating and mapping variable IDs to a base ID.

The IGXSessionIDGen interface defines methods for generating session IDs, creating variable IDs, and mapping variable IDs to the base ID. To implement a custom session ID generator, create a class that implements the IGXSessionIDGen interface, and implement all the interface methods.

## Include File

gxapplogic.h

## Methods

| Method               | Description   |
|----------------------|---|
| GenerateSessID( )    | Generates a new session ID.                               |
| GenerateVariantID( ) | Accepts an input session ID and generates a different ID. |
| MapToBaseID( )       | Maps a variable session ID to a base ID.                  |

## Related Topics

CreateSession( ) and GetSession( ) in the GXAppLogic class

IGXSession2 interface

## GenerateSessID( )

Generates a new session ID.

**Syntax** HRESULT GenerateSessID(  
     DWORD dwFlags,  
     ULONG nSessID,  
     LPSTR pSessID);

**dwFlags** Specify 0. For internal use only.

**nSessID** The size of the buffer to store the returned session ID.

**pSessID** The buffer to store the returned session ID.

**Usage** When an AppLogic calls CreateSession( ) to create a new session, it needs to pass in a pointer to an IGXSessionIDGen object as an argument. If the AppLogic passes NULL, the Netscape Application Server uses the default IGXSessionIDGen to generate a session ID.

To use a different mechanism for generating session IDs, create a class that implements the IGXSessionIDGen interface and implement GenerateSessID( ). When you pass your session ID generator object as an argument to CreateSession( ), it invokes your implementation of GenerateSessID( ).

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** In the following code example, which is specific to the Solaris platform, `GenerateSessID()` is implemented to use a high resolution counter as the session ID:

```
STDMETHODIMP
MySessIDGen::GenerateSessID(DWORD dwFlags, ULONG nSessID, LPSTR
pSessID)
{
    if (!pSessID)
        return GXE_INVALID_ARG;
    WORD64 HiResCounter;
    HiResCounter = gethrtime();
    CHAR id[64]
    sprintf(id, "%lld", HiResCounter);
    strncpy(pSessID, id, nSessID);
    pSessID[nSessID-1] = '\0';

    return NOERROR;
}
```

**Related Topics** `GenerateVariantID()`, `MapToBaseID()`

## GenerateVariantID()

Accepts an input session ID and generates a different ID.

**Syntax** HRESULT GenerateVariantID(  
LPCSTR pBaseID,  
DWORD dwFlags,  
ULONG nVariantID  
LPSTR pVariantID);

**pBaseID** The base session ID from which the variable ID is to be generated.

**dwFlags** Specify 0 (zero). For internal use only.

**nVariantID** The size of the buffer to store the variable session ID.

**pSessID** The buffer to store the variable session ID.

**Usage** If your AppLogic creates a custom class to implement the `IGXSessionIDGen` interface, you need to implement all the methods in the interface, including `GenerateVariantID()`.

You can write `GenerateVariantID()` to implement a way to generate session IDs that change. Changing a session's ID as it is passed between the Web browser and Netscape Application Server provides additional security. Internally, however, the Netscape Application Server uses a base ID that does not change. Therefore, if you implement `GenerateVariantID()` to create variable IDs, you also need to write `MapToBaseID()` to convert variable IDs to a base ID.

If you don't want to generate variable IDs in your application, implement `GenerateVariantID()` to simply return the base ID.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Example** In the following code example, which is specific to the Solaris platform, `GenerateVariantID()` is implemented to generate different IDs from a base ID:

```
STDMETHODIMP
MySessIDGen::GenerateVariantID(LPCSTR pBaseID, DWORD dwFlags, ULONG
nVariantID, LPSTR pVariantID)
{
    if (!pBaseID || !pVariantID || nVariantID <= GXStrLen(pBaseID))
        return GXE_INVALID_ARG;

    CHAR id[64];
    WORD64 HiResCounter;
    HiResCounter = gethrtime();
    sprintf(id, "%lld.%s", HiResCounter, pBaseID);

    strncpy(pVariantID, id, nVariantID);
    pVariantID[nVariantID-1] = '\0';

    return NOERROR;
}
```

**Related Topics** `GenerateSessID()`, `MapToBaseID()`

## MapToBaseID()

Maps a variable session ID to a base ID.

**Syntax** HRESULT MapToBaseID(  
LPCSTR pVariantID,  
DWORD dwFlags,  
ULONG nBaseID,

```
LPSTR pBaseID);
```

**pVariantID** The variable ID to map to the base ID.

**dwFlags** Specify 0 (zero). For internal use only.

**nBaseID** The size of the buffer to store the returned base ID.

**pBaseID** The buffer to store the returned base ID.

**Usage** If your AppLogic creates a custom class to implement the IGXSessionIDGen interface, you need to implement all the methods in the interface, including MapToBaseID( ).

You can write MapToBaseID( ) in conjunction with GenerateVariantID( ) to implement a way to generate session IDs that change. Changing a session's ID as it is passed between the Web browser and Netscape Application Server provides additional security. Internally, however, the Netscape Application Server uses a base ID that does not change. Therefore, if you implement GenerateVariantID( ) to create variable IDs, you also need to implement MapToBaseID( ) to convert these variable IDs to a base ID.

If you don't want to generate variable IDs in your application, implement GenerateVariantID( ) and MapToBaseID( ) to simply return the base ID.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example** In the following code example, MapToBaseID( ) is overridden to convert variable IDs generated by GenerateVariantID( ) to the base ID:

```
STDMETHODIMP
MySessIDGen::MapToBaseID(LPCSTR pVariantID, DWORD dwFlags, ULONG
nBaseID, LPSTR pBaseID)
{
    if (!pVariantID || !pBaseID)
        return GXE_INVALID_ARG;

    LPSTR p = strchr(pVariantID, '.');
    if (!p)
        return GXE_FAIL;

    *p++ = '\0';

    CHAR id[64];
    sprintf(id, "%s", p);
```

```
        strncpy(pBaseID, id, nBaseID);  
        pBaseID[nBaseID-1] = '\\0';  
  
        *--p = '.';  
  
        return NOERROR;  
    }
```

**Related Topics**   [GenerateVariantID\(\)](#), [GenerateSessID\(\)](#)

# IGXState2 interface

The IGXState2 interface represents a node, or state object, in the State tree. A state tree is a hierarchical data storage mechanism. It is used primarily for storing application data that needs to be distributed across server processes and clusters. For example, the session data your application creates and maintains is stored in nodes of a state tree.

Use a state tree in your application if it needs to maintain and share data in a multi-server environment running load-balanced application components. A node has the following attributes:

- A name. Nodes on the same level of the state tree must have unique names, but not otherwise.
- Contents in the form of an IGXValList.
- A timeout value. You can specify if the content of the node automatically expires after a specified time. If you don't specify a timeout value (timeout = 0), the content is saved until the node is deleted explicitly.
- Scope. You can specify if the node data is available at the local, cluster, or enterprise-wide level.

The IGXState2 interface defines methods for creating and deleting nodes, setting and retrieving node contents, and retrieving the attributes of a node.

To create a state tree, use the following methods:

- GetStateTreeRoot( ) method in the GXAppLogic class to create the root node.
- CreateStateChild( ) in this interface to create the child nodes.

## Include File

gxistate.h

## Methods

| Method               | Description   |
|----------------------|---|
| CreateStateChild()   | Creates a child node under the node on which this method is called. |
| DeleteStateChild()   | Deletes a child node.   |
| GetStateChild()      | Gets a specified child node.  |
| GetStateChildCount() | Gets the count of children nodes.                                   |
| GetStateContents()   | Gets the contents of the node.                                      |
| GetStateFlags()      | Gets the flags assigned to the node when it was created.            |
| GetStateName()       | Returns the name of the node.                                       |
| GetStateTimeout()    | Returns the node's timeout value in seconds.                        |
| SaveState()          | Saves updates to the node contents.                                 |
| SetStateContents()   | Sets node contents.   |

## CreateStateChild()

Creates a child node under the node on which this method is called.

**Syntax**

```
HRESULT CreateStateChild(
    LPCSTR pName,
    ULONG Timeout,
    DWORD dwFlags,
    IGXState2 **ppChild);
```

**pName** The name of the child node. If a child node with the given name already exists, this method returns an error.

**Timeout** The unit of timeout is seconds. The meaning of timeout depends on the timeout flag specified in dwFlags. A value of 0 means the contents of the node is saved until deleted explicitly. You can assign a non-zero timeout value only to child nodes that are leaf nodes. Parent nodes can only have a timeout value of 0.

**dwFlags** Specify one of the following flags, or 0 to use the default system settings:

- `GXSTATE_LOCAL` to make the node visible to the local process only.
- `GXSTATE_CLUSTER` to make the node visible to all application components within the cluster.
- `GXSTATE_DISTRIB` to make the node visible to all application components in the enterprise environment.
- `GXSTATE_TIMEOUT_CREATE` to specify that the contents of the node expires *n* seconds from the time the node was created.

The default scope is distributed and the default timeout is 60 seconds from the time the node was last accessed.

**ppChild** A pointer to the created IGXState2 object. When the AppLogic is finished using the object, call the `Release()` method to release the interface instance.

**Usage** Use `CreateStateChild()` to add a child node to a state tree. The application component should already have created the root node of the tree with `GetStateTreeRoot()` in the `GXAppLogic` class.

To create a new child node in a particular position of the tree, traverse the tree until you reach the node that will be the parent of the new child node. Then call `CreateStateChild()`.

- Rules**
- The scope of a parent node must be the same as or greater than the scope of its child nodes. For example, if the scope of a child node is set to the cluster level, its parent node must be set to either the cluster or distributed level.
  - Parent nodes can only have a timeout value of 0.
- Tips**
- To traverse the state tree to find the desired location in which to create a new child node, use `GetStateChild()`. Each successive call to `GetStateChild()` descends one level in the tree.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Example** The following code shows how to create a child node if it doesn't already exist:

```
HRESULT hr;

hr = GetStateTreeRoot(GXSTATE_DISTRIB, "Grammy", &m_pStateRoot);

if (hr == NOERROR && m_pStateRoot)
```

```

{
    IGXState2 *pState = NOERROR;
    hr = m_pStateRoot->GetStateChild("Best Female Vocal",
        &pState);
    if (hr != NOERROR || !pState)
    {
        hr = m_pStateRoot->CreateStateChild("Best Female Vocal",
            0, GXSTATE_DISTRIB, &pState);
    }
}

```

**Related Topics** “Using the State Layer” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

---

## DeleteStateChild()

Deletes a child node from a state tree.

**Syntax** HRESULT DeleteStateChild(  
LPCSTR pName);

**pName** The name of the child node to delete.

**Usage** Use DeleteStateChild() to delete a child node from a state tree when your application no longer needs it. A child node can be deleted only from its parent node. For example, if the state tree has three levels and you want to delete a node at the third level, traverse the tree until you find its parent node at the second level. Then call DeleteStateChild() to delete a specific node.

**Rule** You can delete a parent node only after deleting its child nodes.

**Tip** To traverse the state tree to find the parent node of the child node to delete, use GetStateChild(). Each successive call to GetStateChild() descends one level in the tree.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Using the State Layer” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

---

## GetStateChild()

Gets a specified child node.

**Syntax** HRESULT GetStateChild(  
LPCSTR pName, IGXState2 \*pState);

```
LPCSTR pName,
IGXState2 **ppChild);
```

**pName** The name of the child node to get.

**ppChild** A pointer to the retrieved IGXState2 object. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use GetStateChild() to retrieve a node whose content you want to get or update. Your application component can also use GetStateChild() to traverse a state tree to find the parent node of child nodes to add or delete.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Using the State Layer” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

---

## GetStateChildCount()

Gets the count of children nodes.

**Syntax** HRESULT GetStateChildCount(  
     DWORD dwFlags,  
     ULONG \*pCount);

**dwFlags** Currently unused.

**pCount** Pointer to where the child count is returned.

**Usage** Use this method to return the number of children at any given state node.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Using the State Layer” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

---

## GetStateContents()

Gets the contents of the node.

**Syntax** HRESULT GetStateContents(  
     IGXVallist \*\*ppContents);

**ppContents** Pointer to the returned IGXValList that contains the contents of the current child node. When the AppLogic is finished using the object, call the Release( ) method to release the interface instance.

**Usage** Use GetStateContents( ) to retrieve the contents of the node, or to check if the node contains contents before setting values in the node. This method retrieves the contents that were last saved in the distributed store with SaveState( ).

- Tips**
- To traverse the state tree to find a specific node, use GetStateChild( ). Each successive call to GetStateChild( ) descends one level in the tree.
  - If you update the contents of a node with SetStateContents( ) but do *not* save the contents in the distributed store with SaveState( ), GetStateContents( ) will not return the content set with SetStateContents( ). It will return the contents that were last saved.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Using the State Layer” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

---

## GetStateFlags( )

Gets the flags assigned to the node when it was created.

**Syntax** HRESULT GetStateFlags(  
    DWORD \*pdwFlags);

**pdwFlags** Pointer to the client-allocated variable that contains the returned state flag.

**Usage** Use GetStateFlags( ) to retrieve the flag that represents the node’s scope, lifetime, and timeout criteria. This flag is specified when the state node is created. The following table describes the valid session flags:

| Flag            | Description   |
|-----------------|---|
| GXSTATE_LOCAL   | The node is visible to the local process only.                        |
| GXSTATE_CLUSTER | The node is visible to all application components within the cluster. |

| Flag                   | Description   |
|------------------------|---|
| GXSTATE_DISTRIB        | The node is visible to all application components in the enterprise environment.      |
| GXSTATE_TIMEOUT_CREATE | The contents of the node expires <i>n</i> seconds from the time the node was created. |

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Using the State Layer” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

## GetStateName()

Returns the name of the node.

**Syntax** HRESULT GetStateName(  
LPSTR pName,  
ULONG nName);

**pName** Pointer to the buffer allocated by the client to store the returned node name.

**nName** The size of the buffer to store the node name.

**Usage** Use GetStateName() when the name of the node is unknown and is required for subsequent operations.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Using the State Layer” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

## GetStateTimeout()

Returns the node’s timeout value in seconds.

**Syntax** HRESULT GetStateTimeout(  
ULONG \*pTimeout);

**pTimeout** Pointer to the buffer allocated by the client to store the returned timeout value.

**Usage** Use `GetStateTimeout()` in conjunction with `GetStateFlags()` to determine if and when the contents of the node expires. A timeout value of 0 means the node contents are saved until the node is deleted explicitly.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** “Using the State Layer” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

---

## SaveState()

Saves updates to the node contents.

**Syntax**

```
HRESULT SaveState(  
    DWORD dwFlags);
```

**dwFlags** Specify 0 (zero). Internal use only.

**Usage** Use `SaveState()` after you set or change the contents of a node. This method flushes the node contents into the distributed store.

**Tip** The `GetStateContents()` method retrieves the contents that were last saved in the distributed store with `SaveState()`. Therefore, if you update the contents of a node with `SetStateContents()`, but do *not* call `SaveState()`, `GetStateContents()` will not return the content set with `SetStateContents()`.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** “Using the State Layer” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

---

## SetStateContents()

Sets node contents.

**Syntax**

```
HRESULT SetStateContents(  
    IGXValList *pContents);
```

**pContents** Pointer to the `IGXValList` of values to set in the current node.

**Usage** Use `SetStateContents()` to update the contents of a node.

- Tips**
- To traverse the state tree to find the child node to update, use `GetStateChild()`. Each successive call to `GetStateChild()` descends one level in the tree.
  - Call `SaveState()` after you set or change the contents of a node. This method flushes the node contents into the distributed store. If you call `SetStateContents()` several times before calling `SaveState()`, only the value from the last `SetStateContents()` call is saved.
  - The `GetStateContents()` method retrieves the contents that were last saved in the distributed store with `SaveState()`. Therefore, if you update the contents of a node with `SetStateContents()`, but do *not* call `SaveState()`, `GetStateContents()` will not return the content set with `SetStateContents()`.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Example** The following code shows how to create a child node and set its contents:

```
IState2 tree = getStateTreeRoot(GXSTATE.GXSTATE_DISTRIB, "Grammy");

if (tree!=null)
{
    IState2 child = tree.getStateChild("Best Female Vocal");
    if (child == null)
    {
        child = tree.createStateChild("Best Female Vocal", 0,
            GXSTATE.GXSTATE_DISTRIB);
    }
    if (child != null)
    {
        IValList val = GX.CreateValList();
        val.setValString("winner", "Whitney Houston");
        val.setValString("runner up", "Barbara Streisand");
        child.setStateContents(val);
        child.saveState(0);
    }
}

HRESULT hr;

hr = GetStateTreeRoot(GXSTATE_DISTRIB, "Grammy", &m_pStateRoot);

if (hr == NOERROR && m_pStateRoot)
{
    IGXState2 *pState = NOERROR;
```

```

hr = m_pStateRoot->GetStateChild("Best Female Vocal",
    &pState);
if (hr != NOERROR || !pState)
{
    hr = m_pStateRoot->CreateStateChild("Best Female Vocal",
        0, GXSTATE_DISTRIB, &pState);

if (hr == NOERROR && pState)
{
    pState->GetStateContents(&pVL);
    if (!pVL)
    {
        IGXValList *pVL = GXCreateValList();
        pVL->SetValString("winner", "Whitney Houston");
        pVL->SetValString("runnerup", "Barbara
            Streisand");

        hr = pState->SetStateContents(pVL);
        hr = pState->SaveState(0);

        pVL->Release();
    }
}

```

**Related Topics** “Using the State Layer” in Chapter 8, “Managing Session and State Information,” in *Programmer’s Guide*.

# IGXStreamBuffer interface

The IGXStreamBuffer interface represents a buffer for capturing streamed output during template processing. Use a stream buffer if your AppLogic needs to manipulate the data before sending it to another AppLogic. For example, the AppLogic can collect the data in a stream buffer, then parse it or save it to a file.

To capture the data in a stream buffer, use the EvalOutput( ) method in the GXAppLogic class and pass in an IGXStream object. To manipulate the data in the stream buffer, use the GetStreamData( ) method in this interface.

To create an instance of the IGXStreamBuffer interface, use the GXCreateStreamBuffer( ) helper function.

## Include File

gxstream.h

## Method

|                  |   |
|------------------|---|
| GetStreamData( ) | Returns an array of byte values from the stream buffer. |
|------------------|---|

---

## GetStreamData( )

Returns an array of byte values from the stream buffer.

**Syntax** HRESULT GetStreamData(  
     DWORD flags,  
     LPBYTE pBuff,  
     ULONG nBuff);

**flags** Specify 0 (zero).

**pBuff** Pointer to the client-allocated buffer to store the data.

**nBuff** Length of the client-allocated buffer.

**Usage** Use GetStreamData( ) to retrieve the contents of the stream buffer that was captured during streamed template processing. The AppLogic can then manipulate the data as needed.

**Rule** Call `GetStreamData()` after `EvalOutput()` in the `GXAppLogic` class. The `EvalOutput()` method captures output in the stream buffer if the `AppLogic` passes in an `IGXStream` object.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `EvalOutput()` in the `GXAppLogic` class

# IGXTable interface

The IGXTable interface represents the definition of a table that is part of a relational data source. IGXTable provides methods to perform the following types of operations:

- Add, update, and delete rows in the table.
- Obtain information about table attributes as they are defined in the database catalog. Table attributes include the table name, table columns, data connection, and so on. To obtain additional information about individual columns, use the methods in the IGXColumn interface.

The IGXTable interface is part of the Data Access Engine (DAE) service.

To create an instance of the IGXTable interface, use `GetTable()` in the `IGXDataConn` interface or `GetTable()` in the `IGXColumn` interface.

Each call to `GetTable()` returns a new IGXTable object rather than returning an existing table object.

## Include File

gxidata.h

## Methods

| Method                         | Description   |
|--------------------------------|---|
| <code>AddRow()</code>          | Inserts a new row in the table.   |
| <code>AllocRow()</code>        | Allocates a new, empty row buffer, replacing the previous row buffer if one exists. |
| <code>DeleteRow()</code>       | Deletes one or more rows in the table.  |
| <code>EnumColumnReset()</code> | Resets the column enumeration to the first column in the table.                     |
| <code>EnumColumns()</code>     | Returns the definition of the next column in the table.                             |
| <code>GetColumn()</code>       | Returns the definition of a column with the specified name.                         |
| <code>GetColumnByOrd()</code>  | Returns the definition of the column in the specified ordinal position.             |

| Method                 | Description   |
|------------------------|---|
| GetColumnOrdinal( )    | Returns the ordinal position of the column specified by name.                                     |
| GetDataConn( )         | Returns the data connection object associated with the data source in which the table is defined. |
| GetName( )             | Returns the name of the table.  |
| GetNumColumns( )       | Returns the number of columns in the table object.  |
| SetValueBinary( )      | Specifies a BINARY value of a column in the row buffer.   |
| SetValueBinaryPiece( ) | Specifies a LONG BINARY value of a column in the row buffer.                                      |
| SetValueDateString( )  | Specifies the Date value of a column in the row buffer.   |
| SetValueDouble( )      | Specifies the double value of a column in the row buffer.   |
| SetValueInt( )         | Specifies the int value of a column in the row buffer.  |
| SetValueString( )      | Specifies the String value of a column in the row buffer.   |
| SetValueText( )        | Specifies a TEXT value of a column in the row buffer.   |
| SetValueTextPiece( )   | Specifies a LONGTEXT value of a column in the row buffer.   |
| UpdateRow( )           | Modifies one or more rows in the table with the contents of the row buffer.                       |

## Related Topics

“Inserting Records in a Database,” “Updating Records in a Database,” and “Using Pass-Through Database Commands” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

## AddRow( )

Inserts a new row in the table.

**Syntax** HRESULT AddRow(  
     DWORD dwFlags,  
     IGXTrans \*pTrans);

**dwFlags** Specifies one of the following flags used to execute this insert operation:

- For synchronous operations, the default, specify zero or `GX_DA_EXEC_SYNC`.
- For asynchronous operations, specify `GX_DA_EXEC_ASYNC`.

**pTrans** Pointer to the `IGXTrans` object that contains the transaction associated with this insert operation, or `NULL`.

**Usage** Use `AddRow()` to insert a new record into a table.

- Rules**
- Before adding a row, the `AppLogic` must first call `AllocRow()` to create a row buffer.
  - Next, the `AppLogic` must specify data values for the new row by calling any of the `SetValueXXX()` methods, such as `SetValueString()` or `SetValueBinary()`.
  - The `AppLogic` must specify a value for any column defined as `NOT NULL` and without a `DEFAULT` value, such as keys.
  - The `AppLogic` must be logged into the database with sufficient privileges to insert records in the target table.
  - If the insert operation is part of a transaction, the `AppLogic` must first create an instance of the `IGXTrans` interface using `CreateTrans()` in the `GXAppLogic` class. The `AppLogic` must then call `Begin()` before executing the statement and, after executing the statement, call `Commit()` or `Rollback()` as appropriate.
- Tips**
- To determine whether a column is defined as `NOT NULL`, use `GetNullsAllowed()` in the `IGXColumn` interface.
  - Alternatively, the `AppLogic` can insert records by passing a `SQL INSERT` statement using `SetSQL()` in the `IGXQuery` interface. The statement must comply with `ANSI 92 SQL` syntax.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```
// Get a table
IGXTable *pTable=NULL;
if(((hr=pConn->GetTable("OBTransaction",
&pTable))==GXE_SUCCESS)&&pTable) {

    // Look up the column ordinals for the table
    ULONG transTypeCol=0;
```

```

pTable->GetColumnOrdinal("transType", &transTypeCol);
ULONG postDateCol=0;
pTable->GetColumnOrdinal("postDate", &postDateCol);
ULONG acctNumCol=0;
pTable->GetColumnOrdinal("acctNum", &acctNumCol);
ULONG amountCol=0;
pTable->GetColumnOrdinal("amount", &amountCol);

// Create a transaction
IGXTrans *pTx=NULL;
if(((hr=CreateTrans(&pTx))==GXE_SUCCESS)&&pTx) {
    pTx->Begin();

    // Allocate a new row
    pTable->AllocRow();
    pTable->SetValueString(acctNumCol, pFromAcct);
    pTable->SetValueInt(transTypeCol,TRANSTYPE_WITHDRAWAL);
    pTable->SetValueDateString(postDateCol, dateStr);
    pTable->SetValueDouble(amountCol, amount*-1.0);

    // Add the row to the table
    if(pTable->AddRow(0, pTx)==GXE_SUCCESS) {
        ...
    }
}

```

**Related Topics** IGXTrans interface

“Inserting Records in a Database,” “Updating Records in a Database,” and “Using Pass-Through Database Commands” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

## AllocRow()

---

Allocates a new, empty row buffer, replacing the previous row buffer if one exists.

**Syntax** HRESULT AllocRow();

**Usage** Use AllocRow() to allocate a new row buffer before adding or updating records in a table. The row buffer is a virtual representation of a row in the target table, including all column definitions. The AppLogic writes data values

to the row buffer first, then writes the contents of the row buffer to either a new record using `AddRow()` or to one or more existing records using `UpdateRow()`.

- Rules**
- The AppLogic must call `AllocRow()` before specifying column values with a `SetValueXXX()` method.
  - The AppLogic must call `AllocRow()` every time before calling `AddRow()` or `UpdateRow()`.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```
// Get a table
IGXTable *pTable=NULL;
if((hr=pConn->GetTable("OBTransaction",
&pTable))==GXE_SUCCESS)&&pTable) {

    // Look up the column ordinals for the table
    ULONG transTypeCol=0;
    pTable->GetColumnOrdinal("transType", &transTypeCol);
    ULONG postDateCol=0;
    pTable->GetColumnOrdinal("postDate", &postDateCol);
    ULONG acctNumCol=0;
    pTable->GetColumnOrdinal("acctNum", &acctNumCol);
    ULONG amountCol=0;
    pTable->GetColumnOrdinal("amount", &amountCol);

    // Create a transaction
    IGXTrans *pTx=NULL;
    if((hr>CreateTrans(&pTx))==GXE_SUCCESS)&&pTx) {
        pTx->Begin();

        // Allocate a new row
        pTable->AllocRow();
        pTable->SetValueString(acctNumCol, pFromAcct);
        pTable->SetValueInt(transTypeCol,TRANSTYPE_WITHDRAWAL);
        pTable->SetValueDateString(postDateCol, dateStr);
        pTable->SetValueDouble(amountCol, amount*-1.0);

        // Add the row to the table
        if(pTable->AddRow(0, pTx)==GXE_SUCCESS) {
            ...
        }
    }
}
```

**Related Topics** “Inserting Records in a Database,” “Updating Records in a Database,” and “Using Pass-Through Database Commands” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

## DeleteRow()

---

Deletes one or more rows in the table.

**Syntax** HRESULT DeleteRow(  
     DWORD dwFlags,  
     LPSTR szWhere,  
     IGXTrans \*pTrans);

**dwFlags** Specifies one of the following flags used to execute this delete operation:

- For synchronous operations, the default, specify zero or GX\_DA\_EXEC\_SYNC.
- For asynchronous operations, specify GX\_DA\_EXEC\_ASYNC.

**szWhere** Selection criteria expression for one or more rows to delete. The syntax is the same as the SQL WHERE clause, only without the WHERE keyword. Use ANSI 92-compliant syntax. If an empty string is specified, all rows in the table are deleted.

**pTrans** Pointer to the IGXTrans object that contains the transaction associated with this delete operation, or NULL.

- Rules**
- The AppLogic must be logged into the database with sufficient privileges to delete records in the target table.
  - If the delete operation is part of a transaction, the AppLogic must first create an instance of the IGXTrans interface using CreateTrans() in the GXAppLogic class. The AppLogic must then call Begin() before executing the statement and, after executing the statement, call Commit() or Rollback() as appropriate.

**Tip** Alternatively, the AppLogic can delete records by passing a SQL DELETE statement using SetSQL() in the IGXQuery interface, then executing the query. The statement must comply with ANSI 92 SQL syntax.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

```

Example IGXValList *conn_params;

conn_params = GXCreateValList();
conn_params->SetValString("DSN", "salesDB");
conn_params->SetValString("DB", "salesDB");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");

IGXDataConn *conn = NULL;
HRESULT hr;

hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL, &conn);
if (hr == NOERROR &&
    conn)
{
    IGXTable *table = NULL;
    hr = conn->GetTable("employees", &table);
    if (hr == NOERROR &&
        table)
    {
        table->DeleteRow(0, "lastname='Smith'", NULL);

        table->Release();
    }
    conn->Release();
}
conn_params->Release();

```

**Related Topics** IGXTrans interface

“Deleting Records From a Database” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

“Using Pass-Through Database Commands” in Chapter 5, “Working with Databases” in *Programmer’s Guide*.

---

## EnumColumnReset()

Resets the column enumeration to the first column in the table.

**Syntax** HRESULT EnumColumnReset();

**Usage** Use EnumColumnReset( ) before iterating through and retrieving columns in a table. EnumColumnReset( ) ensures that column retrieval starts from the first column.

Thereafter, use EnumColumns( ) to retrieve each column sequentially. Each EnumColumns( ) call returns an IGXColumn object for the next column.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Getting Information About Columns or Fields” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

“Inserting Records in a Database,” “Updating Records in a Database,” and “Using Pass-Through Database Commands” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

---

## EnumColumns()

Returns the definition of the next column in the table.

**Syntax** HRESULT EnumColumns(  
IGXColumn \*\*ppColumn);

**ppColumn** Pointer to the IGXColumn object that contains the returned next column of data. When the AppLogic is finished using the object, call the Release( ) method to release the interface instance.

**Usage** Use EnumColumns( ) when the column definition is unknown and required for subsequent operations. The AppLogic can use the returned IGXColumn object to determine characteristics of the column, such as its name, data type, size, whether nulls are allowed, and so on.

Before iterating through columns, the client code should call EnumColumnReset( ) to ensure that EnumColumns( ) starts with the first column in the table. Each subsequent EnumColumns( ) call moves to the next sequential column in the table and retrieves its column definition in an IGXColumn object.

- Tips**
- The columns might not be returned in the order in which they are defined in the database catalog.
  - Test for NULL to determine when the last column has been retrieved.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** IGXColumn interface

“Getting Information About Columns or Fields” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

---

## GetColumn()

Returns the definition of a column with the specified name.

**Syntax** HRESULT GetColumn(  
     LPSTR szColumn,  
     IGXColumn \*\*ppColumn);

**szColumn** Name of the column to retrieve.

**ppColumn** Pointer to the IGXColumn object that contains the returned column definition. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use GetColumn() when the column definition is unknown but its name is known. The AppLogic can use the IGXColumn object to determine other characteristics about the column, such as its data type, size, whether nulls are allowed, and so on.

**Rule** The specified column name must exist in the table.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** IGXColumn interface

“Getting Information About Columns or Fields” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

---

## GetColumnByOrd()

Returns the definition of the column in the specified ordinal position.

**Syntax** HRESULT GetColumnByOrd(  
     ULONG Ordinal,  
     IGXColumn \*\*ppColumn);

**Ordinal** Ordinal number (position) of the column in the table. The first column is 1, the second column is 2, and so on.

**ppColumn** Pointer to the IGXColumn object that contains the returned column definition. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use GetColumnByOrd() when the column definition is unknown but its position in the table is known, such as when iterating through columns in the table. The AppLogic can use the IGXColumn object to determine other characteristics about the column, such as its name, data type, size, whether nulls are allowed, and so on.

**Rule** The specified column number must exist in the table.

- Tips**
- Column positions in a table may change between different table objects.
  - Columns are not guaranteed to be in the same order in which the database lists them.
  - To iterate through columns in a table using GetColumnByOrd(), call GetNumColumns() to determine the maximum number of columns in the table, then proceed sequentially through each column using GetColumnByOrd(), beginning with column 1, through the last column.
  - Alternatively, call EnumColumnReset() to start with the first column in the table, then call EnumColumns() repeatedly through the last column.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** IGXColumn interface

“Getting Information About Columns or Fields” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

---

## GetColumnOrdinal()

Returns the ordinal position of the column specified by name.

**Syntax**

```
HRESULT GetColumnOrdinal(
    LPSTR szColumn,
    ULONG *pOrdinal);
```

**szColumn** Name of the column.

**pOrdinal** Pointer to the buffer allocated by the client to contain the returned ordinal position of the specified column. The first column is 1, the second column is 2, and so on.

**Usage** Use `GetColumnOrdinal()` when the ordinal position of a column is unknown and is required for subsequent operations. For example, the ordinal position of a column is a required parameter value for the `SetValue**()` methods, such as `SetValueString()` and `SetValueInt()`.

**Rule** The specified column name must exist in the table.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```
// Create a data connection
IGXDataConn *pConn=NULL;

if(((hr=GetOBDataConn(&pConn))==GXE_SUCCESS)&&pConn) {
    IGXTable *pTable=NULL;

    // Get the table
    if(((hr=pConn->GetTable("OBTransaction",
        &pTable))==GXE_SUCCESS)&&pTable) {

        // Look up the column ordinals for the table
        ULONG transTypeCol=0;
        pTable->GetColumnOrdinal("transType", &transTypeCol);
        ULONG postDateCol=0;
        pTable->GetColumnOrdinal("postDate", &postDateCol);
        ULONG acctNumCol=0;
        pTable->GetColumnOrdinal("acctNum", &acctNumCol);
        ULONG amountCol=0;
        pTable->GetColumnOrdinal("amount", &amountCol);
    }
}
```

**Related Topics** [IGXColumn interface](#)

“Getting Information About Columns or Fields” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

## GetDataConn()

---

Returns the data connection object associated with the data source in which the table is defined.

**Syntax** HRESULT GetDataConn(  
           IGXDataConn \*\*ppDataConn);

**ppDataConn** Pointer to the IGXDataConn object that contains the returned data connection object associated with the data source in which the table is defined. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** Use GetDataConn() when the data connection associated with the table is unknown and is required for subsequent operations.

**Tip** The IGXDataConn object that GetDataConn() returns may not be equal (==) to the IGXDataConn object that CreateDataConn(), in the GXAppLogic class, returned.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “About Database Connections” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

---

## GetName()

Returns the name of the table.

**Syntax** HRESULT GetName(  
           LPSTR pBuffer,  
           ULONG nBuff);

**pBuff** Pointer to the buffer allocated by the client to contain the returned table name.

**nBuff** The length of the pBuffer buffer, in bytes.

**Usage** Use GetName() when the name of the table is unknown and is required for subsequent operations.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

---

## GetNumColumns()

Returns the number of columns in the table object.

**Syntax** HRESULT GetNumColumns(  
    ULONG \*pnCols);

**pnCols** Pointer to the returned number of columns in the table.

**Usage** Use GetNumColumns() when the number of columns defined in the table is unknown and is required for subsequent operations. When iterating through columns in a table, the AppLogic can use this information to specify the maximum number of iterations.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

---

## SetValueBinary()

Specifies a BINARY value of a column in the row buffer.

**Syntax** HRESULT SetValueBinary(  
    ULONG Ordinal,  
    LPBYTE pValue,  
    ULONG nOffset,  
    ULONG nLength);

**Ordinal** Ordinal number (position) of the column in the table definition. The first column is 1, the second column is 2, and so on.

**pValue** A byte array expression to assign to the column.

**nOffset** Number of bytes to skip from the beginning of the byte array. This value specifies the starting point within the array.

**nLength** Number of bytes to set for the byte array.

**Usage** Use SetValueBinary() for BINARY data of which the total size is equal to or smaller than 64K.

- Rules**
- The AppLogic must call AllocRow() before attempting to write to the row buffer.
  - The data type of the column must be BINARY or VARBINARY, or database equivalent.

**Tip** Use SetValueBinaryPiece() for LONGBINARY, LONGVARBINARY, or equivalent type values.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Inserting Records in a Database,” “Updating Records in a Database,” and “Using Pass-Through Database Commands” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

## SetValueBinaryPiece()

---

Specifies a LONGBINARY value of a column in the row buffer.

**Syntax** HRESULT SetValueBinary(  
     ULONG Ordinal,  
     LPBYTE pValue,  
     ULONG nOffset,  
     ULONG nLength);

**Ordinal** Ordinal number (position) of the column in the table definition. The first column is 1, the second column is 2, and so on.

**pValue** A byte array expression to assign to the column.

**nOffset** Number of bytes to skip from the beginning of the byte array. This value specifies the starting point within the array.

**nLength** Number of bytes to set for the byte array.

**Usage** Use SetValueBinaryPiece() to specify LONGBINARY data. LONGBINARY data must be added in 64K increments, therefore, you must use SetValueBinaryPiece() several times to add the data.

- Rules**
- The AppLogic must call AllocRow() before attempting to write to the row buffer.
  - The data type of the column must be LONGBINARY, LONGVARBINARY, or database equivalent.
  - Must be called after AllocRow() but before AddRow() or UpdateRow().

**Tip** Use SetValueBinary() for BINARY, VARBINARY, or equivalent type values.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Inserting Records in a Database,” “Updating Records in a Database” and “Using Pass-Through Database Commands” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

## SetValueDateString()

Specifies the Date value of a column in the row buffer.

**Syntax** HRESULT SetValueDateString(  
     ULONG Ordinal,  
     LPSTR pValue);

**Ordinal** Ordinal number (position) of the target column in the table. The first column is 1, the second column is 2, and so on.

**pValue** A date expression to assign to the column. Use one of the following formats:

- "Fri Oct 10 14:35:59.999 PDT 1997"

The subseconds (.999 in the example) and time zone (PDT in the example) are optional.

- "1997-10-01 14:35:59.999"

The time is optional.

**Rule** The AppLogic must call AllocRow() before attempting to write to the row buffer.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
// Get a table
IGXTable *pTable=NULL;
if(((hr=pConn->GetTable("OBTransaction",
&pTable))==GXE_SUCCESS)&&pTable) {

    // Look up a column ordinal
    ULONG postDateCol=0;
    pTable->GetColumnOrdinal("postDate", &postDateCol);

    // Allocate a new row and set a datestring value
    pTable->AllocRow();
    pTable->SetValueDateString(postDateCol, dateStr);
```

**Related Topics** "Inserting Records in a Database," "Updating Records in a Database" and "Using Pass-Through Database Commands" in Chapter 5, "Working with Databases," in *Programmer's Guide*.

## SetValueDouble()

---

Specifies the double value of a column in the row buffer.

**Syntax** HRESULT SetValueDouble(  
     ULONG Ordinal,  
     double nValue);

**Ordinal** Ordinal number (position) of the target column in the table. The first column is 1, the second column is 2, and so on.

**nValue** A double expression to assign to the column.

**Rule** The AppLogic must call AllocRow() before attempting to write to the row buffer.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
// Get a table
IGXTable *pTable=NULL;
if((hr=pConn->GetTable("OBTransaction",
&pTable))==GXE_SUCCESS)&&pTable) {

    // Look up a column ordinal
    ULONG amountCol=0;
    pTable->GetColumnOrdinal("amount", &amountCol);

    // Allocate a new row and set a double value
    pTable->AllocRow();
    pTable->SetValueDouble(amountCol, amount*-1.0);
```

**Related Topics** “Inserting Records in a Database,” “Updating Records in a Database” and “Using Pass-Through Database Commands” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

## SetValueInt()

---

Specifies the int value of a column in the row buffer.

**Syntax** HRESULT SetValueInt(  
     ULONG Ordinal,  
     DWORD nValue);

**Ordinal** Ordinal number (position) of the target column in the table. The first column is 1, the second column is 2, and so on.

**nValue** An int expression to assign to the column.

**Rule** The AppLogic must call AllocRow() before attempting to write to the row buffer.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
// Get a table
IGXTable *pTable=NULL;
if(((hr=pConn->GetTable("OBTransaction",
&pTable))==GXE_SUCCESS)&&pTable) {

    // Look up a column ordinal
    ULONG transTypeCol=0;
    pTable->GetColumnOrdinal("transType", &transTypeCol);

    // Allocate a new row and set an int value
    pTable->AllocRow();
    pTable->SetValueInt(transTypeCol, TRANSTYPE_WITHDRAWAL);
```

**Related Topics** “Inserting Records in a Database,” “Updating Records in a Database” and “Using Pass-Through Database Commands” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

---

## SetValueString()

**Syntax**

```
HRESULT SetValueString(
    ULONG Ordinal,
    LPSTR pValue);
```

**Ordinal** Ordinal number (position) of the target column in the table. The first column is 1, the second column is 2, and so on.

**pValue** Pointer to the variable that contains the string expression to assign to the column.

**Rule** The AppLogic must call AllocRow() before attempting to write to the row buffer.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

```
// Get a table
```

```

IGXTable *pTable=NULL;
if((hr=pConn->GetTable("OBTransaction",
&pTable))==GXE_SUCCESS)&&pTable) {

    // Look up a column ordinal
    ULONG acctNumCol=0;
    pTable->GetColumnOrdinal("acctNum", &acctNumCol);

    // Allocate a new row and set a string value
    pTable->AllocRow();
    pTable->SetValueString(acctNumCol, pFromAcct);

```

**Related Topics** “Inserting Records in a Database,” “Updating Records in a Database” and “Using Pass-Through Database Commands” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

---

## SetValueText()

Specifies a TEXT value of a column in the row buffer.

**Syntax** HRESULT SetValueText(  
     ULONG Ordinal,  
     LPSTR pValue,  
     ULONG nOffset,  
     ULONG nLength);

**Ordinal** Ordinal number (position) of the column in the table definition. The first column is 1, the second column is 2, and so on.

**pValue** A string expression to assign to the column.

**nOffset** Number of characters to skip from the beginning of the string.

**nLength** Number of characters to set.

**Usage** Use SetValueText() for TEXT data, or database equivalent, of which the total size is equal to or smaller than 64K.

- Rules**
- The AppLogic must call AllocRow() before attempting to write to the row buffer.
  - The data type of the column must be TEXT or database equivalent.

**Tip** Use SetValueTextPiece() for LONGTEXT or equivalent type values.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Inserting Records in a Database,” “Updating Records in a Database” and “Using Pass-Through Database Commands” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

---

## SetValueTextPiece()

Specifies a LONG TEXT value of a column in the row buffer.

**Syntax**

```
HRESULT SetValueText(
    ULONG Ordinal,
    LPSTR pValue,
    ULONG nOffset,
    ULONG nLength);
```

**Ordinal** Ordinal number (position) of the column in the table definition. The first column is 1, the second column is 2, and so on.

**pValue** A string expression to assign to the column.

**nOffset** Number of characters to skip from the beginning of the string.

**nLength** Number of characters to set.

**Usage** Use SetValueTextPiece() for LONGTEXT data. LONGTEXT values must be added in 64K increments, therefore, you must call SetValueTextPiece() repeatedly to add the data.

- Rules**
- The AppLogic must call AllocRow() before attempting to write to the row buffer.
  - The data type of the column must be LONGTEXT or database equivalent.

**Tip** Use SetValueText() for TEXT or equivalent type values.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** “Inserting Records in a Database,” “Updating Records in a Database” and “Using Pass-Through Database Commands” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

## UpdateRow()

---

Modifies one or more rows in the table with the contents of the row buffer.

**Syntax** HRESULT UpdateRow(  
    DWORD dwFlags,  
    LPSTR szWhere,  
    IGXTrans \*pTrans);

**dwFlags** Specifies one of the following flags used to execute this update operation:

- For synchronous operations, the default, specify zero or GX\_DA\_EXEC\_SYNC.
- For asynchronous operations, specify GX\_DA\_EXEC\_ASYNC.

**szWhere** Selection criteria expression for one or more rows to update. The syntax is the same as the SQL WHERE clause, only without the WHERE keyword. Use ANSI 92-compliant syntax. If an empty string is specified, all rows in the table are updated.

**pTrans** Pointer to the IGXTrans object that contains the transaction associated with this update operation, or NULL.

- Rules**
- Before modifying a row, the AppLogic must first call AllocRow() to create the row buffer.
  - Next, the AppLogic must specify data values for the new row by calling any of the following methods: SetValueDateString(), SetValueDouble(), SetValueInt(), SetValueString().
  - For tables defined with one or more UNIQUE keys, the AppLogic can perform a single-record update but not a multiple-record update.
  - The AppLogic must specify a value for any column defined as NOT NULL and without a DEFAULT value, such as keys.
  - The AppLogic must be logged into the database with sufficient privileges to update records in the target table.

- If the update operation is part of a transaction, the AppLogic must first create an instance of the IGXTrans interface using CreateTrans() in the GXAppLogic class. The AppLogic must then call Begin() before executing the statement and, after executing the statement, call Commit() or Rollback() as appropriate.

- Tips**
- The UpdateRow() method overwrites all columns in the target record(s) with the contents of the row buffer. Therefore, retrieve the row first using a query, assign the column values to the row buffer, then change only the column(s) you want to update.
  - To determine whether a column is defined as NOT NULL, use GetNullsAllowed() in the IGXColumn interface.
  - Alternatively, the AppLogic can update records by passing a SQL INSERT statement using SetSQL() in the IGXQuery interface. The statement must comply with ANSI 92 SQL syntax.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```

IGXValList *conn_params;
// Set connection parameters
conn_params = GXCreateValList();
conn_params->SetValString("DSN", "salesDB");
conn_params->SetValString("DB", "salesDB");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");

IGXDataConn *conn = NULL;

HRESULT hr;

// Create a data connection
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL, &conn);
if (hr == NOERROR &&
    conn)
{
    IGXTable *table = NULL;
    hr = conn->GetTable("employees", &table);
    if (hr == NOERROR &&
        table)
    {
        hr = table->AllocRow();
    }
}

```

```
if (hr == NOERROR)
{
    ULONG col;
    table->GetColumnOrdinal("region", &col);
    table->SetValueString(col, "East");

    table->UpdateRow(0, "region='West'", NULL);
}
table->Release();
}
conn->Release();
}
conn_params->Release();
```

### Related Topics IGXTrans interface

“Updating Records in a Database” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*

“Using Pass-Through Database Commands” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

# IGXTemplateData interface

The IGXTemplateData interface represents a hierarchical source of data used for HTML template processing. IGXTemplateData provides methods for iterating through rows in a set of memory-based hierarchical data and retrieving column values.

To create an IGXTemplateData object, an AppLogic calls the GXCreateTemplateDataBasic( ) helper function. The AppLogic populates the IGXTemplateData object with rows of hierarchical data, then passes this GXTemplateDataBasic object as the data parameter in EvalTemplate( ) or EvalOutput( ) in the GXAppLogic class. The Template Engine then draws upon the hierarchical data during template processing using methods in the IGXTemplateData interface.

The Template Engine normally processes the hierarchical template data internally. To provide application-specific special processing and hook into the template generation process, the AppLogic can subclass the GXTemplateDataBasic class and override the IGXTemplateData member methods.

## Include File

gxitmpl.h

## Methods

| Method      | Description  |
|-------------|--|
| GetValue( ) | The Template Engine calls this method to dynamically retrieve the value of the specified field from the current row in the hierarchical template data. |
| IsEmpty( )  | The Template Engine calls this method to determine whether the specified group in the hierarchical result set is empty (contains no rows).             |
| MoveNext( ) | The Template Engine calls this method to retrieve the next row of the specified group in the hierarchical template data object.                        |
| SetHint( )  | Placeholder method for future functionality.   |

## Related Topics

EvalTemplate() and EvalOutput() in the GXAppLogic class

GXTemplateDataBasic class

“Constructing a Hierarchical Result Set with GXTemplateDataBasic” in Chapter 7, “Working with Templates,” in *Programmer’s Guide*.

## GetValue()

---

The Template Engine calls this method to dynamically retrieve the value of the specified field from the current row in the hierarchical template data.

**Syntax** HRESULT GetValue(  
     LPSTR szExpr,  
     IGXBuffer \*\*ppBuff);

**szExpr** Name of a field in the template data object.

**ppBuff** Pointer to the IGXBuffer object that will contain the returned value of the specified field in the current row. After the function is done, the returned buffer should hold a zero-terminated string. This method allocates the IGXBuffer object automatically. When the AppLogic is finished using the object, call the Release() method to release the interface instance.

**Usage** The Template Engine calls GetValue() to retrieve values from the hierarchical template data object for subsequent processing.

**Rule** The specified field name must exist in the template data object.

- Tips**
- When processing result sets, first call IsEmpty() to determine whether rows were returned. Next, for each row in the result set, call GetValue() to retrieve field values, then call MoveNext() to move to the next row in the result set, until the end of the result set is reached.
  - Use methods in the IGXBuffer interface to manipulate the returned memory block.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** EvalTemplate() and EvalOutput() in the GXAppLogic class

GXTemplateDataBasic class

“Constructing a Hierarchical Result Set with GXTemplateDataBasic” in Chapter 7, “Working with Templates,” in *Programmer’s Guide*.

## IsEmpty()

---

The Template Engine calls this method to determine whether the specified group in the hierarchical result set is empty (contains no rows).

**Syntax** HRESULT IsEmpty(  
           LPSTR group,  
           BOOL \*empty);

**group** Name of a group in the hierarchical result set.

**Usage** The Template Engine calls IsEmpty() to test whether the specified group in the IGXTemplateData object contains any rows of data before processing individual fields using GetValue().

**Rule** The specified group name must exist in the hierarchical data set.

**Tip** When processing result sets, first call IsEmpty() to determine whether rows were returned. Next, for each row in the result set, call GetValue() to retrieve field values, then call MoveNext() to move to the next row in the result set, until the end of the result set is reached.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** EvalTemplate() and EvalOutput() in the GXAppLogic class  
 GXTemplateDataBasic class

“Constructing a Hierarchical Result Set with GXTemplateDataBasic” in Chapter 7, “Working with Templates,” in *Programmer’s Guide*.

## MoveNext()

---

The Template Engine calls this method to retrieve the next row of the specified group in the hierarchical template data object.

**Syntax** HRESULT MoveNext(  
           LPSTR group);

**group** Name of a group to process in the hierarchical data of the template data object.

**Usage** The Template Engine calls `MoveNext()` when iterating through rows in the template data object to retrieve the contents of the next sequential hierarchical row of data.

**Rule** The specified group name must exist in the hierarchical data set.

**Tip** When processing result sets, first call `IsEmpty()` to determine whether rows were returned. Next, for each row in the result set, call `GetValue()` to retrieve field values, then call `MoveNext()` to move to the next row in the result set, until the end of the result set is reached.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `EvalTemplate()` and `EvalOutput()` in the `GXAppLogic` class

`GXTemplateDataBasic` class

“Constructing a Hierarchical Result Set with `GXTemplateDataBasic`” in Chapter 7, “Working with Templates,” in *Programmer’s Guide*.

---

## SetHint()

The `SetHint()` method is a placeholder for future functionality. Currently, it is implemented to return 0. If you create a custom template data class that implements the `IGXTemplateData` interface, implement `SetHint()` to return 0.

**Syntax**

```
HRESULT SetHint(  
    LPSTR group,  
    DWORD flags,  
    ULONG max,  
    IGXValList *pVal);
```

# IGXTemplateMap interface

The IGXTemplateMap interface represents a mapping between a template field specification and dynamic data used for HTML template processing. IGXTemplateMap provides the `Get()` method for resolving the `id` attribute in a GX markup tag. Each `id` attribute contains a field name that can be mapped.

To create a field map, an AppLogic calls the `GXCreateTemplateMapBasic()` helper function. The AppLogic then populates the field map using `Put()`, in the `GXTemplateMapBasic` class, for each field mapping, then passes this IGXTemplateMap object as the map parameter in `EvalTemplate()` or `EvalOutput()` in the `GXAppLogic` class. When the Template Engine encounters a GX markup tag with the `id` attribute while processing the template, it calls `Get()` in the IGXTemplateMap interface to resolve the name.

To provide application-specific special processing, an AppLogic can subclass the `GXTemplateMapBasic` class and override the `Get()` method to hook into the Template Engine generation process. For example, the AppLogic can intercept and filter data from a database before the Template Engine processes it.

## Include File

`gxitmpl.h`

## Method

|                    |   |
|--------------------|---|
| <code>Get()</code> | Resolves the <code>id</code> attribute specified in a GX markup tag in the template being processed by the Template Engine. This method is called by the Template Engine. |
|--------------------|---|

## Related Topics

`EvalTemplate()` and `EvalOutput()` in the `GXAppLogic` class

`GXTemplateDataBasic` class

IGXTemplateData interface

“GX Markup Tag Syntax” in Chapter 7, “Working with Templates,” in *Programmer’s Guide*.

## Get()

---

Resolves the `id` attribute specified in a GX markup tag in the template being processed by the Template Engine. This method is called by the Template Engine.

**Syntax**

```
HRESULT Get(  
    LPSTR szExpr,  
    IGXObject *pData,  
    IGXObject *pMark,  
    IGXBuffer **pBuff);
```

**szExpr** In the current GX markup tag in the HTML template being processed, the name of the field, or placeholder, assigned to the `id` attribute. Must be an identical match (case-sensitive).

**pData** Specify NULL. Internal use only.

**pMark** Specify NULL. Internal use only.

**pBuff** Pointer to the IGXBuffer object that contains the returned value. This method allocates the IGXBuffer object automatically. When AppLogic is finished using the object, call the `Release()` method to release the interface instance.

**Usage** GX markup tags are used in an HTML template to identify where dynamic data appears in the output report. In the GX markup tags, the `id` attribute specifies any of the following items: the name of a flat query within a hierarchical query, a field in the hierarchical result set or TemplateDataBasic object, or an HTML template. The type of item specified in the `id` attribute depends on the `type` attribute that is specified in the same GX markup tag.

The Template Engine calls `Get()` to resolve the `id` attribute specified in a GX markup tag in the template being processed by the Template Engine. To provide application-specific special processing, an AppLogic can subclass the `GXTemplateMapBasic` class and override `Get()` to manipulate the Template Engine generation process. For example, an AppLogic can intercept and filter data from a database before the Template Engine processes it.

**Rule** An AppLogic should use Get( ) only to override it after subclassing the GXTemplateMapBasic class.

**Tip** Use methods in the IGXBuffer interface to manipulate the memory block.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** EvalTemplate( ) and EvalOutput( ) in the GXAppLogic class

GXTemplateDataBasic class

IGXTemplateData interface

“GX Markup Tag Syntax” in Chapter 7, “Working with Templates,” in *Programmer’s Guide*.



# IGXTile interface

The IGXTile interface represents a tile, which is a record set that contains multiple records. A tile can also contain nested tiles. Organized like a hierarchical result set, a tile is returned by the `GXProcessOutput()` helper function.

AppLogics use IGXTile together with `GXProcessOutput()` when working with non-HTML results returned by another AppLogic. The following are the general steps for getting the tile:

1. A client AppLogic calls an AppLogic with `NewRequest()`.
2. Through `NewRequest()`, the client passes input and output IGXVallists to the called AppLogic. If the client is an AppLogic, it specifies the value "ocl" for the `gx_client_type` key in the input IGXVallist.
3. The called AppLogic processes the request and sends back results using its output IGXVallist or by calling `EvalOutput()`.
4. The client calls the `GXProcessOutput()` helper function to process the results into an IGXTile object.
5. Using methods in the IGXTile interface, the client traverses the tile and retrieves values to populate user interface controls, such as text boxes or list boxes, on a form.

The tile corresponds to the structure specified by the `tile` and `cell` tags in the template file that the called AppLogic used when it called `EvalOutput()`. The `tile` tag determines the tile or record set, and the `cell` tag, the values in each record.

## Include File

`gxcipm.h`

## Methods

| Method               | Description   |
|----------------------|---|
| GetTileChild()       | Returns the specified child tile.                   |
| GetTileValue()       | Returns the value of a specified field in a record. |
| MoveTileNextRecord() | Moves to the next record in the tile.               |
| MoveTileToRecord()   | Moves to a specific record in the tile.             |

## Example

The following example shows a template file used by a called AppLogic when generating output, and a section of a program that uses GXProcessOutput() and IGXTile methods to process the output:

GXML template file:

```
<gx type=tile id="PRODUCTS" max=100>
<gx type=cell id="PRODUCTS.Category"></gx>
<gx type=cell id="PRODUCTS.ProdName"></gx>
</gx>
<gx type=tile id="CATEGORIES" max=100>
<gx type=cell id="CATEGORIES.CategoryId"></gx>
</gx>
```

Code snippet:

```
// Call this AppLogic
hr = pConn->NewRequest(guid, vIn, vOut, 0);
if (hr != NOERROR)
{
    printf("Failed to invoke NewRequest()\n");
    exit(-1);
}

// Get the root tile from the output vlist
mainTile = NULL;
hr = GXProcessOutput(NULL, 0, vOut, &mainTile);

if (hr == NOERROR)
{
```

```

//Iterate over all categories and print their names
ptile = NULL;
if ((hr = mainTile->GetTileChild("CATEGORIES", &ptile)) != NOERROR)
{
    printf("Unable to get tile child, hr = %d\n", hr);
}
while (ptile && hr == NOERROR)
{
    hr = ptile->GetTileValue("CATEGORIES.Name", sval, sizeof(sval));
    if (hr == NOERROR)
    {
        for (int i=0; i < (depth * 2); i++)
            printf(" ");
        printf("Category %s\n", sval);
    }

    hr = ptile->GetTileValue("CATEGORIES.CategoryId", sval,
sizeof(sval));

    if (hr == NOERROR)
    {
        test_Catalog(pConn, sval, depth+1);
    }
}

hr = ptile->MoveTileNextRecord();
}
if (ptile)
    ptile->Release();

```

## Related Topics

NewRequest() and EvalOutput() in the GXAppLogic class

GXProcessOutput(),  
IGXValList interface

## GetTileChild()

Returns the specified tile.

**Syntax** HRESULT GetTileChild(  
LPSTR name,  
IGXTile \*\*tile);

**name** The name of the child tile in the tile. This name must match a name assigned to the `id` attribute of type `tile` in the template file.

**tile** A pointer to the retrieved IGXTile object. When the client is finished using the object, call the `Release()` method to release the interface instance.

**Usage** Use `GetTileChild()` to retrieve a tile from which to get records and record values. Use it in conjunction with `MoveTileNextRecord()` and `GetTileValue()` to traverse the tile and retrieve record values. The client can call these methods in a loop until all values in a tile have been retrieved.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `GetTileValue()`,  
`MoveTileNextRecord()`

---

## GetTileValue()

Returns the value of a specified field in a record.

**Syntax** HRESULT GetTileValue(  
LPSTR name,  
LPSTR value,  
ULONG valuelen);

**name** The name of the field in the current record. This name must match a name assigned to the `id` attribute of type `cell` in the template file.

**value** Pointer to a buffer allocated by the client to store the returned string value.

**valuelen** The size of the buffer to store the value.

**Usage** Use `GetTileValue()` to retrieve values in a record. Use it in conjunction with `GetTileChild()` and `MoveTileNextRecord()` to traverse the tile and retrieve each value. The client can call these methods in a loop until all values in a tile have been retrieved.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** GetTileChild(),  
MoveTileNextRecord()

---

## MoveTileNextRecord()

Moves to the next record in the tile.

**Syntax** HRESULT MoveTileNextRecord()

**Usage** Use MoveTileNextRecord() to go to the next record in a tile after retrieving values in the current record. Use the method in conjunction with GetTileChild() and GetTileValue() to traverse the tile and retrieve each value. The client can call these methods in a loop until all values in a tile have been retrieved.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** GetTileChild(),  
GetTileValue()

---

## MoveTileToRecord()

Moves to a specific record in the tile.

**Syntax** HRESULT MoveTileToRecord(  
    ULONG ord);

**ord** The position of the record in the tile. The first record in a tile is 1, the second is 2, and so on.

**Usage** Use MoveTileToRecord() when iterating through the tile multiple times. For example, after iterating through all the records, the AppLogic can return to the first record in preparation for the next iteration. If the tile contains many records, the AppLogic can also use MoveTileToRecord() to display only several records at a time.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.



# IGXTrans interface

The IGXTrans interface represents a transaction object used for subsequent transaction processing operations. IGXTrans provides operations for beginning, committing, and rolling back transactions.

After instantiating a transaction object, the AppLogic calls `Begin()` to start the transaction. Next, the AppLogic performs any query, insert, update, or delete operations, passing the transaction object to the respective method in the IGXTable interface. Finally, the AppLogic closes the transaction by calling either `Commit()` to save all changes or `Rollback()` to cancel them. Closing a transaction terminates the transaction object and releases system resources.

The calls that make up a transaction can be in any part of the code; they need not be consecutive. The commands in a transaction are united by the fact that they all have the same transaction object as a parameter.

An application can process several transactions simultaneously. Each transaction works with a different database connection object. Within a single transaction, however, all the commands must access a single database through a single connection object.

To create an instance of the IGXTrans interface, use `CreateTrans()` in the `GXAppLogic` class.

## Include File

`gxitrans.h`

## Methods

| Method                  | Description   |
|-------------------------|---|
| <code>Begin()</code>    | Starts the transaction.                             |
| <code>Commit()</code>   | Commits the transaction, saving any changes.        |
| <code>Rollback()</code> | Rolls back the transaction, abandoning any changes. |

## Example

```
// Create a transaction for several insert operations
IGXTrans *pTx=NULL;

if(((hr=CreateTrans(&pTx))==GXE_SUCCESS)&&pTx) {
    // Begin the transaction
    pTx->Begin();
    IGXResultSet *pRset=NULL;

    // Update User
    if(((hr=pUserPQuery->Execute(0, pUserValList, pTx, NULL,
    &pRset))==GXE_SUCCESS)&&pRset) {

        // The result set is not needed; release it
        pRset->Release();

        // Update Customer
        if(((hr=pCustPQuery->Execute(0, pCustValList, pTx, NULL,
        &pRset))==GXE_SUCCESS)&&pRset) {

            // All is ok. Commit the transaction
            pTx->Commit(0, NULL);
            GXSetValListString(m_pValIn, "ssn", m_pSsn);
            GXSetValListString(m_pValIn, "OUTPUTMESSAGE", "Successfully
            updated customer record");

            if(NewRequest("AppLogic CShowCustPage", m_pValIn, m_pValOut,
            0)!=GXE_SUCCESS)
                HandleOBSYSTEMError("Could not chain to CShowCustPage
                applogic");
        }
    }
    else {
        pTx->Rollback();
        HandleOBSYSTEMError("Could not insert checking account record
        for new customer");
    }
}
else {
    pTx->Rollback();
    HandleOBSYSTEMError("Could not insert checking account record for
    new customer");
}
```

```

    }
    pTx->Release();
}
else
    HandleOBSystemError("Could not start transaction");

```

## Related Topics

CreateTrans() in the GXAppLogic class

AddRow(), UpdateRow(), and DeleteRow() in the IGXTable interface

“Managing Database Transactions” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

## Begin()

---

Starts the transaction.

**Syntax** HRESULT Begin();

**Usage** Use Begin() to start a transaction before performing any operations in the transaction. Subsequent operations belong to the current transaction until either Commit() or Rollback() is called.

- Rules**
- AppLogic must start the transaction explicitly using Begin() before performing any query, insert, update, or delete operations associated with the transaction.
  - AppLogic must complete the transaction explicitly by calling Commit() to save any changes to tables or Rollback() to abandon them. If a database error occurs before either are called, the database server will roll back the transaction automatically.

**Tip** Use transactions judiciously to avoid locking conflicts. For example, avoid deadlocks by not using different open transactions on the same table.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```

// Create a transaction
IGXTrans *pTx=NULL;

if(((hr=CreateTrans(&pTx))==GXE_SUCCESS)&&pTx) {

```

```
// Begin the transaction
pTx->Begin();
```

**Related Topics** CreateTrans() in the GXAppLogic class

AddRow(), UpdateRow(), and DeleteRow() in the IGXTable interface

“Managing Database Transactions” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

---

## Commit()

Commits the transaction, saving any changes.

**Syntax** HRESULT Commit(  
     DWORD dwFlags,  
     IGXObject \*\*ppEvent);

**dwFlags** Specify 0.

**ppEvent** Specify NULL. Internal use only.

**Usage** Use Commit() to commit a transaction and write unsaved changes to disk. Commit() saves the changes, terminates the transaction object, and releases system resources.

- Rules**
- The AppLogic must start the transaction explicitly by calling Begin() before any changes associated with the transaction can be committed.
  - The AppLogic must complete the transaction explicitly by calling Commit() to save any changes to tables or Rollback() to abandon them.
  - The AppLogic cannot reuse an IGXTrans object that has been committed. It must create a new one using CreateTrans() in the GXAppLogic class.

- Tips**
- If an error occurs before the commit operation succeeds, the database server usually rolls back the transaction automatically.
  - The target database server may take time to process a commit request.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
// Update Customer record
if((hr=pCustPQuery->Execute(0, pCustValList, pTx, NULL,
&pRset))==GXE_SUCCESS)&&pRset) {
```

```
// Operation succeeded. Commit the transaction.
pTx->Commit(0, NULL);
GXSetValListString(m_pValIn, "OUTPUTMESSAGE", "Successfully updated
customer record");
```

**Related Topics** CreateTrans() in the GXAppLogic class

AddRow(), UpdateRow(), and DeleteRow() in the IGXTable interface

“Managing Database Transactions” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

## Rollback()

Rolls back the transaction, abandoning any changes.

**Syntax** HRESULT Rollback();

**Usage** Many database servers buffer changes made during a transaction, then update the affected tables only after the commit request is received.

Rolling back a transaction terminates the transaction object and releases system resources.

- Rules**
- The AppLogic must start the transaction explicitly by calling Begin() before any changes associated with the transaction can be rolled back.
  - The AppLogic must complete the transaction explicitly by calling Commit() to save any changes to tables or Rollback() to abandon them.
  - The AppLogic cannot reuse an IGXTrans object that has been rolled back. It must create a new one using CreateTrans() in the GXAppLogic class.

**Tip** If an error occurs before the commit operation succeeds, the database server usually rolls back the transaction automatically.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
// Update User
if(((hr=pUserPQuery->Execute(0, pUserValList, pTx, NULL,
&pRset))!=GXE_SUCCESS)&&pRset) {
    // Query succeeded. Perform update.
else
    pTx->Rollback();
```

```
HandleOBSystemError("Could not insert checking account record for new  
customer");
```

**Related Topics** CreateTrans() in the GXAppLogic class

AddRow(), UpdateRow(), and DeleteRow() in the IGXTable interface

“Managing Database Transactions” in Chapter 5, “Working with Databases,” in *Programmer’s Guide*.

# IGXValList interface

An IGXValList represents a collection of GXVAL objects. This collection is not a sequential list, but an unordered set of GXVAL objects with no implied sequence or progression.

For Netscape Application Server-enabled AppLogics, input arguments and output value(s) are stored in IGXValList objects. Every request to an AppLogic passes a list of input arguments, and every result from an AppLogic returns a list of output values. The GXAppLogic class defines two member variables, `m_pValIn` and `m_pValOut`, to contain the input arguments and output values, respectively, of AppLogic execution.

In an IGXValList, values and objects are mapped to keys. The key name is the name of a GXVAL object. AppLogic code refers to GXVAL object in the IGXValList by its key name. Key names are unique within each IGXValList object.

The IGXValList interface provides methods for adding, retrieving, removing, and counting GXVAL objects in the IGXValList instance. Using methods in the IGXValList interface, the AppLogic can test for input arguments and modify their contents for output values.

Keys may be passed to the AppLogic as a request from an HTML document or from another AppLogic module. In an HTML form, keys are often the field names defined in the form. In this way, the AppLogic can easily identify expected, common, or “well-known” keys, and the AppLogic can ignore irrelevant parameters.

For example, an AppLogic named `getLogin` might prompt users for their username and login, then pass this information, identified as “username” and “password”, to other AppLogics for processing. An AppLogic named `validateLogin` could retrieve the input parameters, find the values associated with the well-known keys “username” and “password”, then take action based on the data that the user entered (testing for its existence, performing a range or length check, looking up the combination in a password table, and so on).

To create an instance of the IGXValList interface, use the `GXCreateValList()` function.

## Include File

gxival.h

## Methods

| Method           | Description   |
|------------------|---|
| Count()          | Returns the number of GXVAL objects in the IGXValList.                                |
| GetNextKey()     | Retrieves the key name of the next GXVAL object in the IGXValList.                    |
| GetVal()         | Copies the specified GXVAL object from the IGXValList.                                |
| GetValBLOB()     | Returns the specified BLOB object.  |
| GetValBLOBSize() | Returns the size of a BLOB IGXValList object.   |
| GetValByRef()    | Gets the specified GXVAL object in the IGXValList.                                    |
| GetValInt()      | Retrieves an integer value from the specified GXVAL object in the IGXValList.         |
| GetValString()   | Retrieves a string value from the specified GXVAL object in the IGXValList.           |
| RemoveVal()      | Removes the specified GXVAL object from the IGXValList.                               |
| ResetPosition()  | Resets the iterator position to the "first" GXVAL object in the IGXValList.           |
| SetVal()         | Adds a GXVAL object to the IGXValList, or overwrites an existing one.                 |
| SetValBLOB()     | Adds a BLOB object to the IGXValList object.  |
| SetValByRef()    | Adds a GXVAL object to the IGXValList, or overwrites an existing one.                 |
| SetValInt()      | Adds a GXVAL object of type integer to the IGXValList, or overwrites an existing one. |
| SetValString()   | Adds a GXVAL object of type string to the IGXValList, or overwrites an existing one.  |

## Related Topics

GXVAL struct

m\_pValIn and m\_pValOut in the GXAppLogic class

Execute() in the GXAppLogic class

“Passing Parameters to AppLogic From Code” and “Returning Output Parameters in an IGXValList Object” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

---

## Count()

Returns the number of GXVAL objects in the IGXValList.

**Syntax** HRESULT Count (  
          ULONG \*pCount);

**pCount** Pointer to the returned count of GXVAL objects.

**Usage** When the contents of an IGXValList are unknown, an AppLogic can iterate through each GXVAL object to test, retrieve, and update information. Use Count() to determine the maximum number of iterations needed to go completely through the IGXValList.

**Rule** Do not add or remove GXVAL objects to or from the IGXValList when iterating through the IGXValList.

**Tips**

- Use Count() in conjunction with GetNextKey() and ResetPosition() to iterate through the IGXValList.
- Adding or deleting GXVAL objects changes the number of objects in a IGXValList. Be sure to update the GXVAL object count after each add or delete operation.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** GetNextKey()

ResetPosition()

GXVAL struct

m\_pValIn and m\_pValOut in the GXAppLogic class

Execute() in the GXAppLogic class

“Passing Parameters to AppLogic From Code” and “Returning Output Parameters in an IGXValList Object” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

## GetNextKey()

Retrieves the key name of the next GXVAL object in the IGXValList.

**Syntax** HRESULT GetNextKey(  
     LPSTR pKey,  
     ULONG nKey);

**pKey** Pointer to a buffer allocated by the client to store the returned key string.

**nKey** The size of the buffer to store the key.

**Usage** When the contents of a IGXValList are unknown, the AppLogic can iterate through each GXVAL object and retrieve its key name. The AppLogic can then take action based on this information, or use the key name in operations that retrieve, update, or remove GXVAL objects in the IGXValList list.

**Rule** Do not add or remove GXVAL objects to or from the IGXValList when iterating through the IGXValList.

**Tip** Use GetNextKey() in conjunction with Count() and ResetPosition() to iterate through the IGXValList.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
CHAR pKey[256];
GXVAL val;

// Reset to the first GXVAL in the ValList before iteration
pValList->ResetPosition();

// Iterate through all the GXVALs in the
// vallist and print them to a file
while (pValList->GetNextKey(pKey, 256) == NOERROR) {
    pValList->GetValByRef(pKey, &val);

    if (GXVT_TYPE(val.vt) == GXVT_LPSTR)
        fprintf(fp, "\t%s=%s (LPSTR)\n", pKey, val.u.pstrVal);
    else
```

```
fprintf(fp, "\t%s=%d (DWORD)\n", pKey, val.u.ulVal);
```

**Related Topics** Count()

ResetPosition()

GXVAL struct

Execute() in the GXAppLogic class

“Passing Parameters to AppLogic From Code” and “Returning Output Parameters in an IGXValList Object” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

---

## GetVal()

Copies the specified GXVAL object from the IGXValList.

**Syntax** HRESULT GetVal(  
           LPSTR pKey,  
           GXVAL \*pVal);

**pKey** Key name of the GXVAL object to copy from the IGXValList.

**pVal** Pointer to the GXVAL allocated by the client to store the copy of the retrieved GXVAL object.

**Usage** Use GetVal() if the data type of the GXVAL object is not known. Use GetValString() instead for string objects, GetValInt() for integer objects, and GetValBLOB() for BLOB objects.

GetVal() makes a deep copy of the GXVAL object.

**Rule** The specified key name must currently exist in the IGXValList.

**Tip** Use the GXVALClear() function to release a GXVAL object when the AppLogic no longer needs it.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** GetValBLOB(), GetValInt(), and GetValString() in the IGXValList interface

GXVAL struct

m\_pValIn and m\_pValOut in the GXAppLogic class

Execute() in the GXAppLogic class

“Passing Parameters to AppLogic From Code” and “Returning Output Parameters in an IGXValList Object” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

## GetValBLOB()

---

Returns a specified BLOB object from the IGXValList.

**Syntax** HRESULT GetValBLOB(  
     LPSTR pKey,  
     LPBYTE pVal,  
     ULONG nBufferLen);

**pKey** Key name of the GXVAL object that contains the BLOB value to retrieve.

**pVal** Pointer to a buffer allocated by the client to store the returned value.

**nBufferLen** Length of the buffer allocated by the client.

**Usage** Use GetValBLOB() when the type of a GXVAL object is a BLOB, but its value is not known and needed for subsequent operations. Use GetValString() instead for string objects and GetValInt() for integer objects. If the type of the GXVAL object is not known, use GetVal().

**Rule** The data type must be TEXT, BINARY, VARBINARY, or database equivalent.

**Tip** Call GetValBLOBSize() before GetValBLOB() to determine the size of the BLOB so the code can allocate the appropriate buffer size for it.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** GetValBLOBSize() and SetValBLOB() in the IGXValList interface

## GetValBLOBSize()

---

Returns the size of a specified BLOB object in the IGXValList.

**Syntax** HRESULT GetValBLOBSize(  
     LPSTR pKey,  
     ULONG \*pBuffLen);

**pKey** Key name of the GXVAL object that contains the BLOB.

**pBuffLen** Pointer to a buffer allocated by the client to store the returned value.

**Usage** BLOB objects can be large. If you want to determine the size of a BLOB object before retrieving it, use `GetValBLOBSize()`.

**Rule** The data type must be TEXT, BINARY, VARBINARY, or database equivalent.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Related Topics** `GetValBLOB()`

---

## GetValByRef()

Gets the specified GXVAL object from the IGXValList.

**Syntax**

```
HRESULT GetValByRef(
    LPSTR pKey,
    GXVAL *pVal);
```

**pKey** Key name of the GXVAL object to get from the IGXValList.

**pVal** Pointer to the GXVAL allocated by the client to store the retrieved GXVAL object.

**Usage** Use `GetValByRef()` if the data type of the GXVAL object is not known, or if iterating through an IGXValList to get each GXVAL object. Use `GetValString()` instead for string objects, `GetValInt()` for integer objects, and `GetValBLOB()` for BLOB objects.

`GetValByRef()` makes a shallow copy of the specified GXVAL object in the IGXValList. If you want a deep copy, call `GetVal()`.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```
CHAR pKey[256];
GXVAL val;

// Reset to the first GXVAL in the ValList before iteration
pValList->ResetPosition();

// Iterate through all the GXVALs in the
// vallist, get each value and print it to a file
while (pValList->GetNextKey(pKey, 256) == NOERROR) {
```

```

pValList->GetValByRef(pKey, &val);

if (GXVT_TYPE(val.vt) == GXVT_LPSTR)
    fprintf(fp, "\t%s=%s (LPSTR)\n", pKey, val.u.pstrVal);
else
    fprintf(fp, "\t%s=%d (DWORD)\n", pKey, val.u.ulVal);

```

**Related Topics** GetValBLOB(), GetValInt(), and GetValString() in the IGXValList interface

GXVAL struct

m\_pValIn and m\_pValOut in the GXAppLogic class

Execute() in the GXAppLogic class

“Passing Parameters to AppLogic From Code” and “Returning Output Parameters in an IGXValList Object” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

## GetValInt()

---

Retrieves an integer value from the specified GXVAL object in the IGXValList.

**Syntax** HRESULT GetValInt(  
LPSTR pKey,  
LONG \*pVal);

**pKey** Key name of the GXVAL object from which to retrieve the integer value.

**pVal** Pointer to a buffer allocated by the client to store the returned value.

**Usage** Use GetValInt() if the data type of the GXVAL object is known to be an integer. Otherwise, use GetValString() instead for string objects, GetValBLOB() for BLOB objects, or GetVal() for objects of other types.

- Rules**
- The specified key name must currently exist in the IGXValList.
  - The data type of the specified GXVAL object must map to the enum value GXVT\_I4.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** GXVAL struct

m\_pValIn and m\_pValOut in the GXAppLogic class

Execute() in the GXAppLogic class

“Passing Parameters to AppLogic From Code” and “Returning Output Parameters in an IGXValList Object” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

## GetValString()

---

Retrieves a string value from the specified GXVAL object in the IGXValList.

**Syntax** HRESULT GetValString(  
     LPSTR pKey,  
     LPSTR pBuffer,  
     ULONG nBuff);

**pKey** Key name of the GXVAL object from which to retrieve the string value.

**pBuff** Pointer to a buffer allocated by the client to store the returned value.

**nBuff** Length of the buffer allocated by the client.

**Usage** Use GetValString() when the data type of the GXVAL object is known to be a string. Otherwise, use GetVal() instead for integer objects, GetValBLOB() for BLOB objects, or GetVal() for objects of other types.

- Rules**
- The specified key name must currently exist in the IGXValList.
  - The data type of the specified GXVAL object must map to the enum value GXVT\_LPSTR.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** GXVAL struct

m\_pValIn and m\_pValOut in the GXAppLogic class

Execute() in the GXAppLogic class

“Passing Parameters to AppLogic From Code” and “Returning Output Parameters in an IGXValList Object” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

## RemoveVal()

---

Removes the specified GXVAL object from the IGXValList.

**Syntax** HRESULT RemoveVal(  
LPSTR pKey);

**pKey** Key name of the GXVAL object to remove from the IGXValList.

**Usage** Use RemoveVal() to delete a GXVAL object that is no longer needed in the IGXValList. For example, if the AppLogic contains overloaded methods, you might want to remove a GXVAL object to ensure that the proper method is executed.

- Rules**
- The specified key name must currently exist in the IGXValList.
  - Do not remove GXVAL objects from the IGXValList when iterating through the IGXValList.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** SetVal() and ResetPosition() in the IGXValList interface

GXVAL struct

m\_pValIn and m\_pValOut in the GXAppLogic class

Execute() in the GXAppLogic class

“Passing Parameters to AppLogic From Code” and “Returning Output Parameters in an IGXValList Object” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

## ResetPosition()

---

Resets the iterator position to the “first” GXVAL object in the IGXValList.

**Syntax** HRESULT ResetPosition()

**Usage** When the contents of an IGXValList are unknown, the AppLogic can iterate through each GXVAL object and retrieve its key name. Before iterating through the IGXValList, the AppLogic needs to call ResetPosition() once to ensure that iteration begins at the “first” GXVAL object in the IGXValList.

**Rule** Do not add or remove GXVAL objects to or from the IGXValList when iterating through the IGXValList.

- Tips**
- The first GXVAL object is not necessarily the first one added to the IGXValList.
  - Use `ResetPosition()` in conjunction with `Count()` and `GetNextKey()` to iterate through the IGXValList.

**Return Value** HRESULT, which is set to `GXE_SUCCESS` if the method succeeds.

**Example**

```
CHAR pKey[256];
GXVAL val;

// Reset to the first GXVAL in the ValList before iteration
pValList->ResetPosition();

// Iterate through all the GXVALs in the
// vallist and print them to a file
while (pValList->GetNextKey(pKey, 256) == NOERROR) {
    pValList->GetValByRef(pKey, &val);

    if (GXVT_TYPE(val.vt) == GXVT_LPSTR)
        fprintf(fp, "\t%s=%s (LPSTR)\n", pKey, val.u.pstrVal);
    else
        fprintf(fp, "\t%s=%d (DWORD)\n", pKey, val.u.ulVal);
}
```

**Related Topics** `Count()` and `GetNextKey()` in the IGXValList interface

GXVAL struct

`m_pValIn` and `m_pValOut` in the `GXAppLogic` class

`Execute()` in the `GXAppLogic` class

“Passing Parameters to AppLogic From Code” and “Returning Output Parameters in an IGXValList Object” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

---

## SetVal()

Copies a GXVAL object to the IGXValList.

**Syntax** HRESULT SetVal(

```
LPSTR pKey,
GXVAL *pVal);
```

**pKey** Key name of the GXVAL object to add to the IGXValList.

**pVal** The GXVAL object, identified by pKey, to add to the IGXValList.

**Usage** Use SetVal() to add an existing GXVAL object to the IGXValList. If a GXVAL object with the same key name already exists, SetVal() overwrites it with the new one.

SetVal() makes a deep copy of the existing GXVAL object to add it to the IGXValList. If you do not want to make a deep copy, use SetValByRef() instead.

**Rule** Do not add new GXVAL objects to the IGXValList when iterating through the IGXValList.

**Tip** To add a new GXVAL object of type integer, string, or BLOB to the IGXValList, use SetValInt(), SetValString(), or SetValBLOB(), respectively.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** GXVAL struct

m\_pValIn and m\_pValOut in the GXAppLogic class

Execute() in the GXAppLogic class

“Passing Parameters to AppLogic From Code” and “Returning Output Parameters in an IGXValList Object” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

## SetValBLOB()

---

Adds a BLOB object to the IGXValList.

**Syntax** HRESULT SetValBLOB(  
LPSTR pKey,  
LPBYTE pBuff,  
ULONG nBuffLen);

**pKey** Key name of the GXVAL object to add to the IGXValList.

**pBuff** The value of the BLOB object to add to the IGXValList.

**nBuffLen** Number of bytes to set for the byte array. The first nBuffLen bytes in the array pBuff hold the value.

**Usage** Use SetValBLOB() to add a GXVAL object that contains a BLOB value to the IGXValList. If a GXVAL object with the same key name already exists, SetValBLOB() overwrites it with the new one.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** GetValBLOB()  
GetValBLOBSize()

---

## SetValByRef()

Copies a GXVAL object to the IGXValList.

**Syntax** HRESULT SetValByRef(  
LPSTR pKey,  
GXVAL \*pVal);

**pKey** Key name of the GXVAL object to add to the IGXValList.

**pVal** The GXVAL object, identified by pKey, to add to the IGXValList.

**Usage** Use SetValByRef() to add an existing GXVAL object to the IGXValList. If a GXVAL object with the same key name already exists, SetValByRef() overwrites it with the new one.

SetValByRef() makes a shallow copy of the existing GXVAL object to add it to the IGXValList. To make a deep copy, use SetVal().

**Rule** Do not add new GXVAL objects to the IGXValList when iterating through the IGXValList.

**Tip** To add a new GXVAL object of type integer, string, or BLOB to the IGXValList, use SetValInt(), SetValString(), or SetValBLOB(), respectively.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Related Topics** GXVAL struct  
m\_pValIn and m\_pValOut in the GXAppLogic class  
Execute() in the GXAppLogic class

“Passing Parameters to AppLogic From Code” and “Returning Output Parameters in an IGXValList Object” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

## SetValInt()

---

Adds a GXVAL object of type integer to the IGXValList.

**Syntax** HRESULT SetValInt(  
     LPSTR pKey,  
     LONG nVal);

**pKey** Key name of the GXVAL object to create or overwrite.

**nVal** The integer value to assign to the GXVAL object identified by pKey.

**Usage** Use SetValInt() to add a GXVAL object of type integer to the IGXValList. If a GXVAL object with the same key name already exists, SetValInt() overwrites it with the new one.

**Rules** When iterating through existing GXVAL objects in the IGXValList, do not add new GXVAL objects to the IGXValList.

**Tips** To add a new GXVAL object of type string or BLOB to the IGXValList, use SetValString() or SetValBLOB(), respectively.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
// Create an IGXValList and set integer values
IGXValList *pAcct1ValList=GXCreateValList();

pAcct1ValList->SetValInt(":balance", 0);
pAcct1ValList->SetValInt(":acctType", 1);
```

**Related Topics** GXVAL struct

m\_pValIn and m\_pValOut in the GXAppLogic class

Execute() in the GXAppLogic class

“Passing Parameters to AppLogic From Code” and “Returning Output Parameters in an IGXValList Object” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.

## SetValString()

---

Adds a GXVAL object of type string to the IGXValList.

**Syntax** HRESULT SetValString(  
LPSTR pKey,  
LPSTR val);

**pKey** Key name of the GXVAL object to create or overwrite.

**val** The string value to assign to the GXVAL object identified by pKey.

**Usage** Use SetValString() to add a GXVAL object of type string to the IGXValList. If a GXVAL object with the same key name already exists, SetValString() overwrites it with the new one.

**Rules** When iterating through existing GXVAL objects in the IGXValList, do not add new GXVAL objects to the IGXValList.

**Tips** To add a new GXVAL object of type integer or BLOB to the IGXValList, use SetValInt() or SetValBLOB(), respectively.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Example**

```
// Create an IGXValList and set string values
IGXValList *pCustValList=GXCreateValList();

if(pUserValList&& pCustValList) {
    pCustValList->SetValString(":ssn", m_pSsn);
    pCustValList->SetValString(":firstName", m_pFirstName);
    pCustValList->SetValString(":lastName", m_pLastName);
}
```

**Related Topics** GXVAL struct

m\_pValIn and m\_pValOut in the GXAppLogic class

Execute() in the GXAppLogic class

“Passing Parameters to AppLogic From Code” and “Returning Output Parameters in an IGXValList Object” in Chapter 4, “Writing Server-Side Application Code,” in *Programmer’s Guide*.



# C++ Functions

This chapter discusses C++ functions in the Netscape Application Server Foundation Class Library.

The following functions are included in this chapter:

|                             |                              |
|-----------------------------|------------------------------|
| GXContextGetAppEventMgr( )  | GXContextGetSessionCount( )  |
| GXCreateBuffer( )           | GXCreateValList( )           |
| GXCreateStreamBuffer( )     | GXCreateTemplateDataBasic( ) |
| GXCreateTemplateMapBasic( ) | GXDeleteCriticalSection( )   |
| GXEnterCriticalSection( )   | GXGetCurrentDateTime( )      |
| GXGetValList( )             | GXGetValListBLOB( )          |
| GXGetValListGUID( )         | GXGetValListString( )        |
| GXGUIDToString( )           | GXInitCriticalSection( )     |
| GXLeaveCriticalSection( )   | GXProcessOutput( )           |
| GXSetValList( )             | GXSetValListBLOB( )          |
| GXSetValListGUID( )         | GXSetValListString( )        |
| GXStringToGUID( )           | GXSYNC_DEC( )                |
| GXSYNC_DESTROY( )           | GXSYNC_INC( )                |
| GXSYNC_INIT( )              | GXSYNC_LOCK( )               |
| GXSYNC_UNLOCK( )            | GXVALClear( )                |
| GXVALCopy( )                | GXWaitForOrder( )            |

## GXContextGetAppEventMgr()

Retrieves the object for managing application events.

**Syntax** HRESULT GXContextGetAppEventMgr(  
 IGXContext \*pContext,  
 IGXAppEventMgr \*\*ppAppEventMgr)

**pContext** A pointer to the IGXContext object, which provides access to NAS services. Specify m\_pContext, a member variable in the GXAppLogic class.

**pAppEventMgr** Pointer to the returned IGXAppEventMgr object.

**Usage** Use GXContextGetAppEventMgr() to retrieve an IGXAppEventMgr object. Through the IGXAppEventMgr interface, you can create and manage application events. Application event objects define events that are triggered at a specified time or triggered explicitly.

**Return Value** IGXAppEventMgr object, or NULL for failure.

**Include File** gxdlmutil.h

**Related Topics** IGXAppEventMgr interface

## GXContextGetSessionCount()

Returns the number of sessions in the cluster.

**Syntax** GXContextGetSessionCount(  
 IGXContext \*pContext,  
 DWORD dwFlags,  
 LPSTR pAppName,  
 ULONG \*pCount)

**pContext** A pointer to the IGXContext object, which provides access to NAS services. Specify m\_pContext, a member variable in the GXAppLogic class.

**dwFlags** Not used.

**pAppName** Name of the application for which sessions are being counted.

**pCount** A ULONG pointer to where the session count is returned.

|                       |   |
|-----------------------|---|
| <b>Usage</b>          | Use <code>GXContextGetSessionCount()</code> to obtain a count of sessions in the cluster. |
| <b>Return Value</b>   | An integer representing the session count.  |
| <b>Include File</b>   | <code>gxdlmutil.h</code>  |
| <b>Related Topics</b> | <code>GetStateChildCount()</code> in the <code>IGXState2</code> interface                 |

## GXCreateBuffer()

Creates a new `IGXBuffer` object, which represents a block of memory.

**Syntax** `IGXBuffer *GXCreateBuffer();`

**Usage** Use to create a memory block that can be shared by multiple objects. Thereafter, use methods in the `IGXBuffer` interface to manage this memory block. When the `AppLogic` is finished using the object, call the `Release()` method to release the interface instance.

- Tips**
- After creating the `IGXBuffer` object, call `Alloc()` in the `IGXBuffer` interface to allocate the memory buffer managed by the `IGXBuffer`.
  - Call `GetAddress()` to obtain the starting address of the memory block.

**Return Value** `IGXBuffer` object.

**Include File** `gxutil.h`

**Example**

```
STDMETHODIMP
OBBaseAppLogic::HandleOBValidationError(LPSTR pMessage)
{
    HRESULT hr = GXE_SUCCESS;
    GXTemplateMapBasic map;

    IGXBuffer *pBuff = GXCreateBuffer();
    if(pBuff)
    {
        pBuff->Alloc(strlen(pMessage)+1);
        strcpy((char*)pBuff->GetAddress(), pMessage);
        map.Put("OUTPUTMESSAGE", pBuff);
        // Send it to the template
        hr=EvalTemplate("GXApp/COOnlineBank/templates/
        ValidationError.html", (IGXTemplateData*) NULL, &map, NULL,
```

```

        NULL);
        pBuff->Release();
    }
    return hr;
}

```

**Related Topics** IGXBuffer interface

GetFields(), GetGroupBy(), GetHaving(), GetOrderBy(), GetSQL(), GetTables(), and GetWhere() in the IGXQuery interface

GetValue() in the IGXTemplateData interface

Get() in the IGXTemplateMap interface

## GXCreateStreamBuffer()

Creates a new IGXStream object, which represents a buffer for capturing streamed output during template processing.

**Syntax** IGXStream \*GXCreateStreamBuffer(  
IGXStream \*pstream);

**pstream** Specify NULL to create a simple stream buffer. Specify another stream buffer to chain two stream buffers.

**Usage** Use GXCreateStreamBuffer() to create a stream buffer to pass to EvalOutput() or EvalTemplate(). The EvalOutput() and EvalTemplate() methods merge a template with data from the IGXTemplateData object and stream the output to the buffer. Use a stream buffer if, for example, your AppLogic needs to manipulate the data before sending it to another AppLogic.

**Tip** The IGXStream object implements the IGXStreamBuffer interface. To manipulate data in a stream buffer, use the GetStreamData() method in the IGXStreamBuffer interface.

**Return Value** IGXStream object.

**Include File** gxutil.h

**Related Topics** EvalOutput() and EvalTemplate() in the GXAppLogic class

IGXStreamBuffer interface

# GXCreateTemplateDataBasic()

Creates a GXTemplateDataBasic object, which represents a hierarchical source of data.

**Syntax** GXTemplateDataBasic \*GXCreateTemplateDataBasic(  
LPSTR name = NULL)

**name** The name of the TemplateDataBasic object.

**Usage** Use GXCreateTemplateDataBasic() to create a hierarchical source of data to pass to EvalOutput() or EvalTemplate(). The EvalOutput() and EvalTemplate() methods merge a template with data from the ITemplateData object and stream an output report.

**Return Value** GXTemplateDataBasic object.

**Include File** gxtmplbasic.h

**Example** In the following code snippet, two TemplateDataBasic objects are created to store the results from a query to avoid running the same query twice. The two TemplateDataBasic objects are then combined into one and passed to evalTemplate() for processing.

```
GXTemplateDataBasic *pAcctsTempDB =
GXCreateTemplateDataBasic("SelCustAccts");

GXTemplateDataBasic *pAcctsTempDB2 =
GXCreateTemplateDataBasic("SelCustAccts2");

if(pAcctsTempDB&& pAcctsTempDB2) {
    char pAcctDesc[200];
    char pAcctNum[200];

    // Get the indices of columns in the result set
    ULONG acctDescCol=0;
    pRset->GetColumnOrdinal("OBAccountType_acctDesc", &acctDescCol);
    ULONG acctNumCol=0;
    pRset->GetColumnOrdinal("OBAccount_acctNum", &acctNumCol);

    char tmpStr[300];

    // Loop through the result set and add rows to the
    // TemplateDataBasic objects
```

## GXCreateTemplateMapBasic()

```
do {
    pRset->GetValueString(acctDescCol, pAcctDesc, 200);
    pRset->GetValueString(acctNumCol, pAcctNum, 200);
    sprintf(tmpStr, "acctDesc=%s;acctNum=%s", pAcctDesc, pAcctNum);
    pAcctsTempDB->RowAppend(tmpStr);
    pAcctsTempDB2->RowAppend(tmpStr);
} while(pRset->FetchNext() != GXE_SUCCESS);

// Create dummy parent to contain the two template objects
GXTemplateDataBasic *pParent=NULL;
if((pParent=GXCreateTemplateDataBasic("Parent"))) {
    // Create one dummy row
    pParent->RowAppend("Dummy=dummy");
    pParent->GroupAppend(pAcctsTempDB);
    pParent->GroupAppend(pAcctsTempDB2);

    // Merge the template data results with a template

    if(EvalTemplate("GXApp/COOnlineBank/templates/Transfer.html",
        pParent, NULL, NULL, NULL) != GXE_SUCCESS)
        Result("<HTML><BODY>Unable to evaluate template.</BODY></
        HTML>");

    pParent->Release();
}
```

**Related Topics** EvalOutput() and EvalTemplate() in the GXAppLogic class

IGXTemplateData interface

GXTemplateDataBasic class

# GXCreateTemplateMapBasic()

Creates a new GXTemplateMapBasic object, which represents a mapping between a template field specification and dynamic data used for template processing.

**Syntax** GXTemplateMapBasic \*GXCreateTemplateMapBasic();

- Usage** Use `GXCreateTemplateMapBasic()` to create a template map object to pass to `EvalOutput()` or `EvalTemplate()`. A template map object is used to link template fields to calculated values or to source data with a non-matching field name but identically-formatted data.
- Return Value** `GXTemplateMapBasic` object.
- Include File** `gxtmplbasic.h`
- Related Topics** `EvalOutput()` and `EvalTemplate()` in the `GXAppLogic` class  
`IGXTemplateMap` interface  
`GXTemplateMapBasic` class

## GXCreateValList()

Creates a new `IGXValList` object.

**Syntax** `IGXValList *GXCreateValList();`

**Usage** Use `GXCreateValList()` to create a new `IGXValList` object. Thereafter, use methods in the `IGXValList` interface to manage this `IGXValList` object. When the `AppLogic` is finished using the object, call the `Release()` method to release the interface instance.

**Return Value** `IGXValList` object.

**Include File** `gxval.h`

**Example**

```
// Set up an IGXValList for inserting data into a database
IGXValList *pCustValList=GXCreateValList();

if(pUserValList&&pCustValList) {
    pCustValList->SetValString(":ssn", m_pSSn);
    pCustValList->SetValString(":prefix", m_pPrefix);
    pCustValList->SetValString(":firstName", m_pFirstName);
    pCustValList->SetValString(":lastName", m_pLastName);

// Create the query to update the OBCustomer table
GXQuery *pCustQuery=NULL;

if((hr=CreateQuery(&pCustQuery))==GXE_SUCCESS)&&pCustQuery) {
```

## GXDeleteCriticalSection()

```
pCustQuery->SetSQL("UPDATE OBCustomer SET prefix = :prefix, firstName  
= :firstName, lastName = :lastName, WHERE ssn = :ssn");  
  
// Execute the query and pass in the IGXValList  
if(((hr=pCustPQuery->Execute(0, pCustValList, pTx, NULL,  
    &pRset))==GXE_SUCCESS)&&pRset)
```

**Related Topics** IGXValList interface

GXGetValList(), GXGetValListBLOB(), GXGetValListGUID(), and  
GXGetValListBLOB() functions

# GXDeleteCriticalSection()

Deletes a critical section object.

**Syntax** void GXDeleteCriticalSection(GXCRIT\_SECTION \*x);

**x** Pointer to the GXCRIT\_SECTION variable that represents the critical section to delete.

**Usage** Use GXDeleteCriticalSection() to destroy a critical section object that AppLogic no longer needs. Calling GXDeleteCriticalSection() releases the system resources allocated for the critical section object.

- Rules**
- The specified critical section variable must be initialized by a previous call to GXInitCriticalSection().
  - Before deleting the object, the AppLogic must release ownership of the specified critical section by calling GXLeaveCriticalSection().
  - Subsequent calls to the critical section are invalid. To use the critical section again, the AppLogic must subsequently initialize the critical section using GXInitCriticalSection().

- Tips**
- Delete a critical section as soon as the AppLogic no longer needs it, such as in a destructor method.
  - In multithreaded programming, use critical sections in your AppLogic to ensure synchronization when multiple threads can manipulate the same object.

**Return Value** void

**Include File** gxplat.h

**Related Topics** “Using Critical Sections” in Chapter 3, “Application Development Techniques,” in *Programmer’s Guide*.

## GXEnterCriticalSection()

Waits for exclusive ownership of a critical section and returns when ownership is granted.

**Syntax** `void GXEnterCriticalSection(GXCRIT_SECTION *x);`

**x** Pointer to the GXCRIT\_SECTION variable that represents the critical section to enter.

**Usage** Use GXEnterCriticalSection() to obtain exclusive thread access to a shared resource before performing any operations on the protected resource. GXEnterCriticalSection() blocks until the thread is granted ownership.

- Rules**
- The specified critical section must be initialized by a previous call to GXInitCriticalSection().
  - The specified critical section must be released by a subsequent call to GXLeaveCriticalSection(). Otherwise, a deadlock may occur.

- Tips**
- Release a critical section as soon as the AppLogic no longer needs it so that other threads may acquire it.
  - In multithreaded programming, use critical sections in your AppLogic to ensure synchronization when multiple threads can manipulate the same object.

**Include File** gxplat.h

**Related Topics** “Using Critical Sections” in Chapter 3, “Application Development Techniques,” in *Programmer’s Guide*.

## GXGetCurrentDateTime()

Returns the current system date and time in a GXDATETIME format.

**Syntax** void GXGetCurrentDateTime(  
GXDATETIME \*pDT);

**pDT** Pointer to the GXDATETIME struct that will be filled with the returned system date and time.

**Usage** Use GXGetCurrentDateTime() to obtain the current system date and time for use in subsequent operations, such as computing the elapsed time or saving timestamp information in a new or modified row in a table.

**Return Value** void

**Include File** gxutil.h

**Example**

```
// Get the current date time
GXDATETIME dt;
GXGetCurrentDateTime(&dt);
char dateStr[50];
sprintf(dateStr, "%d-%d-%d %d:%d:%d", dt.year, dt.month, dt.day,
dt.hour, dt.minute, dt.second);
Log(dateStr);
```

**Related Topics** GXDATETIME struct

## GXGetValList()

Retrieves the data type and value of a GXVAL object in an IGXVallist.

**Syntax** HRESULT GXGetValList(  
IGXVallist \*list,  
LPSTR key,  
GXVALTYPE \*type,  
DWORD \*val);

**list** IGXVallist containing the GXVAL object whose data type and value to retrieve.

**key** Key name of the GXVAL object whose data type and value to retrieve.

**type** Pointer to the GXVALTYPE variable allocated by the client to store the retrieved data type of the GXVAL object.

**val** Pointer to the DWORD variable allocated by the client to store the retrieved value of the GXVAL object.

**Usage** Use the GXGetValList() function when the GXVAL object is 32 bits in size, but its exact type and value are not known and needed for subsequent operations. If the GXVAL object is of type string, BLOB, or GUID, use GXGetValListString(), GXGetValListBLOB(), and GXGetValListGUID(), respectively.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the function succeeds.

**Include File** gxval.h

**Related Topics** GXSetValList()  
 GXVAL struct  
 IGXValList interface

## GXGetValListBLOB()

Returns the BLOB object in an IGXValList.

**Syntax** LPBYTE GXGetValListBLOB(  
 IGXValList \*list,  
 LPSTR key,  
 DWORD \*pSize);

**list** IGXValList containing the GXVAL object whose BLOB value to retrieve.

**key** Key name of the GXVAL object whose BLOB value to retrieve.

**pSize** Pointer to the DWORD variable allocated by the client to store the size of the BLOB.

**Usage** Use the GXGetValListBLOB() function when the type of a GXVAL object is a BLOB, but its value is not known and needed for subsequent operations. If the GXVAL object is of DWORD size or of type integer, use GXGetValList(). If it is of type string or GUID, use GXGetValListString() and GXGetValListGUID(), respectively.

**Tip** GXGetValListBLOB() returns a pointer to the BLOB, therefore, the value can change if subsequent operations change the value in the GXVAL object.

**Return Value** A pointer to the BLOB, or NULL if an error occurs.

**Include File** gxval.h

**Related Topics** GXSetValListBLOB()

GXVAL struct

IGXValList interface

## GXGetValListGUID()

Returns the GUID object in an IGXValList.

**Syntax** GUID GXGetValListGUID(  
    IGXValList \*list,  
    LPSTR key);

**list** IGXValList containing the GXVAL object whose GUID value to retrieve.

**key** Key name of the GXVAL object whose GUID value to retrieve.

**Usage** Use the GXGetValListGUID() function when the type of the GXVAL object is a GUID, but its value is not known and needed for subsequent operations. If the GXVAL object is of DWORD size or of type integer, use GXGetValList(). If it is of type string or BLOB, use GXGetValListString() and GXGetValListBLOB(), respectively.

**Return Value** A copy of the GUID.

**Include File** gxval.h

**Related Topics** GXSetValListGUID()

GXVAL struct

IGXValList interface

## GXGetValListString()

Retrieves the string value of a GXVAL object in an IGXValList.

**Syntax** LPSTR GXGetValListString(  
     IGXValList \*list,  
     LPSTR key);

**list** IGXValList containing the GXVAL object whose string value to retrieve.

**key** Key name of the GXVAL object whose string value to retrieve.

**Usage** Call the GXGetValListString() function to get the value of a GXVAL object of type string. If the GXVAL object is of DWORD size or of type integer, use GXGetValList(). If it is of type BLOB or GUID, use GXGetValListBLOB() and GXGetValListGUID(), respectively.

**Tip** GXGetValListString() returns a pointer to the string, therefore, the value can change if subsequent operations change the value in the GXVAL object.

**Return Value** A pointer to the string.

**Include File** gxval.h

**Example**

```
OBCustomerFormAppLogic::OBCustomerFormAppLogic():
    m_pSsn(NULL),
    m_pUserName(NULL),
    m_pPrefix(NULL),
    m_pFirstName(NULL),
    m_pMiddleName(NULL),
    m_pLastName(NULL)
// Method that gets values from the
// AppLogic's input IGXValList
STDMETHODIMP_(BOOL)
OBCustomerFormAppLogic::GetFormInputs()
{
    m_pSsn=GXGetValListString(m_pValIn, "ssn");
    m_pUserName=GXGetValListString(m_pValIn, "userName");
    m_pPrefix=GXGetValListString(m_pValIn, "prefix");
    m_pFirstName=GXGetValListString(m_pValIn, "firstName");
    m_pMiddleName=GXGetValListString(m_pValIn, "middleName");
    m_pLastName=GXGetValListString(m_pValIn, "lastName");
```

**Related Topics** GXSetValListString()

GXVAL struct

IGXValList interface

## GXGUIDToString()

Converts a GUID to a string.

**Syntax** HRESULT GXGUIDToString(  
     REFIID idClass,  
     LPSTR szClass);

**idClass** The GUID to convert to string.

**szclass** The client-allocated string buffer that will be filled with the string representation of the GUID.

**Usage** Use GXGUIDToString() if you need the GUID string for debugging purposes, or if you need to pass a GUID as a string. The NewRequest() method, for example, takes a GUID string as a parameter.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the function succeeds.

**Include File** gxutil.h

**Related Topics** GXStringToGUID()

## GXInitCriticalSection()

Initializes a critical section object.

**Syntax** void GXInitCriticalSection(GXCRIT\_SECTION \*x);

**x** Pointer to a previously declared GXCRIT\_SECTION variable that represents the critical section to initialize.

**Usage** Use GXInitCriticalSection() to allocate a critical section object to be used in subsequent operations to synchronize thread access to a particular process.

- Rules**
- The AppLogic must declare the critical section variable as type GXCRIT\_SECTION before initializing it.
  - The specified critical section must be initialized by a call to GXInitCriticalSection() before subsequent critical section operations.
  - The specified critical section must be destroyed by a subsequent call to GXDeleteCriticalSection().

- Tips**
- Consider calling `GXInitCriticalSection()` in a constructor method.
  - Destroy the critical section object using `GXDeleteCriticalSection()` as soon as the `AppLogic` no longer needs it.
  - In multithreaded programming, use critical sections in your `AppLogic` to ensure synchronization when multiple threads can manipulate the same object.

**Return Value** void

**Include File** `gxplat.h`

**Related Topics** “Using Critical Sections” in Chapter 3, “Application Development Techniques,” in *Programmer’s Guide*.

## GXLeaveCriticalSection()

Releases ownership of a critical section object.

**Syntax** `void GXLeaveCriticalSection(GXCRIT_SECTION *x);`

**x** Pointer to the `GXCRIT_SECTION` variable that represents the critical section to leave.

**Usage** Use `GXLeaveCriticalSection()` to release exclusive thread access to shared resources after completing operations on the protected resource. Releasing ownership allows other threads to acquire the critical section.

- Rules**
- The specified critical section must be initialized by a previous call to `GXInitCriticalSection()`.
  - The thread must already have ownership of the specified critical section by a previous call to `GXEnterCriticalSection()`.

- Tips**
- The `AppLogic` can call `GXEnterCriticalSection()` and `GXLeaveCriticalSection()` repeatedly before calling `GXDeleteCriticalSection()`.
  - Leave a critical section as soon as the `AppLogic` no longer needs it so that other threads may acquire it.

- In multithreaded programming, use critical sections in your AppLogic to ensure synchronization when multiple threads can manipulate the same object.

**Include File** gxplat.h

**Related Topics** “Using Critical Sections” in Chapter 3, “Application Development Techniques,” in *Programmer’s Guide*.

## GXProcessOutput()

Processes the results in an AppLogic’s output IGXValList (vOut) and returns an IGXTile object from which the caller can extract data.

**Syntax**

```
HRESULT ProcessOutput(
    IGXContext *pContext,
    DWORD flags,
    IGXValList *pValList
    IGXTile **ppTile);
```

**context** The IGXContext object, which gives the AppLogic access to Netscape Application Server services. Pass in the AppLogic’s context member variable.

**flags** Specify 0. Internal use only.

**pValList** The output IGXValList that contains the results returned by a called AppLogic.

**ppTile** Pointer to the returned IGXTile object.

**Usage** Use GXProcessOutput() to process non-HTML results that are returned in the following situation:

1. A client (AppLogic or OCL client application) calls an AppLogic with NewRequest().
2. Through NewRequest(), the client passes input and output IGXValLists to the called AppLogic. The client specifies the value "ocl" for the gx\_client\_type key in the input IGXValList.
3. The called AppLogic processes the request and returns results in the output IGXValList.

`GXProcessOutput()` returns the processed data as an `IGXTile` object. Data in the `IGXTile` object is organized like a hierarchical result set. The client can use methods in the `IGXTile` interface to loop through the result set and retrieve values.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the function succeeds.

**Include File** `gxcipm.h`

**Example**

```
// Call an AppLogic
hr = NewRequest(guid, vIn, vOut, 0);
if (hr != NOERROR)
{
    printf("Failed to invoke NewRequest()\n");
    exit(-1);
}
// Get the root tile from the output vallist
mainTile = NULL;

hr = GXProcessOutput(NULL, 0, vOut, &mainTile);

if (hr == NOERROR)
{
    //Iterate over all categories and print their names
    ptile = NULL;
    if ((hr = mainTile->GetTileChild("CATEGORIES", &ptile)) != NOERROR)
    {
        printf("Unable to get tile child, hr = %d\n", hr);
    }
    while (ptile && hr == NOERROR)
    {
        hr = ptile->GetTileValue("CATEGORIES.Name", sval, sizeof(sval));
        if (hr == NOERROR)
        {
            for (int i=0; i < (depth * 2); i++)
                printf(" ");
            printf("Category %s\n", sval);
        }

        hr = ptile->MoveTileNextRecord();
    }
    if (ptile)
        ptile->Release();
}
```

**Related Topics** NewRequest() in the GXAppLogic class  
IGXTile interface

## GXSetValList()

Specifies the data type and DWORD-sized value of a GXVAL object in an IGXValList.

**Syntax** HRESULT GXSetValList(  
    IGXValList \*list,  
    LPSTR key,  
    GXVALTYPE type,  
    DWORD val);

**list** IGXValList that contains the GXVAL object whose data type and value to set. If the GXVAL object does not already exist, GXSetValList() creates it.

**key** Key name of the GXVAL object whose data type and value to set. If the GXVAL object does not already exist, GXSetValList() creates it.

**type** The data type to assign to the GXVAL object.

**val** The DWORD sized value to assign to the GXVAL object.

**Usage** Call the GXSetValList() function to assign a DWORD sized value to a GXVAL object in an IGXValList. If the GXVAL object does not already exist, GXSetValList() creates it, then copies it to the IGXValList.

If you want to assign a string, a BLOB, or a GUID value to a GXVAL object, use GXSetValListString(), GXSetValListBLOB(), and GXSetValListGUID(), respectively.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the function succeeds.

**Include File** gxval.h

**Related Topics** GXGetValList()

GXVAL struct

IGXValList interface

# GXSetValListBLOB()

Specifies a BLOB value for a GXVAL object in an IGXValList.

**Syntax** HRESULT GXSetValListBLOB(  
 IGXValList \*list,  
 LPSTR key,  
 LPBYTE val,  
 DWORD size);

**list** IGXValList that contains the GXVAL object whose BLOB value to set. If the GXVAL object does not already exist, GXSetValListBLOB() creates it.

**key** Key name of the GXVAL object whose BLOB value to set. If the GXVAL object does not already exist, GXSetValListBLOB() creates it.

**val** The BLOB value to assign to the GXVAL object.

**size** The size of the BLOB.

**Usage** Call the GXSetValListBLOB() function to assign a BLOB value to a GXVAL object in an IGXValList. If the GXVAL object does not already exist, GXSetValListBLOB() creates it, then copies it to the IGXValList.

If you want to assign a DWORD-sized value, a string, or a GUID value to a GXVAL object, use GXSetValList(), GXSetValListString(), and GXSetValListGUID(), respectively.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the function succeeds.

**Include File** gxval.h

**Related Topics** GXGetValListBLOB()

GXVAL struct

IGXValList interface

# GXSetValListGUID()

Specifies a GUID value for a GXVAL object in an IGXValList.

**Syntax** HRESULT GXSetValListGUID(  
 IGXValList \*list,  
 LPSTR key,  
 LPBYTE val,  
 DWORD size);

## GXSetValListString()

```
IGXValList *list,  
LPSTR key,  
GUID *);
```

**list** IGXValList that contains the GXVAL object whose GUID value to set. If the GXVAL object does not already exist, GXSetValListGUID() creates it.

**key** Key name of the GXVAL object whose GUID value to set. If the GXVAL object does not already exist, GXSetValListGUID() creates it.

**GUID \*** Pointer to the GUID to copy to the specified GXVAL object.

**Usage** Call the GXSetValListGUID() function to assign a GUID value to a GXVAL object in an IGXValList. If the GXVAL object does not already exist, GXSetValListGUID() creates it, then copies it to the IGXValList.

If you want to assign a DWORD-sized value, a string, or a BLOB value to a GXVAL object, use GXSetValList(), GXSetValListString(), and GXSetValListBLOB(), respectively.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the function succeeds.

**Include File** gxval.h

**Related Topics** GXGetValListGUID()

GXVAL struct

IGXValList interface

## GXSetValListString()

Specifies a string value for a GXVAL object in an IGXValList.

**Syntax** HRESULT GXSetValListString(  
IGXValList \*list,  
LPSTR key,  
LPSTR val);

**list** IGXValList that contains the GXVAL object whose string value to set. If the GXVAL object does not already exist, GXSetValListString() creates it.

**key** Key name of the GXVAL object whose string value to set. If the GXVAL object does not already exist, GXSetValListString() creates it.

**val** The string value to set in the specified GXVAL object.

**Usage** Call the `GXSetValListString()` function to assign a string value to a GXVAL object in an `IGXValList`. If the GXVAL object does not already exist, `GXSetValListString()` creates it, then copies it to the `IGXValList`.

If you want to assign a DWORD-sized value, a BLOB, or a GUID value to a GXVAL, use `GXSetValList()`, `GXSetValListBLOB()`, and `GXSetValListGUID()`, respectively.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the function succeeds.

**Include File** `gxval.h`

**Related Topics** `GXGetValListString()`

`GXVAL` struct

`IGXValList` interface

## GXStringToGUID()

Converts a string to a GUID.

**Syntax** `HRESULT GXStringToGUID(  
LPSTR szClass,  
GUID *idclass);`

**szClass** The string to parse as a GUID. It must be in GUID format:

```
{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}
```

**idclass** Pointer to the client-allocated GUID in which to return the parsed GUID value.

**Usage** Use `GXStringToGUID()` if you do not want to work directly with a GUID struct. You might find it easier to set the value of a GUID by passing it a string than by assigning values to the 128-bit members in a GUID structure.

**Return Value** `HRESULT`, which is set to `GXE_SUCCESS` if the function succeeds.

**Include File** `gxutil.h`

**Related Topics** `GXGUIDToString()`

## GXSYNC\_DEC()

Decrements a variable under the protection of a spin lock.

**Syntax** `LONG GXSYNC_DEC(  
    LONG *pv,  
    GXSYNCVAR *pLock)`

**pv** Pointer to the LONG variable to decrement.

**pLock** Pointer to the spin lock to use while decrementing.

**Return Value** The decremented LONG value.

**Usage** Use GXSYNC\_DEC() to decrement a variable, by one (1), using a spin lock to ensure synchronized access to it.

GXSYNC\_DEC() calls GXSYNC\_LOCK() automatically before decrementing the variable and calls GXSYNC\_UNLOCK() automatically after decrementing the variable.

Alternatively, use GXSYNC\_INC() to increment a variable using a spin lock.

**Rule** The specified spin lock must be initialized by a previous call to GXSYNC\_INIT().

**Include File** `gxutil.h`

**Related Topics** “Using Spin Locks” in Chapter 3, “Application Development Techniques,” in *Programmer’s Guide*.

## GXSYNC\_DESTROY()

Destroys a spin lock.

**Syntax** `void GXSYNC_DESTROY(  
    GXSYNCVAR *pSyncVar)`

**pSyncVar** Pointer to the previously initialized spin lock to destroy.

**Return Value** `void`

- Usage** Use `GXSYNC_DESTROY()` to destroy a spin lock that AppLogic no longer needs. Calling `GXSYNC_DESTROY()` releases the system resources allocated for the spin lock.
- Rules**
- The specified spin lock must be initialized by a previous call to `GXSYNC_INIT()`.
  - Subsequent calls to the spin lock are invalid. To use the spin lock again, the AppLogic must subsequently initialize the spin lock using `GXSYNC_INIT()`.
- Tips**
- Destroy a spin lock using `GXSYNC_DESTROY()` as soon as the AppLogic no longer needs it.
  - Use spin locks to ensure synchronous access for only short processes consisting of just one or several brief operations. Extensive or careless use of spin locks (such as for longer processes like memory allocation or ODBC calls) can reduce AppLogic performance. For longer processes, use alternative means instead, such as the lock manager or semaphore locks.

**Include File** `gxutil.h`

**Example**

```
// declare class that uses spin locks
class myClass {
    GXSYNCVAR m_sync;
    class::MyClass(){ // constructor
        GXSYNC_INIT(&m_sync);} // initialize sync var
    class::~MyClass(){ // destructor
        GXSYNC_DESTROY(&m_sync);} // destroy sync var
};
```

**Related Topics** “Using Spin Locks” in Chapter 3, “Application Development Techniques,” in *Programmer’s Guide*.

## GXSYNC\_INC()

Increments a variable under the protection of a spin lock.

**Syntax** `LONG GXSYNC_INC(`  
`LONG *pv,`  
`GXSYNCVAR *pLock)`

**pv** Pointer to the LONG variable to increment.

**pLock** Pointer to the spin lock to use while incrementing.

**Return Value** The incremented LONG value.

**Usage** Use GXSYNC\_INC() to increment a variable, by one (1), using a spin lock to ensure synchronized access to it.

GXSYNC\_INC() calls GXSYNC\_LOCK() automatically before incrementing the variable and calls GXSYNC\_UNLOCK() automatically after incrementing the variable.

Alternatively, use GXSYNC\_DEC() to decrement a variable using a spin lock.

**Rule** The specified spin lock must be initialized by a previous call to GXSYNC\_INIT().

**Include File** gxutil.h

**Related Topics** “Using Spin Locks” in Chapter 3, “Application Development Techniques,” in *Programmer’s Guide*.

## GXSYNC\_INIT()

Initializes a spin lock.

**Syntax** void GXSYNC\_INIT(  
GXSYNCPVAR \*pSyncVar)

**pSyncVar** Pointer to the GXSYNCPVAR synchronization variable, representing a spin lock, to initialize.

**Return Value** void

**Usage** Use GXSYNC\_INIT() to declare and allocate a synchronization variable of type GXSYNCPVAR to be used to synchronize access to shared resources, via a spin lock, in subsequent operations.

- Rules**
- The spin lock must be initialized by a call to GXSYNC\_INIT() prior to subsequent spin lock operations.
  - The specified spin lock must be destroyed by a subsequent call to GXSYNC\_DESTROY().

- Tips**
- Destroy a spin lock using `GXSYNC_DESTROY()` as soon as the `AppLogic` no longer needs it.
  - Use spin locks to ensure synchronous access for only short processes consisting of just one or several brief operations. Extensive or careless use of spin locks (such as for longer processes like memory allocation or ODBC calls) can reduce `AppLogic` performance. For longer processes, use alternative means instead, such as the lock manager or semaphore locks.

**Include File** `gxutil.h`

**Example**

```
// declare class that uses spin locks
class myClass {
    GXSYNCVAR m_sync;
    class::MyClass(){ // constructor
        GXSYNC_INIT(&m_sync);} // initialize sync var
    class::~MyClass(){ // destructor
        GXSYNC_DESTROY(&m_sync);} // destroy sync var
};
```

**Related Topics** “Using Spin Locks” in Chapter 3, “Application Development Techniques,” in *Programmer’s Guide*.

## GXSYNC\_LOCK()

Acquires a spin lock.

**Syntax** `void GXSYNC_LOCK(  
GXSYNCVAR *cs)`

**cs** Pointer to the spin lock to acquire.

**Return Value** `void`

**Usage** Use `GXSYNC_LOCK()` to acquire exclusive access to the shared resource(s) that the specified spin lock protects. While an `AppLogic` owns the spin lock, other clients cannot acquire it.

- Rules**
- The specified spin lock must be initialized by a previous call to `GXSYNC_INIT()`.
  - The specified spin lock must be released by a subsequent call to `GXSYNC_UNLOCK()`. Otherwise, a deadlock occurs.

- Tips**
- The AppLogic must not wait or go to sleep while it owns a spin lock.
  - Release a spin lock as soon as the AppLogic no longer needs it so that other clients may acquire it.
  - Use spin locks to ensure synchronous access for only short processes consisting of just one or several brief operations. Extensive or careless use of spin locks (such as for longer processes like memory allocation or ODBC calls) can reduce AppLogic performance. For longer processes, use alternative means instead, such as the lock manager or semaphore locks.

**Include File** gxutil.h

**Example**

```
// Use a spin lock for exclusive access to a variable
GXSYNC_LOCK(&SyncVar); // acquire the spin lock
m_ID1++;
if (m_ID1 == 0)
    m_ID2++;
GXSYNC_UNLOCK(&SyncVar); // release the spin lock
```

**Related Topics** “Using Spin Locks” in Chapter 3, “Application Development Techniques,” in *Programmer’s Guide*.

## GXSYNC\_UNLOCK()

Releases an acquired spin lock.

**Syntax** void GXSYNC\_UNLOCK(  
GXSYNCVAR \*cs)

**cs** Pointer to the spin lock to release.

**Return Value** void

**Usage** Use GXSYNC\_UNLOCK() to release a spin lock that was acquired in a preceding GXSYNC\_LOCK() call. Releasing the spin lock allows other clients to acquire it.

- Rules**
- The specified spin lock must be initialized by a previous call to GXSYNC\_INIT().
  - The specified spin lock must be acquired by a previous call to GXSYNC\_LOCK().

- Tips**
- Release a spin lock as soon as the AppLogic no longer needs it so that other clients may acquire it.
  - Use spin locks to ensure synchronous access for only short processes consisting of just one or several brief operations. Extensive or careless use of spin locks (such as for longer processes like memory allocation or ODBC calls) can reduce AppLogic performance. For longer processes, use alternative means instead, such as the lock manager or semaphore locks.

**Include File** gxutil.h

**Example**

```
// Use a spin lock for exclusive access to a variable
GXSYNC_LOCK(&SyncVar); // acquire the spin lock
m_ID1++;
if (m_ID1 == 0)
    m_ID2++;
GXSYNC_UNLOCK(&SyncVar); // release the spin lock
```

**Related Topics** “Using Spin Locks” in Chapter 3, “Application Development Techniques,” in *Programmer’s Guide*.

## GXVALClear()

Clears the contents of a GXVAL object and releases any secondary allocated memory that the GXVAL object may have been pointing to.

**Syntax** HRESULT GXVALClear(  
GXVAL \*pVal);

**pVal** Pointer to the GXVAL object to clear.

**Usage** When your AppLogic no longer requires GXVAL objects that you created with the GXVALCopy() function or the GetVal() method in the IGXValList interface, call the GXVALClear() function.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the function succeeds.

**Include File** gxval.h

**Related Topics** GXVAL struct

# GXVALCopy()

Copies a GXVAL object to another.

**Syntax** HRESULT GXVALCopy(  
     GXVAL \*pSrc,  
     GXVAL \*pDst);

**pVal** Pointer to the source GXVAL object to copy.

**pDst** Pointer to the destination GXVAL object to which the source GXVAL object is to be copied.

**Usage** Use GXVALCopy() to work with a copy of a GXVAL object. You must create the destination GXVAL object before calling GXVALCopy(). When your AppLogic no longer requires the copy of the GXVAL object, call GXVALClear() to release it.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the function succeeds.

**Include File** gxval.h

**Related Topics** GXVAL struct

# GXWaitForOrder()

Waits for asynchronous, flat database queries to be completed within a specified time frame.

**Syntax** HRESULT GXWaitForOrder(  
     IGXOrder \*\*pOrder,  
     ULONG nOrder,  
     ULONG \*pnOrder,  
     IGXObject \*pEventSrc,  
     ULONG nTimeout);

**pOrder** Pointer to an array of IGXOrder objects. Each element in the array corresponds to an asynchronous operation.

**nOrder** Number of IGXOrder objects in the array.

**pnOrder** Pointer to the variable that contains the returned index of the order that is finished, if any. If the returned pnOrder equals -1, and error occurred. Otherwise, pnOrder equals the index of the finished order (0 to n-1).

**pEventSrc** Pointer to an IGXObject variable that provides the blocking services, such as the IGXContext object (m\_pContext) in the GXAppLogic class.

**nTimeout** Maximum number of seconds to wait before expiring, if none of the asynchronous queries is finished.

**Usage** Use GXWaitForOrder() to wait for one or more asynchronous operations, such as asynchronous database queries, to return the completed results from the database server on which they were submitted. Asynchronous queries that were started using ExecuteQuery() in the IGXDataConn interface may return results sets that are not yet finished. An AppLogic module must wait for the result set to be finished before using the result set.

When running asynchronous queries, the AppLogic needs to determine when a particular query has finished processing on the database server. The GXWaitForOrder() function will block efficiently until either one of the following conditions occurs:

- The status of the IGXOrder object associated with one of the queries changes to GX\_STATE\_DONE, a macro-based constant defined in gxioorder.h.
- The specified timeout limit has been exceeded.

**Rules** • To run an asynchronous query, the AppLogic must specify GX\_DA\_EXEC\_ASYNC as the dwFlags parameter of ExecuteQuery() in the IGXDataConn interface.

- To retrieve an IGXOrder object, use GetOrder() in the IGXResultSet interface on an unfinished result set.
- When the AppLogic is finished using the IGXOrder object, call the Release() method to release the interface instance.

**Tips** • GXWaitForOrder() replaces the GXOrderWait() and GXOrderWaitTimeout() functions in the previous release.

- The GXWaitForOrder() function will return as soon as any error occurs or an asynchronous operation (IGXOrder) in the input array is complete or a timeout happens. Remove any completed IGXOrder objects from the array

before calling GXWaitForOrder() again on the same array. Also, check the IGXOrder using GetState() in the IGXOrder interface to determine whether the asynchronous query completed successfully or returned an error.

**Return Value** HRESULT, which is set to GXE\_SUCCESS if the method succeeds.

**Include File** gxorder.h

**Example**

```

IGXOrder *pOrder;
ULONG     nOrder;
HRESULT hr, ReqResult;

if (NewRequestAsync(asyncGUIDStr, m_pValIn,
                    m_pValOut, 0, &pOrder) == GXE_SUCCESS)
{
    Log("Successfully invoked async AppLogic\n");

    // wait for async applogic to finish (max 100 seconds)
    hr = GXWaitForOrder(&pOrder, 1, &nOrder, m_pContext, 100);
    if (hr != NOERROR)
    {
        return Result("Error in executing async request:
                      order wait returned an error");
    }
    else
    {
        pOrder->GetState(NULL, &ReqResult, NULL);
        if (ReqResult != NOERROR)
            return Result("Error in executing async request");
    }
}
else
{
    Log("Failed to invoke async AppLogic\n");
}

```

**Related Topics** ExecuteQuery() in the IGXDataConn interface

GetOrder() in the IGXResultSet interface

IGXOrder interface

# C++ Macros and Structures

This chapter discusses the macros and structures in the Netscape Application Server Foundation Class Library.

## Macros

- GXDLM\_DECLARE
- GXDLM\_IMPLEMENT
- GXDLM\_IMPLEMENT\_BEGIN
- GXDLM\_IMPLEMENT\_END
- GXGUID\_EQUAL

## Structures

- GUID struct
- GXDATETIME struct
- GXVAL struct

## GXDLM\_DECLARE

Associates a C++ class in a dynamically loadable, shared library module (DLM) with a GUID.

**Syntax** `GXDLM_DECLARE(class_name, clsid)`

**class\_name** The C++ class to associate with a GUID. This class can be any Component Object Model (COM) class, such as a GXAgent-derived class.

**clsid** The GUID to associate with the specified class. You should already have defined the GUID object from the GUID struct.

**Usage** Use the `GXDLM_DECLARE` macro in conjunction with the `GXDLM_IMPLEMENT_BEGIN`, `GXDLM_IMPLEMENT`, and `GXDLM_IMPLEMENT_END` macros to provide a DLM with Netscape-specific exported C functions. Netscape Application Server expects to find these exported functions when it loads the DLM at runtime. The exported functions are required to fully initialize the DLM and to create C++ instances from it.

- Rules**
- Call `GXDLM_DECLARE` once for each AppLogic module or exported C++ class in a DLM.
  - Call `GXDLM_DECLARE` in a header file.

**Include File** `gxdlm.h`

## GXDLM\_IMPLEMENT

Establishes to the Netscape Application Server the entry point in a dynamically loadable, shared library module (DLM) for one exported C++ class.

**Syntax** `GXDLM_IMPLEMENT(class_name, clsid)`

**class\_name** The C++ class to establish.

**clsid** The GUID associated with the class.

**Usage** Use the `GXDLM_IMPLEMENT` macro in conjunction with the `GXDLM_DECLARE`, `GXDLM_IMPLEMENT_BEGIN`, and `GXDLM_IMPLEMENT_END` macros to provide a DLM with Netscape-specific

exported C functions. Netscape Application Server expects to find these exported functions when it loads the DLM at runtime. The exported functions are required to fully initialize the DLM and to create C++ instances from it.

- Rules**
- Call GXDLM\_IMPLEMENT once for each exported C++ class in an DLM that you want the Netscape Application Server to access and create dynamically.
  - GXDLM\_IMPLEMENT calls must be made between the GXDLM\_IMPLEMENT\_BEGIN and GXDLM\_IMPLEMENT\_END calls.
  - There can be only one GXDLM\_IMPLEMENT\_BEGIN and GXDLM\_IMPLEMENT\_END block in a DLM.
  - Call GXDLM\_IMPLEMENT in a C++ source (.cpp, non-header) file.

**Include File** gxdlm.h

## GXDLM\_IMPLEMENT\_BEGIN

Establishes to the Netscape Application Server the entry point to the dynamically loadable, shared library module (DLM).

**Syntax** GXDLM\_IMPLEMENT\_BEGIN ( )

**Usage** Use the GXDLM\_IMPLEMENT\_BEGIN macro in conjunction with the GXDLM\_DECLARE, GXDLM\_IMPLEMENT, and GXDLM\_IMPLEMENT\_END macros to provide a DLM with Netscape-specific exported C functions. Netscape Application Server expects to find these exported functions when it loads the DLM at runtime. The exported functions are required to fully initialize the DLM and to create C++ instances from it.

- Rules**
- Call GXDLM\_IMPLEMENT\_BEGIN before GXDLM\_IMPLEMENT.
  - There can be only one GXDLM\_IMPLEMENT\_BEGIN and GXDLM\_IMPLEMENT\_END block in a DLM.
  - Call GXDLM\_IMPLEMENT\_BEGIN in a C++ source (.cpp, non-header) file.

**Include File** gxdlm.h

## GXDLM\_IMPLEMENT\_END

Indicates that all exported C++ classes in the dynamically loadable, shared library module (DLM) have been established with Netscape Application Server.

**Syntax** `GXDLM_IMPLEMENT_END( )`

**Usage** Use the `GXDLM_IMPLEMENT_END` macro in conjunction with the `GXDLM_DECLARE`, `GXDLM_IMPLEMENT_BEGIN`, and `GXDLM_IMPLEMENT` macros to provide a DLM with Netscape-specific exported C functions. Netscape Application Server expects to find these exported functions when it loads the DLM at runtime. The exported functions are required to fully initialize the DLM and to create C++ instances from it.

- Rules**
- Call `GXDLM_IMPLEMENT_END` after `GXDLM_IMPLEMENT`.
  - There can be only one `GXDLM_IMPLEMENT_BEGIN` and `GXDLM_IMPLEMENT_END` block in a DLM.
  - Call `GXDLM_IMPLEMENT_END` in a C++ source (.cpp, non-header) file.

**Include File** `gxdlm.h`

## GXGUID\_EQUAL

Determines whether two GUIDs are equivalent.

**Syntax** `GXGUID_EQUAL(guid1, guid2)`

**guid1** The first GUID to use in the comparison.

**guid2** The second GUID to use in the comparison.

### Usage

Use `GXGUID_EQUAL` to compare if two AppLogic modules are the same. This information is necessary when implementing the `QueryInterface( )` method.

**Return Value** True if the GUIDs are the same.

**Include File** `gx.util.h`

See Also `QueryInterface()` in the `IGXObject` interface

## GUID struct

A GUID structure holds a globally unique identifier (GUID), which identifies AppLogic modules and Netscape Application Server services. This identifier is a 128-bit value.

**Syntax**

```
typedef struct _GUID {
    unsigned long Data1;
    unsigned short Data2;
    unsigned short Data3;
    unsigned char Data4[9];
} GUID;
```

**Data1** Specifies the first eight hexadecimal digits of the GUID.

**Data2** Specifies the first group of four hexadecimal digits of the GUID.

**Data3** Specifies the second group of four hexadecimal digits of the GUID.

**Data4** Specifies an array of eight elements that contains the third and final group of eight hexadecimal digits of the GUID in elements 0 and 1, and the final 12 hexadecimal digits of the GUID in elements 2 through 7.

## GXDATETIME struct

A `GXDATETIME` structure contains date and time data.

**Syntax**

```
typedef struct tagGXDATETIME {
    short year;
    unsigned short month;
    unsigned short day;
    unsigned short hour;
    unsigned short minute;
    unsigned short second;
    unsigned short fraction;
    unsigned short timezone;
} GXDATETIME;
```

**year** Year. Range (A.D.) : 1 to 32767. Range (B.C.): -32768 to -1.

**month** Number of the month. Range: 1 to 12.

**day** Number of the day of the month. Range: 1 to 31.

**hour** Hours since midnight. Range: 0 to 23.

**minute** Minutes after the hour. Range: 0 to 59.

**second** Seconds after the minute. Range: 0 to 59.

**fraction** Milliseconds after the second.

**timezone** Time zone information.

**Include File** gxitypes.h

## GXVAL struct

A GXVAL structure represents a single value of a particular data type. Parameters that are passed to an AppLogic, or results that are retrieved from an AppLogic, are contained in an IGXVallist object that contains one or more GXVAL objects.

```
Syntax typedef struct tagGXVAL {
    GXVALTYPE vt;
    WORD wReserved1;
    WORD wReserved2;
    WORD wReserved3;
    union
    {
        unsigned char cVal;
        short iVal;
        long lVal;
        floatfltVal;
        double dblVal;
        SCODE codeVal;
        unsigned short boolVal;
        unsigned char bVal;
        unsigned short uiVal;
        IUnknown *punkVal;
        LPSTR pstrVal;
        void *pvoidVal;
    } u;
```

```
} GXVAL;
```

**vt** The data type of the GXVAL object's value. The following table lists the enum values you can use:

| Enum value   | Type                               |
|--------------|------------------------------------|
| GXVT_I2      | 2-byte signed integer              |
| GXVT_I4      | 4-byte signed integer              |
| GXVT_R4      | 4-byte real number                 |
| GXVT_R8      | 8-byte real number                 |
| GXVT_ERROR   | 4-byte error code for internal use |
| GXVT_BOOL    | BOOL True or False                 |
| GXVT_UNKNOWN | IUnknown FAR pointer               |
| GXVT_I1      | 1-byte signed char                 |
| GXVT_UI1     | 1-byte unsigned char               |
| GXVT_UI2     | 2-byte unsigned short              |
| GXVT_UI4     | 4-byte unsigned long               |
| GXVT_I8      | 64-bit signed integer              |
| GXVT_UI8     | 64-bit unsigned integer            |
| GXVT_LPSTR   | null terminated string             |
| GXVT_CLSID   | GUID                               |
| GXVT_BLOB    | Large binary object                |

**wReserved1, wReserved2, wReserved3** Reserved.

**cVal** A 1-byte signed integer number, byte, char, or ASCII character.

**iVal** A 2-byte signed integer number, or short.

**lVal** A 4-byte signed integer number, or int.

**fltVal** A 4-byte real number, or float.

**dblVal** An 8-byte real number, or double.

**codeVal** A 4-byte error code. Internal use only.

**boolVal** True=any non-zero number. False=0.

**bVal** A 1-byte unsigned char.

**uiVal** A 2-byte unsigned integer number, or short.

**ulVal** A 4-byte unsigned integer number, or long.

**punkVal** An IUnknown FAR pointer.

**pstrVal** A generic char pointer.

**pvoidVal** A generic void pointer.

**Include File** gxival.h

**Related Topics** IGXValList interface

GXVALClear( )

GXVALCopy( )

# A

## Return Codes

Many methods and functions in the Netscape Server Foundation Class Library return the HRESULT type as an error code. The following table lists the HRESULT types defined in `gxgenericerr.h`:

| HRESULT               | Value      |
|-----------------------|------------|
| GXE_SUCCESS           | 0          |
| GXE_ERROR             | 0x80240001 |
| GXE_INVALID_ARG       | 0x80240002 |
| GXE_INVALID_INTERFACE | 0x80240003 |
| GXE_NOT_SUPPORTED     | 0x80240004 |
| GXE_EOF               | 0x80240005 |
| GXE_READ_FAILED       | 0x80240006 |
| GXE_WRITE_FAILED      | 0x80240007 |
| GXE_ALLOC_FAILED      | 0x80240008 |
| GXE_INVALID_NAME      | 0x80240009 |
| GXE_INVALID_EXPR      | 0x8024000a |
| GXE_INVALID_INDEX     | 0x8024000b |
| GXE_TOO_SMALL         | 0x8024000c |
| GXE_FAIL              | 0x8024000d |

| HRESULT              | Value      |
|----------------------|------------|
| GXE_NOINTERFACE      | 0x8024000e |
| GXE_MEM_ALLOC_FAILED | 0x8024000f |

# Index

## A

AddAction() 129  
AddConn() 184  
AddQuery() 194  
AddRef() 230  
AddRow() 316  
Alloc() 135  
AllocRow() 318  
asynchronous operations 400

## B

Begin() 353  
Binary Large Objects (BLOBs) 383, 391

## C

ChangeMode() 214  
classes  
    GXAppLogic 29  
    GXSession2 89  
    GXTemplateDataBasic 91  
    GXTemplateMapBasic 97  
    overview 27  
Close() 140, 220, 259  
CloseConn() 166  
Commit() 354  
constants  
    GX\_DA\_TYPE\_BINARY 162  
    GX\_DA\_TYPE\_DATE 162  
    GX\_DA\_TYPE\_DATETIME 162  
    GX\_DA\_TYPE\_DOUBLE 162  
    GX\_DA\_TYPE\_ERROR 162  
    GX\_DA\_TYPE\_LONG 162  
    GX\_DA\_TYPE\_STRING 162

    GX\_DA\_TYPE\_TIME 162  
    GX\_STATE\_ACTIVE 266  
    GX\_STATE\_CANCEL 266  
    GX\_STATE\_DONE 266  
    GX\_STATE\_UNKNOWN 266  
    GXACL\_ALLOWED 56  
    GXACL\_DONTKNOW 56  
    GXACL\_NOTALLOWED 56  
    GXORDER\_STATE\_ACTIVE 234  
    GXORDER\_STATE\_CANCEL 234  
    GXORDER\_STATE\_DONE 234  
    GXORDER\_STATE\_UNKNOWN 234

Count() 202, 359  
CreateDataConn() 32  
CreateDataConnSet() 35  
CreateEvent() 120  
CreateHierQuery() 36  
CreateMailbox() 37  
CreateQuery() 39  
CreateSequence() 283  
CreateSession() 40  
CreateStateChild() 304  
CreateTrans() 42  
CreateTrigger() 167  
critical sections 380, 381, 386, 387

## D

dates 381, 407  
DeleteActions() 131  
DeleteCache() 44  
DeleteEvent() 106, 120  
DeleteRow() 320  
DeleteStateChild() 306

DelQuery() 196  
DestroySession() 45  
DisableEvent() 107, 121  
DisableTrigger() 169  
Drop() 280  
DropTrigger() 169

## E

EnableEvent() 107, 121  
EnableTrigger() 170  
EnumActions() 131  
EnumColumnReset() 259, 321  
EnumColumns() 259, 322  
EnumCount() 185  
EnumEvents() 107, 122  
EnumNext() 186  
EnumReset() 188  
EvalOutput() 45  
EvalTemplate() 48  
Execute() 51, 140, 196, 238  
ExecuteMultipleRS() 143  
ExecuteQuery() 170

## F

FetchNext() 260  
functions  
GXContextGetAppEventMgr() 374  
GXContextGetSessionCount() 374  
GXCreateStreamBuffer() 376  
GXCreateTemplateDataBasic() 377  
GXCreateTemplateMapBasic() 378  
GXCreateValList() 379  
GXDeleteCriticalSection() 380  
GXEnterCriticalSection() 381  
GXGetCurrentDate() 381  
GXGetValList() 382  
GXGetValListBLOB() 383  
GXGetValListGUID() 384

GXGetValListString() 384  
GXGUIDToString() 386  
GXInitCriticalSection() 386  
GXLeaveCriticalSection() 387  
GXProcessOutput() 388  
GXSetValList() 390  
GXSetValListBLOB() 391  
GXSetValListGUID() 391  
GXSetValListString() 392  
GXStringToGUID() 393  
GXSYNC\_DESTROY() 394  
GXSYNC\_INC() 394, 395  
GXSYNC\_INIT() 396  
GXSYNC\_LOCK() 397  
GXSYNC\_UNLOCK() 398  
GXVALClear() 399  
GXVALCopy() 400  
GXWaitForOrder() 400

## G

GenerateSessID() 298  
GenerateVariantID() 299  
Get() 342  
GetAddress() 136  
GetAppEvent() 52  
GetAttributes() 131  
GetColumn() 203, 261, 323  
GetColumnByOrd() 204, 262, 323  
GetColumnOrdinal() 263, 324  
GetConnInfo() 172  
GetConnProps() 173  
GetCurrent() 280  
GetDataConn() 325  
GetDriver() 174  
GetErrorCode() 189  
GetErrorCodeNum() 190  
GetErrorFacility() 191  
GetErrorMessage() 191  
GetEvent() 124

GetFields() 242  
 GetGroupBy() 243  
 GetHaving() 244  
 GetMoreResults() 146  
 GetName() 132, 153, 326  
 GetNext() 281  
 GetNextKey() 360  
 GetNullsAllowed() 155  
 GetNumColumns() 264, 326  
 GetOrder() 265  
 GetOrderBy() 245  
 GetParams() 147  
 GetPrecision() 156  
 GetResultSet() 148, 206  
 GetRowNumber() 206, 265  
 GetScale() 158  
 GetSequence() 286  
 GetSession() 53  
 GetSessionApp() 290  
 GetSessionData() 291  
 GetSessionFlags() 292  
 GetSessionID() 293  
 GetSessionTimeout() 294  
 GetSize() 137, 159  
 GetSQL() 246  
 GetState() 233  
 GetStateChild() 306  
 GetStateChildCount() 307  
 GetStateContents() 307  
 GetStateFlags() 308  
 GetStateName() 309  
 GetStateTimeout() 309  
 GetStateTreeRoot() 54  
 GetStatus() 266  
 GetStreamData() 313  
 GetTable() 160, 175  
 GetTables() 176, 247  
 GetTileChild() 347  
 GetTileValue() 348  
 GetType() 162  
 GetVaBLOB() 362  
 GetVal() 361  
 GetValBLOBSize() 362  
 GetValByRef() 363  
 GetValInt() 364  
 GetValString() 365  
 GetValue() 338  
 GetValueBinary() 267  
 GetValueBinaryPiece() 267  
 GetValueDateString() 207, 269  
 GetValueDouble() 208, 270  
 GetValueInt() 209, 271  
 GetValueSize() 271  
 GetValueString() 209, 272  
 GetValueText() 273  
 GetValueTextPiece() 274  
 GetWhere() 248  
 GroupAppend() 92  
 GUIDs 384, 386, 391, 393, 406, 407  
 GUID struct 407  
 GX\_DA\_TYPE\_BINARY 162  
 GX\_DA\_TYPE\_DATE 162  
 GX\_DA\_TYPE\_DATETIME 162  
 GX\_DA\_TYPE\_DOUBLE 162  
 GX\_DA\_TYPE\_ERROR 162  
 GX\_DA\_TYPE\_LONG 162  
 GX\_DA\_TYPE\_STRING 162  
 GX\_DA\_TYPE\_TIME 162  
 GX\_STATE\_CANCEL 266  
 GX\_STATE\_DONE 266  
 GX\_STATE\_UNKNOWN 266  
 GXACL\_ALLOWED 56

GXACL\_DONTKNOW 56  
 GXACL\_NOTALLOWED 56  
 GXAppLogic class  
   CreateDataConn() 32  
   CreateDataConnSet() 35  
   CreateHierQuery() 36  
   CreateMailbox() 37  
   CreateQuery() 39  
   CreateSession() 40  
   CreateTrans() 42  
   DeleteCache() 44  
   described 29  
   DestroySession() 45  
   EvalOutput() 45  
   EvalTemplate() 48  
   Execute() 51  
   GetAppEvent() 52  
   GetSession() 53  
   GetStateTreeRoot() 54  
   IsAuthorized() 55  
   IsCached() 57  
   LoadHierQuery() 58  
   LoadQuery() 61  
   Log() 64  
   LoginSession() 65  
   LogoutSession() 68  
   m\_pContext 29  
   m\_pStream 29  
   m\_pValIn 30  
   m\_pValOut 30  
   NewRequest() 69  
   NewRequestAsync() 71  
   RemoveAllCachedResults() 74  
   RemoveCachedResult() 75  
   Result() 76  
   SaveSession() 77  
   SetCacheCriteria() 78  
   SetSessionVisibility() 81  
   SetVariable() 82  
   SkipCache() 83  
   StreamResult() 84  
   StreamResultBinary() 85  
   StreamResultHeader() 86  
 GXContextGetAppEventMgr() 374  
 GXContextGetSessionCount() 374  
 GXCreateStreamBuffer() 376  
 GXCreateTemplateDataBasic() 377  
 GXCreateTemplateMapBasic() 378  
 GXCreateValList() 379  
 GXDATETIME struct 407  
 GXDeleteCriticalSection() 380  
 GXDLM\_DECLARE 404  
 GXDLM\_IMPLEMENT 404  
 GXDLM\_IMPLEMENT\_BEGIN 405  
 GXDLM\_IMPLEMENT\_END 406  
 GXEnterCriticalSection() 381  
 GXGetCurrentDateTime() 381  
 GXGetValListString() 384  
 GXGetValList() 382  
 GXGetValListBLOB() 383  
 GXGetValListGUID() 384  
 GXGUID\_EQUAL 406  
 GXGUIDToString() 386  
 GXInitCriticalSection() 386  
 GXLeaveCriticalSection() 387  
 GXORDER\_STATE\_ACTIVE 234  
 GXORDER\_STATE\_CANCEL 234  
 GXORDER\_STATE\_DONE 234  
 GXORDER\_STATE\_UNKNOWN 234  
 GXProcessOutput() 388  
 GXSession2 class  
   described 89  
 GXSetValList() 390  
 GXSetValListBLOB() 391  
 GXSetValListGUID() 391  
 GXSetValListString() 392  
 GXStringToGUID() 393  
 GXSYNC\_DESTROY() 394  
 GXSYNC\_INC() 394, 395  
 GXSYNC\_INIT() 396

GXSYNC\_LOCK() 397  
GXSYNC\_UNLOCK() 398  
GXTemplateDataBasic() 94  
GXTemplateDataBasic class  
  described 91  
  GroupAppend() 92  
  GXTemplateDataBasic() 94  
  RowAppend() 93  
GXTemplateMapBasic class  
  described 97  
  Put() 98  
GXVALClear() 399  
GXVALCopy() 400  
GXVAL struct 408  
GXWaitForOrder() 400

## I

IGX\_STATE\_ACTIVE 266  
IGXAppEvent interface  
  DeleteEvent() 106  
  described 103  
  DisableEvent() 107  
  EnableEvent() 107  
  EnumEvents() 107  
  QueryEvent() 110  
  RegisterEvent() 110  
  SetEvent() 113  
IGXAppEventManager interface  
  CreateEvent() 120  
  DeleteEvent() 120  
  described 117  
  DisableEvent() 121  
  EnableEvent() 121  
  EnumEvents() 122  
  GetEvent() 124  
  RegisterEvent() 125  
  TriggerEvent() 127  
IGXAppEventObj interface  
  AddAction() 129  
  DeleteActions() 131  
  described 129

  EnumActions() 131  
  GetAttributes() 131  
  GetName() 132  
  SetAttributes() 132  
IGXBuffer interface  
  Alloc() 135  
  described 135  
  GetAddress() 136  
  GetSize() 137  
  SetData() 137  
IGXCallableStmt interface  
  Close() 140  
  described 139  
  Execute() 140  
  ExecuteMultipleRS() 143  
  GetMoreResults() 146  
  GetParams() 147  
  GetResultSet() 148  
  SetParams() 149  
IGXColumn interface  
  described 151  
  GetName() 153  
  GetNullsAllowed() 155  
  GetPrecision() 156  
  GetScale() 158  
  GetSize() 159  
  GetTable() 160  
  GetType() 162  
IGXDataConn interface  
  CloseConn() 166  
  CreateTrigger() 167  
  described 165  
  DisableTrigger() 169  
  DropTrigger() 169  
  EnableTrigger() 170  
  ExecuteQuery() 170  
  GetConnInfo() 172  
  GetConnProps() 173  
  GetDriver() 174  
  GetTable() 175  
  GetTables() 176  
  PrepareCall() 177  
  PrepareQuery() 179  
  SetConnProps() 181

- IGXDataConnSet interface
  - AddConn() 184
  - described 183
- IGXEnumObject interface
  - described 185
  - EnumCount() 185
  - EnumNext() 186
  - EnumReset() 188
- IGXError interface
  - described 189
  - GetErrorCode() 189
  - GetErrorCodeNum() 190
  - GetErrorFacility() 191
  - GetErrorMessage() 191
- IGXHierQuery interface
  - AddQuery() 194
  - DelQuery() 196
  - described 193
  - Execute() 196
- IGXHierResultSet interface
  - Count() 202
  - described 199
  - GetColumn() 203
  - GetColumnByOrd() 204
  - GetResultSet() 206
  - GetRowNumber() 206
  - GetValueDateString() 207
  - GetValueDouble() 208
  - GetValueInt() 209
  - GetValueString() 209
  - MoveNext() 210
  - MoveTo() 211
- IGXLock interface
  - ChangeMode() 214
  - described 213
  - Lock() 215
  - Unlock() 216
- IGXMailbox interface
  - Close() 220
  - described 219
  - Open() 221
  - Retrieve() 222
  - RetrieveCount() 224
  - RetrieveReset() 225
- Send() 226
- IGXObject interface
  - AddRef() 230
  - described 229
  - QueryInterface() 231
  - Release() 232
- IGXOrder interface
  - described 233
  - GetState() 233
- IGXPreparedQuery interface
  - described 237
  - Execute() 238
  - SetParams() 239
- IGXQuery interface
  - described 241
  - GetFields() 242
  - GetGroupBy() 243
  - GetHaving() 244
  - GetOrderBy() 245
  - GetSQL() 246
  - GetTables() 247
  - GetWhere() 248
  - SetFields() 249
  - SetGroupBy() 250
  - SetHaving() 251
  - SetOrderBy() 252
  - SetSQL() 253
  - SetTables() 254
  - SetWhere() 255
- IGXResultSet interface
  - Close() 259
  - described 257
  - EnumColumns() 259
  - FetchNext() 260
  - GetColumn() 261
  - GetColumnByOrd() 262
  - GetColumnOrdinal() 263
  - GetNumColumns() 264
  - GetOrder() 265
  - GetRowNumber() 265
  - GetStatus() 266
  - GetValueBinary() 267
  - GetValueBinaryPiece() 267
  - GetValueDateString() 269

- GetValueDouble() 270
- GetValueInt() 271
- GetValueSize() 271
- GetValueString() 272
- GetValueText() 273
- GetValueTextPiece() 274
- MoveTo() 274
- RowCount() 275
- WasNull() 276
- IGXSequence interface
  - described 279
  - Drop() 280
  - GetCurrent() 280
  - GetNext() 281
- IGXSequenceMgr interface
  - CreateSequence() 283
  - described 283
  - GetSequence() 286
- IGXSession2 interface
  - described 289
  - GetSessionApp() 290
  - GetSessionData() 291
  - GetSessionFlags() 292
  - GetSessionID() 293
  - GetSessionTimeout() 294
  - SaveSession() 295
  - SetSessionData() 296
- IGXSessionIDGen interface
  - described 297
  - GenerateSessID() 298
  - GenerateVariantID() 299
  - MapToBaseID() 300
- IGXState2 interface
  - CreateStateChild() 304
  - DeleteStateChild() 306
  - described 303
  - GetStateChild() 306
  - GetStateChildCount() 307
  - GetStateContents() 307
  - GetStateFlags() 308
  - GetStateName() 309
  - GetStateTimeout() 309
  - SaveState() 310
  - SetStateContents() 310
- IGXStreamBuffer interface
  - described 313
  - GetStreamData() 313
- IGXTable interface
  - AddRow() 316
  - AllocRow() 318
  - DeleteRow() 320
  - described 315
  - EnumColumnReset() 321
  - EnumColumns() 322
  - GetColumn() 323
  - GetColumnByOrd() 323
  - GetColumnOrdinal() 324
  - GetDataConn() 325
  - GetName() 326
  - GetNumColumns() 326
  - SetValueBinary() 327
  - SetValueBinaryPiece() 328
  - SetValueDateString() 329
  - SetValueDouble() 330
  - SetValueInt() 330
  - SetValueString() 331
  - SetValueText() 332
  - SetValueTextPiece() 333
  - UpdateRow() 334
- IGXTemplateData interface
  - described 337
  - GetValue() 338
  - IsEmpty() 339
  - MoveNext() 339
  - SetHint() 340
- IGXTemplateMap interface
  - described 341
  - Get() 342
- IGXTile interface
  - described 345
  - GetTileChild() 347
  - GetTileValue() 348
  - MoveTileNextRecord() 349
  - MoveTileToRecord() 349
- IGXTrans interface
  - Begin() 353
  - Commit() 354
  - described 351

- Rollback() 355
- IGXTResultSet interface
  - EnumColumnReset() 259
- IGXValList interface
  - Count() 359
  - described 357
  - GetNextKey() 360
  - GetVal() 361
  - GetValBLOB() 362
  - GetValBLOBSize() 362
  - GetValByRef() 363
  - GetValInt() 364
  - GetValString() 365
  - RemoveVal() 366
  - ResetPosition() 366
  - SetVal() 367
  - SetValBLOB() 368
  - SetValByRef() 369
  - SetValInt() 370
  - SetValString() 371
- interfaces
  - IGXAppEvent 103
  - IGXAppEventMgr 117
  - IGXAppEventObj 129
  - IGXBuffer 135
  - IGXCallableStmt 139
  - IGXColumn 151
  - IGXDataConn 165
  - IGXDataConnSet 183
  - IGXEnumObject 185
  - IGXError 189
  - IGXHierQuery 193
  - IGXHierResultSet 199
  - IGXLock 213
  - IGXMailbox 219
  - IGXObject 229
  - IGXOrder 233
  - IGXPreparedQuery 237
  - IGXQuery 241
  - IGXResultSet 257
  - IGXSequence 279
  - IGXSequenceMgr 283
  - IGXSession2 289
  - IGXSessionIDGen 297
  - IGXState2 303

- IGXStreamBuffer 313
- IGXTable 315
- IGXTemplateData 337
- IGXTemplateMap 341
- IGXTile 345
- IGXTrans 351
- IGXValList 357
  - overview 101
- IsAuthorized() 55
- IsCached() 57
- IsEmpty() 339
- IUnknown interface 229

## L

- LoadHierQuery() 58
- LoadQuery() 61
- Lock() 215
- Log() 64
- LoginSession() 65
- LogoutSession() 68

## M

- m\_pContext 29
- m\_pStream 29
- m\_pValIn 30
- m\_pValOut 30
- macros
  - GXDLM\_DECLARE 404
  - GXDLM\_IMPLEMENT 404
  - GXDLM\_IMPLEMENT\_BEGIN 405
  - GXDLM\_IMPLEMENT\_END 406
  - GXGUID\_EQUAL 406
- MapToBaseID() 300
- MoveNext() 210, 339
- MoveTileNextRecord() 349
- MoveTileToRecord() 349
- MoveTo() 211, 274

## **N**

NewRequest() 69  
NewRequestAsync() 71

## **O**

Open() 221

## **P**

PrepareCall() 177  
PrepareQuery() 179  
Put() 98

## **Q**

QueryEvent() 110  
QueryInterface() 231

## **R**

RegisterEvent() 110, 125  
Release() 232  
RemoveAllCachedResults() 74  
RemoveCachedResult() 75  
RemoveVal() 366  
ResetPosition() 366  
Result() 76  
Retrieve() 222  
RetrieveCount() 224  
RetrieveReset() 225  
Rollback() 355  
RowAppend() 93  
RowCount() 275

## **S**

SaveSession() 77, 295  
SaveState() 310  
Send() 226

SetAttributes() 132  
SetCacheCriteria() 78  
SetConnProps() 181  
SetData() 137  
SetEvent() 113  
SetFields() 249  
SetGroupBy() 250  
SetHaving() 251  
SetHint() 340  
SetOrderBy() 252  
SetParams() 149  
    IGXPreparedQuery interface 239  
SetSessionData() 296  
SetSessionVisibility() 81  
SetSQL() 253  
SetStateContents() 310  
SetTables() 254  
SetVal() 367  
SetValBLOB() 368  
SetValByRef() 369  
SetValInt() 370  
SetValString() 371  
SetValueBinary() 327  
SetValueBinaryPiece() 328  
SetValueDateString() 329  
SetValueDouble() 330  
SetValueInt() 330  
SetValueString() 331  
SetValueText() 332  
SetValueTextPiece() 333  
SetWhere() 255  
SeVariable() 82  
SkipCache() 83  
spin locks 394, 395, 396, 397, 398  
StreamResult() 84  
StreamResultBinary() 85

StreamResultHeader() 86  
structures  
  GUID struct 407  
  GXDATETIME struct 407  
  GXVAL struct 408

## **T**

times 381, 407  
TriggerEvent() 127

## **U**

Unlock() 216  
UpdateRow() 334

## **V**

variables  
  m\_pContext 29  
  m\_pStream 29  
  m\_pValIn 30  
  m\_pValOut 30

## **W**

WasNull() 276