KnowNow LiveServer Developer's Guide

KnowNow, Inc.
997 East Arques Avenue
Sunnyvale, California 94085 USA

Legal Notices

Copyright© 2000–2008 KnowNow, Inc. All rights reserved. No part of this work covered by copyright herein may be reproduced in any form or by any means—graphic, electronic or mechanical—including photocopying, recording, taping, or storage in an information retrieval system, without prior written permission of the copyright owner.

KnowNow™ and the KnowNow logo are registered trademarks of KnowNow, Inc.. Other KnowNow product names used herein may be trademarks or registered trademarks of KnowNow, Inc..

All other brand, company, and product names referenced in this publication may be trademarks, registered trademarks, or service marks of their respective holders and are used here for informational purposes only.

Statement of Conditions

KNOWNOW, INC. PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL KNOWNOW BE LIABLE FOR ANY LOSS OF PROFITS, LOSS OF BUSINESS, LOSS OF USE OR DATA, INTERRUPTION OF BUSINESS, OR FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY KIND, EVEN IF KNOWNOW HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES ARISING FROM ANY DEFECT OR ERROR IN THIS PUBLICATION OR IN THE KNOWNOW SOFTWARE.

KnowNow, Inc. may revise this publication from time to time without notice. Some states or jurisdictions do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

Version

Software version: Release 3.5

Documentation version: April 30, 2008

Table of Contents

Preface .... **xix**

Audience xix
What This Guide Covers xix
Additional Sources of Information xx
Document Conventions xxi
How To Contact KnowNow, Inc. xxii

Chapter 1 The LiveServer's Operations **1**

Introducing the KnowNow Architecture 2
  Key LiveServer Concepts 3
Session Management 5
Event Communication Flow 7
Topics and Events 9
  Topic-based Event Routing 9
  Types of Topics 10
    Topics 11
    Meta-Topics 11
    Journals 11
  Tunnel Routes 12
  Managing Topics 13
    Creating Topics 13
    Setting Topic Properties 13
    Referencing Topics 13
    Managing Topics with Modules 13
    Topic Space 14
Introducing Connectors 15
  Connector Operations 16
  Connector Features 17
  The KnowNow Connectors 17
    Supported Compilers 17
    KnowNow JavaScript Connector 18
    KnowNow Windows Connector Suite 18
    KnowNow LiveJava Connector 19

Introducing Modules 20

Chapter 2 Events 21

Publish and Subscribe Operations and Events 22
The URL Root 22
Sending Commands to the LiveServer 23
The Anatomy of an Event: Headers 25
Event Size 27
Request Side 27
Response Side 28
Using Event Headers 33
Header Names 33
Sending Commands with Event Headers 34
Duplicate Event Squashing 35
Using the HTTP Interface 36
Sending Commands to the LiveServer from a Browser 36
POST 37
GET 37
Rules for Using Headers 37
Tunnel Route Formats 39
The simple Tunnel Route Format 39
The js Tunnel Route Format 40
Event Format 41
Callbacks 42
An Overview of do_method Commands 44
The Format of do_method Commands 44
The Categories of do_method Commands 44
Command Classes 45
Miscellaneous Class 45
kn_ Headers and do_method Values 45
do_method Command Reference 49
add_journal 49
add_notify 49
add_route 49
add_topic 50
batch 50
blank 51
clear_topic 51

delete_notify 51
delete_route 51
delete_topic 51
help 52
lib 52
notify 52
route 52
set_topic_property 54
test 54
update_notify 54
update_route 55
whoami 55
An Overview of kn_ and Other Headers..56
kn_ Headers Summary Table 56
Globally Supported kn_ Headers 59
Special LiveServer Headers 59
Status Event Headers..60
status 60
Specifying Time in Headers..60
kn_ Header Reference 62
kn_atomic 62
kn_batch 62
kn_block..62
kn_connection..62
kn_default_kn_expires 63
kn_deleted 63
kn_deletions..63
kn_event_id 64
kn_expires..64
kn_filtername..65
Attaching Filters..65
Detaching Filters 66
kn_filteroptions..66
kn_filterparams 66
kn_from 67
kn_history_* 67
kn_hold_new_events 69
Example of Use..69

kn_id 70
Shutting Down and Restarting the LiveServer 70
kn_max_queue_size 71
kn_module 71
kn_owner 71
kn_payload 72
kn_provider_id 72
kn_provider_name 73
kn_request_format 73
kn_request_id 73
kn_request_manager 73
kn_request_response 73
kn_response_flush 73
kn_response_flush_interval 74
kn_response_format 74
kn_response_uri 74
kn_retry 75
kn_route_checkpoint 75
kn_route_cluster_state 75
kn_routed_from 75
kn_route_id 76
kn_route_location 76
kn_since_checkpoint 76
kn_status 76
kn_status_from 76
kn_status_to 76
kn_temporary 77
Unreaped Routes and Topics and Temporary Topics 78
Example Settings 79
kn_temporary_expires 79
kn_timer_* 79
kn_time_t 80
kn_to 81
kn_topic_nodelete 81
kn_topic_nopersist 81
kn_topic_nopost 81
kn_topic_order 82
kn_topic_allow_update 82

kn_uri 82
Detecting Presence or Deletions 83
Sensing Deletions....83

Chapter 3 Web Services 85

Overview....86
Making Web Services Requests....87
Creating Offhost Routes....87
Obtaining WSDLs....88
Route Creation Examples 89
Creating Requests....89
Creating SOAP Requests....90
Creating Your Own SOAP 90
Letting the LiveServer Generate the SOAP 90
Creating REST Requests....91
Retrying Requests 91
Managing Your Responses 91
The LiveServer's SOAP and WSDL Interfaces 93
SOAP and do_method Commands 93
Using SOAP and WSDL 93
SOAP and WSDL Examples 93

Chapter 4 LiveServer Modules.... 95

Introducing Modules 96
Module Basics 96
Module Names....97
Module Start-up and Shutdown 97
Further Information on Modules....98
Filter Modules....99
Creating and Using Filter Modules....99
Filter Module File Formats.... 100
Route Filter Modules 100
Topic Filter Modules 100
The LiveServer Filter Modules 101
The KnowNow Expr Filter 102
Using Expr With Topics or Routes.... 102
Expr Syntax 103
Delimiter Characters.... 103
Expr Parameters 104

average . . . 105
count . . . 105
header . . . 105
lowercase . . . 105
max . . . 105
min . . . 105
op . . . 106
regex . . . 108
replace and replaceHeader . . . 108
std . . . 109
sum . . . 109
topic . . . 109
type . . . 109
uppercase . . . 109
value . . . 110
Expr Example . . . 110
The KnowNow Filterduplicates Filter . . . 111
Filterduplicates Parameters . . . 112
k or key . . . 112
Example . . . 112
h or hash . . . 113
i or include . . . 113
nomarkup . . . 114
sensitive . . . 114
s or separator . . . 114
x or exclude . . . 114
The KnowNow Tcl Email Filter . . . 115
The KnowNow XPath Filter . . . 116
The KnowNow XSLT Module . . . 118
The KnowNow XSLTtoHeaders Module . . . 120
Security Modules . . . 121
Understanding the Module API . . . 122
Start-up and Shutdown Processing of Modules . . . 122
Loading Modules into the LiveServer . . . 122
Start-up Processing . . . 123
Shutdown Processing . . . 124
Event Management . . . 124
Attaching Filter Modules to Topics or Routes . . . 124

Detaching Filter Modules 126
Events and Filter Modules 127
The LiveServer Module API Quick Reference 130
Required Module Functions. 130
KnModuleStart 131
KnApiEvent 131
KnRequest 132
Understanding the Security Module API 132
Authentication 133
Authorization. 133
Encapsulating HTTP Requests 133
KnAuthObject_Web. 134
KnAuthObject_Command. 134
KnAuthObject_Event 134
KnAuthObject_Topic and KnAuthObject_Route. 134
Offhost Routing. 135
Return Values 136

Chapter 5 Common Connector Capabilities 137

Basic Connector Capabilities 138
Connector Rules 138
Connectors, Tunnels, and Journals. 138
Publishing and Subscribing 139
Publishing in Batch Mode. 139
Unsubscribing 139
Session Management 139
Handling Status Events. 140
Using Filter Modules 140
LiveServer/ESS Installations 141
Advanced Connector Capabilities 142
All Since Last Connect (ASLC). 142
Cursors 143
Exception Handling 143
Heartbeat. 143
HTTP 1.0 Keepalive. 144
Licensing 144
Logging 144
Presence. 144

Proxies 144
Reconnect/retry. 145
Request/response. 145
Topic Properties 145
Cursors. 146
Static (Client Side) Cursors. 146
Dynamic (Server Side) Cursors 146
Cursor Capabilities 147
Request/response 148
Request/response Basics 149
Request/response Tasks 149
Setting Up Request/response. 150
Creating and Registering Service Managers 151
Registering Service Managers Directly with the LiveServer 151
Using Connectors to Register Service Managers. 152
Registering Providers 152
Creating a Requestor Journal 153
Using Request/response 154
Sending Requests 154
Blocking versus Non-Blocking Requests 155
Updating Requests 156
Sending Responses Back to the Requestor. 156
Deleting Service Managers and Providers 156
Request/response Authorization 157
Providers. 157
Requestors 158
Administrators 158
Monitoring 159

Chapter 6 Using the JavaScript Connector 161

Using the JavaScript Connector 162
Including the JavaScript Connector. 162
Understanding the Connector's Frames. 163
JavaScript Access and Security Domains 163
Debugging Your Applications 164
Publishing Events. 166
Event Identifiers 167
Publishing Form Data 167

Subscribing to Topics . . . 168
  Understanding Destinations . . . 168
  Subscription Options . . . 169
Unsubscribing . . . 171
Using Status Handlers . . . 172
JavaScript ActiveX Shim . . . 173
  Configuring the Browser . . . 173
  Application Configuration . . . 175
  Shim API . . . 175
  Shim Properties . . . 176
Simple Event Mapping Support . . . 177
  Mapping Outbound Objects to Events . . . 177
  Mapping Inbound Events to Objects . . . 178
Localization . . . 179
Writing Cross-Domain Web Applications . . . 181
  Modifying the Application Code . . . 182
    HTML Pages Not Using the Connector . . . 182
    Using a Single IP Address . . . 183
Using Command-Line Parameters . . . 184
  Command-Line Examples . . . 186
API Overview . . . 188
  The self Object . . . 188
  The kn Object . . . 188
  The KN Object . . . 188
  The kn_argv Object . . . 188
  Status Handler Objects . . . 189
  String and Character Support . . . 189
  The self Object . . . 189
    self Properties . . . 190
      kn_appFrameAttributes . . . 190
      kn_blank . . . 190
      kn_defaultHacks . . . 190
      kn_defaultOptions . . . 190
      kn_displayName . . . 190
      kn_hashCache . . . 190
      kn_lang . . . 190
      kn_lastTag . . . 190
      kn_lbversion . . . 190

kn_maxHits . . . 191
kn_queryString . . . 191
kn_response_flush . . . 191
kn_server . . . 191
kn_strings . . . 191
kn_tunnelID . . . 191
kn_tunnelMaxAge . . . 192
kn_tunnelURI . . . 192
kn_userid . . . 192
self Methods . . . 192
$$ . . . 192
$$_ . . . 192
kn__hacks . . . 193
kn__debug . . . 193
kn__options . . . 193
kn_charCodesFromString . . . 193
kn_createContext . . . 193
kn_debug . . . 194
kn_decodeRequest . . . 194
kn_defaultOnError . . . 194
kn_defaultOnMesssage . . . 194
kn_defaultOnStatus . . . 195
kn_defaultOnSuccess . . . 195
kn_doStatus . . . 195
kn_encodeRequest . . . 195
kn_escape . . . 195
kn_formatString . . . 195
kn_hacks . . . 196
kn_htmlEscape . . . 196
kn_inspectAsHTML . . . 196
kn_inspectAsText . . . 196
kn_inspectInWindow . . . 196
kn_isReady . . . 196
kn_lbversion . . . 196
kn_localize . . . 197
kn_localizeDefault . . . 197
kn_onTunnelStatus . . . 197
kn_onTunnelStop . . . 197

kn_opts 197
kn_options 198
kn_presence_register 198
kn_presence_obj 198
kn_publish 199
kn_publishForm 199
kn_redrawCallback 199
kn_resolvePath 199
kn_sendCallBack 199
kn_stringFromCharcode 200
kn_subscribe 200
kn_tunnelLoadCallBack 200
kn_unescape 200
kn_unsubscribe 200
kn_utf8decode 200
kn_version_check 201
kn_version_define 201
kn_version_toFloat 202
kn_watchdog_register 202
The kn Object 202
kn object Properties 202
kn.displayname 203
kn.documents 203
kn.lastError 203
kn.lastError.date 203
kn.leaderWindow 203
kn.ownerWindow 203
kn.tunnelURI 203
kn.tunnelID 203
kn.userid 203
kn object Methods 204
kn.ADD_JOURNAL 204
kn.ADD_NOTIFY 204
kn.ADD_ROUTE 204
kn.ADD_TOPIC 204
kn.BATCH 205
kn.CLEAR_TOPIC 205
kn.DELETE_NOTIFY 205

kn.DELETE_ROUTE 205
kn.DELETE_TOPIC 205
kn.NOTIFY 205
kn.ROUTE 206
kn.SAVE_CONFIG 206
kn.SET_TOPIC_PROPERTY 206
kn.UPDATE_NOTIFY 206
kn.UPDATE_ROUTE 206
kn.clearHandler 207
kn.iw 207
kn.publish 207
kn.publishForm 207
kn.sendQueue 207
kn.setHandler 208
kn.subscribe 208
kn.unsubscribe 208
The KN Object 208

Chapter 7 Using the Windows Connectors 211

Introducing the Windows Connector Suite 213
Connector APIs Overview 214
Supported Tools and Platforms 214
SSL and the Windows Connectors 215
Pre-installation Tasks and Connector Dependencies 216
LiveActiveX Installation Note 216
Live.NET Connector Requirements 216
LivePDA Connector Requirements 217
Installing the Windows Connectors 219
Uninstalling the Windows Connectors 225
Using the Connector APIs 226
Sample Code and Additional Documentation 226
Connector Libraries 227
Essential Tasks 228
API Classes for the Windows Connectors 229
Providing Access to the APIs 230
C++ Static Libraries 230
COM/ActiveX 231
.NET 231

The Connector Class 232
Creating a Connector 232
Number of Connector Instances 232
Examples. 233
Specifying a LiveServer 233
Connection Parameters. 233
Connection Status Handlers 234
Checking the Connection Status 235
Request Status Events 236
The IRequestHandler Class 236
OnError 237
OnStatus 237
OnSuccess 237
Publish Operations 239
Enabling and Using Offline Queuing 239
Offline Queuing API Calls 240
Offline Queuing Status Codes 241
Offline Queuing Examples 241
Subscribe Operations 242
Subscribe 242
SubscribeASLC. 243
The IListener Class 244
Other Subscribe Operations 245
Unsubscribe 246
Cursors. 247
Heartbeat 248
API Support for the Heartbeat Feature 248
Logging 249
Live.NET Connector and LivePDA Connector Logging 249
Configuring Logging 250
Using Configuration Files 250
Using the API. 251
Logging Messages and Log Level 251
Presence 253
PresenceSubscribe 253
PresenceUnsubscribe 253
Request/response 254
AddRequest 254

InitRequestBlocking 255
AddRequestBlocking 255
CleanupRequestBlocking 256
UpdateRequest 256
AddResponse 256
AddProvider 257
RemoveProvider 257
AddRequestManager 257
Disconnecting from the LiveServer 258
The LiveActiveX Connector: Use Fully Qualified Names 259
The LiveC++ Connector: Exceptions and Error Codes 260

Chapter 8 Using the Live.NET and LivePDA Connectors 267

About the Live.NET and LivePDA Connectors 268
Accessing the Live.NET Connector's API Documentation 268
About the WinForm Components 269
An Overview of the KnowNow WinForm Components 269
Live Data Source 269
KNChatlet 270
KNEventViewer 270
KNImage 270
KNMarqueeBand 270
KNTree 270
Windows Form and Control Methods 271
About the KnowNow ASP.NET Controls 272
An Overview of the KnowNow ASP.NET Controls 272
KNChatlet 273
KNEventViewer 273
KNImage 273
KNList 274
KNMarqueeBand 274
KNTable 274
KNTree 275
Cascading Style Sheet Support 275
System Configuration for the KnowNow ASP.NET Controls 275
KnowNow ASP.NET Deployment 276
Case 1 276
Case 2 277

Case 3 . . . 277
Programming Notes . . . 278
Bundling the Live.NET Connector with Custom Applications . . . 278
Providing Access to the APIs . . . 278
Accessing Header Values . . . 279
Live.NET Connector Exception Handling . . . 279
Installing Web Parts that Use the Live.NET Connector . . . 279

Chapter 9 Using the LiveJava Connector . . . 285

System Requirements . . . 287
Installing the LiveJava Connector . . . 288
jar Files . . . 288
Installing on Windows . . . 290
Installing on UNIX . . . 295
Using the LiveJava Connector . . . 296
Communicating with the Connector . . . 296
System Properties . . . 297
KN.log.level . . . 297
KN.log.file . . . 297
KN.forceHTTP_1.0 . . . 298
KN.disableKeepAlives . . . 298
Setting Multiple System Properties . . . 298
Connecting to a LiveServer through a Proxy . . . 298
Connecting to a LiveServer Using SSL . . . 299
Testing SSL . . . 299
Publishing Events . . . 301
Handling Status Events . . . 302
Sample Status Event . . . 302
Exceptions . . . 304
Subscribing . . . 305
Subscribing a Topic to a Callback . . . 305
All Since Last Connect (ASLC) . . . 306
Subscribing One Topic to Another Topic . . . 307
Subscription Options . . . 307
Unsubscribing . . . 309
Heartbeat . . . 310
API Support for the Heartbeat Feature . . . 310
Logging . . . 312

Presence 314
presenceSubscribe 314
PresenceUnsubscribe 314
Request/response 315
Journal Connection Callback 316
Simple Event Mapping Support 318
Mapping Outbound Objects to Events 318
Mapping Inbound Events to Objects 320
Using the Java Messaging Service 323
Java Messaging System API 323
JMS Overview 323
Administered Objects 324
Connection Factories 324
Destinations 325
Connections 325
Sessions 325
Message Producers 325
Message Consumers and Message Listeners 326
Messages 326
Unsupported JMS Interfaces 327
Unsupported JMS Methods 327
A Guide to the Java API 329
Creating and Managing Events 329
Cursors 329
Publishing, Subscribing, and Unsubscribing 330
Managing Topics, Events, and Routes 330
Setting Queue Options 330

Index 333

Preface

Audience

This book, the *KnowNow LiveServer Developer's Guide,* is for enterprise software developers who are creating applications that will use the KnowNow LiveServer and its Connectors. It provides programming information on the LiveServer and on the various Connectors (which manage communications between applications and the LiveServer) so that you can customize your applications to take best advantage of the LiveServer's capabilities.

This book is for software developers. It is assumed that the reader possesses a general knowledge of event-driven programming and HTTP/HTML as well as specific knowledge of the programming language(s) for the Connector(s) you are interested in using.

What This Guide Covers

This book describes how you can use the features of the KnowNow LiveServer and of the JavaScript Connector, LiveJava Connector, LiveC++ Connector for Windows, LiveActiveX Connector, Live.NET Connector, and LivePDA Connector through their respective application programming interfaces (APIs).

This book covers fundamental LiveServer architecture and concepts and provides guidelines for using the APIs and customizing your applications to take best advantage of the desired Connector.

This book contains the following chapters and glossary:

- Chapter 1 , "The LiveServer's Operations": Gives an overview of the LiveServer's features, architecture, terminology, and concepts.
- Chapter 2 , "Events": Gives detailed information on LiveServer events (communications between LiveServers and applications).

- Chapter 3 , "Web Services": Describes how to create Web services requests using the LiveServer as a proxy for those requests.
- Chapter 4 , "LiveServer Modules": Describes the kinds of modules in the LiveServer platform and provides an overview of the LiveServer's application programming interface (API), which you can use to create your own custom modules.
- Chapter 5 , "Common Connector Capabilities": Provides an overview of capabilities that most or all of the Connectors share, such as publishing and subscribing as well as other capabilities.
- Chapter 6 , "Using the JavaScript Connector": Provides information on how the JavaScript-based JavaScript Connector works and discusses its API. Using this information, you can customize your JavaScript application to use the JavaScript Connector to perform publish, subscribe, and unsubscribe operations.
- Chapter 7 , "Using the Windows Connectors": Provides information on the Windows suite of Connectors (LiveC++ Connector for Windows, LiveActiveX Connector, Live.NET Connector, and LivePDA Connector) and the provided Connector APIs. Using the information in this chapter, you can customize your Windows application to use one of the Windows Connectors to perform publish, subscribe, and unsubscribe operations.
- Chapter 9 , "Using the LiveJava Connector": Provides information on how the LiveJava Connector works and discusses its API so you can best customize your Java application to use the LiveJava Connector to perform publish, subscribe, unsubscribe, and request/response operations.
- An index provides quick access to the information in this book.

Additional Sources of Information

Additional information can be found in the following sources. The LiveServer documentation is available by choosing links on the LiveServer's opening screens. All KnowNow documentation is also available at the KnowNow Web site.

- The *KnowNow LiveServer Administration Guide* provides information on installing, configuring, and troubleshooting the LiveServer. It includes information on using the KnowNow utilities for backing up and restoring your database files, and includes a glossary of terms.
- The *KnowNow LiveServer Developer's Guide* (this book) provides information on the LiveServer's operations, events, modules, and Connectors.
- The KnowNow APIs are documented in separate HTML files that are included with the LiveServer software package. These HTML files provide linked, hierarchical views of the APIs, along with descriptions of the various calls and functions. Each

Connector has its own API and API documentation, both of which are provided with the Connector. For information on the location of each Connector's documentation, see the specific chapter for that Connector.

- A number of excellent samples are provided with the LiveServer, such as for making Web services request, for the Java filter, and so on. These samples are located in the LiveServer's installation directory.
- The *LiveServer_release_notes.html* file contains late-breaking information that could not make it into this book.
- The *KnowNow LiveBrowser Developer's Guide* provides information on using the LiveBrowser and its components.
- The KnowNow Web site, http://www.knownow.com, provides additional documentation and support.

Document Conventions

The following text conventions are used in this book to indicate information.

Italic	Italics are used the first time a term is introduced, for the titles of books, and for variables.
Boldface type	Boldface type is used for emphasis and for the names of menu, button, and other controls/commands, as well as for parameter names.
`Code`	Code excerpts and command line sequences are shown in this type face.
Ellipsis `. . .`	An ellipsis is used in code examples and syntax to indicate that nonrelevant information is not shown.
/name/ and */name*	A name enclosed in slashes is a directory name. It does not necessarily mean that the directory is at the root level; in fact, in most cases, the directory is a subdirectory in a hierarchy. A name starting with a slash represents a topic name; as with directories, the topic could be anywhere in a topic hierarchy.

How To Contact KnowNow, Inc.

We welcome any comments you may have about our software products or our documentation. Our goal is to provide functional and easy-to-use products that help you work more efficiently. For more information, contact us at

KnowNow, Inc.
997 East Arques Avenue
Sunnyvale, California 94085 USA

Phone: (408) 585-1800
Fax: (408) 585-1801

Web site: http://www.knownow.com

For technical support, or if you have comments or suggestions, contact Support at

support@knownow.com
(408) 585-1835

For sales inquiries, contact sales@knownow.com or call (408) 585-1866.

Chapter 1 The LiveServer's Operations

This chapter describes the important concepts and functions of the LiveServer. This chapter complements the introductory chapter in the *KnowNow LiveServer Administration Guide*, which provides a more general overview of the LiveServer's functions. Most of the topics introduced in this chapter are discussed in more detail in other chapters in this book. An overview of the LiveServer's operations is presented under the following headings:

- "Introducing the KnowNow Architecture" on page 2
- "Event Communication Flow" on page 7
- "Topics and Events" on page 9
- "Introducing Connectors" on page 15
- "Introducing Modules" on page 20

Introducing the KnowNow Architecture

KnowNow extends the use of HTTP and HTTPS beyond the limitations of client-initiated transactions. KnowNow tools free you from the constraints of the Web's normal request-response conventions and allow you to create Web-based applications that communicate asynchronously using messages, which are, along with all other communications and transactions, called events.

Note: All communications between applications and the LiveServer are called events. For more information on events, see Chapter 2 , "Events.".

Events can be initiated by either the client or server. By publishing events, applications can pass along information to any interested client application as soon as it becomes available. By subscribing to events, client applications can obtain the information of interest as soon as it is available. These actions are all moderated by the LiveServer, which acts as a central routing hub for all communications between applications. With a LiveServer in the mix, any application can exchange data with any other application.

The ability to publish and subscribe does not require you to adopt new, proprietary protocols, nor do does it require you to commit to a specific technology. With KnowNow, you can integrate dynamic data sources directly with your applications (which can be, for example, Excel spreadsheets, Java-based programs, and Web-enabled enterprise solutions) without downloading additional Java or ActiveX plug-ins.

Figure 1-1. The KnowNow architecture.

KnowNow LiveServer

Module

Module

Browser using a Connector

Application using a Connector

In addition, the LiveServer supports Web services requests, so you can create SOAP, WSDL, and REST requests.

Key LiveServer Concepts

The KnowNow LiveServer enables application-to-application communication across the Internet by providing real-time event subscription and publication services. Each LiveServer routes events from publishers to subscribers. A LiveServer can send and receive events from any application to many applications, including Web browsers, in real time, anywhere on the Internet. The LiveServer is reliable, secure, easily managed, and highly scalable. The LiveServer can coexist with your existing application servers, databases, and enterprise application integration (EAI) solutions.

The key concepts for a LiveServer are as follows:

- The communication flow, as described under "Event Communication Flow" on page 7.
- Topics and routes. Topics are the means by which the LiveServer organizes and manages events. Routes are the means by which the LiveServer sends and receives events. Topics and routes are described under "Topics and Events" on page 9.
- Connectors, which are code components that manage communications between your application and a LiveServer. Connectors are briefly described in "Connector Operations" on page 16. In addition, each language-specific Connector has its own chapter in this book, with the exception of the LiveSheet Connector, which is described in the *KnowNow LiveSheet for Excel® User's Guide* (which is shipped in electronic form with the LiveSheet Connector).
- Modules, which come in various flavors and are used for a variety of tasks. The LiveServer has certain core and helper modules it uses, and you can in addition write your own custom modules to perform security tasks as well to perform as topic and route filtering. Modules are briefly introduced under "Introducing Modules" on page 20, and are discussed in more detail in Chapter 4 , "LiveServer Modules."

At a very high level, the actions of the KnowNow platform boil down to this: The LiveServer communicates with Connectors, which in turn communicate with your application. Your application then communicates with the Connector, sending and receiving events through it to other applications. The LiveServer receives events from publishing applications and routes them to the subscribing applications that are interested in receiving those events.

Note: It is possible to have more than one LiveServer in a system; in fact, LiveServers can be used in clusters, as described in the *KnowNow LiveServer Administration Guide*.

Session Management

LiveServer provides a set of authentication policies for the Flat Files and LDAP permissions systems. Selection of these policies is controlled through the **Authentication policy** parameter using the KnowNow System Administration console as described in the *KnowNow LiveServer Administration Guide*. For information on Connector API support for session management, see "Session Management" on page 139.

LiveServer's authentication policies direct LiveServer to use particular configurations of two components: knperm and knsession. For policies that are based only on HTTP Basic authentication (policy 1, which requires HTTP credentials, or policy 4, which allows anonymous access), the session module is not enabled at all. For other policies, the knsession AOLServer module is added into LiveServer's configuration. KnSession is a framework for handling cookie-based authentication of HTTP requests.

Policies 2, 3, 5, and 6 invoke the knsession module. Policies 2, 3, and 5 also then configure a particular module that fits into the KnSession framework (knlogin) to handle particular authentication policies where the cookies and sessions are entirely maintained by LiveServer. Policy 6 is reserved for invoking other KnSession-based modules that integrate with other authentication systems.

KnSession always "ignores" HTTP requests that have HTTP Basic authentication, allowing KnPerm (and the core AOLServer) to handle those. This provides easy compatibility for existing Connector-based components; they need not be re-coded to use the session login capability just because it's turned on. Also, KnSession assumes that login form handling is only possible in a browser, so if the User-Agent HTTP header value cannot be recognized as a browser, KnSession acts just like policy 1: HTTP Basic Required).

The knlogin plugin to knsession provides for policies 2, 3, and 5. Policies 2, 3, and 5 implement various kinds of authentication requirements. All of them use a login form to present requests for credentials; the difference is in when that form is required and whether anonymous access is allowed. For policy 2 (which requires credentials and does not allow anonymous access), all browser sessions must be authenticated, and the form does not provide for an anonymous option. For policy 3 (which requires a login but allows anonymous access), the same form is presented at the same times as in policy 2, but a checkbox allowing anonymous logins is presented.

Policy 5 (which allows anonymous sessions by default, but also allows for providing login credentials for certain pages) is a "session-based" version of Policy 4. In this case, a session is automatically provided when the user goes to the LiveServer pages (no credentials or login form is required). However, when a user accesses a privileged page, they are redirected to a login form.

In all these cases, the login active page is deployed at the URL "/login/login" and the logout (remove session) page is deployed at URL "/login/logout". It is possible to customize the login page to match your application's style; see "runtime/pages/login/login.adp" for an example. The login page URL can be configured in the KnowNow System Administration console; for additional information, see the *KnowNow LiveServer Administration Guide*.

Policy 6 (custom session management) is provided for development teams that need to integrate with external cookie-based authentication systems. We *highly* recommend contacting KnowNow customer support for this sort of integration. It's easy, but we have not completed formal documentation of this sort of work.

Event Communication Flow

With KnowNow in the picture, the communication model stripped to its most essential components is like this:

- On the publish side, information flows over an HTTP router into the LiveServer. This information can consist of commands that affect topics, though it can also consist of other kinds of information, such as status messages or data.
- On the subscribe side, client applications send a subscription request asking to be notified whenever anything happens in relation to one or more specific topics. The subscription request establishes a journal (a special kind of topic) for that subscriber. Each subscriber has only one journal. (The Connector sets up the journal.)
- The LiveServer creates routes between the topic and the subscriber's journal and between the topic and the publisher's journal. There is one route per subscriber.

The way in which these communications are shared is through the LiveServer and its language-specific Connectors. The KnowNow LiveServer provides easily managed, scalable, real-time, publish-subscribe event notification across the Internet. LiveServers run as server daemons on the same machines as your Web servers or application servers, or alongside them on separate host machines. The LiveServer uses HTTP or HTTPS and the related Web standards for all operations. You can use more than one LiveServer on more than one machine, and you can also cluster LiveServers to act in concert.

In the KnowNow architecture, all communications between Connectors, a LiveServer, and your application are called events, and the processes by which events are shared are called publishing and subscribing. Connectors, which are specific to particular programming languages, are the intermediaries between your applications and the LiveServer. (Although they don't need to be; there are some things that can be done without Connectors, but life is much easier with them.) When you create your application, you use the KnowNow application programming interface for your chosen language to have your application communicate with the appropriate Connector, which will in turn communicate with the LiveServer. If you already have an application, you can customize it so that it can communicate with the correct Connector

Connectors are introduced in more detail under "Introducing Connectors" on page 15, and are discussed in depth in the chapters on the individual Connectors.

In order to handle events, the LiveServer uses organizational items called topics to sort and share events. Topics are discussed in more detail under "Topics and Events" on page 9.

The LiveServer also supports request/response for those Connectors that support it. For information on request/response, see "Request/response" on page 148.

The LiveServer itself comes with a set of API calls for creating modules for managing topics and routes and for managing authentication and authorization. For more on modules and the LiveServer API, see Chapter 4 , "LiveServer Modules."

Topics and Events

With KnowNow products, you can create applications that communicate asynchronously using events. Events are simply messages sent by an application through a Connector to a LiveServer. (The process of sending those events is called publishing.) The LiveServer then forwards the event (also through a Connector) to one or more other applications that have subscribed to the event. An event could be used to:

- Trigger an update in the current prices in a stock portfolio display.
- Re-calculate an online customer's total purchase price when an item is added or a quantity is changed.
- Indicate a change in traffic on a set of Web servers.
- Update a chat window with the latest message.
- And so on.

Any data your application controls could be considered an event that can be shared with other applications or with itself.

Events have an identification string, timestamp, an expiration timestamp, metadata headers, and a content payload. The data contained in the event payload can be in any format.

Topic-based Event Routing

An event sent by an authorized client to a LiveServer for publishing is always associated with a topic. Topics are the way in which the LiveServer organizes the events it receives for publishing. Topics are stored in the form of directories on the LiveServer host machine and are managed by the LiveServer.

Any client application that is connected to a LiveServer can act as an event publisher, a topic subscriber, or both. Topics themselves can also subscribe to other topics, which allows you to construct interconnected topic groups and distribute one event into many topics or combine events from multiple topics into a single event stream.

Consider the hypothetical topic tree shown below, which uses topics to organize events for stock quotes. Starting at the bottom of the tree, a separate topic is created for each stock; in this example, the topics are being created to handle price information events.

Example 1-1. Stock exchange topics.

```
/StockExchange
        /AMEX
            /AMC
```

```
        /BerkHBcp
        ...
    /DAX
        /ADSGF
        /SI
        ...
    /NASDAQ
        /ATI
        /BHA
        ...
    /NYSE
        /AET
        /BA
        ...
```

At the next level up, the AMEX topic represents an individual stock exchange and subscribes to all topics representing stock traded on the AMEX exchange. The other topics at the same level (DAX, NASDAQ, and so on) perform the same kinds of tasks as the AMEX topic does.

Finally, the StockExchange topic subscribes to all the stock exchange topics.

A client application that is interested in the information contained in these topics could do any of the following tasks:

- Subscribe to events for an individual stock to receive price information only for that stock.
- Subscribe to all events for a particular stock exchange and receive events for all stocks to which that exchange has subscribed.
- Subscribe to the StockExchange topic and receive price information on all stocks.

This is just one way in which topics can be used. The way in which topics are organized and in which they subscribe to other topics will vary according to what is most useful for the information.

Types of Topics

When your application publishes an event, it always specifies a topic to which that event is to be published. A topic defines how an event is to be routed to all the subscribers to that topic. A route can point to another topic or to a generic listener.

Topics fall into three basic categories:

- Topics
- Meta-Topics

- Journals

Topics

In a route topology, topics are the primary nodes to which data is posted. Topics organize events. Events are published to topics; applications then subscribe to those topics to receive those events (and can also unsubscribe from topics as well). Topics can have meta-topics that are used to store information relevant to the topic, such as hierarchy and routing data. Topic names start with a forward slash; they can also be called a path. Topics are stored in the LiveServer under a directory called /kn, and are usually specified relative to the /kn directory. Topics can be nested just like directories. For example, /sales could be a topic directly under /kn, and /northwest could be a topic under /sales, and /forestry could be a topic under /northwest, so the topic name for /forestry would be specified like this: /sales/northwest/forestry.

Meta-Topics

Meta-topics provide a unique feature of introspection to the LiveServer, meaning that you can use them to understand what the structure and behavior of a topic. Each topic has a set of meta-topics named .../kn_subtopics and .../kn_routes. The meta-topic /kn_subtopics stores events that represent the topic's children (that is, a topic's subtopics); /kn_routes stores events that represent the routes out of a topic. Meta-topics are created only as needed. You should never post (notify) to meta-topics; the protocol used by LiveServer does not define the operation of such notifications.

An application can subscribe to a topic's meta-topics in the same way it subscribes to any other topic, with similar results. For instance, if an application subscribes to /*topicname*/kn_subtopics, it will be notified every time subtopics are added to or deleted from that topic. The same is true for routes; if an application subscribes to a topics /*topicname*/kn_routes meta-topic, it will receive events every time a route is added or removed from that topic.

You can also physically view the directory hierarchy to see what is in /kn_topics and /kn_routes by using theKnowNow System Administration console, whose use is described in the *KnowNow LiveServer Administration Guide*.

Journals

A journal is a special kind of topic that provides connections to a client over special routes called tunnel routes. You can think of a journal as a session. A journal topic is in essence a URL that is accessible on the Internet. Connectors use the journal topic to communicate between the LiveServer and an application that cannot otherwise receive a POST from the LiveServer.

Only journals are allowed to have tunnel routes. The names of journals must end with /kn_journal. As with other topics, journals can have meta-topics as well. Events for tunnel routes are created in the /kn_routes meta-topic of the journal, but can be used for introspection purposes only. For instance, you may subscribe to a journal's /kn_routes meta-topic to monitor the events that are generated when a client connects to or disconnects from the LiveServer over a persistent connection. However, you would not be able to modify those events or affect the behavior of the tunnel routes in any way.

Journals know how to multiplex events from many topics so the Connectors can demultiplex them on the other end. Journals are normal topics without duplicate squashing or event updating. The advantage of journals is that clients can subscribe to a single topic using a single HTTP session and all events destined for that subscriber are delivered over the one tunnel route.

Most KnowNow applications organize the names of individuals' journals by using a convention in the /kn/who topic. The first level of this convention names the individual; for example, /kn/who/person. Then, all of that person's sessions (journals) are organized under that topic's s subtopic. Each session (journal) is named with a random number, to support multiple simultaneous sessions by that user.

/kn/who/*person*/s/9390810457/kn_journal

Tunnel Routes

Tunnel routes are smart routes from the LiveServer to clients (Connectors and standalone HTTP applications). Tunnel routes use persistent connections to send events to desktops and applications while maintaining the policies of Internet administrators for their firewalls, network address translators (NATs), and Web proxies. Tunnel routes always have an endpoint, which is a journal; therefore, they are only allowed for journals.

LiveServer messages sent over tunnel routes come in two flavors:

- JavaScript, encoded as **js**. This format is used over a tunnel route to pass events to a JavaScript Connector that is running in a browser.
- Simple, encoded as **simple**. This format is used by all other types of Connectors.

Both tunnel route formats send keepalives occasionally to maintain their persistent connections with the LiveServer. All tunnels and routes have an owner as specified in kn_owner. By setting the kn_expires header, a client can request that the LiveServer close a tunnel route at a certain time. For more on tunnel route formats, see "Tunnel Route Formats" on page 39. For more on event headers, see "Using Event Headers" on page 33. For more on managing any kind of route, see "Command Classes" on page 45 and "Route Filter Modules" on page 100.

All tunnel routes support flushing, as described under "kn_response_flush" on page 73.

Managing Topics

Whatever the type of topic you are creating, you will need to create and name it, reference it, and possibly write modules to perform certain actions on topics and routes.

Creating Topics

Your application can create a topic by simply referencing the topic in a request to the LiveServer. In other words, if the LiveServer receives a publish command that asks the LiveServer to publish an event to a topic, and that topic does not exist, the LiveServer creates that topic, then publishes the event to it.

Setting Topic Properties

Topics have properties, such as an expiration time. You can set properties for topics by either using the KnowNow System Administration console or by using the do_methods and kn_ headers that are related to topics. Information on using the KnowNow System Administration console is in the *KnowNow LiveServer Administration Guide*; information on the do_methods and kn_ headers is in Chapter 2 , "Events." In particular, see "set_topic_property" on page 54.

Referencing Topics

A topic is identified by a uniform resource identifier (URI); for example:

/chat/technology/xml

To avoid naming conflicts, the topic name should start with a domain name; for example,

/knownow.com/chat/technology/xml

A topic name can contain Roman letters, digits, slashes, underscores, dashes, periods, and non-ASCII characters.

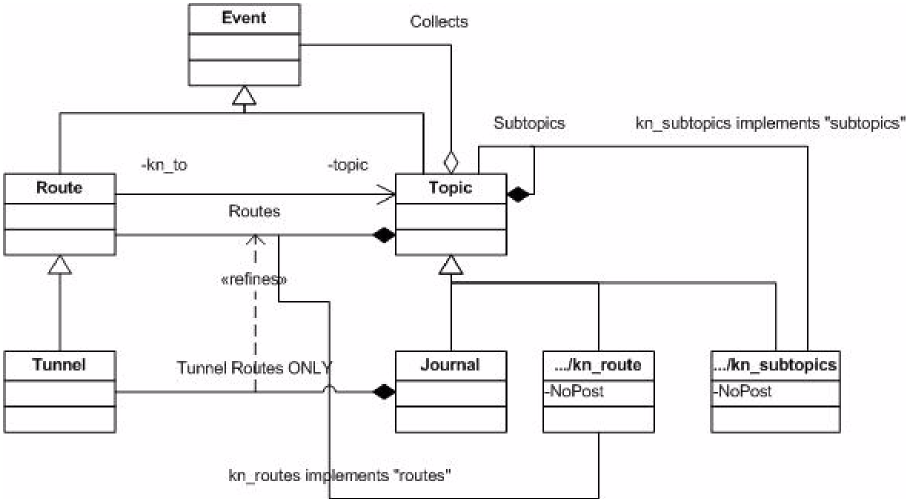
Managing Topics with Modules

To manage topics and their routes, or to manage authentication/authorization, you can write modules. Modules are blocks of code, written using the LiveServer API, that perform specific actions; in the case of topic and route modules, which are called filter modules, they perform specific actions on topics or routes based on events. All modules are loaded into the LiveServer using the KnowNow System Administration console. Once loaded, filter modules can be attached to a topic or route as described under "Filter Modules" on page 99; security modules normally manage the entire LiveServer installation.

Topic Space

Figure 1-2 is a UML (unified modeling language) diagram illustrating how the LiveServer thinks of topic space objects.

Figure 1-2. Implementation of topics.



Introducing Connectors

KnowNow Connectors make it possible for your applications to easily communicate with the LiveServer, giving you a convenient way to publish, subscribe, and unsubscribe using your native application environment. Each Connector is shipped with language-specific application programming interfaces (APIs). A Connector may also include wrappers for specific application needs. All the Connector APIs include all components needed for managing the publish, subscribe, and unsubscribe operations, as well as offline queuing, transport, status handlers, and other capabilities.

Connectors are employed in situations where it is less than ideal to send or receive HTTP POSTs, such as when

- HTTP/1.1 implementation is not conveniently accessible in the integration environment.
- The LiveServer needs to be connected to from behind a Network Address Translation (NAT) router, a firewall, or other network configuration that won't allow inbound HTTP connections.
- You wish to take advantage of a simple API instead of writing to the HTTP API.

Figure 1-3. Connector architecture.

With the exception of the LiveSheet Connector, which is documented in the *KnowNow LiveSheet for Excel® User's Guide*, each KnowNow Connector has its own chapter in this book, as well as separate API documentation in HTML format. (For a description of the capabilities that are common to all Connectors, see Chapter 5 , "Common Connector Capabilities.") The language-specific chapters provide information for interfacing with that type of Connector, and also provide an overview of the important components of each Connector's API. The API documentation documents each call in each Connector's API. Using the information in this book and in the separate API documentation, you can customize your application to communicate more readily with the desired Connector.

For example, if your application is a Java application, you could use the KnowNow Java API to customize your application. You could also read the chapter on common Connector capabilities as well as the chapter on the LiveJava Connector in order to understand how the LiveJava Connector works and how to use the KnowNow Java API to customize your application. You could refer to the separate API documentation for detailed information on each call that you are interested in using.

Connector Operations

The LiveServer communicates with Connectors, which in turn communicate with your application. Your application also communicates with the Connector, sending and receiving events through it to other applications. As described under "The KnowNow Connectors" on page 17, there is a Connector for each of the supported languages. In order to use a Connector and its capabilities, you would normally use the appropriate Connector API to customize your application.

The supported languages are JavaScript, Java, C, and C++, as well as COM, ActiveX, and .NET for Windows. Later chapters in this book describe Connectors at the language-specific level, including providing an overview of the capabilities of the APIs that are available for each language.

Connectors perform the following major operations:

- They manage publish, subscribe, and unsubscribe requests.
- They perform operations based on the results of status events relating to connection status and the status of publish, subscribe, and unsubscribe events.
- Optionally, they can manage offline queuing, either automatically by enabling offline queuing, or with more custom control using additional calls that support offline queuing. Offline queuing stores events while the connection to the LiveServer is down, then automatically sends those events to the LiveServer when the connection is re-established.
- Some Connectors have additional capabilities, such as request/response.

Connector Features

Connectors offer capability, flexibility, and compatibility in the following ways:

- Connectors provide application programming interfaces (APIs) for JavaScript, Java™, C, C++, ActiveX, .NET, and the Pocket PC, giving you a convenient and easy way to publish, subscribe, and unsubscribe using your native application environment.
- Connectors enable your application(s) to exchange data with other applications through a LiveServer.
- Connectors connect to a LiveServer using HTTP or HTTPS, so they can communicate across existing firewalls without requiring re-configuration.

A Connector toolkit of components is available that simplifies integration with commonly used applications, developer environments, and legacy browsers.

The KnowNow Connectors

The KnowNow Connectors are as follows:

- "KnowNow JavaScript Connector" on page 18
- "KnowNow Windows Connector Suite" on page 18
- "KnowNow LiveJava Connector" on page 19

Each Connector has a language-specific application programming interface with which you create one or more instances of the Connector within your application. The Connector APIs provide a full set of functions for managing publish, subscribe, and unsubscribe operations, as well as offline queueing and status handling.

No matter what the language, each individual Connector communicates with only one LiveServer; therefore, if your application needs to communicate with more than one LiveServer, you will need to create an additional Connector for each additional LiveServer.

Supported Compilers

For customizing your applications with a KnowNow Connector using the Connector APIs, the following compilers are supported:

- Microsoft Visual C++ Version 6 for Windows NT 4.0, Windows 2000, and Windows XP
- Sun WorkShop 6 Update 1 on Solaris™
- Java JDK 1.3.1 and later

- GNU C/C++ compiler

Additional supported tools and platforms may be discussed in the specific Connector chapter.

KnowNow JavaScript Connector

The KnowNow JavaScript Connector enables an application running in a Web browser to communicate with the LiveServer. It includes an ActiveX shim. The JavaScript Connector is installed when you install the LiveServer (as described in the *KnowNow LiveServer Administration Guide*). The KnowNow JavaScript Connector is described in detail in Chapter 6 .

KnowNow Windows Connector Suite

The LiveServer has available a suite of Connectors that work on the Windows platform. The Connectors in this suite make it possible for any Windows application to send and receive KnowNow events. The APIs for the suite of Windows Connectors include support for C++ as well as for the Windows-specific COM, ActiveX, and .NET interfaces, and the Pocket PC. In order for your Windows applications to best and most easily use the capabilities of the LiveServer, you have a choice of using any of the following Connectors:

- LiveC++ Connector for Windows
- LiveActiveX Connector
- Live.NET Connector
- LivePDA Connector

Which Connector you use depends on what language your application is written in. Each of these Connectors performs the same kinds of functions. In addition, the Live.NET Connector offers WinForm and ASP.NET components. Application programming interfaces (APIs) support a wide range of tools and Windows platforms.

Three APIs (C++, COM/ActiveX, and .NET) offer language-specific functions for modifying or creating your applications to take advantage of the capabilities of the desired Connector. By using the APIs, your application can make the proper calls to the Connector and properly receive information from the Connector.

The LiveC++ Connector API is written in C++; the Live.NET Connector API is written in .NET. The remaining APIs (Com/ActiveX) are wrappers around the LiveC++ Connector core. Each API shares the same names for objects, functions, and calls, making it easy to move from using one API to another, and are threadsafe.

The Windows suite of Connectors is described in detail in Chapter 7 , "Using the Windows Connectors" and Chapter 8 , "Using the Live.NET and LivePDA Connectors."

KnowNow LiveJava Connector

The KnowNow LiveJava Connector is supported on Java 1.3.1 virtual machines. It can be used to connect Java applications with each other and the LiveServer. The LiveJava Connector's native code wrappers are available on Solaris™, Linux, and Windows as shared libraries. The LiveJava Connector is described in detail in Chapter 9 , "Using the LiveJava Connector."

Introducing Modules

Although the LiveServer itself is provided by KnowNow, and you may not need to do anything with it in order to use it, you can customize the LiveServer by creating and loading modules to perform specific functions, such as topic and route management and authentication, authorization, and other security features. The LiveServer uses core modules that are already provided for your use. Some of these modules can be replaced with your own custom versions; others must be used as they are shipped from KnowNow. All modules are configured using the KnowNow System Administration console. To create these modules, use the LiveServer APIs. Modules and the module API are described in Chapter 4 , "LiveServer Modules."

Chapter 2 Events

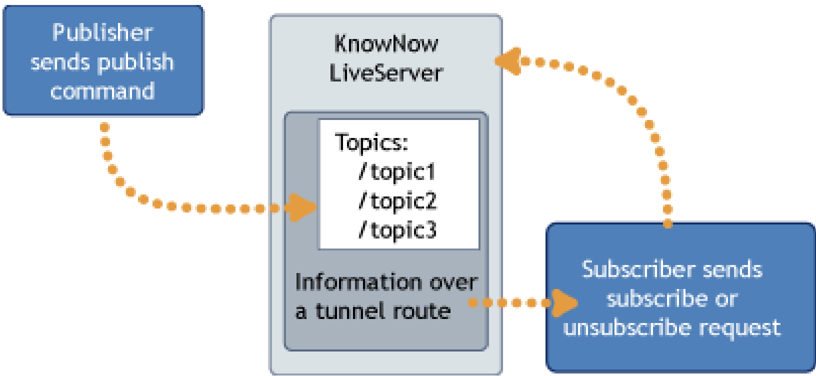
All communications between applications and the LiveServer are called events. Events contain certain specific types of information in special formats. This chapter describes the LiveServer's events and event headers under the following headings:

- "Publish and Subscribe Operations and Events" on page 22
- "The Anatomy of an Event: Headers" on page 25
- "Event Size" on page 27
- "Using Event Headers" on page 33
- "Duplicate Event Squashing" on page 35
- "Using the HTTP Interface" on page 36
- "Tunnel Route Formats" on page 39
- "An Overview of do_method Commands" on page 44
- "do_method Command Reference" on page 49
- "An Overview of kn_ and Other Headers" on page 56
- "kn_ Header Reference" on page 62
- "Detecting Presence or Deletions" on page 83

Publish and Subscribe Operations and Events

When a publishing application sends events to a topic in the LiveServer, the LiveServer places that information in the correct topic and routes it out to the interested subscriber client applications as illustrated in Figure 2-1.

Figure 2-1. The KnowNow publish and subscribe communication flow.



The URL Root

There is more to this model, though. Each piece of information that comes into the LiveServer either contains or does not contain what is called the *URL root*. The URL root is simply a part of the HTTP command that, when present, is a flag to the LiveServer telling it to handle that information as a LiveServer request. If the information flowing into the LiveServer does **not** contain the URL root, the information is handled by the LiveServer as a normal HTTP request, just as any Web server might.

The LiveServer's URL roots is **/kn**. In the following HTTP request, the first part indicates the protocol, an IP address, machine name, or domain name, and the port. The next section, the section that is composed of **/kn?**, uses one of the LiveServer's URL roots. Everything after the URL root comprises LiveServer commands.

```
http://localhost:8000/kn?do_method=add_notify;kn_payload=payload;myheader=stuff
```

To break this command down into its component parts,

- The first part, **http://localhost:8000**, is the protocol, machine name, and port.
- The second part, **/kn?**, is the LiveServer's URL root, with a question mark indicating that one or more commands follow.
- The third part, **do_method=add_notify;kn_payload=payload;myheader=stuff**, is a set of commands being sent to the LiveServer.

Sending Commands to the LiveServer

The LiveServer receives commands in headers, which are header names paired with a value (in the format *header_name*=*value*; for example, do_method=add_notify or kn_to=/mytopic). The order in which headers are sent does not matter. Headers are described in more detail under "Using Event Headers" on page 33.

With the concept of the URL root, a somewhat more complete look at the LiveServer would include the fact that it is sending and receiving information depending on what type of information it is. An incoming URL (request) is handled either by a Web browser or by a Connector according to these rules:

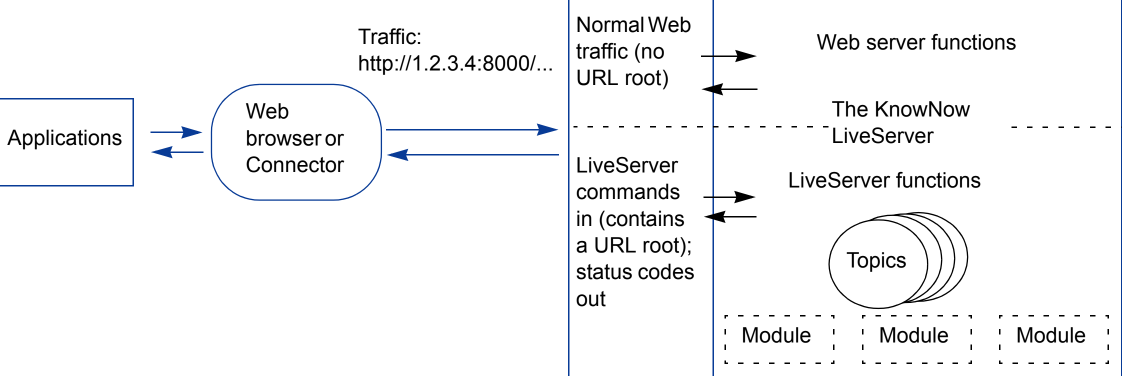
- If the request is handled by a browser, the browser converts the URL to the needed HTTP GET/POST syntax and sends it on to the LiveServer.
- If the request is handled by a Connector, the Connector takes the URL and calls whatever is needed, such as the **publish** command, then converts the URL to HTTP protocol and sends it on to the LiveServer.

Figure 2-2 illustrates how a URL is treated depending on whether it has a URL root or not. Once the request arrives at the LiveServer, the LiveServer examines it to see if it contains a URL root.

- If the URL **does not** contain a URL root, it is handled as a normal URL/Web request.
- If it **does** contain a URL root, the LiveServer further manages the traffic (the event), either directly as a publish, subscribe, or unsubscribe request, or by handing off the event to one of the modules that you have created and loaded into the LiveServer. If you have loaded in a security module, the LiveServer first hands off the event to the security module to authorize and authenticate the requestor before the LiveServer processes the request.

In either case, the outgoing response traffic can be one of the normal HTTP status codes (such as 200, 401, or 404). For example, if the incoming traffic was a request to delete a topic, and the topic does not exist, the outgoing status code would be a 404, indicating that the topic doesn't exist, along with, possibly, an explanatory message. For further information on status events, see "Status Event Headers" on page 60.

Figure 2-2. The LiveServer's activities.



The Anatomy of an Event: Headers

Each event processed by the LiveServer contains a set of one or more headers. As explained under "The URL Root" on page 22, these headers are preceded by a URL root that tells the LiveServer to process it specially.

Headers are the core of events. They carry commands and information to the LiveServer. Each header consists of a name and a value, using the syntax **name=value**. Many of these headers, such as the do_method commands, are provided by KnowNow for your use. In the case of the do_method command used in the example under "The URL Root" on page 22, the **name** part of the header is **do_method**, and the **value** part of the header is **add_notify**. Each name and value is a string of Unicode or UCS characters (including reserved characters and characters not yet allocated) and can be of any length. The order in which headers appear within an event is not preserved; therefore, event headers can appear in any order within an event. Each header must have a unique name.

KnowNow recommends that headers follow the naming and meaning of HTTP headers when appropriate, using kn_payload for message bodies. For example, a property named "content-type" whose value is "text/plain" would indicate, to a receiver, that the kn_payload property is a text string.

However, since header names are case-sensitive, we recommend using lowercase equivalents of the analogous HTTP header names.

A sample event might include the headers and values shown in Table 2-1. (If you are viewing this documentation in a Web browser, the Greek and Japanese words in the **Value** column and in the discussion of this example should appear as Greek letters and Japanese *katakana* respectively. If they do not, you may need to add language support to your browser.)

Table 2-1. A sample event.

Header Name	Value
header1	value1
kn_payload	This is the kn_payload.**LF**This is Greek: κοσμε
header2	**SP**value2**LF**Japanese: チヤシト
header**LF**three is**NUL**long	1
kn_id	979273892_861_2249

Any characters that are not alphanumeric should be escaped with the percent symbol (%) and two hexadecimal digits containing the byte value of the character.

Some parts of the example in Table 2-1 include characters with special meanings:

- LF. The linefeed character (character 10, U+000A).
- NUL. A null character (character 0, U+0000).
- SP. The literal space character (character 32, U+0020).
- κοσμε. The Greek word *kosme* in Greek characters (a five-character string, U+03BA,U+1F79,U+03C3,U+03BC,U+03B5).
- チャシト. The Japanese word *chyatto* ("chat") in katakana characters (a four-character string, U+30C1,U+30E3,U+30C3,U+30C8).

These headers would look like the following in a command (normally, these would appear in a single line, but here they are broken out into separate lines for clarity; note how commands are separated by semicolons):

```
header1="value1";
kn_payload="This is the kn_payload.LFThis is Greek: κοσμε";
header2="SPvalue2LFJapanese: チャシト";
headerLFthree is NULlong="1";
kn_id="979273892_861_2249"
```

The headers beginning with **kn_** are KnowNow headers; the others are custom headers that you might create. The KnowNow headers are described in greater detail under "Using Event Headers" on page 33.

Event Size

This section addresses the question of the size of the KnowNow message (event), including the HTTP header, on the Internet of an event with an empty kn_payload., assuming that it is for only the required KnowNow header properties.

The discussion considers both the request and response sides of a typical HTTP transaction with a LiveServer.

Request Side

On the publication side, the shortest publication request seems to be a 26-byte HTTP/0.9 GET request:

```
GET /kn?do_method=notify
```

On the subscription side, the shortest tunnel route request is an HTTP/0.9 GET request, which is 20 bytes:

```
GET /kn/kn_journal
```

Also on the subscription side, the shortest static route request is an HTTP/0.9 GET request, which is 27 bytes:

```
GET /kn?kn_to=/kn_journal
```

Upgrading the request to HTTP/1.0 adds another nine bytes for the space HTTP/1.0 version in the request line.

To get around URL length limitations, Connectors often send their requests using the HTTP/1.0+ POST method, such as this 37-byte notification:

```
POST /kn HTTP/1.0
Content-Length: 0
```

Upgrading the request to HTTP/1.1 adds a Host: HTTP header with the host name of the LiveServer; this adds at least nine bytes, plus the length of the host name.

Using a longer topic name increases the request size by the length of the topic name.

Including headers and values in a request increases the request size proportional to the length of headers and values, with the exact length being data-dependent (varying from one to three times the number of bytes, depending on the particular byte values). There is also a small overhead per header (one byte per header-value pair for the equals sign (=) separating the name from the value, plus one byte for all but the first header-value pair to account for the semicolon (;) or ampersand (&) separating it from the previous header-value pair).

Nesting requests inside larger "batched" requests adds additional overhead, since each batched request is encoded individually (as above), and then the encoded request is further encoded as the value of a kn_batch header.

As an example, consider an event with an 11-byte payload string of

```
one+one=two
```

and an expiration time 30 seconds into the future ("+30"). One encoded form of that event might be the 42-byte string

```
kn_payload=one%2bone%3dtwo;kn_expires=%2b3
```

A simple notification using that event might be an 80-byte POST request:

```
POST /kn HTTP/1.0
Content-Length: 42
kn_payload=one%2bone%3dtwo;kn_expires=%2b3
```

If we encapsulate this inside a one-item "batched" request, we must add an explicit do_method header with the value notify. The encoded batched request might be the 100-byte string

```
do_method=batch;kn_batch=do_method%3Dnotify%3Bkn_payload%3Done%252bone%253dtwo%3B
kn_expires%3D%252b3
```

An HTTP/1.0 POST request containing that batch might be 139 bytes:

```
POST /kn HTTP/1.0
Content-Length: 100
do_method=batch;kn_batch=do_method%3Dnotify%3Bkn_payload%3Done%252bone%253dtwo%3B
kn_expires%3D%252b3
```

Response Side

The JavaScript-formatted response to a fairly typical publish request is 729 bytes:

```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Expires: Thu, 01 Jan 1970 00:00:00 GMT
MIME-Version: 1.0
Date: Wed, 14 May 2003 20:41:43 GMT
Server: KnowNow LiveServer/1.8.0.34
Content-Type: text/html; charset=utf-8
Content-Length: 465
Connection: close

<html><head><title>200 Notified.</title>
<script> <!--

// -->
```

```
</script>
</head>
<body>
<h1>200 Notified.</h1><!----><script type="text/javascript"><!--
if (parent.kn_sendCallback) parent.kn_sendCallback({elements:[
{
       name : "status",
       value : "200 Notified."
},
{
       name : "kn_id",
       value : "754773BB-2690-48E9-A69C-1B09A6A6B876"
},
{
       name : "kn_time_t",
       value : "1052944716"
},
{
       name : "kn_payload",
       value : ""
}
]}, window);
// -->
</script><!--
-->
</body></html>
```

If the easier-to-parse **simple** response format is requested (by adding the kn_response_format header with the value **simple** to the original request), the response is only 379 bytes:

```
HTTP/1.0 200 OK
Content-Type: text/plain; charset=utf-8
Expires: Thu, 01 Jan 1970 00:00:00 GMT
MIME-Version: 1.0
Date: Wed, 14 May 2003 20:43:56 GMT
Server: KnowNow LiveServer/1.8.0.34
Content-Type: text/plain; charset=utf-8
Content-Length: 113
Connection: close

91
kn_id: 6FE36D2F-3A39-4A39-A496-152650A78B7A
kn_time_t: 1052945036
status: 200=20Notified.
```

For tunnel routes, the response is not fixed-size; instead, the response starts with status event, such as this 898-byte JavaScript-formatted response:

```
HTTP/1.0 200 Watching Topic.
```

```
Connection: close
Pragma: No-cache
Content-Type: text/html; charset=utf-8

<html><head><title>200 Watching Topic.</title>
<script><!--
// -->
</script>
</head>
<body onload="if (parent.kn_tunnelLoadCallback)
parent.kn_tunnelLoadCallback(window)">
<h1>200 Watching Topic.</h1><!--
--><script type="text/javascript"><!--
if(parent.kn_heartbeatCallback)parent.kn_heartbeatCallback(30,self);
// -->
</script><!--
--><script type="text/javascript"><!--
if (parent.kn_sendCallback) parent.kn_sendCallback({elements:[
{
        name : "status",
        value : "200 Watching topic."
},
{
        name : "journal",
        value : "/kn/kn_journal"
},
{
        name : "kn_journal_reconnect_scheme",
        value : "kn_history_since_event_id"
},
{
        name : "kn_id",
        value : "82CC5576-5218-4CE5-AA1F-4AEBA0CD3E62"
},
{
        name : "kn_time_t",
        value : "1052945979"
},
{
        name : "kn_payload",
        value : ""
}
]}, window);
// -->
</script><!--
```

... or this 299-byte **simple**-formatted response:

```
HTTP/1.0 200 Watching Topic.
Connection: close
```

```
Pragma: No-cache
Content-Type: text/plain; charset=utf-8

172
kn_id: 0D7FA6B2-CFDE-4F93-8B40-3D1BDF31260B
kn_time_t: 1052946087
status: 200=20Watching=20topic.
journal: /kn/kn_journal
kn_journal_reconnect_scheme: kn_history_since_event_id
```

An event notification in JavaScript format might look like this 559-byte packet:

```
--><script type="text/javascript"><!--
if (parent.kn_sendCallback) parent.kn_sendCallback({elements:[
{
        name : "kn_history_since_event_id",
        value : "6"
},
{
        name : "kn_route_location",
        value : "/kn/B15FA952-CA2F-46B8-B0AF-2AC1583C9CBA"
},
{
        name : "kn_route_id",
        value : "B15FA952-CA2F-46B8-B0AF-2AC1583C9CBA"
},
{
        name : "kn_routed_from",
        value : "http://sales.knownow.com/kn"
},
{
        name : "kn_id",
        value : "0C6873CB-21A9-4A60-9447-D9A9C7F9DCA4"
},
{
        name : "kn_time_t",
        value : "1052946330"
},
{
        name : "kn_payload",
        value : ""
}
]}, window);
// -->
</script><!--
```

... or this 268-byte packet (**simple**-formatted):

```
246
kn_route_location: /kn/9B17F7E3-231E-4630-A711-A3599159ACFD
```

```
kn_route_id: 9B17F7E3-231E-4630-A711-A3599159ACFD
kn_routed_from: http=3A//sales.knownow.com/kn
kn_id: E53EFDD1-7544-4617-9499-524ED72F95EB
kn_time_t: 1052946194
kn_history_since_event_id: 1
```

In JavaScript format, an event packet not immediately followed by another event packet is flushed by a trailing block of 8192 zero (NUL) bytes.

There is no equivalent for **simple**-formatted tunnels.

Periodically, an otherwise-idle JavaScript-formatted tunnel receives a “heartbeat” string such as the following 129 bytes:

```
--><script type="text/javascript"><!--
if(parent.kn_heartbeatCallback)parent.kn_heartbeatCallback(30,self);
// -->
</script><!--
```

In a **simple**-formatted tunnel, the “heartbeat” string is a single space byte.

Responses to HTTP/0.9 requests eliminate the overhead of the response headers; the LiveServer does not respond using HTTP/1.1. Since HTTP/0.9 support is not fully usable (too many newlines required in the request), HTTP/1.0 is the only fully-supported protocol version.

Using Event Headers

As explained under "The URL Root" on page 22, each URL root event that is processed by the LiveServer contains one or more headers, which send commands and parameters to the LiveServer. Headers are defined in more detail under "The Anatomy of an Event: Headers" on page 25. The LiveServer recognizes a number of headers that have been defined by KnowNow; in addition, you can create your own custom headers. This section provides information on event headers under the following headings:

- "Header Names" on page 33
- "Sending Commands with Event Headers" on page 34

Header Names

Event headers can have one of three types of names:

- Headers that begin with do_.

  Most of these headers are named do_method and are followed by an equals sign and an additional term. Headers starting with do_method are always commands. KnowNow provides a set of do_method headers that you can use; however, although you will be using these headers, the characters do_ and do_method are reserved for KnowNow use; you cannot create your own custom headers that use do_. do_method has a number of values that you can use, as described under "An Overview of do_method Commands" on page 44.

  In addition to the do_method commands, there are also routing command headers, which begin with do_ and are used with do_method=route. These headers provide parameters specifically for the do_method=route command, as described under "route" on page 52.

- Headers that begin with kn_. kn_ headers are usually parameters that are used in conjunction with do_method commands. There are different types of kn_ headers as discussed in this chapter; some are used with routes, some are used to contain data, and some are assigned and used only by the LiveServer.

  As with do_method, KnowNow provides a set of kn_ headers that you can use. All kn_ headers are used in conjunction with do_method commands. Some kn_ headers are required for all events; others are specific to one or more do_method commands; some are optional for some or all do_method commands. As with do_ and do_method, the characters kn_ are reserved for KnowNow use; you cannot create your own custom headers that begin with kn_.

- Custom headers, which are headers that are named uniquely by you and that contain whatever values you wish them to contain. You can write custom headers for applications that are using the LiveServer.

A sample event could include some of these headers as shown in the following example; values in Italic typeface would be replaced with actual values.

```
do_method=add_notify;kn_payload=payload;mycustomheader=customheadercontents
```

Sending Commands with Event Headers

Event headers are sent to the LiveServer as part of a URL that contains the URL root. They are most often sent in sets, especially as some of the headers must be used in conjunction with each other. In the following example, each header is presented on its own line for clarity; normally, the set of headers would be presented in a single HTTP line, with each being separated by a semicolon.

```
do_method=add_notify;
kn_to=/mytopic;
kn_id="987654321";
kn_payload="payloadcontent";
mycustomheader="customheadercontent";
```

In this example, the add_notify command is telling the LiveServer to add or replace a notify event to a topic. kn_to identifies the topic, and kn_id is a unique identification number for this event. kn_payload contains the event's payload, while mycustomheader is a custom header providing additional event information or processing.

Duplicate Event Squashing

Duplicate squashing is the process of throwing away (completely ignoring) duplicate events. Topics and journals both perform duplicate event squashing; however, because journals are a special kind of topic, they have special behaviors that are different from normal named topics, including with duplicate squashing. With journals, duplicates are determined differently than as they are with normal named topics.

Normal named topics identify as a duplicate any event that has the same data in kn_id, kn_payload, and kn_time_t the same data in as a previous event. When such a duplicate event comes in, the topic throws it away.

Journals look at additional headers in the event to determine whether an event is a duplicate. In order to understand the difference, we need to take a look at the different types of headers. There are route headers and data headers. Route headers, which go over routes, are also called hop-by-hop headers (or hop-by-hop routes). The hop-by-hop headers are

- kn_route_* (that is, any header that starts with kn_route_, such as kn_route_location)
- kn_routed_from

Data headers contain information regarding the event, such as kn_payload and kn_id. (There are also headers that are owned and used by the LiveServer, such as all headers starting with kn_event*.)

In general, normal named topics ignore the route headers when considering whether the event is new or has been updated, and look only at the data headers kn_id, kn_payload, and kn_time_t to determine whether the event is a duplicate. If the kn_id, kn_payload, and kn_time_t headers are the same as those of a previous event, the event is considered a duplicate and is squashed. If the kn_id, kn_payload, or the kn_time_t (or all three) are different from a previous event, then the event is considered unique and is not squashed as a duplicate.

In the case of journals, in addition to looking at the kn_id, kn_payload, and the kn_time_t, the journal also looks at the kn_route_location header. The default behavior of journals is this: If the kn_route_location header of an event is the same as that of a previous event (along with the kn_id, kn_payload, and kn_time_t), then the event is considered a duplicate. If the kn_route_location is different from that of a previous event, even if the kn_payload and the kn_id are the same as those in the previous event, then the event is considered to be unique and is not squashed as a duplicate.

Using the HTTP Interface

Your application can use the either the HTTP GET or POST methods to send a request to a LiveServer (to select operations and set LiveServer options.) Your application specifies one of the do_method values listed under "An Overview of do_method Commands" on page 44, along with any of the required kn_ headers for the desired operation. For most requests, the LiveServer returns a status event that indicates the success or failure of a previous request.

For information on making Web services requests, see Chapter 3 , "Web Services."

This section provides information on using HTTP commands under the following headings:

- "Sending Commands to the LiveServer from a Browser" on page 36
- "POST" on page 37
- "GET" on page 37
- "Rules for Using Headers" on page 37

Sending Commands to the LiveServer from a Browser

It is sometimes useful to use your browser to send commands to the LiveServer—for example, when debugging your application. To do this, just browse to a URL that contains a command. Here are some examples.

Note: In the examples below, replace localhost:8000 with the host name and port number for your LiveServer; for example, "host1.example.com:8000."

- Publish an event containing the payload "100" to the topic /what/prices/ibm

```
http://localhost:8000/kn/what/prices/ibm?do_method=notify;kn_payload=100
```

or

```
http://localhost:8000/kn?do_method=notify;kn_to=/what/prices/ibm;kn_payload=100
```

- Create a route between topicA and topicB

```
http://localhost:8000/kn/topicA?do_method=route;kn_to=/topicB
```

or

```
http://localhost:8000/kn?do_method=router;kn_from=/topicA;kn_to=/topicB
```

- A journal is a topic that ends in /kn_journal. It represents a particular user's long-lived connection to the LiveServer. A subscription is just a route from the sub-

scribed-to topic, into the journal. So, to subscribe the user at /who/anonymous/s/10023/kn_journal to the topic /what/prices/ibm:

```
http://localhost:8000/kn/what/prices/ibm?do_method=route;kn_to=/who/anonymous/s/10023/kn_
journal
```

or

```
http://localhost:8000/kn?do_method=route;kn_from=/what/prices/ibm;kn_to=/who/anonymous/s/
10023/kn_journal
```

- Same as above, but with kn_history_since_n of 10

```
http://localhost:8000/kn/what/prices/ibm?do_method=route;kn_to=/who/anonymous/s/10023/kn_
journal;kn_history_since_n=10
```

or

```
http://localhost:8000/kn?do_method=route;kn_from=/what/prices/ibm;kn_to=/who/anonymous/s/
10023/kn_journal;kn_history_since_n=10
```

POST

The HTTP POST method is preferred for some requests (route and notify), as it avoids lengthy URLs.

If an HTTP POST request does not specify the do_method header, and other headers are provided, the LiveServer treats the POST request as a notify request.

GET

The GET method is preferred for request types whose results are intended to be cached for long periods of time, such as lib, blank, and whoami.

If an HTTP GET request does not specify the do_method header, and other headers are provided, the LiveServer treats the GET as a route request. If an HTTP GET request specifies no parameters at all, the LiveServer treats it as a help request.

Rules for Using Headers

- For HTTP POST requests, headers are sent as POST data in application/x-www-form-urlencoded format.
- For HTTP GET requests, headers are sent as URL query strings in application/x-www-form-urlencoded format.
- Header names and values use the UTF-8 character set encoding.

- Path information, the part of the URL path following the server URL, can be used to specify a default topic for route and notify requests, but is not considered to be a parameter.
- To select the desired LiveServer operation or option, your application can use the headers described under “An Overview of do_method Commands” on page 44.
- Headers described under “Globally Supported kn_ Headers” on page 59 are available as event headers and, except where otherwise noted, are propagated along routes.
- With the exception of header names starting with kn_, headers not listed here are generally available for use by your application as application-specific event headers.
- All header names starting with do_ and kn_ are reserved for use by the LiveServer.
- Headers described under “Status Event Headers” on page 60 are used in status events generated by the LiveServer to indicate the success or failure of a request.

Tunnel Route Formats

LiveServer messages sent over tunnel routes can be formatted in various ways. The tunnel route format is set by **kn_response_format**. The formats are

- Simple, encoded as **simple**. This format is used by all Connectors other than the JavaScript Connector (which uses the **js** format). It is an x-www-form-url encoded HTTP message. With this format, the LiveServer passes events as header/value pairs. This format can also be used for offhost routes, in which case each event is one HTTP message with the header/value pairs in the body. For more on this format, see "The simple Tunnel Route Format" on page 39.
- JavaScript, encoded as **js**. This format is used over a tunnel route to pass events to a JavaScript Connector that is running in a browser. The JavaScript tunnel route format knows to pad events to allow notification on some Web browsers; this is the way to accommodate a browser optimization that only renders Web pages after some specific amount of data is received. Connectors using the JavaScript tunnel route format have the ability to adjust the amount of event padding. For more on this format, see "The js Tunnel Route Format" on page 40.

The **simple** and **js** tunnel route formats send keepalives occasionally to maintain their persistent connections with LiveServer. All tunnel route keepalives are a single byte. By setting the **kn_expires** header, a client can request that the LiveServer close a tunnel route at a certain time.

The simple Tunnel Route Format

The **simple** tunnel route format delivers events in a simple format resembling RFC-822, possibly with whitespace padding before and after encoded events. The sample event described under "The Anatomy of an Event: Headers" on page 25 would be formatted as follows:

Example 2-1. simple event format.

```
170
header1: value1
header2: =20value2=0AJapanese: チヤシト
header=0Athree=20is=20=00long: 1
kn_id: 979273892_861_2249

This is the kn_payload.
This is Greek: κοσμε
```

Each encoded event is preceded by a decimal byte count (not a character count) for the encoded form and a linefeed (character 10, U+000A). Each event is followed by a linefeed. These are not part of the encoded event, and are not included in the byte count. So, for example, the "layout" of an event would be as follows:

Example 2-2. Layout of an event.

```
Byte count plus a linefeed character (not included in the event's byte count)
The event (one or more lines, all included in the byte count)
A following linefeed (not included in the byte count)
```

When encoded, the sample event in Example 2-1 is 170 bytes.

Encoded events include a sequence of zero or more headers followed by a linefeed and the **kn_payload** value. Each header in an encoded event consists of a header name, a colon, a single space, a header value, and a linefeed. The header names and the header values may both include bytes which are encoded in MIME Quoted-Printable style (in other words, =*XX*, where *XX* is the uppercase hexadecimal representation of the quoted byte value).

The LiveServer will always escape the following bytes: colon (U+003A), the equals sign (U+003D), and leading and trailing spaces (U+0020) in header names and values. Other bytes may be escaped, as in the above example, where all control characters are escaped. UTF-8 sequences of two or more bytes may have none, some, or all bytes escaped.

The list of headers never includes kn_payload. The kn_payload value is not encoded in MIME Quoted-Printable style.

Header names, header values, and the kn_payload value are UTF-8 encoded. In Example 2-1, チャシト is encoded (and counted) as the 12 bytes 0xE3, 0x83, 0x81, 0xE3, 0x83, 0xA3, 0xE3, 0x83, 0x83, 0xE3, 0x83, 0x88 and κοσμε is encoded (and counted) as the 11 bytes 0xCE, 0xBA, 0xE1, 0xBD, 0xB9, 0xCF, 0x83, 0xCE, 0xBC, 0xCE, 0xB5.

The js Tunnel Route Format

The **js** tunnel route format generates HTML with JavaScript callbacks for event delivery and for dynamic document handling.

- Event Format describes the JavaScript format used for event delivery.
- "Callbacks" on page 42 lists the supported callbacks.

Event Format

The LiveServer packages events in JavaScript event objects when using the **js kn_response_format**. Each event object includes an array of <INPUT>-style headers with name and value strings corresponding to UTF-16 encoded event header names and values.

If a particular header name or value has not already been decoded from UTF-8 into UTF-16 by the LiveServer, the letter U will be appended to the property name, making it nameU or valueU.

The UTF-8 encoded string must include JavaScript UTF-16 characters with values corresponding to the bytes in the UTF-8 sequence; that is, it must be encoded as UTF-8 and then represented using one JavaScript character per UTF-8 byte. The LiveServer might choose to provide only a UTF-8 version of a particular string, it might choose to provide only a UTF-16 version, or it might choose to provide both UTF-8 and UTF-16 versions. The sample event described under "The Anatomy of an Event: Headers" on page 25 would be structured as shown in Example 2-3.

Example 2-3. Event example.

```
event =
{
    elements:
    [
        {
            nameU: "header1",
            value: "value1"
        },
        {
            name: "kn_payload",
            valueU: "This is the kn_payload.\nThis is Greek: " +
                "\xce\xba\xe1\xbd\xb9\xcf\x83\xce\xbc\xce\xb5"
        },
        {
            nameU: "header2",
            valueU: " value2\nJapanese: " +
                "\xe3\x83\x81\xe3\x83\xa3\xe3\x83\x83\xe3\x83\x88"
        },
        {
            name: "header\nthree is\0long",
            value: "1"
        },
        {
            name: "kn_id",
            nameU: "kn_id",
            value: "979273892_861_2249"
```

```
            valueU: "979273892_861_2249"
        }
    ]
}
```

Typical usage of an event is shown in Table 2-2.

Table 2-2. Typical usage of an *event*.

Command	Action
event.elements.length	Number of header name-value pairs in an *event* (five, in this example)
event.elements[0].nameU	UTF-8 encoded name of the first header in an *event* ("element1," in this example)
event.elements[1].name	UTF-16 encoded name of the second header in an *event* ("kn_payload," in this example)
event.elements[2].valueU	UTF-8 encoded value associated with the third header in an *event* ("value2\nJapanese: \xe3\x83\x81\xe3\x83\xa3\xe3\x83\x83\xe3\x83\x88," in this example)

Callbacks

The LiveServer communicates with JavaScript-enabled Web browsers through the following callbacks:

- parent.kn_tunnelLoadCallback(window), which is called when a tunnel route connection (as produced by do_method=route with no kn_to or a javascript: kn_to) is closed.
- parent.kn_sendCallback(event, window), which is called whenever an event is received. The format of an event is described under "The Anatomy of an Event: Headers" on page 25.
- parent.kn_redrawCallback(window), which is called while do_method=blank is loading. Used by the KNDocument class to render HTML pages dynamically.

The **window** parameter passed to all three callbacks can be used to determine which frame a callback was called in.

Note: It is recommended that JavaScript applications **not** use these callbacks directly. Instead, they should use the interface described in the JavaScript Connector Library.

An Overview of do_method Commands

The do_method commands send commands relating to topics, routes, and other aspects of the LiveServer's functioning. The following sections provide information on these commands:

- "The Format of do_method Commands" on page 44, which describes what do_method commands look like and how they are used
- "The Categories of do_method Commands" on page 44, which provides some overview information on do_method commands
- "kn_ Headers and do_method Values" on page 45, which describes how these two types of headers interact

For an alphabetical listing of all the do_method commands, see "do_method Command Reference" on page 49.

The Format of do_method Commands

The do_method commands are all used in the form do_method=*command*. The commands are all defined. The names of the commands indicate their purpose. For example, the add_topic command is for creating a new topic, and would be used like this:

```
do_method=add_topic
```

In addition, the do_method commands are often used in conjunction with kn_ headers, which are also all defined and which perform additional tasks, such as setting parameters. For example, the kn_to header can be used to specify a topic or location or route, depending on which do_method command it is used with.

Starting on page 49, the do_method commands are listed in alphabetical order, with descriptions of their functions and their related kn_ headers, as well as other needed information.

The Categories of do_method Commands

The do_method commands fall into two types of categories: command classes and the miscellaneous class.

Command Classes

There are four LiveServer do_method command classes. These classes affect different aspects of events, topics, and routes, such as adding, deleting, or modifying them. All the values in the command classes support status events.

- The batch class, which contains the batch command. With the exception of add_journal, any of the command classes can be included inside a batch command (or, in the case of the batch command itself, inside another batch command).
- The route class, which contains commands relating to routes: add_journal, add_route, delete_route, route, and update_route.
- The notify class, which contains commands relating to event notifications: add_notify, delete_notify, notify, and update_notify.
- The topic class, which contains commands relating to topics: add_topic, clear_topic, delete_topic, and set_topic_property.

From the point of view of permissions, the following classes also are permissions classes: route, notify, and topic. For more information on permissions, see the *KnowNow LiveServer Administration Guide*.

Miscellaneous Class

The miscellaneous class contains commands for performing other functions, such as obtaining a help document. The commands in the miscellaneous class are blank, help, lib, test, and whoami.

kn_ Headers and do_method Values

As mentioned on page 44, most kn_ headers provide parameters for do_method commands. Not all kn_ headers can be used with all do_method commands.

- Some kn_ headers are supported by all do_method commands. These globally supported headers are listed under “Globally Supported kn_ Headers” on page 59. Some of these are generated with default values if you do not explicitly specify them.
- Of the kn_ headers that are not globally supported, some are supported for use in conjunction with one or more of these do_method values; these are referred to as locally supported kn_ headers. Some are supported only by a single do_method command.

For each do_method command listed in the “do_method Command Reference” on page 49, the associated kn_ headers are also discussed. In addition, you can find organizational and conceptual information on the kn_ headers starting under “An Overview of kn_ and Other Headers” on page 56.

All do_method values that contain the word **topic** or **notify** in their names look at the kn_to parameter to see if the destination of the request is on the current LiveServer. If the request is not on the current LiveServer, the request is forwarded to the proper LiveServer.

All route-class commands look at the kn_from parameter to see if the source of the route is on the current LiveServer. If it is not, the request is forwarded to the proper LiveServer. The LiveServer will proxy the request and then return the response information from the destination LiveServer.

Table 2-3 provides a list of all the do_methods with the kn_ headers that they use. It also lists other uses, users, and providers of the kn_ headers. For the reverse version of this table, see Table 2-4 on page 56.

Table 2-3. Summary of the do_method commands (and other users) and their associated kn_ headers.

do_method Command or Other Uses/Users	Associated kn_ Headers
add_journal	kn_response_format, kn_status_from
add_notify	kn_provider_id, kn_provider_name, kn_response_format, kn_response_uri, kn_status_from, kn_to
add_route	kn_deletions, kn_from, kn_module, kn_response_format, kn_status_from, kn_to
add_topic	kn_atomic, kn_default_kn_expires, kn_filtername, kn_filteroptions, kn_filterparams, kn_response_format, kn_temporary, kn_temporary_expires, kn_status_from, kn_to
All do_method commands	kn_expires, kn_id, kn_payload, kn_timer_*, kn_time_t
All tunnel route creation commands	kn_response_flush, kn_response_flush_interval
Assigned by the LiveServer	kn_event_id, kn_route_id
Assigned by the LiveServer; used with all do_method commands	kn_routed_from, kn_route_location

Table 2-3. Summary of the do_method commands (and other users) and their associated kn_ headers. *(continued)*

do_method Command or Other Uses/Users	Associated kn_ Headers
batch	kn_batch, kn_response_format, kn_status_from
clear_topic	kn_default_kn_expires, kn_filtername, kn_filteroptions, kn_filterparams, kn_response_format, kn_status_from
Clusters: Holds the values for the state of cluster nodes	kn_route_cluster_state
delete_notify	kn_response_format, kn_status_from
delete_route	kn_response_format, kn_status_from
delete_topic	kn_default_kn_expires, kn_filtername, kn_filteroptions, kn_filterparams, kn_response_format, kn_status_from
notify	kn_deleted, kn_provider_id, kn_provider_name, kn_response_format, kn_status_from, kn_status_to, kn_to
request/response actions	kn_block, kn_request_id, kn_request_manager, kn_request_response
route	kn_connection, kn_deletions, kn_filtername, kn_filteroptions, kn_filterparams, kn_from, kn_history_*, kn_hold_new_events, kn_owner, kn_request_format, kn_status_from, kn_to
route class commands that don't apply to tunnel routes	kn_uri
set_topic_property	kn_atomic, kn_default_kn_expires, kn_deleted, kn_filtername, kn_filteroptions, kn_filterparams, kn_max_queue_size, kn_response_format, kn_status_from, kn_temporary, kn_temporary_expires, kn_to, kn_topic_allow_update, kn_topic_nodelete, kn_topic_nopersist, kn_topic_nopost, kn_topic_order

Table 2-3. Summary of the do_method commands (and other users) and their associated kn_ headers. *(continued)*

do_method Command or Other Uses/Users	Associated kn_ Headers
update_notify	kn_deleted, kn_provider_id, kn_provider_name, kn_response_format, kn_response_uri, kn_status_from, kn_to
update_route	kn_deletions, kn_from, kn_response_format, kn_status_from, kn_to

do_method Command Reference

The do_method commands are listed in this section in alphabetical order. All these commands support the headers listed under "Globally Supported kn_ Headers" on page 59. In addition, some of these do_method commands support some subset of the kn_ headers that are described under "kn_ Header Reference" on page 62; see Table 2-3 on page 46 for a summary of the do_method commands and a full list of the kn_ headers that are used with each. In some cases, when used in conjunction with a specific do_method, certain kn_ headers have specific meanings. Where such specific information applies for a kn_ header for any do_method command, that information is briefly supplied with the do_method command. For a complete description of each kn_ header, see the kn_ header reference pages.

add_journal

Member of the route class. Supports status events.

Adds a journal. This is the same as using do_method=route with a null kn_to or using a kn_to that starts with JavaScript.

- kn_from. Provides the name of the journal.
- kn_response_flush and kn_response_flush_interval, which set a flush buffer and interval. These are used with tunnel route creation commands.
- kn_response_format. Selects a tunnel route format.

add_notify

Member of the notify class. Supports status events.

Adds or replaces a notify event to a topic (same as the do_method=notify command).

- kn_response_format. Selects a tunnel route format.
- kn_to. Provides the route's destination.

add_route

Member of the route class. Supports status events.

Adds or replaces a route. This is the same as the do_method=route command, except that add_route will not create journals or delete routes, depending on the kn_to values.

- kn_deletions.Sends a deletion event.

- kn_from. Provides the starting topic to start the route from.
- kn_history_since_event_id. For details on this header, see the discussion under "kn_history_*" on page 67.
- kn_response_flush and kn_response_flush_interval, which set a flush buffer and interval. These are used with tunnel route creation commands.
- kn_response_format. Selects a tunnel route format.
- kn_to. Provides the URI for the end point of the route.

add_topic

Member of the topic class. Supports status events.

Adds a new topic to the LiveServer.

- kn_atomic. Sets a topics property as described under "kn_atomic" on page 62.
- kn_response_format. Selects a tunnel route format.
- kn_temporary and kn_temporary_expires. Make a topic temporary.
- kn_to. Provides the name of the topic to create.

batch

The only member of the batch class. Supports status events. Most Connectors supply an API command, batchPost, that supports this functionality.

Sends multiple route requests, notify requests, or some of each type. Each request is contained in its own kn_batch parameter. do_method=batch generates a status event for the batch request itself as well as a status event for each request contained in the batch. All status events from the batched events use thekn_response_format and kn_status_to parameters supplied with the top-level batch request.

The kn_batch parameter is batch request in application/x-www-form-urlencoded format; for example,

```
do_method=notify;kn_to=/what/chat;kn_payload=hi%20there
```

Several kn_batch parameters can occur in a single batch request.

- kn_batch. Contains a single request.
- kn_response_format. Selects a tunnel route format.
- kn_status_to. Location to which to send status events resulting from the batched set of commands.

blank

Member of the miscellaneous class. Does not support status events.

Returns an empty HTML document with a JavaScript callback. Essential for enforcing the JavaScript *same-domain* security policy. This is useful because JavaScript access to about:blank is sometimes restricted. For more information on cross-domain environments, see "Writing Cross-Domain Web Applications" on page 181.

clear_topic

Member of the topic class. Supports status events.

Deletes all events from a topic on the LiveServer. Return code: 404 Event does not exist. If you wish to delete just one event, use delete_notify. The header kn_topic_nodelete. prevents a topic from being deleted, even if you use this do_method.

- kn_to. The name of the topic to delete all events from.

delete_notify

Member of the notify class. Supports status events.

Deletes a single, specified event from a topic. Return code: 404 Event does not exist. If you wish to delete all events from a topic, use clear_topic.

- kn_id. ID of the event to delete.
- kn_response_format. Selects a tunnel route format.
- kn_to. The name of the topic where the event exists.

delete_route

Member of the route class. Supports status events.

Deletes a route. Return code: 404 Route does not exist.

- kn_from. The name of the topic where the route begins.
- kn_id. ID of the route to delete.
- kn_response_format. Selects a tunnel route format.

delete_topic

Member of the topic class. Supports status events.

Deletes a topic and all events in it. You can also delete a topic by setting kn_expires to "+0". For information on temporary topics, see kn_temporary. The header kn_topic_nodelete. prevents a topic from being deleted, even if you use this do_method.

help

Member of the miscellaneous class. Does not support status events.

Returns a (possibly empty) help document.

lib

Member of the miscellaneous class. Does not support status events.

Returns the JavaScript Connector library and includes the results of do_method=whoami. This service is typically referenced near the beginning of an HTML page to make the page into an interactive Web application. For more information, see Chapter 6 , "Using the JavaScript Connector."

notify

Member of the notify class. Supports status events.

Posts an event to the destination specified by kn_to. If no kn_to parameter is given, kn_to defaults to the path information.

The following headers are of particular interest to notify.

- kn_event_id. Assigned by the LiveServer.
- kn_response_format. Selects a tunnel route format.
- kn_status_to. The topic to which status events are to be redirected.
- kn_to. The topic to publish an event to.

route

Member of the route class. Supports status events.

Establishes a route from kn_from to kn_to.

Every route, including tunnels, has an owner as assigned in the kn_owner header.

You can use the route command to delete a route by using an empty kn_to.

If kn_to is not specified (that is, if it is missing entirely), or if it is a string starting with **javascript:**), this command opens a persistent HTTP connection with a stream of events from the specified source topic, which must be a kn_journal. Not supplying a kn_to is the equivalent of **javascript:**.

Routes are deleted when any of the following are true:

- Their kn_expires has expired.
- They are off host, and the foreign host can't be reached when routing some event.
- Their destination is on host, and it is to a journal that no longer exists.
- Their destination is on host, and it is to a temporary topic that no longer exists.

The LiveServer will **not** delete routes that

- have an on-host destination to normal topics that do not exist
- are still available for delivery

Headers of interest include

- kn_connection. Sets a tunnel format.
- kn_deletions. Sends a deletion event.
- kn_from. Starting point of route. If no kn_from parameter is given, kn_from defaults to the path information.
- kn_history_*. This series of headers makes it possible to retrieve events using various time and age criteria. These initial route publication (IRP) headers are used in conjunction with do_method=route requests to control the immediate transmission of previously-transmitted events along a new route. If multiple IRP headers are specified, the route will be populated with the events which meet all the criteria specified by these headers. A brief discussion of how reconnection works is in order, since it applies to the IRP headers. A client (a Connector or an application) can tell the LiveServer that it is trying to reconnect to a journal it thinks already exists. The LiveServer infers that a journal is being reconnected when any IRP parameter is provided. If a client indicates that it is reconnecting, and the journal does not exist, the client will get a 404 return code. This informs the client that it must create the journal anew (in other words, without a reconnection parameter) and, more importantly, recreate its subscriptions.
- kn_response_flush and kn_response_flush_interval, which set a flush buffer and interval. These are used with tunnel route creation commands.
- kn_response_format. Selects a tunnel route format.
- kn_to. End point of route. If this is blank, deletes the route.

set_topic_property

Member of the topic class. Supports status events.

Modifies a topic's properties on the LiveServer. This is the same as calling update_notify on the topic's corresponding subtopic event. You can also set topic properties using the topic-related kn_ headers through the KnowNow System Administration console as described in the *KnowNow LiveServer Administration Guide*.

- kn_atomic. Sets a topic's property as described under "kn_atomic" on page 62.
- kn_default_kn_expires. Default expiration time for events on this topic. If this topic is a journal, then it is limited by the **Maximum Session Reconnect Time** parameter (which is set using the KnowNow System Administration console as described in the *KnowNow LiveServer Administration Guide*). This parameter limits the maximum amount of time that a journal can be set to maintain events. The default value of **Maximum Session Reconnect Time** is five minutes.
- kn_filtername. The name of a filter module to apply to the topic.
- kn_filterparams. The parameters for the named filter module.
- kn_max_queue_size. The maximum number of events to be stored in this topic.
- kn_response_format. Selects a format for tunnel routes.
- kn_temporary and kn_temporary_expires. Make a topic temporary.
- kn_topic_nodelete. Makes a topic undeletable.
- kn_topic_nopersist. Makes a topic not persistable.
- kn_topic_nopost. Makes a topic so that it cannot be posted to.
- kn_to. The name of the topic for which you are setting the properties.

test

Member of the miscellaneous class. Does not support status events.

Performs a LiveServer self-test and, if okay, delivers HTTP status 200 OK.

update_notify

Member of the notify class. Supports status events.

Updates an existing event. If the event does not exist, a 404 is returned. If the event does exist, only the headers that are set in this request will be changed in the event. All other headers will remain the same. Return code: 404 Event does not exist.

- kn_id. ID of the event to update.

- kn_response_format. Selects a tunnel route format.
- kn_to. The name of the topic where the event exists.

update_route

Member of the route class. Supports status events.

Updates an existing route. If the route does not exist, a 404 is returned. If the route does exist, only the headers that are set in the request will be changed. All other headers will remain the same. Return code: 404 Route does not exist.

- kn_deletions. Sends a deletion event.
- kn_from. The name of the topic where the route begins.
- kn_id. ID of the route to update.
- kn_response_format. Selects a tunnel route format.
- kn_to. The route's destination.

whoami

Member of the miscellaneous class. Does not support status events.

This command is specific to the JavaScript API. It returns JavaScript to set LiveServer- and session-specific window string properties (properties of the **self** object). All returned string properties are optional and UTF-8 encoded. It also returns a document.domain setting when the **Configuration > Service Network > Domain** parameter is set. (This parameter is set using the KnowNow System Administration console.) The window string properties are:

- kn_displayname. Remote user name (default: Anonymous User or Guest User).
- kn_server. URL or path name for the LiveServer (default: /kn). This could be fully qualified. You can override this default by accessing and changing the **Configuration > JavaScript > Domain** parameter in the KnowNow System Administration console. For instructions on using the KnowNow System Administration console, see the *KnowNow LiveServer Administration Guide*.
- kn_userid. Remote user ID (default: anonymous or guest).

For more information on the self object, see Chapter 6 , "Using the JavaScript Connector."

An Overview of kn_ and Other Headers

There are additional headers that are used in conjunction with the do_method commands, generally to set parameters for the commands. These headers generally start with kn_ and fall into the following categories:

- headers that can be used with all do_methods (described in more detail under "Globally Supported kn_ Headers" on page 59)
- headers that are supported by a subset of do_methods or that, in some cases, are specific to one do_method; the do_methods that are associated with each kn_ header are listed in Table 2-4.
- special LiveServer headers; the values of these headers are set by the LiveServer and cannot be changed; for a list of the headers that have such values, see "Special LiveServer Headers" on page 59
- status event headers (described in more detail on "Status Event Headers" on page 60)

All the kn_ headers are summarized in Table 2-4. They are also described in detail in alphabetical order under "kn_ Header Reference" on page 62. Time as used in the kn_ headers is described under "Specifying Time in Headers" on page 60.

kn_ Headers Summary Table

Table 2-4 summarizes the kn_ headers and, if applicable, the do_method commands they are used with. Information for the majority of these kn_ headers is provided in the "kn_ Header Reference" on page 62. (Some information is provided elsewhere as indicated in the table.) For a list of the commands within a given class of do_methods (for example, a list of all route commands), see "The Categories of do_method Commands" on page 44. For the reverse version of this table, see Table 2-3 on page 46.

Table 2-4. Summary of the kn_ headers and their associated do_method commands.

Header	Used with do_method= (or, used for or with...)
kn_atomic	add_topic, set_topic_property
kn_batch	batch
kn_block	Used in request/response
kn_connection	route
kn_default_kn_expires	All do_method commands in the topic class (add_topic, clear_topic, delete_topic, set_topic_property)

Table 2-4. Summary of the kn_ headers and their associated do_method commands.

Header	Used with do_method= (or, used for or with...)
kn_deleted	notify, set_topic_property, update_notify
kn_deletions	add_route, route, update_route
kn_event_id	Assigned by the LiveServer
kn_expires	All do_method commands
kn_filtername	All do_method commands that are in the topic and route classes (see "Command Classes" on page 45)
kn_filteroptions	All do_method commands that are in the topic and route classes (see "Command Classes" on page 45)
kn_filterparams	All do_method commands that are in the topic and route classes (see "Command Classes" on page 45)
kn_from	add_route, route, update_route
kn_history_*	A set of headers that are used with the route command
kn_hold_new_events	route
kn_id	All do_method commands
kn_mailto, kn_mailcc, kn_mailbcc, kn_mailfrom, and kn_mailsubject	Used with the Tcl email filter (documented under "The KnowNow Tcl Email Filter" on page 115)
kn_max_queue_size	set_topic_property
kn_module	add_route
kn_owner	All route commands (see "Command Classes" on page 45)
kn_payload	All do_method commands
kn_provider_id and kn_provider_name	add_notify, notify, update_notify
kn_request_format	Used with route commands to create Web services requests
kn_request_id, kn_request_manager, and kn_request_response	Used in request/response

Table 2-4. Summary of the kn_ headers and their associated do_method commands.

Header	Used with do_method= (or, used for or with...)
kn_response_flush and kn_response_flush_interval	All tunnel route creation commands
kn_response_format	All do_method commands that allow status events
kn_response_uri	add_notify, update_notify
kn_retry	Used in Web services requests
kn_route_checkpoint	Deprecated. Use kn_history_since_event_id instead.
kn_route_cluster_state	Holds the values for the state of cluster nodes.
kn_routed_from	Assigned by the LiveServer; used with all do_method commands
kn_route_id	Assigned by the LiveServer
kn_route_location	Assigned by the LiveServer; used with all do_method commands
kn_status_from	All do_method commands that are in one of the command classes (batch, notify, route, or topic command classes; see "Command Classes" on page 45)
kn_status_to	notify
kn_temporary and kn_temporary_expires	add_topic, set_topic_property
kn_timer_* (kn_timer_fired, kn_timer_interval, kn_timer_nextupdate, and kn_timer_updated)	Used to create and manage periodic events
kn_time_t	All do_method commands
kn_to	add_notify, add_route, add_topic, notify, route, set_topic_property, update_notify, update_route
kn_topic_nodelete, kn_topic_nopersist, and kn_topic_nopost	set_topic_property

Table 2-4. Summary of the kn_ headers and their associated do_method commands.

Header	Used with do_method= (or, used for or with...)
kn_topic_order and kn_topic_allow_update	set_topic_property
kn_uri	All commands in the route class (except those that apply to tunnel routes); see "Command Classes" on page 45

Globally Supported kn_ Headers

Certain kn_ headers are supported by all the do_method commands and for all events. These headers are:

- kn_expires
- kn_id
- kn_payload
- kn_response_format
- kn_routed_from
- kn_route_location
- kn_time_t

Information on each of these kn_ headers is provided in the "kn_ Header Reference" on page 62.

In some cases, you will need to explicitly provide these headers; in other cases, you will not. If you do not explicitly provide these headers, the LiveServer will assign a default value for each of them. For example, if you do not provide a kn_id value when creating a topic, the LiveServer will create one for you. However, if you wish to delete that topic, you will need to use kn_id to explicitly specify the topic's ID.

Special LiveServer Headers

Certain headers have special meaning to the LiveServer. They are set by the LiveServer every time an event passes over a route. As the event traverses the route, the previous values of these headers are removed and new ones may be inserted. These headers are

- kn_event_id
- kn_routed_from
- kn_route_id
- kn_route_location

Status Event Headers

The LiveServer generates status events to indicate the success or failure of any LiveServer commands (any commands that generate a status event). The LiveServer also generates status events for requests with unknown do_method values.

The LiveServer generates separate status events for each request contained within a batch request, as well as a status event for the batch request itself. Where the status event is returned depends on how kn_status_to is set.

If kn_status_to is not specified, the status events are returned in the body of the HTTP response and the status code and the reason phrase of the HTTP response are taken from the status field of the first status event. The kn_response_format parameter determines how the events are represented.

If kn_status_to has been specified, the HTTP response uses 204 No Content or 200 OK as the HTTP status code and reason phrase. In accordance with RFC 2616, there is no response body. Instead, the events are forwarded to the topic whose URL is specified as the value of kn_status_to.

The status event headers are:

- status
- kn_payload
- kn_route_location

status

The status header contains a status code and brief description in HTTP format as a three-digit HTTP reason code followed by a space and an HTTP reason phrase.

- Codes 1*xx*, 2*xx* and 3*xx* codes generally indicate success.
- Codes 4*xx* and 5*xx* generally indicate errors.

If this is the first event being returned over the requesting connection (when kn_status_to was not set and the request is not part of a do_method=batch request), the status will also be sent as the HTTP status code and reason phrase.

Specifying Time in Headers

Some headers use the following characters for time. If no character is specified, then + (in the future) is assumed.

- The plus sign (+) specifies that the given number is an amount of time, in seconds, in the future.

- The minus sign (-) specifies that the given number is an amount of time, in seconds, in the past.
- The at sign (@) specifies that the specific time is in UNIX time, where the starting time is January 1, 1970.

kn_ Header Reference

The kn_ headers are listed in alphabetical order in this section. You can use do_method commands (listed under "do_method Command Reference" on page 49) to change the values of many of these headers. You can also use the KnowNow System Administration console to change these values. The KnowNow System Administration console is described in the *KnowNow LiveServer Administration Guide*.

Note: Any header that starts with kn_event* is owned and set by the LiveServer.

kn_atomic

Use this header to set a topic's atomic property. Any topic for which this is provided (using set_topic_property or add_topic) will cause all data for that topic to be written to disk before any response is given. In other words, if you publish to a topic marked as atomic, then when you receive the 200 OK, you know not only that it was routed to all destinations, but it was also persisted to disk.

kn_batch

Contains a single request in application/x-www-form-urlencoded format. Several kn_batch parameters may occur in a single batch request, with each header separated by semicolons. For example,

do_method=notify;kn_to=/what/chat;kn_payload=hi%20there

kn_block

This header is used only in request/response and is tied to kn_module=kn_request_response.

If a route is marked as a request/response provider, it can be marked as a blocking request/response provider.

For information on request/response, see "Request/response" on page 148.

kn_connection

This header is recognized at tunnel creation; therefore, it is used with route. Its value has the format

typestring[;*version*[;*sigstring*]]

where the *version* and *sigstring* are optional.

If no kn_connection header is provided, and the tunnel type (kn_response_format) is **js**, then the kn_connection value defaults to **LiveBrowser;0.0**. If no kn_connection header is provided, and the response format is **simple**, then the value will default to **Other;0.0**.

kn_default_kn_expires

Applies to creation of a topic and to the creation, updating, or setting of properties on a topic. Sets the default kn_expires value for the topic being modified. Any event that does not already have a kn_expires header that is posted into a topic that has a kn_default_kn_expires header will have its kn_expires set to the value in the topic's kn_default_kn_expires.

The default kn_expires for journals cannot exceed the LiveServer's **Maximum Session Reconnect Time** parameter. Most applications will use relative syntax for this (for example, +300 to mean 5 minutes from post).

Events with immediate expiration are still delivered if possible; they will simply be deleted (by the reaper) at the earliest opportunity. So setting kn_default_kn_expires to zero provides for a topic with minimum "memory."

Our recommendation is to publish events with kn_expires set to **infinity** or other explicit expiration times, rather than relying on LiveServer or topic defaults.

kn_deleted

This header is set to **true** and sent along with deletion events. It can be used anywhere, and can come from the LiveServer without human intervention.

Note that you can get kn_deleted events for events you never received. This can happen if you are receiving events from a topic with a kn_deletions=true route that excludes some events; when those events are deleted, you will still get their deletion events.

kn_deletions

Topics that have routes with the kn_deletions header set send a deletion event along the route any time an event is deleted from the topic. This event contains the same kn_id as the event that is being deleted, an empty payload, and a header set inside it with kn_deleted set to **true**.

You can subscribe to any topic for deletion events using this header. When an event in that topic is deleted, that event is sent to you with the kn_deleted header set to **true**. For example, you could create the three subscriptions shown in Table 2-5.

Table 2-5. kn_deletions example.

Route	Routed From	Route options
1	A	(empty)
2	A	kn_history_since_event_id=<hash>&kn_history_until_time=+0
3	A/kn_routes	kn_deletions=true

Route 1 is the new events route. Route 2 is the old events route. Route 3 is the old events completed route. On this third route, you're looking for the specific event where the kn_uri matches the kn_uri of the second route and the header kn_deleted=**true**. You'll get that as soon as the LiveServer deletes the second route, which it will do implicitly as soon as it has satisfied the kn_history_until_time clause.

kn_event_id

This header is a guaranteed unique value that is assigned by the LiveServer. It is a read-only value (i.e., it cannot be overridden). This header is valid on all LiveServers, clustered or single. It differs from kn_id in that kn_id can be overwritten and clients/Connectors can assign kn_id values.

kn_expires

The date and time when the event or route will expire. This can be expressed in the same format as kn_time_t, or as a positive offset in seconds from the event's arrival time; for example, +15 to expire 15 seconds after arrival. You can also use **infinity** or **now**, as shown in Table 2-6.

Table 2-6. Values for kn_expires

Value	Action
+*n*	A positive offset in seconds from the event's arrival time.
infinity	This event will never expire.

Table 2-6. Values for kn_expires *(continued)*

Value	Action
now	This event is already expired.
timestamp_value	Event timestamp expressed as seconds since Thursday, January 1, 00:00:00, 1970, UTC. By default, this value is set to the event's creation time. An event created on Friday, January 12, 04:31:32 2001 UTC would have a kn_time_t value of 979273892.

If this header is not specified, the event may be subject to a default expiration time configured by the LiveServer administrator.

By setting the kn_expires header, a client can request that the LiveServer close a tunnel route at a certain time. Some tunnels are created with an initial kn_expires so that they will close at a particular time. This is used to simulate polling behavior and to support HTTP clients that cannot access the response body until the response has finished.

When a tunnel route with no kn_to expires, the persistent connection is closed. Any header beginning with kn_route_ (kn_route_id, kn_route_location, and so on) is also removed.

kn_filtername

Use this header to specify the name of a filter module. Use whatever URI you assigned to the filter module when using the KnowNow System Administration console. The filter name must start with /kn_system/filters. For information on setting parameters for filters, see "kn_filterparams" on page 66. For more information on naming filters, see "Module Names" on page 97.

Attaching Filters

This is really a family of headers. Any header which begins with the characters kn_filtername specifies one filter module to be attached to a specific stream of events (i.e., route or topic). When identifying filters, if you wish to attach more than one filter to a route or topic, add an extension to the header to indicate the order in which you wish the filter to be applied. The extension can be anything you like, or nothing. For example, you could use the header

```
kn_filtername.1=/kn_system/filters/one
```

to attach one filter, and

```
kn_filtername.2=/kn_system/filters/two
```

to attach another one to the same stream. Each of these is an "instance" of the filters you are attaching. The characters following kn_filtername identify the instance and specify an execution order (in this example, those instances are **.1** and **.2**).

Filters are executed in alphabetical and numeric order of the instance name. The empty string sorts first. Here are some examples:

```
kn_filtername=/kn_system/filters/.../zzz_filter
kn_filtername.1=/kn_system/filters/.../bbb_filter
kn_filtername.2=/kn_system/filters/.../aaa_filter
```

If these were all applied to the same topic, the filter that is identified with no extension (kn_filtername=) would be applied first. After that, the filter identified with kn_filtername.1 would be applied next, and finally the filter identified with kn_filtername.2 would be applied last.

The first filter to return anything other than KnErrorSuccess stops the chain, which means the event is not propagated.

Detaching Filters

If you wish to remove a given filter, just update the event (route or journal) with the header, setting the values to the null string. For example, in the example above, if you wished to detach just kn_filtername.1, but you wish to keep kn_filtername and kn_filtername.2, update the event with the following headers:

```
kn_filtername.1=
kn_filterparams.1=
```

kn_filteroptions

Contains ownership options for filters. If the value contains the string as=kn_owner on a route filter, then the filter will run as the owner of the route. If it contains as=kn_user, it will run as the user who submitted the request. If no option is provided, the default is as=kn_user. (For the KnowNow modules, see "Filter Modules" on page 99.)

kn_filterparams

Contains any parameters a filter might need. This is a string passed to the filter module when it is attached to the route or topic. Different filter modules (e.g., XSLT, Expr, etc.) have different parameters—and even very different parameter languages—so see the documentation for each specific module for the appropriate values and syntax. (For the KnowNow modules, see "Filter Modules" on page 99.)

Like kn_filtername, this is really a family of headers. Each instance of a filter (see kn_filtername) has its own parameters header, with an extension that matches the extension of the instance of kn_filtername that it applies to. So, for example, when kn_filtername.1=/kn_system/filters/one is attached, it gets its parameters from the header kn_filterparams.1.

kn_from

Specifies the source for any route-class commands; for example, /postit or http://myserver/kn/postit. The LiveServer looks at kn_from for LiveServer commands.

kn_history_*

The LiveServer supports a set of headers whose names start with kn_history_. These headers specify a number of events to read since or before specified times, number of events, or event IDs. Table 2-7 summarizes the kn_history_* headers. For information on how to specify time in these headers, see "Specifying Time in Headers" on page 60. Additional information on these headers follows the table.

Table 2-7. The kn_history_* headers.

Header	Value Type	Sample Value	Meaning
kn_history_since_age	Decimal seconds since 1970 (negative by default). Replaces do_max_age, which is deprecated.	300	Get events that are newer than 300 seconds old.
		-300	Get events that are newer than 300 seconds old.
		@300	Get events that have happened since 5 minutes after 12:00 1 January 1970 GMT.
		infinity	Get all events.
kn_history_since_event_id	Event ID (hexadecimal number). Compares to kn_event_id.	3b4e3340e cb73d129f2 d923a	Get events since the event with this specific kn_event_id.

Table 2-7. The kn_history_* headers. *(continued)*

Header	Value Type	Sample Value	Meaning
kn_history_since_n	Integer number of events. Replaces do_max_n, which is deprecated.	10	Get most recent *n* events.
kn_history_since_time	Decimal seconds since 1970 (absolute by default). Compares to a given time. Supports +/-/@ syntax, with @ as the default.	300	Get events that have happened since 5 minutes after 12:00 1 January 1970 GMT.
		-300	Get events that are newer than 300 seconds old.
		0	Get all events.
kn_history_until_age	Decimal seconds since 1970 (negative by default).	300	Get all events that are older than 300 seconds ago.
		infinity	Don't get any events.
kn_history_until_event_id	Event ID (hexadecimal number).	3b4e3340e cb73d129f2 d923a	Deliver up to and including the event with the given event_id.
kn_history_until_n	Integer number of events.	10	Stop after delivering *n* events.
kn_history_until_time	Decimal seconds since 1970 (absolute by default).	300	Don't get any events after 1970-01-01t00:05:00 GMT.
		0	Don't get any events.

The following information applies to these headers:

- In using these headers, you should have at most one kn_history_since_* header, and at most one kn_history_until_* header. If more than one of each of these kinds of headers is supplied, the LiveServer will just pick one, and there is no way of telling which one it will choose.
- A kn_history_until_* header only makes sense if there is a kn_history_since_* header as well. The kn_history_until_* header is ignored otherwise.
- You can't use **since** to identify times in the future when the LiveServer starts delivering to you.

- The **until** attributes are
  - kn_history_until_age (get all events that are older than a specified age)
  - kn_history_until_event_id (stop at a particular event ID)
  - kn_history_until_n (deliver at most *n* events)
  - kn_history_until_time (similar to kn_history_until_age, but @ based)

  Any route with an “until” attribute will be deleted as soon as its “until” criteria is met or the LiveServer has exhausted the event list for the topic during replay. In other words, you can’t use **until** to identify event times or events in the future when the LiveServer deletes your route; once the LiveServer has finished routing all events, it’s done.

Also note that kn_since_checkpoint is supported, but deprecated. It is preferable to use one of these headers instead.

kn_hold_new_events

This header guarantees that any events posted to the kn_from topic in the route while the initial route publication (IRP) is being processed will not be delivered to the kn_to topic until after the IRP is complete. All initial route population (IRP) events are delivered to the tunnel before any new events are delivered. New notify requests that are routed to that journal will block while waiting for the IRP to complete.

Used with kn_history_since_age and kn_history_since_n.

Example of Use

To illustrate how this header is used and how it is useful, let us say you have a /topicA with 1,000 events in it, and you send this command that does **not** include the kn_hold_new_events header:

```
http://kn.example.com/kn
    /topicA?do_method=route&kn_to=who/me/kn_journal&kn_history_since_age=infinity
```

While you are receiving those 1,000 events, user “they” publishes a new event “event1” into /topicA. In this example, you could receive that event anywhere in the stream of 1,001 events, which may not be what you want.

If, on the other hand, if you send this command that includes kn_hold_new_events:

```
http://kn.example.com/kn/topicA?do_method=route&kn_to=who/me/kn_journal
        &kn_history_since_age=infinity&kn_hold_new_events
```

the event “event1” will not be delivered until after the original 1,000 events.

Note that only the kn_from topic is locked. If "they" publish the event directly to the kn_to topic ("who/me/kn_journal"), it will still be interleaved with the IRP events.

kn_id

This header is generated by your application. It contains an ID for the event. If an event arrives at the LiveServer without an assigned kn_id, the LiveServer assigns one. If you wish to delete a topic, you will need to use kn_id to explicitly specify the ID of the topic to delete. You can also use kn_id in conjunction with other headers to shut down and restart the LiveServer as described under "Shutting Down and Restarting the LiveServer" on page 70.

IDs are unique to events within each topic; if an event is published or routed to a non-journal topic which already contains an event with the same kn_id, the new event is considered an update or possibly even a duplicate. Updated events are moved to the end of the topic's event stream so that a later route request with kn_history_since_n or kn_history_since_age parameters will return the updated events. Duplicate events are ignored as described under "Duplicate Event Squashing" on page 35.

By default, the ID is a pseudo-random string, such as 979273892_861_2249.

You can also use kn_id to control processes dynamically. For information on the Process Manager, see the *KnowNow LiveServer Administration Guide*.

Shutting Down and Restarting the LiveServer

The do_method=shutdown command is no longer functional. Instead, to achieve the same results for individual node shutdowns, post an event with kn_id equal to **restart** to the following topic:

/kn_system/nodes/*nodename*/operations

You can restart all the LiveServers in a cluster by posting the same event to this topic:

/kn_system/operations

You can monitor the state of a node by subscribing to its topic (for example, /kn_system/nodes/*nodename*). By using kn_history_since_n=1, you will see the state of that node, as known by the node you are on, and interpreting the value of the kn_route_cluster_state header in the one event you will receive.

For restarting a cluster (or a single node), the recommended procedure is:

1. Get the list of nodes by subscribing to /kn_system/nodes/kn_subtopics.
2. Get the status of each node by subscribing to /kn_system/nodes/*nodename* with kn_history_since_n=1.

3. Issue an event with a payload of **restart** to /kn_system/operations.
4. Monitor the states of all nodes.

You are guaranteed that the node you are attached to will be among the last nodes to be restarted; also, in a cluster, that node will not restart unless some other node has completed its restart. Use the LiveBrowser's reconnect features to reconnect to your journal (or refresh your screen programmatically) and monitor its restart.

kn_max_queue_size

By default, topics can have an unlimited number of events. You can limit the maximum number of events that are contained within a topic at any given time by setting the kn_max_queue_size property on the topic. If kn_max_queue_size is used, the topic acts as a FIFO (first in, first out) queue where only the last *n* events will be stored. You can use a do_method command to set this property, or you can set the **Max Queue Size** field when creating or editing topics using the KnowNow System Administration console (as described in the *KnowNow LiveServer Administration Guide*).

The following example uses the set_topic_property do_method command to set a maximum event queue length of 50 in the topic /mytopic. In this example, if the topic has 50 unexpired events, and a new event arrives, the oldest event in the queue immediately expires in order to make room for the new event:

```
do_method=set_topic_property;kn_to=/mytopic;kn_max_queue_size=50
```

kn_module

This header is used only in request/response. It only takes kn_request_response as a value; it is ignored if it contains any other values.

kn_owner

In determining whether a route command is valid, the LiveServer uses the permissions of the owner of the route (permissions as granted in the perms file). Every route has an owner as assigned in the kn_owner header.

The kn_owner is assigned by the LiveServer and cannot be overridden by including it as a header in a notify or route request. The assignment is made from the HTTP credentials used for the do_method=route request. Whether the do_method was implicit or explicit makes no difference.

For example, if you send this command

```
http://rcobb:mypwd@knownow.example.com/kn/topicA?do_method=route&kn_to=topicB
```

the owner is rcobb. This is implicitly what is sent when you've been required to sign on to a Web site. If no credentials are required by your LiveServer (for example, you haven't loaded an authentication module, or you've allowed anonymous users to create routes), then the owner assigned will be "" (the null user name).

You can see the kn_owner header delivered from the LiveServer if you subscribe to the /kn_routes subtopic of a topic.

kn_payload

Contains the event's payload, analogous to a message body. If kn_payload is not present, the LiveServer inserts this header with the empty string as the value. Also used as a status event; for more information on status events, see "Status Event Headers" on page 60.

Associated with kn_payload are **content-type** and **deleted**.

- **content-type** supplies MIME-style content type for the data in kn_payload; for example, text/plain, image/gif, text/html, application/octet-stream, or text/xml. In general, all text/* media types should undergo character set and encoding conversion to UCS or Unicode (encoded as UTF-8) before being sent in a request to the LiveServer. The use of content-type qualifiers (for example, charset=utf-8) is discouraged. Since UCS or Unicode (encoded as UTF-8) is already the implied character set and encoding for event element names and values, a charset qualifier is unnecessary.
- **deleted** is a flag to mark deleted events. If the value of the deleted element is the string **true**, an application receiving it should process the event as a notification of deletion. Any other value for this element should be ignored. It is recommended that the kn_payload of a deleted event be empty, and that a deleted event have a kn_expires element with the value +3600.

We recommend encoding all binary (non-Unicode) payloads in base 64 format.

As a status event, ideally, kn_payload contains a detailed and human-readable description of what happened in plain text format. In practice, it is often empty. (It is also optional.)

kn_provider_id

This header is used in request/response.

When a service manager is used, it gets the requests made by the requestor and updates the kn_provider_id field. This field tells the LiveServer which provider to send the request to. When a service manager isn't used, the requestor must set the kn_provider_id field so the LiveServer knows where to send the request. Therefore, even though a provider may be assigned to a topic, the LiveServer doesn't send events to the provider until the kn_provider_id says so.

kn_provider_id can only be used in an add_notify request if there is **not** a Service Manager for the specific provider's topic. Again, this is only relevant to route creation or change.

kn_provider_name

This header is used in request/response. For information on request/response, see "Request/response" on page 148.

kn_request_format

This header is only used with a route command, and is used for creating Web services requests. It has two possible values: ws_soap and ws_rest. For information on creating Web services requests, see Chapter 3 , "Web Services."

kn_request_id

This header is used in request/response. For information on request/response, see "Request/response" on page 148.

kn_request_manager

This header is used in request/response. If kn_module is set to kn_request_response, this header is used to mark a route as belonging to the service manager. For information on request/response, see "Request/response" on page 148.

kn_request_response

This header is used in request/response. It is assigned to kn_module. For information on request/response, see "Request/response" on page 148.

kn_response_flush

All tunnel routes support the related headers kn_response_flush and kn_response_flush_interval. These headers are only applicable to route commands *from* kn_journals; in other words, use them only with tunnel route creation commands.

Usage of these headers is as follows: On any tunnel creation event, the kn_response_flush header specifies the size of the flush buffer in bytes. The flush buffer can be as large as 64KB. The existence of this specification implies that flushing will be done.

The default values of kn_response_flush depend on the tunnel format. For **js** and **simple** tunnel formats, the default values for kn_response_flush are as follows.

- JavaScript-format (**js**) tunnels default to flushing with 4KB buffers.
- Simple-format (**simple**) tunnels default to no flushing.

The flush character is always ASCII NUL (\000).

Once you have specified a flush buffer size, use kn_response_flush_interval to specify a flush wait interval.

kn_response_flush_interval

If you have specified a flush buffer using kn_response_flush, then use kn_response_flush_interval to set the interval in microseconds that an event will wait (for other events to be delivered on that tunnel) before that event is flushed down the pipe.

By default, the value of kn_response_flush_interval is 20000 (.020 seconds).

kn_response_format

Selects a tunnel route format, **simple**, **js**, or **JSON**, for events returned in an HTTP response. For more information on tunnel route formats, see "Tunnel Route Formats" on page 39.

kn_response_uri

This header is used in request/response.

The kn_response_uri is set by the requestor to tell the provider where to send the response. A provider subscribes to a topic and waits for requests. A requestor wants to know something from the provider. It needs a way for the provider to talk to it, so it creates a journal. It then creates a request and sets the kn_response_uri header to point to its journal. It sends that request to the topic. The provider gets the request on the topic and builds a response. It looks at the request for the kn_response_uri header. It then sends the response to that location, and the requestor gets the response through its journal.

kn_retry

This header is used for Web services requests. It specifies how many times the LiveServer should re-send a Web services request before it gives up. The default value is **20**. The LiveServer will retry for the specified number of tries with an exponentially increasing number of seconds between retries.

For example, if kn_retry is set to **8**, the LiveServer will retry for a total of eight times. The first time, it will retry after 1 second, then it will wait 2 seconds and try again, then it will wait 4 seconds and try again, then 8 seconds, then 16, then 32, then 64, and finally 128 seconds.

The LiveServer will not always retry even if you have a number of retries set in this header. The retry attempts depend on the reason for the request not being received. For example, the LiveServer will **not** retry if the reason is a permissions error. It **will** keep retrying if the host is unreachable, for example.

The retry count is posted to the topic specified in the route event header. You can use the Topics panel in the KnowNow System Administration console to browse to that topic and view the number of retries for that event.

For more information on Web services, see Chapter 3 , "Web Services."

kn_route_checkpoint

This header has been replaced with kn_history_since_event_id.

kn_route_cluster_state

The state events for nodes use this header. The four values for this header are:

- startup
- recovering
- active
- offline

kn_routed_from

Absolute URI of the topic that the event was most recently forwarded from. Use of this header by client applications is **not** recommended.

kn_route_id

The kn_id of the route that the event was most recently forwarded along. Use of this header by client applications is **not** recommended.

kn_route_location

A status event header. Initially, this header contains the value of the kn_status_from parameter from the original request that generated this status (if that request included the kn_status_from header). If this event passes through any routes, kn_route_location will be replaced with the URI of the most recent route the event traversed, such as

http://myserver/kn/what/chat/kn_routes/964832125_29689_47

For more information on status events, see "Status Event Headers" on page 60.

For routed events, this URI can be overridden by the kn_uri parameter.

kn_since_checkpoint

Supported, but deprecated. Use one of the kn_history_* headers instead.

kn_status

Used in request/response. In kn_status, the word **done** is a reserved string and must be provided in order for the response to be sent to the requestor. For information on request/response, see "Request/response" on page 148.

kn_status_from

The LiveServer generates status events to indicate the success or failure of route, notify, and batch requests, as well as requests with unknown do_methods. Each such request, whether in a batch or by itself on an HTTP connection, generates a status event. kn_status_from is a value to place in the kn_route_location header of status events, as described under "Status Event Headers" on page 60.

kn_status_to

Applies to all LiveServer commands. The value of this header specifies the topic to which status events are redirected; for example,

/who/anonymous/s/22690909/kn_journal

The kn_status_to value is placed in the kn_route_location element of status events.

If kn_status_to is not specified, status events will be returned in the HTTP response. For more information, see "Status Event Headers" on page 60.

kn_temporary

By default, topics expire like normal events. Their kn_expires header determines when they will be deleted. You can also explicitly delete the topic using do_method=delete_topic (or by setting kn_expires to "+0").

However, in many cases it's impossible to know a specific time when a topic should expire—in other words, it should expire when nothing is using it, but no sooner, but it is difficult to know when that time will be. Temporary topics are useful in this situation. Temporary topics can have a kn_expires of **infinity** (the default for topics other than journals), but, as soon as they have no subtopics, or routes leaving them, they will be assigned a new expiration time (as described under kn_temporary_expires).

You can create a temporary topic in either of two ways.

- You can place the topic in /who/wms, /who/java, or /who/*anystring*/s/ and use the KnowNow System Administration console to set the **Temporary Topic Recognize** parameter to **true** (this parameter is documented in the *KnowNow LiveServer Administration Guide*).
- You can set this header, kn_temporary, to **true** to make a specific topic temporary. If you use this method, you can then place the topic anywhere, instead of just in /who/wms., /who/java, or /who/*anystring*/s/.

kn_temporary overrides any value set in **Temporary Topic Recognize**, so that, for example, if **Temporary Topic Recognize** is set to **false**, you can still use kn_temporary to create a temporary topic, and you can use kn_temporary to create a temporary topic anywhere and not just as a subtopic to /who/wms, /who/java, or /who/*anystring*/s/.

You can also set kn_temporary to **false** for a specific topic in /who/wms, /who/java, or /who/*anystring*/s/, and that topic will not be temporary, even if **Temporary Topic Recognize** is set to **true**.

All the subtopics of a topic are reaped when a topic is reaped. Journals are reaped when their journal reconnect timeouts expire. When you want to create a temporary topic, you can create it as a subtopic of your own /kn_journal. That topic will then go away when your session goes away.

Use kn_temporary_expires to set the expiration time for the temporary topic. You can also use the KnowNow System Administration console to set the **Temporary Topic Expires** parameter to a desired default expiration time.

Table 2-8 summarizes the functions of and differences between the temporary topic headers and the KnowNow System Administration console parameters.

Table 2-8. Headers and parameters for temporary topics.

Header or Parameter	Description
kn_temporary	A header used with add_topic or set_topic_property to make a specific topic temporary.
kn_temporary_expires	A header used with add_topic or set_topic_property to set the expiration time (in seconds) for a specific temporary topic.
Temporary Topic Recognize	A parameter that is set using the KnowNow System Administration console. It specifies that all topics in a specific location are to be considered temporary. Can be overridden with kn_temporary.
Temporary Topic Expires	A parameter that is set using the KnowNow System Administration console. It specifies the default expiration time for temporary topics. Can be overridden with kn_temporary_expires.

Unreaped Routes and Topics and Temporary Topics

Temporary topics have a number of useful applications. For example, you may have decided to create co-dependent temporary topics by creating a route loop between them. This method ensures that if either topic is deleted, the other will (eventually) be. These loops are not deleted by the reaper because they have their uses—as long as applications use explicit deletions (or rely on non-infinite kn_expires values) for at least one topic in the loop.

However, there are situations where you don't create such a loop intentionally, and it causes leaked topics and routes. For example, if you have topic /kn/who/a/s/x/y/z routed to /kn/who/a/s/x/y, both with **infinity** as their kn_expires, then neither of these topics will ever be reaped, even if they form an "island" with no incoming or outgoing routes. Why? Well, "z" has a route, so it's not reaped, and "y" has a subtopic, so it's not reaped.

Example Settings

Assuming for the sake of example that you have a topic named /who/java/myjava, Table 2-9 describes the behavior for conflicting settings of kn_temporary and **temporary_topic_recognize**.

Table 2-9. Comparing kn_temporary and temporary_topic_recognize.

If kn_temporary is set to...	And Temporary Topic Recognize is set to...	Then this will happen
True for /who/java/myjava	**False** (i.e., topics in the specific temporary topic directories are **not** recognized as temporary)	The /who/java/myjava topic is recognized as temporary
False for /who/java/myjava	**True** (i.e., topics in /who/java are recognized as temporary)	The /who/java/myjava topic is not recognized as temporary

kn_temporary_expires

Sets a relative expiration time for a temporary topic. Once the temporary topic has no subtopics or routes, its expiration time will be set based on this value. It overrides the default value set for the **Temporary Topic Expires** parameter, which is documented in the *KnowNow LiveServer Administration Guide*.

For example, if this header is set to 300, the topic will expire five minutes after the topic has no subtopics or routes. If any subtopics or routes are established within those five minutes, the expiration time of the topic will be reset to the value it had previously.

The difference between this header and the KnowNow System Administration console parameter is that this header sets an expiration time for a specific temporary topic, while the KnowNow System Administration console parameter sets an expiration time for all temporary topics.

For information on the time format and possible values, see kn_expires. Note that you do not need to use a plus sign (+) with the time value for this header.

For information on how to specify that a topic is to be temporary, see kn_temporary.

kn_timer_*

Periodic ("timer") events are events which are updated regularly by the LiveServer. The headers associated with periodic events are

- kn_timer_fired
- kn_timer_interval

- kn_timer_nextupdate
- kn_timer_updated

An event can be made periodic by providing a kn_timer_interval header on publication:

```
kn_timer_interval=interval
```

where *interval* is encoded as a positive default-relative KnTime (e.g., +300; default-relative means that 300 implies +300, not 5 minutes after midnight 1/1/1970 GMT; see also "Specifying Time in Headers" on page 60 and "kn_history_*" on page 67).

Upon publication, the header kn_timer_updated is set to the LiveServer time (as in the kn_time header), and the header kn_timer_nextupdate is set to the next LiveServer time for it to fire.

At the first reaper interval after the kn_timer_nextupdate, kn_timer_fired will be set to the current LiveServer time, and kn_timer_nextupdate will be increased by the kn_timer_interval.

If there is no kn_timer_interval, the kn_timer_nextupdate header will be removed from the event.

Using kn_timer_nextupdate without a kn_timer_interval is a way to allow a client application to determine the timing (e.g., based on load). However, any application which relies on its own updates is inherently less reliable than one that relies on the LiveServer's internal update, since an event could be lost in the client application (via a crash, for example), and the new fire time never created. A more reliable way to achieve the same effect is to simply update the kn_timer_nextupdate field while retaining the kn_timer_interval.

A more typical use case for using kn_timer_nextupdate without an interval is to provide for triggering a particular one-time-only operation in the future.

kn_time_t

Timestamp recording the event receipt time. Time is expressed as seconds (with fractional microseconds) since Thursday, January 1, 00:00:00, 1970, UTC format; for example, an event created on Friday, January 12, 04:31:32 2001 UTC would have a kn_time_t value of

979273892.54

The format of this value is nine.six digits; that is, nine digits preceding a decimal point, and six following, when sent by LiveServer. LiveServer accepts values with or without decimal points, and will preserve up to six digits of precision after the decimal point. For example, 9, 9.0, and 9.000000 are all January 1, 00:00:09 1970 UTC.

If not provided by the client, LiveServer will set the value to the event's arrival time. The kn_time_t is *not* updated as LiveServer routes the event.

As LiveServer receives published events, it sets a granularity of 10 milliseconds (1/100 of a second) to kn_time_t. To avoid out-of-order events after a database recovery, you must publish at a rate lower than 100 events per second.

kn_to

Specifies the topic to publish the event to (a route's destination) for a route or notify command; for example, /what/chat, /what/チャシト, /what/κοσμε, http://myserver/kn/what/chat. Basically, kn_to is used to set up routes.

If the kn_to parameter is not specified, kn_to defaults to the path information. LiveServer looks at kn_to for any topic or notify commands.

kn_topic_nodelete

This header is used only with set_topic_property.

If this header is provided with any value other than **false**, it makes it impossible to delete this topic using do_method=delete_topic. It also makes it impossible to clear events out of the topic using do_method=clear_topic.

Events in kn_topic_nodelete topics will still expire, and can be individually deleted by updating their expiration times.

kn_topic_nopersist

This header is used only with set_topic_property.

If this header is provided with any value other than **false**, it directs the LiveServer to not persist the specified topic. If any events are in the topic before this property is added, those events will persist until they expire. This is probably not what you want, so you may wish to issue the set_topic_property command before posting events to the topic.

kn_topic_nopost

This header is used only with set_topic_property and is used by the LiveServer's internal statistics module.

This header makes the specified topic impossible to notify into from outside of a LiveServer (in-process) module. If a regular topic has a route to a kn_topic_nopost topic, any normal notifies (posts) into that regular topic will **not** appear in the kn_topic_nopost topic.

kn_topic_order

This header is used to specify a topic order. it supports four values:

- **none**
- **causal**
- **agreed**
- **safe**

ASLC applications will typically need **agreed** or **safe** topics.

kn_topic_allow_update

The value of this header can be either **true** or **false**. If **false**, events in this topic may not be updated; attempts to submit an event with a matching kn_id but non-duplicate data will be rejected with a 40x (not permitted) error message if directly submitted. If a route attempts to do so, the route will be in error and be deleted.

kn_uri

Sets the kn_route_location for all events on a route. If kn_uri is set as part of the HTTP header, it overrides the value set in kn_route_location.

Detecting Presence or Deletions

The LiveServer thinks someone is present if:

1. There is a journal for that person (/kn/who/*person*/.../kn_journal (where ... can be any number of subtopics).
2. There is an active tunnel routing out from that journal.
3. The active tunnel contains the kn_user_id or userid of that person.

Applications can monitor tunnel creation and destruction by subscribing to topics. In particular, to do so, subscribe to /kn/who/kn_subtopics to find out when new journals are created. Then subscribe to each journal's /kn/who/*person*/kn_routes to see their tunnels arrive and depart.

Presence can be monitored as described under "Presence" on page 144.

Sensing Deletions

To forward deletion notifications along a route, add the kn_deletions header with the value **true** to the route. Deletion notifications are distinguished from other notifications by their additional kn_deleted header with the value **true**.

Additionally, many applications use an update with an empty kn_payload to mean "deleted."

For correct behavior in the presence of expiration, deletion, and updates, applications are advised to observe the following guidelines:

1. Publish your own (accurate) timestamps, preferably with floating-point accuracy, in kn_time_t.
2. Sort incoming events by timestamp.
3. An incoming event with a more recent timestamp, or with an equal timestamp but received later, may be considered an update to an earlier event with the same value for the kn_id header.
4. Do not propagate updates that don't differ from the most recent previous value, or that differ only in the hop-by-hop headers (kn_routed_from and any headers that have names starting with kn_route_).
5. When the most recent update to a particular kn_id has an empty kn_payload or the value **true** in its kn_deleted header, that kn_id is considered deleted or expired.
6. Whenever possible, set event expiration times to the values your application needs. These kn_expires values may be specified in absolute terms (a la kn_time_t) or in

relative terms (for example, '+30' for a half-minute lifetime.) Note that the special kn_expires value of **infinity** or '**+infinity**' is used to indicate that an event should be kept as long as possible.

7. Subscribers should obey expiration times on events they receive.
8. To keep system-wide inconsistencies to a minimum, try to ensure that no two publishers will send updates with a particular kn_id during a short time interval ("short" here is probably five minutes or so in the worst case, 30 seconds in more usual cases). Ideally, build applications so that they do not use updates.
9. To "delete" an event, publish an update to the appropriate kn_id with an empty kn_payload value, and a kn_expires value of '+255' or so. (255 seconds is merely a heuristic, not some magical limit. It should be enough time for an update to propagate to a reasonable majority of endpoints.)

Chapter 3 Web Services

In addition to being a Web server, the LiveServer also supports Web service requests to other LiveServers or to other Web services. You can make SOAP and REST Web service requests using the LiveServer as described in this chapter under the following headings:

- "Overview" on page 86
- "Making Web Services Requests" on page 87
- "The LiveServer's SOAP and WSDL Interfaces" on page 93

Overview

Most Web services are implemented using SOAP or HTTP. In addition to supporting HTTP, the LiveServer also supports simple object access protocol (SOAP). SOAP is a clean, lightweight, XML-based protocol for exchange of information in a decentralized, distributed environment. The SOAP protocol consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules, and a convention for representing remote procedure calls and responses.

The SOAP standards documentation can be found at http://www.w3.org/TR/soap/.

A Web services description language (WSDL) defines a SOAP interface. For the XML document that describes the LiveServer's WSDL interfaces, see http://www.w3.org/TR/wsdl.

In addition, the LiveServer also supports REST (representational state transfer) requests. REST isn't a standard but an architectural style for accessing Web services.

Using SOAP, WSDL, and REST, you can use the LiveServer as a proxy for making Web services requests to, for example, Google or Amazon.com.

Making Web Services Requests

If you wish to use the LiveServer to make and manage requests to other Web services, you can do so following these basic steps:

1. Create offhost routes to other Web services. There are different ways to do this depending on which interface (SOAP or REST) you wish to use.
2. Create your requests.
3. Manage the responses to those requests.

The LiveServer ships with some samples you can look at to see how these tasks are put together; this section describes the individual tasks under the following headings:

- "Creating Offhost Routes" on page 87
- "Creating Requests" on page 89
- "Managing Your Responses" on page 91

Creating Offhost Routes

The first step in making and managing requests to other Web services is to create an offhost route to each Web service. In this case, an offhost route is a route from a topic on the LiveServer to a URL off the host (i.e., to a Web service).

In creating each route,

- Use a route creation command (do_method=route, for example).
- In kn_from, use the topic you are routing from.
- Assign the offhost URL to kn_to. (The kn_to header identifies a machine to use—i.e., the destination, a protocol to use, and a format to use.) Note that if you do not assign a URL in the kn_to, the route you are creating is assumed to **not** be an off-host route. The offhost URL can be one of the following:
  - The URL of the WSDL (for example, http://api.google.com:80/GoogleSearch.wsdl). If you are creating a SOAP request where the request is only in the form of header/value pairs as described under "Letting the LiveServer Generate the SOAP" on page 90, use this option. (If the other Web service is another LiveServer, you can obtain the WSDL for that LiveServer as described under "Obtaining WSDLs" on page 88.)

  - The service URL as defined in the Web service's WSDL (for example, http://api.google.com:80/search/beta2).
    - If you are creating a SOAP request to be forwarded to the Web service in the payload as described under "Creating Your Own SOAP" on page 90, use this option.
    - If you are creating a REST request, this is the only option you have for the kn_to. Use this option by entering the Web service's REST URL. For more on REST requests, see "Creating REST Requests" on page 91.
- Use kn_request_format to specify which format the request is in (ws_soap for SOAP, ws_rest for REST). If you do not supply a kn_request_format, it is assumed that the request is a LiveServer request in LiveServer format (which could be LiveServer, Mod-pubsub, and so on).
- If you wish to pass permissions (i.e., user names and passwords), use kn_user and kn_password, as well as the appropriate permissions operations defined in the WSDL for that host. Permissions set this way will apply to any requests made to that offhost route/Web service.

Once you've created the offhost route, you can use the KnowNow System Administration console to check the status of the route. (For instance, to see if the route creation failed or was successful.) For example, the kn_offhost_retries parameter in the /from topic (whatever you have called that topic; you could also call it /publish or /requests or whatever suits you) shows how many connection retries have been attempted. The /status (or whatever you have called it, such as /responses) topic contains status properties.

Obtaining WSDLs

A WSDL provides, among other information, the names of the operations that you can perform with the Web service that provides that WSDL. Therefore, in order to construct Web services requests for a given provider, you will need to obtain the WSDL for whatever server is supplying the Web services. For providers other than KnowNow, you will need to obtain the WSDL directly from those providers.

If you are using another LiveServer to supply Web services, you can obtain that LiveServer's WSDL by sending the following request to it:

```
http://localhost:8000/knSOAP?wsdl
```

As needed, you can use other machine or domain names (or an IP address) and other ports, and the HTTPS protocol.

Here is a sample WSDL request to which a LiveServer will respond by sending its WSDL interface:

```
GET /knSOAP?wsdl
```

Route Creation Examples

The LiveServer ships with GoogleSpelling.cpp , a small C++ code sample showing, among other things, how to make a search request to the Google and BabelFish Web Services using SOAP. In that sample, the offhost route for a SOAP request is created like this (remembering that this is just a snippet from that code and is not complete code in itself):

```
Message route;
route.Set("do_method", "route");
route.Set("kn_from", fromtopic);
route.Set("kn_to", "http://api.google.com:80/search/beta2");
route.Set("kn_request_format", "ws_soap");
```

And here is a snippet of code (from the same sample file) for creating the route for an Amazon.com REST request:

```
Message route;
route.Set("do_method", "route");
route.Set("kn_from", fromtopic);
route.Set("kn_to", "http://webservices.amazon.com/onca/xml?Service=AWSECommerceService");
route.Set("kn_request_format", "ws_rest");
```

Creating Requests

Once you have created the offhost route, you make the actual service request by posting an event to the topic you are creating the route from (i.e, the kn_from topic).

This event must have different types of information in it, depending on what kind of request you are making. However, each event that you post needs to specify the name of the topic where you wish to receive your responses. Use kn_status_to to specify that topic name. Your Web service response will be posted to kn_status_to, so you need to subscribe to that topic if you want to pick up those responses.

The different requirements for different types of requests are described in these sections:

- "Creating SOAP Requests" on page 90
- "Creating REST Requests" on page 91
- "Retrying Requests" on page 91

Creating SOAP Requests

There are two ways to create SOAP requests, depending on whether you want to use your own SOAP generation tool, or whether you would instead rather have the LiveServer generate the SOAP.

Creating Your Own SOAP

If you wish to provide your own SOAP, first, generate the SOAP request using your favorite SOAP generation tool. Then place that entire SOAP request in the kn_payload of your request. See "Creating Offhost Routes" on page 87 for related information on creating the offhost route for this kind of request.

Letting the LiveServer Generate the SOAP

If you want the LiveServer to generate the SOAP for you, post the operation and parameters needed for the Web services request as header/value pairs to the LiveServer. If you wish to pass permissions parameters, use the appropriate WSDL operations as needed.

You must create the headers using the following formats:

- ws_*operationname*
- ws_*parameter1*
- ws_*parameter2*
- and so on

In naming these, use the formulas

```
ws_ + the Web service's operation name
ws_ + one of the Web service's parameter names
```

To obtain the operation and parameter names, refer to that Web service's WSDL. (Case is not important.)

For example, in SOAPRequestor.cpp, in invoking Google's Spelling Suggestion Service, these header/value pairs are specified like so:

```
m.Set("ws_operation", "doSpellingSuggestion");
m.Set("ws_key", "sJu/2PNQFHJ2+0Z6JBQhtcbyh1Vg4dBW");
m.Set("ws_phrase", "knownow");
```

where **operation**, **key**, and **phrase** have been defined by Google's WSDL for Google's Spelling Suggestion Service.

In addition, the kn_to of the route you create needs to point to the WSDL of the Web service you are sending the request to. See "Creating Offhost Routes" on page 87 for related information on creating the offhost route for this kind of request.

Then, create a regular event. The operations and the Web services parameters go into headers. The LiveServer will fetch the WSDL from the WSDL URL you provided in the kn_to, will parse the WSDL using the header/value pairs you supplied, and will generate the SOAP, which it then sends to the Web service. You will also need to specify a kn_status_to as described under "Managing Your Responses" on page 91.

Creating REST Requests

LiveServer REST requests don't take a payload. Instead, they take the REST parameters prepended with ws_, just as described under "Letting the LiveServer Generate the SOAP" on page 90. These parameters are accepted by REST Web services as header/value pairs. If you want to use a Web site's REST protocol, you will need to know what header/value pairs are accepted by that Web service. Normally, each REST Web site documents its header/value pairs.

See "Creating Offhost Routes" on page 87 for related information on creating the off-host route for this kind of request. You will also need to specify a kn_status_to as described under "Managing Your Responses" on page 91.

Here is a snippet of code from SOAPRequestor.cpp for an Amazon.com REST request:

```
m.Set("ws_SubscriptionId", "1QN071H6A3N0T39CH7G2");
m.Set("ws_Operation", "ItemSearch");
m.Set("ws_SearchIndex", "Books");
m.Set("ws_Keywords", "Machu+Pichu");
m.Set("ws_ResponseGroup", "Small");
```

Retrying Requests

You can use the kn_retry header to tell the LiveServer to try resending the request for a specified number of times. See "kn_retry" on page 75 for more information on that header.

Managing Your Responses

The LiveServer does not modify the response from the Web service. Instead, the kn_payload in the response will contain the actual text of the response from the Web service. If you made a SOAP request, it will be a SOAP response; if a REST request, it will be an XML response. (The HTTP response goes into the status header.) The kn_payload response is posted in the topic you've assigned for that purpose (such as /status or /response). You can use the LiveServer's XPath and XSLT filters if you wish; for more information on those filters, see "The KnowNow XPath Filter" on page 116 and "The KnowNow XSLT Module" on page 118.

There are two things to watch for in the response event:

- The kn_id in the response event will be the same as the kn_id in the corresponding request you posted.
- There will be a header status that has the actual HTTP response status that was received by the LiveServer from the Web service.

Here is code subscribing to the Amazon.com REST service (again, from SOAPRequestor.cpp):

```
wstring stopic = ConvertToWide(statustopic);
c.Subscribe(stopic, &mListener, m, 0);
printf("Subscribed to %s \n", statustopic);

m.Set("do_method", "notify");
m.Set("kn_to", fromtopic);
m.Set("kn_status_to", statustopic);
```

The LiveServer's SOAP and WSDL Interfaces

This section provides some additional information on the LiveServer's SOAP and WSDL interfaces.

SOAP and do_method Commands

Using SOAP, you can call the LiveServer with an XML document containing do_method commands. (For more information on do_method commands, see "An Overview of do_method Commands" on page 44)

For example, if you weren't using SOAP, you could send the LiveServer the following GET URL:

```
do_method=notify&kn_payload=make+money&kn_to=important/stuff
```

and the LiveServer would publish the event.

However, with SOAP, you can instead send an XML document to the LiveServer containing the same command. The LiveServer will extract the do_method and payload out of the XML and then publish the event.

With the LiveServer's implementation of SOAP, most of the do_method commands are implemented in just the same way as before, except in the way they are invoked. (The subscribe call is not implemented since that doesn't fit into the SOAP model.) The do_method commands that create, update, and delete events, topics, and routes are implemented.

Using SOAP and WSDL

Publish SOAP requests to the following URL:

```
http://localhost:8000/knSOAP
```

Where *localhost* and *8000* can be replaced with other machine or domain names (or an IP address) and other ports. You can also use HTTPS.

SOAP and WSDL Examples

This section contains several SOAP and WSDL examples.

Here is a sample request that performs a do_method=whoami command (this command would all be on one line):

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"><soapenv:Body><ns1:whoami
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="urn:do_method"/></soapenv:Body></soapenv:Envelope>
```

And here is its response:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
    <soapenv:Body>
        <ns1:whoamiResponse
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="urn:do_method">
            <kn_displayname xsi:type="xsd:string">knadmin</kn_displayname>
            <kn_userid xsi:type="xsd:string">knadmin</kn_userid>
            <kn_server xsi:type="xsd:string">/kn</kn_server>
        </ns1:whoamiResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

Here is a sample event publication:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"><soapenv:Body><ns1:notify
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="urn:do_method"><kn_to
xsi:type="xsd:string">test/SOAPcommands/publishSimpleEvent</kn_to><kn_payload
xsi:type="xsd:string">sample
payload</kn_payload></ns1:notify></soapenv:Body></soapenv:Envelope>
```

and its response:

```
HTTP/1.0 200 OK
Content-Type: text/xml
Expires: Fri, 03 Feb 2006 18:43:53 GMT
Connection: close
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
    <soapenv:Body>
        <ns1:notifyResponse
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="urn:do_method">
        <success xsi:type="xsd:string">200 Notified.</success>
        </ns1:notifyResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

Chapter 4 LiveServer Modules

Although the LiveServer itself is provided by KnowNow, and you may not need to do anything with it in order to use it, you can customize the LiveServer by creating and loading modules to perform specific functions, such as topic and route filtering, management, authentication and authorization, and other security features. The LiveServer uses core modules that are already provided for your use. Some of these modules can be replaced with your own custom versions; others must be used as they are shipped from KnowNow. This chapter provides an overview of LiveServer modules and the module API.

- "Introducing Modules" on page 96
- "Filter Modules" on page 99
- "The KnowNow Expr Filter" on page 102
- "The KnowNow Filterduplicates Filter" on page 111
- "The KnowNow Tcl Email Filter" on page 115
- "The KnowNow XPath Filter" on page 116
- "The KnowNow XSLT Module" on page 118
- "The KnowNow XSLTtoHeaders Module" on page 120
- "Security Modules" on page 121
- "Understanding the Module API" on page 122
- "The LiveServer Module API Quick Reference" on page 130

Introducing Modules

Modules are code components—either dynamic linked libraries (DLLs) or shared objects (SOs)—that you write using the KnowNow LiveServer API, and then load into the LiveServer using the KnowNow System Administration console. Essentially, a module is a plug-in that performs specific actions with events, topics, routes, and security.

In this section, the following headings provide essential information on modules:

- "Module Basics" on page 96
- "Module Names" on page 97
- "Module Start-up and Shutdown" on page 97
- "Further Information on Modules" on page 98

Module Basics

The LiveServer is shipped with several core modules, as well as some modules that perform specific types of tasks. Essentially, there are three types of modules:

- Core modules, required for LiveServer operations. These core modules have been written by KnowNow and can be used as they are; they need no further modification or customization. Use the KnowNow System Administration console to configure these modules as described in the *KnowNow LiveServer Administration Guide*.
- Filter modules, which filter or transform events at the individual topic and route level. The LiveServer ships with several filter modules. You can also create and use any number of your own filter modules. Filter modules are described in more detail under "Filter Modules" on page 99.
- Helper modules, which support specific types of operations, such as security modules for authentication and authorization, and modules to support persistence. Like the core modules, the KnowNow helper modules are also described in the *KnowNow LiveServer Administration Guide*. Unlike the core modules, you can also write your own helper modules to replace the modules that are shipped with the LiveServer. Security modules are described in more detail under "Security Modules" on page 121.

No matter what kind of module it is, you can use the KnowNow System Administration console to specify parameters for that module, such as the module's location and other attributes. Some parameters are already specified, such as for the core LiveServer modules. In addition to the existing parameters in the KnowNow System Administration console, you can use parameters to register and configure your own modules as well. To do so, use the KnowNow System Administration console as described in the *KnowNow LiveServer Administration Guide*.)

You can also use kn_ headers to configure and attach filters. For more information, see "kn_filtername" on page 65.

Module Names

Each module is named with a topic name under the /kn_system/filters topic. For example, an XSLT module might be named /kn_system/filters/xslt. Since modules are named with topics, client applications can traverse the /kn_system/filters hierarchy to determine the list of installed modules on a particular LiveServer.

Note that filter module names *must* be /kn_system/filters/*some_value*. If you accidentally named your filter /kn_system/filters/, LiveServer won't start.

To recover from a bad filter name,

1. Rename the filter.
2. Open the knsettings.conf file.
3. Find the bad filter definition and change the filter name in that file to match the new filter name.
4. Restart LiveServer.

For information on attaching filter modules, see "kn_filtername" on page 65.

Module Start-up and Shutdown

Once you have created a module, you need to load the module into the LiveServer to make it available. To do so, use the KnowNow System Administration console to define that module, then restart the LiveServer.

Whatever kind of module you create, the module must define two function calls: KnModuleStart for the module's start-up (initialization) and KnModuleStop for its shutdown. When the LiveServer loads modules at startup, it calls each module's KnModuleStart function to initialize the module, then loads the module into memory. Before the LiveServer shuts down, it calls each module's KnModuleStop function so that each module can be stopped gracefully. For more information on start-up and shutdown processing and other tasks that modules must perform, see "The LiveServer Module API Quick Reference" on page 130.

Further Information on Modules

If you need to, review the discussion of modules and events under "Publish and Subscribe Operations and Events" on page 22. With the understanding of how the communication flow is managed by KnowNow, and of how modules fit into the overall picture, you can examine what each type of customizable module does as discussed in the remainder of this chapter.

Filter Modules

Filter modules can monitor, block, transform, or reroute events. To use a filter module, you attach it to topics or to individual routes.

- When a filter module is attached to a topic, control is passed to that module each time an event is received or published for that topic.
- When a filter module is attached to a route, control is passed to that module each time an event is traversing the route.

You can attach more than one filter to a topic or route. For rules on the order in which multiple filters on a single topic or route are executed, see "kn_filtername" on page 65.

Once attached to a topic or a route, filter modules examine and modify the contents of each event using criteria you specify. For example, a filter module could monitor traffic on a specific route, possibly also writing that monitored information to a log. Or, it could block the routing of a specific event, or transform that event in some way (such as by adding, removing, or modifying headers), or reroute that event by sending it somewhere else instead of to another destination or in addition to another destination.

Of course, depending on the circumstances, filter modules can perform more than one of these functions at a time, such as both monitoring and rerouting. Also, one filter module can be assigned to more than one route or topic at a time, and each instance of that module can have a different set of parameters (in other words, for each route or topic, the module could use different parameters). For more on passing parameters to a filter module, see "Creating and Using Filter Modules" on page 99.

Creating and Using Filter Modules

In addition to using the filter modules that ship with the LiveServer, you can use the LiveServer API to create your own custom filter modules. Once you have created the desired filter module(s), you load all filter modules into the LiveServer as described under "Introducing Modules" on page 96. (You can always create and add more filter modules later, and also remove any kind of module at any time simply by using the KnowNow System Administration console. For information on using the KnowNow System Administration console, see the *KnowNow LiveServer Administration Guide*.) You can then use those filters as desired.

Once you have created a filter module, you can attach it to the desired topic or route either by using the KnowNow System Administration console or by using calls from the LiveServer API (using the do_method set_topic_property and the kn_ headers kn_filtername and kn_filterparams). As the LiveServer processes the events it receives, if there is a filter module for that event, it passes control to the assigned filter module.

Filter modules are attached using different commands depending on whether they are attached to a topic or a route. However, in all cases, the filter is identified using kn_filtername, and parameters are passed to the filter using kn_filterparams. For more information on these headers, see "kn_filtername" on page 65 and "kn_filterparams" on page 66.

You can create and use any number of filter modules, and a single filter module can be assigned to more than one topic or route. You can also attach different filters to a given topic or route.

Filter Module File Formats

As mentioned under "Introducing Modules" on page 96, filter modules must be in either DLL or SO format, depending on whether you are running on the Windows platform or on Solaris/Linux platforms. For information on how to load modules into the LiveServer and on what tasks all modules need to perform, see "Start-up and Shutdown Processing of Modules" on page 122. For information on the API calls needed to create filter modules, see "The LiveServer Module API Quick Reference" on page 130.

Route Filter Modules

Route filter modules affect events being passed along on a specific route. Route filter modules can attach code to a route to manage events as they are being passed over that route. Subscribers can request that a route filter module be attached to a route to a topic they are interested in. Even if a route filter module is attached to several routes, subscribers will only be interested in and can only affect the route filter modules that are associated with the routes to topics they are subscribing to.

Route filter modules are attached either at subscription time (when the route is created), or later. To attach at subscription time, use the do_method=route command. To attach it later, use the do_method=update_route command. For more on these commands, see "route" on page 52 and "update_route" on page 55.

Topic Filter Modules

Topic filter modules affect events that are being published into the topic they are attached to. Topic filter modules can be attached to a topic in one of two ways: at the time the topic is created, using a do_method=add_topic command, or later, when updating or modifying that topic, using the do_method=set_topic_property command. For more on these commands, see "add_topic" on page 50 and "set_topic_property" on page 54.

The LiveServer Filter Modules

The LiveServer ships with the following useful filter modules that you can use to perform specific types of tasks:

- "The KnowNow Expr Filter" on page 102
- "The KnowNow Filterduplicates Filter" on page 111
- "The KnowNow Tcl Email Filter" on page 115
- "The KnowNow XPath Filter" on page 116
- "The KnowNow XSLT Module" on page 118
- "The KnowNow XSLTtoHeaders Module" on page 120

The KnowNow Expr Filter

The LiveServer ships with a filter module, Expr, that makes it possible for you to use regular expressions. In addition, the filter feature of the KnowNow ESS Administration console has an option for creating regular expressions; that feature uses the Expr module.

Note: The filter module implements extended regular expressions using the Henry Spencer regex library. We assume you are already familiar with regular expressions in general and with the Henry Spencer implementation in particular. If not, there are many good references available on these topics. One such work is *Mastering Regular Expressions*, second edition, by Jeffrey E. F. Friedl.

These filters are described under the following headings:

- "Using Expr With Topics or Routes" on page 102
- "Expr Syntax" on page 103
- "Expr Parameters" on page 104
- "Expr Example" on page 110

Using Expr With Topics or Routes

You can attach the KnowNow Expr filter module to a topic or route using either the KnowNow System Administration console or KnowNow do_methods or headers.

To use the LiveServer's Expr filter,

1. Attach the Expr filter to a topic or add it to a route. You can do so in two ways:
   a. You can use the KnowNow System Administration console (as described in the *KnowNow LiveServer Administration Guide*).
   b. Or you can use the route or set_topic_property do_methods in conjunction with the kn_filtername header.
2. Add a parameter (in the form of a regular expression using one or more of the Expr parameters) to the filter. You can do so in two ways:
   a. You can use the KnowNow System Administration console.
   b. Or you can use the kn_filterparams header.

The syntax is described under "Expr Syntax" on page 103. The parameters are described under "Expr Parameters" on page 104. In the descriptions of the more complex parameters, some examples are provided, and a general example is provided under "Expr Example" on page 110.

Expr Syntax

The format for creating regular expressions is as follows:

```
Param=value[:param=value]...
```

For example,

header=*header_name*:value=*regex*:type=*action*

In this example syntax, header, value, and type are filter module parameters, and *action* is an option of the type parameter.

- *header_value* represents the name of the header to which you wish to apply the filter. For more information, see "header" on page 105.
- The regular expression is provided using the value parameter. *regex* represents the regular expression to be used. For more information, see "value" on page 110.
- *action* represents the action to be taken when the filter criterion is fulfilled. The action is selected by specifying one of the options for the type parameter, which are provided under "type" on page 109.

The anchor characters are begin (^) and end ($).

Delimiter Characters

The delimiters are the colon and the equal sign:

```
: =
```

To get a colon (:) or an equal sign (=) into your regular expression, escape the desired character with a backslash (\). For example, \: matches a colon and \= matches the equal sign. Use a double backslash (\\) to indicate a backslash.

Delimiters other than = and : are not interpreted by Expr. (Though they are interpreted in the "straight string" fields of the KnowNow ESS filter—i.e., in the filter fields other than the **Regular Expression** field.)

An Expr parameter starts at the beginning of the parameter string or after a colon and continues until the next colon. If there is an equal sign in that parameter, then the parameter name is the string from the character after the colon to the equal sign, and the parameter value is everything from the character after the equal sign to the colon.

Regarding the *values* of parameters, if you want to include an equal sign or colon *in* the value, precede the = or : by a backslash (\). Otherwise, all \ characters are passed unmodified to the value parser (or the regular expression parser in the case of value or regex options).

For example,

- `regex=Test` matches any string containing `Test`.
- regex= *space* Test *space* matches any string containing *space* Test *space*.
- `regex=Test1=Test2` matches any string containing `Test1=Test2`, as does `regex=Test1\=Test2`.
- `regex=Test1:Test2` is illegal, since Expr does not have `Test2` parameter.
- `regex=Test1\:Test2` matches any string containing `Test1:Test2`.

Expr Parameters

The filter modules take parameters in the form of a list, using colons to delineate the value pairs. Use normal regular expression rules when creating regular expressions. For debugging purposes, some messages are printed into server.log when the parameters are parsed. The filter will not be attached if the filter parameters don't make sense.

The parameters can be divided into the following categories:

- Cumulative: average, count, max, min, std, sum. These have global scope. For all parameters that perform cumulative work, the values are good until LiveServer is stopped. For example, if you restart LiveServer and you are using the sum parameter, the sum is reset to zero. These options only work for events that pass through the topic or route after LiveServer startup; they don't include any data for events that were in the database when LiveServer recovered.
- Pre-evaluation value manipulators: lowercase and uppercase (which have global scope), and some op parameter options (ignoretags, dontignoretags, and nomarkup). These stack, though most uses of these will be to set them at the top of the stack and use that setting throughout.

  If you don't set the parameter at level *n* of the stack, you get the setting from level *n*-1. So, for example,

  ```
  op=nomarkup:op=push:header=feedTitle:value=coffee:op=pop:
  value=fair
  ```

  would normalize (because of the nomarkup option) both the feedTitle and the kn_payload value before matching against /coffee/ and /fair/ respectively.

- Regex mode flags as specified using certain options for the op parameter: insensitive, sensitive, notequal, and equal. These stack as above.
- Boolean operators as specified using the op options **and** and **or**. These stack as above.

- String side-effects: replace and replaceHeader. Operates at the end of the calculation.
- Actions as specified using the type parameter. These are global for the filter and only evaluated in terms of the final evaluation of the entire expression.
- Event information: header, regex, topic, and value.

These parameters are listed in alphabetical order below.

average

Averages a given header. If no header is specified, by default, it averages the payload. There is no way to reset the average.

count

Counts a given header. If no header is specified, by default, it counts the payload.

header

If this parameter is specified, the regular expression provided by the value parameter is applied to the header of the event specified by the **header** parameter instead of to the payload (in kn_payload). If no **header** is specified, it defaults to the kn_payload header. See also "replace and replaceHeader" on page 108.

For example, if you set the following parameters as follows:

```
header=severity:value=1:type=allow
```

Then you are saying to only allow events through that have severity=1 in their header.

lowercase

Converts all letters to lowercase in a given header. If no header is specified, by default, this parameter applies to the payload.

max

Finds a maximum value in a given header. If no header is specified, by default, this parameter uses the payload.

min

Finds a minimum value in a given header. If no header is specified, by default, this parameter uses the payload.

op

The **op** parameter sets the state for all comparisons that follow it. It provides the following options:

- **and** and **or**. `op=and` specifies that all comparisons afterwards are ANDed with the previous results. `op=or` specifies that all comparisons afterwards are ORed with the previous results. Note that these operators are *not* short-cut operators like Perl or C's && or ||; all expressions are evaluated (as in Perl's "and" and "or" operators). Also, each operator is evaluated in turn; there is no built-in precedence for these operators. If you want to control evaluation precedence, use `op=push` and `op=pop`.
- **debug**. If you set this option, there will be tracing in server.log showing what values are compared using the expression filter. This option is useful for debugging your regular expressions.
- **dontignoretags** and **ignoretags**. If set in an expression pattern, these options cause XML tags to either be considered (**dontignoretags**) or not be considered (**ignoretags**) as part of the expression. (Tags are the mark-up tags used in XML coding, such as <body>.) If this option is not specified, `op=dontignoretags` is the implied default. For example,

  ```
  op=ignoretags:regex=Test:type=allow
  ```

  would match `<b>T</b>est`, but

  ```
  regex=Test:type=allow
  ```

  which uses **dontignoretags** as an implicit default, would *not* match `<b>T</b>est`.

  These two options are related to the **nomarkup** option. For more information, see that option.

- **equal** and **notequal**. These options are used when comparing patterns to set an action to take when the result is true or false. The **equal** option is the default. Therefore,

  ```
  regex=Test:type=allow
  ```

  matches `Test`, but

  ```
  op=notequal:regex=Test:type=allow
  ```

  does *not* match `Test`.

op=equal is a stackable operation. This means it is taken into account to its level until op=notequal is found. Also, op=equal will action to the next levels if the operation is not changed.

For example, suppose you have

```
op=push:value=x:op=or:value=y:op=pop:op=notequal:op=push:value=a:op=and:op=p
ush:op=equal:value=b:op=or:value=c:op=pop:op=and:value=d:op=pop
```

This expression is evaluated as follows:

a. For the op=push:value=x:op=or:value=y:op=pop portion of this expression, the implicit op=equal is used.

b. Then op=notequal is encountered. This is used for the op=push:value=a:op=and and the op=and:value=d:op=pop portions.

c. Finally, the explicit op=equal is used for the op=push:op=equal:value=b:op=or:value=c:op=pop portion.

- **insensitive** and **sensitive**. op=insensitive specifies that all comparisons following this statement are case **in**sensitive. op=sensitive specifies that all comparisons following this statement are case sensitive. By default, comparisons are case insensitive, so you don't need to specify op=insensitive unless you have already previously specified op=sensitive.
- **nomarkup**. Extracts text as it would be rendered from XML or HTML by expanding <br> and <br/> to \n, eliminating all other tags, then expanding the HTML ASCII entities , ", &, <, and > to their corresponding characters. This option is similar to the **ignoretags** option, except that it also expands the specified entities.

  Each of the three options, **dontignoretags**, **ignoretags**, and **nomarkup**, overrides the previous occurrence of any of these options. So, for example, after the following sequence

  ```
  op=dontignoretags:op=nomarkup:op=ignoretags
  ```

  op=ignoretags will be in effect.
- **pop** and **push**. Expression results are pushed onto or popped off of a stack using these options.

  When you use op=push, the following are saved to a stack:

d. The expression result (if any). For example, for (((E))), the first two pushes won't have E.

e. The operand (if any).

f. Any pre-evaluation value manipulators (`op=ignoretags`, `op=dontignoretags`, `op=nomarkup`).

g. The regex mode flags (`op=equal`, `op=notequal`, and `op=sensitive` or `op=insensitive`).

When you use `op=pop`, it retrieves what was saved in the push in reverse order.

If multiple fields are compared, by default, *all* of the **op** options must be true for the expression to be true.

For example, consider the following expression:

```
value=a:op=sensitive:value=b:op=or:op=sensitive:value=C:type=allow
```

This expression can be translated as

```
Find events where the payload contains ((uppercase A OR lowercase a) AND
(lowercase b)) OR (uppercase C OR lowercase c)
```

With this expression, the following payloads would be valid:

```
Ab
ab
zzCzz
```

and the following payloads would be invalid:

```
zzAzz
zzABzz
```

Note that regular expressions are unanchored both at the beginning and the end in all cases.

regex

See value.

replace and replaceHeader

Use **replace** if you want to replace a payload with another value. **replace** contains the new value for the header indicated by **replaceHeader**. If there is no **replaceHeader**, then it contains the new value for the payload. The replace operation is performed when the regular expression in value evaluates to **true** and the type parameter is set to **transform**.

std

Calculates a standard deviation on a given header. If no header is specified, by default, it uses the payload.

sum

Sums the values of a given header. If no header is specified, by default, it uses the payload.

topic

This is the topic the event is sent to if the regular expression in the value parameter evaluates as **true** and the type parameter is set to **reroute**.

type

Specifies the action to take when the regular expression specified in value is evaluated.

Table 4-1. Actions of the **type** parameter.

type=	The regular expression in value evaluates as	Action taken
allow	false	The event is rejected.
deny	true	The event is rejected.
transform	true	The event's payload is replaced by the value from the parameter **replace**.
reroute	N/A	The event is rerouted to the topic provided in the parameter topic.

The **type** parameter defaults to neither allow nor deny. That is, it does not filter if **type** is not set. This default is useful if you use **type** to transform data.

uppercase

Converts all letters to uppercase in the given header. If no header is specified, by default, it applies to the payload.

value

This is the parameter you use to provide the regular expression. The regular expression is matched against what is in kn_payload unless the header parameter is specified. As an alternative, you can use **regex** instead of **value**; they both do the same thing.

Expr Example

Example 4-1 discusses a sample expression that uses some of the Expr parameters.

Example 4-1. Using Expr parameters.

Let us say you entered an expression like this:

```
sum=:count=x:lowercase=name:min=cost
```

If an event has the following two values, one for a Hawaiian shirt, and the other for a t-shirt:

```
(payload=3,x=5,name=Hawaiian Shirt,cost=50)
(payload=2,x=12,name=t-shirt,cost=12)
```

The result would be two events. The first event would be

```
(payload=3,x=1,name=Hawaiian Shirt,cost=50)
```

The second event would be

```
(payload=5,x=2,name=t-shirt,cost=12)
```

The KnowNow Filterduplicates Filter

The LiveServer ships with a filter module, Filterduplicates, that changes the behavior of duplicate detection and removal for topics it is attached to.

Normally, in LiveServer, duplicate detection works as follows:

1. When a new event is being considered for entry into a topic, its kn_id header is examined. LiveServer looks for any other event in that topic with the same kn_id. If none exists, the event is added; it is (by definition) not a duplicate since its key is different from all other keys in the topic.
2. LiveServer then compares the other “non-system” headers in the event. If the value of any non-system header is different, LiveServer considers this an update. Otherwise it’s a duplicate. The system headers are those used internally and supplied by LiveServer; e.g., the **kn_routed_from** header. These are often also referred to as the hop-by-hop headers.
3. Duplicates are ignored. Updates are applied by removing the old event from the event list and adding the new one at the “tail” of the event list. LiveServer is multi-threaded, so the tail can be moving while this happens.

With the Filterduplicates filter, you can change this behavior. This module must be attached as a topic filter (we often call these admission filters). It provides mechanisms for:

- Changing the kn_id (mapping other fields to create a new kn_id value)
- Hashing the new kn_id to a shorter value
- Ignoring headers beyond the hop-by-hop headers
- Specifying specific headers that should be compared, ignoring all others

So, for example, if you have incoming data identified by source and a serial number (that is, its kn_ids look something like *ap102101* in one topic, and *reuters9389101* in another), but the same data could arrive from multiple sources, what you might do is route them both to the same topic (for example, /myNews/) and change the key to reflect the real data (key=guid). If there’s already a good key that both feeds agree on, you would use the header that contains that key value as the new kn_id. Otherwise, you might concatenate a few other headers to create a good key.

Just that operation can remove a lot of duplicates. On the other hand, each source may provide its own additional headers. For the aggregated feed, you’d like to ignore those headers and just provide the important data. To do that, add parameters to the filter to ignore those headers (`exclude=reuters-advertisement,exclude=ap-routing-header`).

Alternatively, the sources may provide a lot of irrelevant headers, and you may want to just compare based on the salient headers. In that case, you would use “include” parameters—e.g., `include=description,include=link,include=author`.

Filterduplicates does *not* provide mechanisms for merging in the header information that was ignored during duplicate checking, nor does it provide a means for preserving the old kn_id value if you've mapped a new one. It also does not provide schemes for changing the values of non-key headers.

Filterduplicates Parameters

Filterduplicates takes the following parameters:

- “k or key” on page 112
- “h or hash” on page 113
- “i or include” on page 113
- “nomarkup” on page 114
- “sensitive” on page 114
- “s or separator” on page 114
- “x or exclude” on page 114

These parameters can be combined, separated by commas; for example,

kn_filterparams: k=title,i=author,i=description,sensitive=false,nomarkup=true

k or key

Syntax:

- k=*header*
- key=*header*

Forms the new kn_id from the listed header. This parameter can appear any number of times; each header value will be appended to form the new key, separated by the separator (discussed under “s or separator” on page 114).

No other headers are affected.

Example

```
kn_filterparams: k=title,key=author
```

Incoming event:

```
kn_id: 118
title: KnowNow News
```

```
author: lisa
```

Outgoing event:

```
kn_id: KnowNow News|lisa
title: KnowNow News
author: lisa
```

h or hash

Syntax:

- h
- h=*anything*
- hash
- hash=*anything*

If set, uses LiveServer's internal fast string hash algorithm over the key formed using the key headers, and then sets the kn_id to the string representation of that number. By default, hash is not set.

The hash algorithm is the one used for string traits in the SimpleTraits library; it was apparently invented by Robert Jenkins and documented in a *Dr. Dobb's* article. Mr. Jenkins reprints this at http://burtleburtle.net/bob/hash/doobs.html.

We recommend not using the `=anything` form. If you really want to put a value there, use `hash=hash32`.

i or include

Syntax:

- i=*header*
- include=*header*

Use the given header to compare for duplicates. This parameter may appear multiple times. If it appears at all, all headers not mentioned will be ignored. We always ignore internal LiveServer headers.

Specify *either* include headers *or* exclude headers; results are undefined if both are provided.

If neither exclude nor include headers are provided, normal LiveServer duplicate squashing will be used.

nomarkup

Syntax:

- nomarkup=true
- nomarkup=false

Extracts text as it would be rendered from XML or HTML by expanding `<br>` and `<br />` to `\n`, eliminating all other tags, then expanding the HTML ASCII entities ` `, `"`, `&`, `<`, and `>` to their corresponding characters.

sensitive

Syntax:

- sensitive=true
- sensitive=false

Specifies that all comparisons are either case sensitive or case insensitive.

s or separator

Syntax:

- s=*value*
- separator=*value*

Use the value to separate the values of the given headers in the new kn_id. By default, the separator is the pipe (|).

x or exclude

Syntax:

- x=*header*
- exclude=*header*

Ignores the given header. LiveServer always ignores internal LiveServer headers. This parameter may appear multiple times.

Specify *either* include headers *or* exclude headers, not both; results are undefined if both are provided.

If neither exclude nor include headers are provided, normal LiveServer duplicate squashing will be used.

The KnowNow Tcl Email Filter

The LiveServer ships with a Tcl email filter that sends SMTP email in response to events. You can attach this filter to topics or routes. You specify the host using the Service Network parameter **SMTP Server**. **SMTP Server** is configured using the KnowNow System Administration console; for more information, see the *KnowNow LiveServer Administration Guide*.

If the Tcl procedure kn_mail_filter is attached to a topic or route (as the filter parameter for /kn_systems/filters/tcl), an event published to that topic will be transformed into an email message using the following headers:

- kn_mailto: a comma-separated list of recipient email addresses
- kn_mailcc: a comma-separated list of email addresses to send copies of the message to
- kn_mailbcc: a comma-separated list of email addresses to send blind copies to
- kn_mailfrom: the email address of the sender
- kn_mailsubject: the subject of the message

The body of the mail message is the payload of the event.

The KnowNow XPath Filter

If you have an XML document and you want to evaluate it for one or more properties, such as whether a node of *X* value is somewhere in the tree, you can use XPath to perform that evaluation. Of course, you need to know something about the hierarchy of the XML document that you are searching in order to write a meaningful expression.

The LiveServer ships with an XPath filter that makes it possible for you to perform these evaluations. You can use the KnowNow System Administration console to attach this filter module to a topic or a route as described on page 102. This section tells how to use the XPath filter module.

Note: We assume you are already familiar with XPath and XML (and probably XSLT as well, since XPath is a major element in XSLT) and that you therefore know how to create XPath expressions; if not, there are many good references available on these topics, starting with http://www.w3.org/TR/xpath.

The XPath filter takes XPath expressions as parameters.

To use the LiveServer's XPath filter,

1. Attach the XPath filter to a topic or add it to a route. You can do so using the KnowNow System Administration console (as described in the *KnowNow LiveServer Administration Guide*), or using the route or set_topic_property do_methods in conjunction with the kn_filtername header.
2. Add a parameter (in the form of an XPath expression) to the filter using either the KnowNow System Administration console or the kn_filterparams header.

Example 4-2 shows how XPath filters are used.

Example 4-2. XPath parameter example.

When an XPath filter is added with the following parameter:

```
kn_filterparams=/doc/name[1]
```

and the payload on an event is

```
<?xml version=\"1.0\"?>
       <doc>
           <name first="Tristan" last="Wylde-LaRue">Mr. Wylde-LaRue</name>
           <name first="Cordelia" last="Wylde-LaRue">Ms. Wylde-LaRue</name>
       </doc>
```

then the KnowNow XPath module will allow the event to be published. (Because there are two names in the XML, the filter /doc/name[1] passes since the record exists.)

On the other hand, an XPath filter with a parameter of

```
kn_filterparams=/doc/name[3]
```

would **not** allow the event to be published. (Because there are only two names, the filter rejects the event, since there are only two records in the XML.)

Both these filters can be passed to KnowNow's XPath module as parameters.

The KnowNow XSLT Module

The LiveServer ships with an XSLT filter module. XSLT is a language for transforming XML documents into other XML documents; for example, to find a certain string and generate new XML with the string transformed into whatever you specified. XSLT has the equivalent of an include statement—you can use that to point to a URL that contains the statement you wish to use.

You can use the KnowNow System Administration console to attach the XSLT filter module to a topic or a route as described on page 102.

Note: We assume you are already familiar with XSLT; if not, there are many good references available on this topic. For a full definition of XSLT, see http://www.w3.org/TR/xslt.

The XSLT module behaves much like the KnowNow XPath filter, except instead of searching for a match (or something that doesn't match), it transforms the document. The KnowNow XSLT filter takes as a parameter an XSLT transformation description you provide, and changes every event that comes through on a topic or route.

To use the LiveServer's XSLT filter,

1. Attach the XSLT filter to a topic or add it to a route. You can do so using the KnowNow System Administration console (as described in the *KnowNow LiveServer Administration Guide*), or using the route or set_topic_property do_methods in conjunction with the kn_filtername header.
2. Add a parameter (in the form of an XSLT expression) to the filter using either the KnowNow System Administration console or the kn_filterparams header.

Example 4-3 shows how XSLT filters work.

Example 4-3. XSLT example.

Using the XSLT filter, the following XML (where the style sheet has been placed in kn_filterparams):

```
<?xml version="1.0"?>
    <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
        <xsl:template match="doc">
            <out><xsl:value-of select="."/></out>
        </xsl:template>
    </xsl:stylesheet>
```

would convert this XML (provided in the event's payload):

```
<?xml version="1.0" encoding="UTF-8"?>
       <doc>Hello</doc>
```

into this (viewable in the event's payload after the event has been published to a topic or route to which this filter has been attached):

```
<?xml version="1.0" encoding="UTF-8"?>
       <out>Hello</out>
```

The KnowNow XSLTtoHeaders Module

In addition to the XSLT module, the LiveServer ships with an XSLTtoHeaders filter module. The LiveServer's XSLTtoHeaders module behaves like the KnowNow XSLT filter, except that the resulting XML is parsed and placed in the resulting event. This module works on all children of a parent node (but not on the parent node itself). When you use this module, all the children of the top level element are converted to header values.

XSLTtoHeaders can take a kn_payload (the payload must be in XML) and generate a new kn_payload; for example, this could be used to convert a document into header/value pairs.

Example 4-4. XSLTtoHeaders example.

Using the XSLTtoHeaders filter, with the appropriate style sheet provided in kn_filterparams, the following XML, provided in the event's payload

```
<?xml version="1.0" encoding="UTF-8"?>
       <item>
           <description>Book title</description>
           <title>For Whom the Bell Tolls</title>
       </item>
```

would be converted into this (viewable in the event's payload after the event has been published to a topic or route to which this filter has been attached):

```
kn_payload:Book title
title:For Whom the Bell Tolls
```

Security Modules

Security modules perform authentication and authorization functions when users attempt to access the LiveServer. Using the LiveServer API, you can create security modules that provide user authentication or authorization (or both).

- Authentication modules validate a user by checking the credentials supplied with a request, such as the user's name and password.
- Authorization modules check the user's credentials supplied with a request to determine whether access should be granted to a particular URL on the LiveServer.

The LiveServer comes with a basic security module, nsperm, that demonstrates the principal calls, as well as an LDAP security module. For information on configuring security with the LiveServer, see the *KnowNow LiveServer Administration Guide*.

For more information on the module API, including specifics for the security module API calls, see "Understanding the Module API" on page 122 and "The LiveServer Module API Quick Reference" on page 130.

If one or more modules are loaded into the LiveServer, as the LiveServer processes requests, it passes control to the appropriate module. In the case of security modules, if such a module exists, all requests are first passed to the authentication module, the authorization module, or, if you have combined the functionality of both into a single module, to the combined module. In keeping with standard HTTP authentication conventions, control is passed to your registered authorization function **before** the LiveServer processes the request.

Understanding the Module API

The LiveServer communicates with modules using well-defined callback functions during three basic phases of operation: startup, event management, and shutdown. Your module must implement callback functions for each of these phases. To create the module and have it perform these tasks, use the LiveServer module API. For additional information on the module API, see "The LiveServer Module API Quick Reference" on page 130.

The following topics provide guidelines and information for creating and using modules:

- "Start-up and Shutdown Processing of Modules" on page 122
- "Event Management" on page 124
- "The LiveServer Module API Quick Reference" on page 130

Start-up and Shutdown Processing of Modules

All modules are loaded into memory when the LiveServer starts up, and remain there until the LiveServer shuts down. The following sections provide an overview of the necessary tasks and calls for start-up and shutdown processing:

- "Loading Modules into the LiveServer" on page 122
- "Start-up Processing" on page 123
- "Shutdown Processing" on page 124

In between the two functions for startup and shutdown, your module can implement other tasks as desired.

Loading Modules into the LiveServer

In order to load a module into the LiveServer, you must use the KnowNow System Administration console to define the module(s) you wish the LiveServer to use, then shut down and restart the LiveServer so that it loads in the information for those modules. When the LiveServer restarts, it loads the module(s)'s DLL(s) or SO(s) into itself, then calls the module's initialization function KnModuleStart, as described in the next section. For instructions on loading and configuring modules, see the *KnowNow LiveServer Administration Guide*.

Start-up Processing

As part of your module initialization at startup time, the LiveServer invokes your module's KnModuleStart function, which each module of any type must implement. This function is called only once, so it is not required to be a threadsafe implementation. Your module's implementation of this function can provide any required initialization, such as connecting to an external system or initializing an internal XSLT engine.

Security modules must also implement KnModuleStart so that either the KnAuthorizeSet or the KnAuthenticateSet function (or both) is invoked to register your authorization or authentication function with the LiveServer. Once the security module is correctly allocated and passed in, after that, any time the LiveServer receives a request, the LiveServer immediately passes control to the registered authorization or authentication function before it processes the request. You can also create any additional desired functionality in your start module.

As part of the initialization, your security module's KnModuleStart implementation can perform any additional functionality as desired, like connecting to an external system such as LDAP or RSA's SecurID®.

Note: The LiveServer will not invoke the registered authentication function directly. Instead, the authorization module must invoke the authentication function. Though you can use two separate security modules, you are not required to decouple the authorization and authentication functions: You can design a single authorization module to provide both functions.

Upon completion, your module's KnModuleStart implementation should return an indication of the overall success or failure of the initialization processing. If a failure indication is returned, the LiveServer ignores the module from that point on.

Example 4-5. Sample implementation of KnModuleStart.

```
KN_EXPORT KnError KnModuleStart(const KnString &path, const KnSet &info)
{
    KnLog(KnLogNotice, "SAMPLE_MODULE:  KnModuleStart()");
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Path = %s", path.c_str());

    for (size_t i = 0; i < info.size(); ++i)
    {
        KnLog(KnLogNotice, "SAMPLE_MODULE:  Info [%u] = %s : %s",
              i, info[i].m_key.c_str(), info[i].m_value.c_str());
    }

    return KnErrorSuccess;
```

```
}
```

Shutdown Processing

When the LiveServer is shutting down, it calls your module's KnModuleStop function. Your module's implementation of this function should provide any necessary clean-up processing, such as disconnecting from any external system and freeing any resources that may have been allocated. This function is called only once, so it's not required to be a threadsafe implementation.

Example 4-6. Sample implementation of KnModuleStop.

```
KN_EXPORT KnError KnModuleStop()
{
    KnLog(KnLogNotice, "SAMPLE_MODULE:  KnModuleStop()");

    return KnErrorSuccess;
}
```

Event Management

Between startup and shutdown, the LiveServer manages events. You can use the module API to attach filter modules to topics or routes, or detach them, or modify their parameters while the LiveServer is running. Information on how to do so is provided under the following headings:

- "Attaching Filter Modules to Topics or Routes" on page 124
- "Detaching Filter Modules" on page 126
- "Events and Filter Modules" on page 127

Attaching Filter Modules to Topics or Routes

This section provides an overview of what you need to do to attach filter modules to topics or routes programmatically. You can also attach filter modules using the KnowNow System Administration console as described in the *KnowNow LiveServer Administration Guide*. For information that is specific to your programming language, see the appropriate language-specific chapter later in this book.

Once the LiveServer has loaded your filter module and called the KnModuleStart function, client applications can attach the filter module to topics or routes within the LiveServer.

- If a filter module is attached to a topic, control is passed to the module each time an event is published to that topic. The LiveServer calls the KnFilterTopicAttach function each time your module is attached to a topic.
- When a filter module is attached to a route, control is passed to the module each time an event is traversing the route. The LiveServer calls the KnFilterRouteAttach function each time your module is attached to a route.

An individual filter module can be attached multiple times within the LiveServer's topic and route hierarchy. The KnFilterTopicAttach or the KnFilterRouteAttach functions are like constructors called with parameters for a particular instantiation of the module. This means that you can provide different sets of parameters for each instance of the filter module.

Note: Your KnFilterTopicAttach or the KnFilterRouteAttach implementation must be threadsafe.

KnFilterTopicAttach always provides the URL of the topic that events are going into. KnFilterRouteAttach always contains the destination URI of the route on which the filter module has been attached. The client application attaching your module may optionally specify a set of parameters to be passed to the module as part of the attachment. This allows modules to be parameterized for each individual attachment, just as a class can be instantiated as several different objects.

For example, you might create a generic XSLT transformer module that is attached in five different locations. Each attachment of the same module can be given its own XSLT sheet to govern the transform it will provide on an individual topic or route.

The sample implementation of KnFilterTopicAttach shown in Example 4-7 simply logs the passed parameters and the URI before returning.

Example 4-7. Sample implementation of KnFilterTopicAttach.

```
KN_EXPORT KnError KnFilterTopicAttach(const KnString &moduleParams, const KnString &uri,
bool &transform)
{
    KnLog(KnLogNotice, "SAMPLE_MODULE:  KnFilterTopicAttach()");
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Parameters = %s", moduleParams.c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Uri = %s", uri.c_str());

    // Do not allow the module to modify events as they pass through
    transform = false;

    return KnErrorSuccess;
```

```
}
```

As with Example 4-7, the sample implementation of KnFilterRouteAttach shown in Example 4-8 simply logs the passed parameters and the URI before returning.

Example 4-8. Sample implementation of KnFilterRouteAttach.

```
KN_EXPORT KnError KnFilterRouteAttach(const KnString &moduleParams, const KnString
&source, const KnString &dest, bool &transform)
{
    KnLog(KnLogNotice, "SAMPLE_MODULE:  KnFilterRouteAttach()");
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Parameters = %s", moduleParams.c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Source Uri = %s", source.c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Destination Uri = %s", dest.c_str());

    // Allow the module to modify events as they pass through
    transform = true;

    return KnErrorSuccess;
}
```

Detaching Filter Modules

This section provides an overview of what you need to do to detach filter modules from topics or routes. For information that is specific to your programming language, see the appropriate language-specific chapter later in this book. You can of course also use the KnowNow System Administration console to detach filter modules.

The LiveServer calls the KnFilterTopicDetach function whenever a filter module is removed from a topic. The LiveServer calls KnFilterRouteDetach when a filter module is detached from a route. Your implementation should clean up any per-attachment resources that may have been allocated.

Note: Your KnFilterTopicDetach or the KnFilterRouteDetach implementation must be threadsafe.

The sample implementation of KnFilterTopicDetach shown in Example 4-9 simply logs the passed parameters and the URI before returning.

Example 4-9. Sample implementation of KnFilterTopicDetach.

```
KN_EXPORT KnError KnFilterTopicDetach(const KnString &moduleParams, const KnString &uri)
{
```

```
    KnLog(KnLogNotice, "SAMPLE_MODULE:  KnFilterTopicDetach()");
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Parameters = %s", moduleParams.c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Uri = %s", uri.c_str());

    return KnErrorSuccess;
}
```

As with Example 4-9, the sample implementation of KnFilterRouteDetach shown in Example 4-10 simply logs the passed parameters and the URI before returning.

Example 4-10. Sample implementation of KnFilterRouteDetach.

```
KN_EXPORT KnError KnFilterRouteDetach(const KnString &moduleParams, const KnString
&source, const KnString &dest)
{
    KnLog(KnLogNotice, "SAMPLE_MODULE:  KnFilterRouteDetach()");
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Parameters = %s", moduleParams.c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Source Uri = %s", source.c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Destination Uri = %s", dest.c_str());

    return KnErrorSuccess;
}
```

Events and Filter Modules

Once a filter module is attached to a topic, the LiveServer invokes your module's KnFilterTopicEvent function whenever an event is being published to a topic. The LiveServer calls KnFilterRouteEvent when traversing a route to which the module is attached. The LiveServer passes the KnFilterTopicEvent or the KnFilterRouteEvent callback using the following arguments:

- URL of the topic for topic modules; for route modules, there is a source and destination URL.
- Event as a KnApiEvent object.
- Any parameters associated with this attachment.
- A discard flag you can set to true (1) if you want the LiveServer to throw the event away.

Note: Your KnFilterTopicEvent/KnFilterRouteEvent implementation must be threadsafe. Since KnFilterTopicEvent/KnFilterRouteEvent can be called simultaneously from multiple threads, make sure locks are used to serialize access to any shared data.

Your implementation of the KnFilterTopicEvent/KnFilterRouteEvent functions can

- Examine the headers and values in the event and even modify the contents (if the module was attached as a transform module).
- Use the KnEventPost function to send the event to another topic on the LiveServer or to URI anywhere on the Internet.
- Completely block the event by setting the discard flag to **true**.

The sample implementations of KnFilterTopicEvent and KnFilterRouteEvent shown in Example 4-11 and Example 4-12 simply log the passed parameters, the URI, and the event contents. They then set the discard flag to **true** and return.

Example 4-11. Sample implementation of KnFilterTopicEvent.

```
KN_EXPORT KnError KnFilterTopicEvent(const KnString &moduleParams, const KnString &uri,
KnApiEvent &event, bool &discard)
{
    KnLog(KnLogNotice, "SAMPLE_MODULE:  KnFilterTopicEvent()");
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Parameters = %s", moduleParams.c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Uri = %s", uri.c_str());

    //  Print the contents of the event to the log file.
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event kn_id:             %s",
event.getKnId().c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event kn_time_t:         %s",
event.getKnTimeT().c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event kn_expires:        %s",
event.getKnExpires().c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event kn_payload:        %s",
event.getKnPayload().c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event kn_route_id:       %s",
event.getKnRouteId().c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event kn_to:             %s",
event.getKnTo().c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event kn_from:           %s",
event.getKnFrom().c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event kn_route_location: %s",
event.getKnRouteLocation().c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event expiration time:   %u",
event.getExpirationTime());

    //  Tell the router to keep the event for its intended destination
    //
    discard = false;

    return KnErrorSuccess;
```

```
}
```

Example 4-12. Sample implementation of KnFilterRouteEvent.

```
KN_EXPORT KnError KnFilterRouteEvent(const KnString &moduleParams, const KnString
&source, const KnString &dest, KnApiEvent &event, bool &discard)
{
    KnLog(KnLogNotice, "SAMPLE_MODULE:  KnFilterRouteEvent()");
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Parameters = %s", moduleParams.c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Source Uri = %s", source.c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Destination Uri = %s", dest.c_str());

    //  Print the contents of the event to the log file.
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event kn_id:             %s",
event.getKnId().c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event kn_time_t:         %s",
event.getKnTimeT().c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event kn_expires:        %s",
event.getKnExpires().c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event kn_payload:        %s",
event.getKnPayload().c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event kn_route_id:       %s",
event.getKnRouteId().c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event kn_to:             %s",
event.getKnTo().c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event kn_from:           %s",
event.getKnFrom().c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event kn_route_location: %s",
event.getKnRouteLocation().c_str());
    KnLog(KnLogNotice, "SAMPLE_MODULE:  Event expiration time:   %u",
event.getExpirationTime());

    //  Tell the router to keep the event for its intended destination
    //
    discard = false;

    return KnErrorSuccess;
}
```

The LiveServer Module API Quick Reference

Using the LiveServer module API, you can define and use your own modules for monitoring, blocking, transforming, or rerouting events, as well as to perform authorization, authentication, and other security features. The LiveServer API is written in the C++ language. The API calls are listed and defined in a set of HTML files that are packaged with the LiveServer; you can access them by opening the /*LiveServer_Root*/kn_module_api/html/index.html file.

This section provides a quick reference to the functions in the API which you can use to create modules for managing topics, routes, and security.

Note: This section is provided to give you an idea of the capabilities of the API. For specific information on each call, information on what have changed since this document was written, and information on new calls, see the API documentation included in your LiveServer installation.

This section is divided into the following sections:

- "Required Module Functions" on page 130
- "KnApiEvent" on page 131
- "KnRequest" on page 132
- "Understanding the Security Module API" on page 132

Required Module Functions

Your module must provide implementations of some or all of the following functions. These functions are invoked by the LiveServer at various stages.

These functions are required of all modules:

- KnModuleStart
- KnModuleStop
- For clusters, KnShutdownNode and KnShutdown. These are general-purpose APIs that provide "with restart" and "without restart" flags.

These functions are required only for topic and route modules:

- KnFilterTopicAttach
- KnFilterTopicDetach
- KnFilterTopicEvent
- KnFilterRouteAttach

- KnFilterRouteDetach
- KnFilterRouteEvent

KnModuleStart

The LiveServer's module API passes two parameters into KnModuleStart. One of them, a path argument, is the path name of the module's shared library, and can be somewhat useful, e.g., in setting up for further dynamic loads. The other contains information as shown in Table 4-2.

Table 4-2. Contents of the "info" set.

Parameter	Contents
server_name	Will usually be "LiveServer"
server_version	Includes the build number
host_name	Canonical host name for the LiveServer (in the case of clusters, the load balancer name)
realm_name	HTTP security realm name for the LiveServer (usually "runtime")
url_root	Usually "/kn"
page_root	Directory that pages are rooted in
home_directory	AOLServer home directory
configuration_directory	Location of the configuration directory
module_name	Name of the module (as inferred from the load path [strip leading directory names and trailing .dll or .so extension])

Neither the path nor the info parameter tells the module what it's called in the configuration file. The Ns_ModuleStart does tell the module that. This makes it easier to get configuration information without hard-coding names into the module.

KnApiEvent

This class is used to represent an event and provides methods for retrieving and setting the event's contents.

KnRequest

This structure is used to represent a request sent to a LiveServer. It contains members that represent the request's contents.

```
struct KnRequest
{
        KnString line;                /* request line sent by the client */
        KnString method;              /* method requested by the client */
        KnString protocol;            /* protocol part of URL or empty */
        KnString host;                /* host part of URL or empty */
        uint16_t port;                /* port specified in URL, or 0 */
        KnString srcUrl;              /* URL where the request came from */
        KnString dsturl;              /* normalized URL, without any query data */
        KnString query;               /* any query data appended to URL */
        KnString peer;                /* IP address of the client */
        double   version;             /* version of the client request */
        KnString authUser;            /* decoded user name */
        KnString authPasswd;          /* decoded user password */
        size_t   contentLength;       /* number of bytes sent by the client */
};
```

Understanding the Security Module API

The security module API provides classes for the creation of authentication and authorization modules that you can then load into the LiveServer to manage user authentication and authorization. Included in the API are classes that define the storage of user data with the creation of routes. This allows LiveServer administrators to determine which user created the subscription. LiveServer administrators may need this information if they delete a subscription that is continuously failing.

You can decide what kind of information you wish your module to receive. The struct KnRequest includes constants for such things as the IP address of the requestor (peer), the TCP port of the requestor (port), and other important information, any or all of which you can use in your implementation.

In addition, the security module API classes include callback functions that you must implement in your security modules for use during LiveServer start-up and shutdown.

Authentication

The security module API provides a single authenticate function, which the LiveServer calls only once. The authenticate function is given access to the unprocessed HTTP request in its entirety as the parameter req. The information in KnRequest provides the information; it is up to the security module to define what req is. The authenticate function returns a KnError with a message indicating success, failure, unauthorized user, and so on. Any response other than success causes the LiveServer to bounce the request back.

Authorization

Authorization is very similar to authentication, except there are two authorize functions as opposed to the single authenticate function.

- The first authorize function is called when any request other than a LiveServer request enters the LiveServer. It is used for Web page authorization, to verify whether the requestor has access to Web pages. This function is called only once for any given request.
- The second authorize function is called for fulfilling LiveServer requests.

You do not need to define both; for example, you may not want to lock down your Web pages, but only your server access, in which case you would only define the second authorize function. The following discussion provides more details about these functions. For more information on LiveServer lockdown, see the *KnowNow LiveServer Administration Guide*.

Encapsulating HTTP Requests

KnRequest is a struct in the security module API that provides constants for your use in authorization. In addition to such things as constants for the IP address, TCP port, incoming HTTP headers, URLs, decoded user name, and so on, it also defines KnAuthObject_e. KnAuthObject_e defines five types for use with the authorize functions for encapsulating HTTP requests:

- KnAuthObject_Web
- KnAuthObject_Command
- KnAuthObject_Event
- KnAuthObject_Topic
- KnAuthObject_Route

The first authorize function uses the first type (KnAuthObject_Web). The second authorize function uses the remaining four types. The second authorize function also uses two methods, KnAuthMethod_Access and KnAuthMethod_Modify. You need to include these two functions in your authorization module (as needed), but you do not need to implement them.

KnAuthObject_Web

When an incoming HTTP request is for a Web page, your module uses the first authorize function to assign the request to the KnAuthObject_Web type.

KnAuthObject_Command

Certain do_method headers are handled one at a time by the authorize function. These do_method headers are

- lib
- whoami

For more information on the do_method commands, see "do_method Command Reference" on page 49.

KnAuthObject_Event

When an event is added, deleted, or updated in a topic, the authorize function is called to see if the user has permission to make those changes in that topic.

KnAuthObject_Topic and KnAuthObject_Route

When a topic is being added, deleted, or updated, the authorize function is called to see if the user has permission to make those changes to that topic. The same is true for routes. Because the interactions can be more complex, these two calls warrant further explanation.

As you may recall from the discussion about meta-topics under "Topics and Events" on page 9, topics and routes are stored as events inside the two meta-topics /kn_subtopics and /kn_routes. It is possible to publish and subscribe to these meta-topics to receive information about changes to topics and routes.

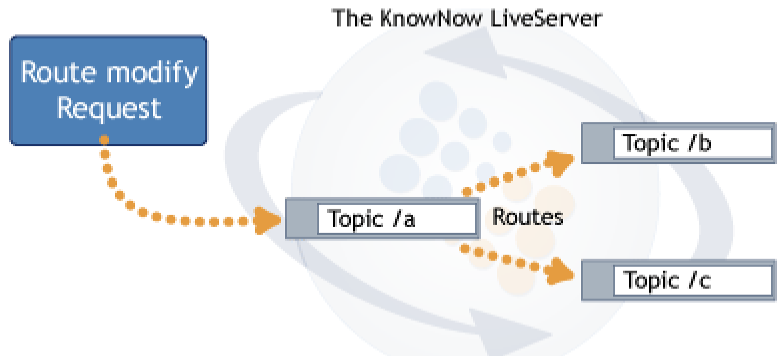
If a user has permission to subscribe to or create subtopics for a given topic, the LiveServer assumes that the user has permission to create topics on that topic's tree.

Let us say, for example, that a user wishes to create a subtopic /four to topic /one (so that the tree would look like this: /one/two/three/four), but where the two "higher up" topics, /two and /three, do not exist. The LiveServer, through the authorization module, will check to see if the user has permission to create /four. If so, it will create /two/three, even if the user might not have access to those two subtopics.

In a more complex example, let's examine the case where a user is modifying (adding, deleting, or changing) a route that someone is subscribed to. When the request comes into the LiveServer, the LiveServer calls the authorization module twice; once to make sure that the user has permission to modify that route, and the second time to make sure that the user has permission to create an event in the topic that the user is routing to.

Using Figure 4-1 as an example, if the user wishes to create a route between topic /a and topic /b, the first call makes sure that the user has permission to create that route, and the second call makes sure that the user has permission to create an event in topic /b.

Figure 4-1. The LiveServer calls the authorization module twice in some cases.



Each time an event traverses the route from /a to /b, the credentials of the person who created that route are checked to see if that original person still has permission to send events along, even if it is another user sending the new event.

Offhost Routing

If an event is being sent to an offhost route where the receiving entity requires credentials, you can specify those credentials through a do_method command. In such offhost events, where the event is being sent directly to another entity that accepts HTTP posts (such as another router, Web server, and so on), and the event is being sent to a full URL via HTTP or HTTPS, the credentials used for that route are defined using the following do_method command and associated kn_ headers, which provide information on the source and a full URL for the destination:

```
do_method=add_route
kn_from=/Source
kn_to=http://www.example.com/more/page/html
```

In addition, one of two headers is sent with this add_route command:

- kn_cookie=cookie
- kn_authorize=AuthorizationScheme

The add_route command passes along whatever is defined in either of these two headers.

Return Values

An authorization function is expected to return one of the values listed in Table 4-3.

Table 4-3. Authorization values.

Value	Meaning
KnErrorSuccess	Access to the resource is allowed.
KnErrorUnauthorized	Access to the resource cannot be granted because the supplied credentials are incomplete.
KnErrorForbidden	Access to the resource is denied

Chapter 5 Common Connector Capabilities

All Connectors manage communications between the LiveServer and your application. While you do not need to use a Connector, a Connector can make your life much easier by simplifying the calls and operations your application needs to perform. Therefore, when creating or customizing your applications to perform publish and subscribe operations with the LiveServer, you may want to have your application use one of the KnowNow Connectors instead of dealing directly with the LiveServer.

This chapter describes the capabilities that are common to most or all of the KnowNow Connectors under the following headings. For information about how each specific Connector accomplishes these capabilities, including information that is unique to each Connector, see the chapters on the specific Connector.

- "Basic Connector Capabilities" on page 138
- "Advanced Connector Capabilities" on page 142
- "Cursors" on page 146
- "Request/response" on page 148

Basic Connector Capabilities

All Connectors perform publish, subscribe, and unsubscribe operations and manage other communications between your application and the LiveServer. Each Connector needs to communicate with a LiveServer. The specifics of how those operations and communications are accomplished are described in the individual Connector chapters. The discussions in the following sections apply to all Connectors:

- "Connector Rules" on page 138
- "Connectors, Tunnels, and Journals" on page 138
- "Publishing and Subscribing" on page 139
- "Unsubscribing" on page 139
- "Session Management" on page 139
- "Handling Status Events" on page 140
- "Using Filter Modules" on page 140
- "LiveServer/ESS Installations" on page 141

Connector Rules

The following rules apply to Connectors:

- One or more Connector instances can talk with the same LiveServer.
- Each instance of a Connector talks with only one LiveServer.
- Each instance of a Connector can have multiple publications and subscriptions.
- Each application can have multiple instances of a Connector, one for each LiveServer of interest.

Connectors, Tunnels, and Journals

When a Connector connects, it does the following:

1. It creates a journal. By convention, the journal is named /kn/who/*person*.
2. It creates a tunnel. A tunnel is a route from a journal to a network endpoint. (Therefore, tunnels are also referred to as tunnel routes.)

The following facts apply to tunnel routes and journals:

- It is possible to create more than one tunnel route from a journal, though the Connectors discourage this.
- Tunnel routes are not identified with an end user.

- Every tunnel route is counted as a connection.
- KnowNow thinks of every journal as a session.
- Tunnel routes come and go much more often than journals; in fact, you can configure journals to stay alive for days.

Publishing and Subscribing

Each Connector's API provides a publish function for publishing an event to a topic on a LiveServer. Events that are published to a topic are available for your or another application to subscribe to; that is, when an application subscribes to a topic, the LiveServer notifies all subscribers, and may also perform other tasks depending on routes that may have been established and filters that may have been applied to those topics and routes. One or more subscribe methods are provided to support subscribing to topics and to support advanced capabilities, such as all since last connect.

Publishing in Batch Mode

With the exception of the LiveC++ Connector, all Connectors support publishing in batch mode with a batchPost API. For more information, see both "batch" on page 50 and the individual Connector's API documentation on batchPost.

Unsubscribing

Each Connector's API supplies a function for unsubscribing from a topic. When an application unsubscribes from a topic, the LiveServer stops delivering to that application events that have been published to that topic.

Session Management

As described under "Session Management" on page 5, LiveServer operates in session management mode when configured to operate using authentication policies 2, 3 or 5 as specified in the *KnowNow LiveServer Administration Guide*. (These policies are governed by the **Authentication policy** parameter.) For Connectors to operate properly in session management mode, applications will need to make two additional API calls: login and logout.

- login. An application calls the Connector's login method after opening a connection to the LiveServer and before any other Connector operations.
- logout. An application calls logout after unsubscribing to all topics and prior to closing a connection to end a session.

You *must* call login and logout when the LiveServer is operating in authentication policy mode 2, 3, or 5. Note that if you call login and/or logout when the LiveServer is operating in policy 1 or 4, the Connectors will behave correctly, so it doesn't hurt to always call them.

A server-side session is maintained when login is called when policies 2, 3, and 5 are in effect. Correspondingly, when one of those policies is in effect, the session is destroyed when logout is called.

We also support *n*-tier deployment wherein the authentication layer can be different than the LiveServer.

Cookie APIs make it possible for users to log into a particular session. An API is also provided for adding and removing cookies that will be sent to LiveServer. This feature is particularly useful in sharing sessions among different Connectors. Note that the getter/setter methods for cookies depend on the type of Connector.

For detailed information, see each specific Connector's API documentation.

Handling Status Events

When the LiveServer processes a request made by your application, it often generates status events that describe the outcome of the request. These status events are normally written in response to an inbound HTTP request to the LiveServer. Your application can implement a handler to handle the status events or redirect the events to any URI, including other topics on the LiveServer.

Using Filter Modules

The LiveServer ships with some filter modules you can use immediately. You can also create your own custom filter modules. Filter modules perform filtering operations on topics and routes, such as logging the contents of each event, or performing event data transformation or translation. When a filter module is attached to a topic, control is passed to the module each time an event is received or published for that topic. When a filter module is attached to a route, control is passed to the module each time an event is traversing the route. Your application can attach a configured module to a topic or a route. A single module can be attached to more than one route or topic. Your application can also detach a module from a topic or a route.

For an overview of the kinds of modules the LiveServer supports and what the LiveServer filter modules can do, see "Filter Modules" on page 99 For information on using the KnowNow System Administration console to load, attach, detach, and unload filter modules, see the *KnowNow LiveServer Administration Guide*.

LiveServer/ESS Installations

In a LiveServer/ESS installation, if HTTPS is enabled and HTTP is disabled, ESS will not start. The issue here is that the server must be able to provide the local host or a local interface IP address to the certain processes, and Connectors must be able to connect to that IP address. However, the host name for that URL will not match the host name in the certificate. In addition, in many clustered situations, the cluster name itself is not reachable from the specific server node. So it is required that the client (in this case, the Connector) be able to ignore the mismatched host name.

To resolve this issue, disable certificate validation for clients connecting to LiveServer using HTTPS protocol by setting one of the following Java system properties to **true**:

- HTTPClient.disableCertificateVerification
- KN.disableCertificateVerification

Advanced Connector Capabilities

With a few exceptions as noted in the text, all Connectors share the capabilities discussed in this section. The individual chapters for each Connector that supports these capabilities include information that is unique to that Connector. For information on using the specific calls, refer to the Connector's API documentation.

- "All Since Last Connect (ASLC)" on page 142
- "Cursors" on page 143
- "Exception Handling" on page 143
- "Heartbeat" on page 143
- "HTTP 1.0 Keepalive" on page 144
- "Licensing" on page 144
- "Logging" on page 144
- "Presence" on page 144
- "Proxies" on page 144
- "Reconnect/retry" on page 145
- "Request/response" on page 145
- "Topic Properties" on page 145

All Since Last Connect (ASLC)

Each Connector has a version of a subscribe command that includes all since last connect (ASLC) capabilities. ASLC makes it possible to, for example, receive all events that had been posted since the last connection. When you set up a subscription using the subscribe command that supports ASLC, you can specify a filter that will enable you to replay all messages in a topic in one of two ways:

- based on the time that the LiveServer receives the event (using kn_time_t)
- based on the event ID (using kn_event_id)

For example, suppose you are subscribing to a topic and are receiving events from that topic, and then there is a network failure between your application and the LiveServer. If another application is still connected and is still publishing to that topic, of course your application will not see those events. If you are using the ASLC capability, when you reconnect, you can retrieve all unexpired events from the point of failure in one of the two ways listed above.

If events have short expiration times, you may wish to use the time-based ASLC feature rather than the event ID. If you are using an event ID, and the event has expired, any events that occurred during the time between the last connection and the next reconnection will not be retrieved, and instead only new events will be retrieved.

Let us say, for example, that the last event received before disconnecting was event 4, and events 5, 6, 7, and 8 were received on the LiveServer since the last connection. If event 4 has not yet expired when the Connector reconnects, the Connector will receive events 5, 6, 7, and 8, and of course any new events while the connection is active. If event 4 has expired, however, the Connector will not receive events 5, 6, 7 and 8, because the LiveServer no longer "knows" about event 4, and therefore has no reference. Instead, the Connector will receive all new events that are generated while connected this time.

Cursors

Events can be manipulated as pages (with a configurable number of events per page), with cursor APIs that make it possible to move through events in a topic by pages (move to next page, move to previous page), jump to the start of a set of events in a topic, jump to the end of a set of events in a topic, and so on. For more information, see "Cursors" on page 146.

Exception Handling

The Windows Connectors and the LiveJava Connector each have a KNException class that supports error codes and native exception handling so that you can more easily write code to manage exceptions (such as a publish failure or a LiveServer connection failure).

Heartbeat

The Windows Connectors and the LiveJava Connector support the LiveServer's heartbeat capability, which makes it possible to determine whether a Connector is present and for a Connector to confirm the presence of a LiveServer. The LiveServer posts events to a topic named /kn_system/heartbeat, and the Connector subscribes to that topic to check on the health of the LiveServer.

The heartbeat feature can be started and stopped.

For more information, see the chapters on those Connectors.

HTTP 1.0 Keepalive

KnowNow's HTTP 1.0 Keepalive capabilities keep connections open while publishing events, thereby removing the time taken in opening and closing the connection every time and hence increasing the publishing rate. If you don't need the improved performance provided by this feature (for example, if you are publishing less frequently), you can override this feature. The Windows Connectors' Parameters class and the LiveJava Connector's KNJServer class support this capability.

Licensing

For licensing purposes, Connectors need to identify themselves to the LiveServer. APIs make it possible for a Connector to provide its Connector type and version number. For more information, see "kn_connection" on page 62.

Logging

For debugging purposes, the Windows Connectors and the LiveJava Connector support logging. The JavaScript Connector has no logging support, but it offers similar functionality through kn_debug flags.

Presence

Presence allows applications to monitor topics so that information about subscribers can be known: whether they are subscribing, unsubscribing, and so on.

To start listening to a topic for the purposes of presence, use the appropriate method (such as the Windows Connectors' PresenceSubscribe, the LiveJava Connector's presenceSubscribe, or the LiveBrowser's kn_presence_obj) to specify a topic, a listener, and handlers.

If you want to stop paying attention to a topic, use the appropriate method (such as the Windows Connectors' PresenceUnsubscribe) to specify the topic to stop listening to.

Proxies

In order to ensure that events are delivered, Connectors can detect and handle proxies and reverse proxies in the following ways:

- They can flush a proxy.
- They can handle a reverse proxy cache.
- They can handle a reverse proxy URL address instead of a port number.

These operations are automatically performed; you don't need to do any coding to invoke these actions.

Reconnect/retry

Connectors can detect a disconnection from the LiveServer. When a disconnection occurs, the Connector attempts to reconnect. When it is successful at reconnecting, the Connector fetches all the events that were published while the connection was down. These operations are automatically performed; you don't need to do any coding to invoke these actions.

Request/response

Request/response enables client applications to submit requests to named service providers and to receive the results in a reliable manner using the LiveServer's publication/subscription system. For a detailed discussion of this capability, see "Request/response" on page 148.

Topic Properties

You can apply certain properties to a topic using the Windows Connectors' SetTopicProperty method or the LiveJava Connector setTopicProperty method. Valid properties are:

- kn_default_kn_expires
- kn_expires
- kn_filtername
- kn_filterparams
- kn_max_queue_size
- kn_response_format
- kn_temporary
- kn_temporary_expires
- kn_to

Cursors

Cursors make it possible to view events in a topic a "page" at a time. They also support moving through each page of events, or paging forward and backward in a set of events. With the exception of the JavaScript Connector, all the Connectors support cursors.

On the LiveServer side, support for cursors is provided through the kn_history_* set of headers. On the Connector side, each Connector that supports cursors has APIs that support the LiveServer's cursor capabilities. The Windows Connectors use a class named ICursor. The LiveJava Connector uses a class named Cursor.

There are three types of cursors: one type of static (or client side) cursor and two types of dynamic (or server side) cursors.

Static (Client Side) Cursors

A static cursor uses only the set of events that existed at cursor initialization; basically, a static snapshot of events from the LiveServer. When subscriber applications ask to initialize the cursor, the Connector fetches all the events from LiveServer, starting from the position given by the application, and caches all these events in memory. When subscriber applications ask for the next or previous page, the page is returned from the cached events on the client side. Any events that occurred on the LiveServer side after cursor initialization are not included in the set.

Dynamic (Server Side) Cursors

With dynamic cursors, when the subscriber application requests a page of events, the Connector subscribes to the topic on the LiveServer that holds the events of interest. (As opposed to retrieving all those events and stashing them on the client side.) Therefore, a dynamic cursor uses both the events that existed at the time the cursor was initialized as well as new events as they arrive, providing applications with real-time updates. When the Connector receives all the events for a page from the LiveServer, it returns the prepared page to the application. Any events that occurred on the LiveServer side after cursor initialization are also included in the set. Because the events are not cached, memory usage is much lighter.

Because of how dynamic cursors work, a request for a page is synchronous and is blocked until the Connector receives all the events for that page.

Sometimes, the LiveServer has fewer events than are asked for. In this case, the dynamic cursor returns as many events as are available on the LiveServer, even if there are fewer than would normally fill a page, and then it waits for enough events to fill the page. As soon a new event arrives, the cursor uses a listener callback to notify the application.

Both types of dynamic cursors have all the above capabilities. In addition, a second type of dynamic cursor can also notify subscriber applications about event deletions (also via a listener callback). With the first type of dynamic cursor, when it is initialized, it obtains existing and new events, but if an event is deleted after the events were retrieved, the deleted events remain in the list. With the second type of dynamic cursor, when an event is deleted, a notification is sent from the LiveServer so that the application can prune the deleted events from the list, if desired.

Cursor Capabilities

During initialization, subscribers specify the number of events in a topic that are to be considered one page. One or more cursors can be opened using one Connector instance. When initializing the cursor, subscribers also specify

- The cursor type (static or dynamic).
- The starting position type. There are four starting position types:
  - Start with an event after a specified time. For more information on this type, see the description of "kn_history_since_age" on page 67.
  - Start with events younger than *n* time units old. For more information on this type, see the description of "kn_history_since_n" on page 68.
  - Start using an event ID.
  - Start using an event time
- If an event ID or event time is used, a value for the event ID or time.

Subscribers can perform the following types of operations with cursors:

- They can ask for events from any page.
- They can ask to move to the next or previous page in the set of pages, or to the start or end of a set of pages.
- They can ask for the last *n* events (for example, the last ten events), or all events in the last *n* time units (for example, all events in the last ten seconds).
- They can browse by event ID and event time.
- They can reset the cursor using an event ID or event time (using the ICursor.SetPos or Cursor.setPos call).

Request/response

Request/response enables client applications (requestors) to submit requests to named service providers (referred to as providers in this discussion) and to receive the results in a reliable manner using the LiveServer's publication/subscription system. The LiveServer and all Connectors support request/response.

Note: This discussion describes the LiveServer's request/response system. An additional request/response system supported by the LiveBrowser, referred to as LWWS, is described in the *KnowNow LiveBrowser Developer's Guide*.

All communications between requestors and providers are facilitated through the KnowNow LiveServer. Requestors and providers must both use either HTTP or HTTPS to communicate with the LiveServer. The LiveServer accepts requests, directs them to the appropriate provider, collects the responses, and then passes them back to the original requestor. Essentially, the LiveServer serves as a bridge between requestors and providers, greatly simplifying communications between the two types of clients.

Some advantages of using the KnowNow system for request/response are that it

- is simple
- makes it so that requestors do not need to know anything about the providers (such as how many there are, where they are located, what protocol they speak, and so on)
- makes it so that clients can make requests and receive responses using simple publish and subscribe commands, just as with other LiveServer functions

Although it isn't required, the most common scenario with request/response is for you to use a service manager for each topic for which you wish to manage request/response, and then to register each service manager with the LiveServer. The LiveServer treats service managers, along with providers, as special kinds of routes. The service manager manages services for that topic on the provider end, performing load balancing or whatever other kinds of actions you wish.

Requestors and providers can also use a Connector to simplify communications with the LiveServer and to handle the request/response endpoint semantics. The Connectors provide a simple set of commands to make requests, and an event-driven response handler for receiving the responses. Connectors are also useful for connecting requestors and providers to the LiveServer, but are not required for using some of the basic request/response features.

The topics in this section are as follows:

- "Request/response Basics" on page 149
- "Request/response Tasks" on page 149
- "Setting Up Request/response" on page 150
- "Using Request/response" on page 154
- "Request/response Authorization" on page 157

Request/response Basics

The request/response semantics are implemented with KnowNow's LiveServer framework. This provides you with the power of request/response together with the flexibility and simplicity of a publish and subscribe system.

There are two required kinds of parties in a request/response system—requestors and providers.

- A requestor is an application that is interested in receiving information from a provider service. It can communicate with that service by posting requests to a single topic on the LiveServer. The requestor eventually receives a response, either an error event or an event that contains the results of that request from the provider. The requestor handles that response when that response comes back.
- A provider is the application that accepts a request, processes it, and then returns a response through the topic of interest.

A third party, the service manager, manages requests for a particular topic. The service manager is not required but is typically used when there are multiple providers.

With request/response, one or more requestors can submit requests to the KnowNow LiveServer for fulfillment by one or more providers. As those requests come in, they are queued in the topics of interest. Then the LiveServer either routes those requests on to the providers, or, if there is a service manager for that topic, the LiveServer passes the requests on to the service manager, which then assigns provider IDs to the requests so they can be sent on by the LiveServer to the provider(s) for fulfillment. When a provider finishes servicing the request, it sends a response back to the topic by way of the LiveServer, and the LiveServer sends the results back to the requestor.

Request/response Tasks

At a high level, here are the tasks that need to take place for request/response. The first three tasks are setup tasks that are performed only once per service manager or provider. Most of the remaining tasks occur for each request.

1. If you wish to use one or more service managers, you need to create them. (There can be only one service manager per topic.)

2. Service managers must register themselves with the LiveServer. (Service managers can also be deleted.) For instructions on these tasks, see "Creating and Registering Service Managers" on page 151 and "Deleting Service Managers and Providers" on page 156.
3. Providers need to register themselves with the LiveServer. (Providers can also be deleted.) For instructions on these tasks, see "Registering Providers" on page 152and "Deleting Service Managers and Providers" on page 156.

This ends the setup phase. The remaining tasks are part of the normal request/response interactions.

1. Requestors send requests to the topic they are interested in. Requests are events that are being passed back and forth. The request is queued in the topic for further handling. For instructions on sending requests, see "Creating a Requestor Journal" on page 153.
2. The LiveServer notifies the service manager each time a request is queued in the topic the service manager is managing. The service manager then updates the request by assigning a provider ID, after which the LiveServer sends the request off to the correct provider. For instructions on this task, see "Updating Requests" on page 156.
3. The provider sends its response back to the LiveServer, which then passes it on to the correct requestor. For instructions on this task, see "Sending Responses Back to the Requestor" on page 156.

Note: For the sake of simplicity, in the following examples and discussion, only one topic and only one service manager are used. The topic is referred to as /RRTopic (for request/response topic), and the service manager is referred to as the service manager. Requestors and providers are identified as R1, R2, P1, P2, and so on.

Setting Up Request/response

The initial tasks of setting up request/response can be accomplished by either communicating directly with the LiveServer or by using a Connector. These tasks are

- "Creating and Registering Service Managers" on page 151
- "Registering Providers" on page 152
- "Creating a Requestor Journal" on page 153

Creating and Registering Service Managers

A service manager is a client application that controls the flow of requests to providers for a given topic. It can be used to perform load balancing or other tasks. The service manager communicates with the LiveServer via HTTP. From the LiveServer's point of view, a service manager is a special kind of route that controls all activity associated with a particular request/response topic. Requests flow to the service manager through a tunnel connection. Requests flow back to the LiveServer via HTTP POSTs. You are not required to create a service manager, but in most cases, you will probably want to.

Service managers have essentially three tasks they need to perform in order to participate in activities relating to the topic of interest.

- One time only, a service manager must register itself with the LiveServer. This task can be accomplished by going directly to the LiveServer, or by using the LiveJava Connector.
- For each provider, the service manager must register that provider with the LiveServer.
- For each request, the service manager assigns provider IDs to that request. Information on that task is provided under "Updating Requests" on page 156.

When you register a service manager with the LiveServer, the following requirements and restrictions kick in for requests to request/response topics:

- A request/response topic can only have one service manager.
- For request/response topics that have service managers, all new requests are sent to the service manager. The service manager now controls everything that is going on in the topic it is servicing.
- Incoming requests must **not** provide a provider ID. The service manager is now the only one to assign provider IDs to requests. The request/response topic rejects all new incoming requests that already have a kn_provider_id header (sending back a "403 Invalid Request" response). Any valid requests are sent to the service manager to have provider IDs assigned.

Registering Service Managers Directly with the LiveServer

The service manager registers itself with the LiveServer in two steps.

1. First, it calls the add_journal command.

```
do_method=add_journal
kn_from=/who/SMAdmin/connector/123/kn_journal
```

The kn_from command identifies the name of the service manager's journal.

2. Next, the service manager creates a route from the request/response topic to the service manager's journal:

```
do_method=add_route
kn_from=/RRTopic
kn_to=/who/SMAdmin/connector/123/kn_journal
kn_id=kn_request_manager
kn_module=kn_request_response
```

   kn_from identifies the request/response topic that the service manager will be managing. kn_to identifies the service manager's journal as created in step 1. kn_id and kn_module must use the kn_request_manager and kn_request_response values exactly as shown. This tells the LiveServer that this route is used for the service manager.

Using Connectors to Register Service Managers

To register a service manager using a Connector, use the appropriate command with the appropriate parameters as described under "Registering Service Managers Directly with the LiveServer" on page 151. The commands for adding a service manager are AddRequestManager for the Windows Connectors and addRequestManager for the LiveJava Connector.

It is expected that the service manager has defined the URI and subscribed to it before calling AddRequestManager or addRequestManager. The Connector and the LiveServer take care of the rest.

Registering Providers

Providers provide responses to requests for particular topics. Like service managers, providers are treated by the LiveServer as a special type of route from the request/response topic.

For a provider to be used, it needs to be registered with the LiveServer. Providers are registered with the LiveServer by creating a journal connection to the provider in the LiveServer, and then by adding a route to that journal as described in this section. If there is a service manager for the topics the provider is interested in, the service manager will take care of the registration. If not, the provider needs to register itself with the LiveServer. In either case, the required actions and commands are the same.

Once a provider is registered, for each request that is sent to that provider, the provider provides a response. Information on the KnowNow commands to accomplish this task is provided under "Sending Responses Back to the Requestor" on page 156.

Registering a provider with the LiveServer is a two-step process (which needs to be performed for each provider).

1. First, create a tunnel connection from the LiveServer to the provider. (This step is not required, but is the most usual scenario.) To create the tunnel connection, use the add_journal command.

   ```
   do_method=add_journal
   kn_from=/who/P1/connector/456/kn_journal
   ```

   The kn_from command is the name of the provider's journal. If using the LiveJava Connector to register the service manager, the Connector will assign the number.

2. Next, create a route from the request/response topic to the journal for the provider's tunnel.

   ```
   do_method=add_route
   kn_from=/RRTopic
   kn_to=/who/P1/connector/456/kn_journal
   kn_id=P1
   kn_module=kn_request_response
   ```

   kn_from identifies the request/response topic that the provider is servicing. kn_to identifies the provider 's journal as created in step 1. kn_id uniquely identifies the provider. kn_module must use kn_request_response exactly as shown. The route will only pass to the provider requests in which the kn_provider_id is equal to the provider ID for the provider on that route.

   The API methods for registering a provider are AddProvider for the Windows Connectors and addProvider for the LiveJava Connector. The provider uses this method to register itself with the system.

Creating a Requestor Journal

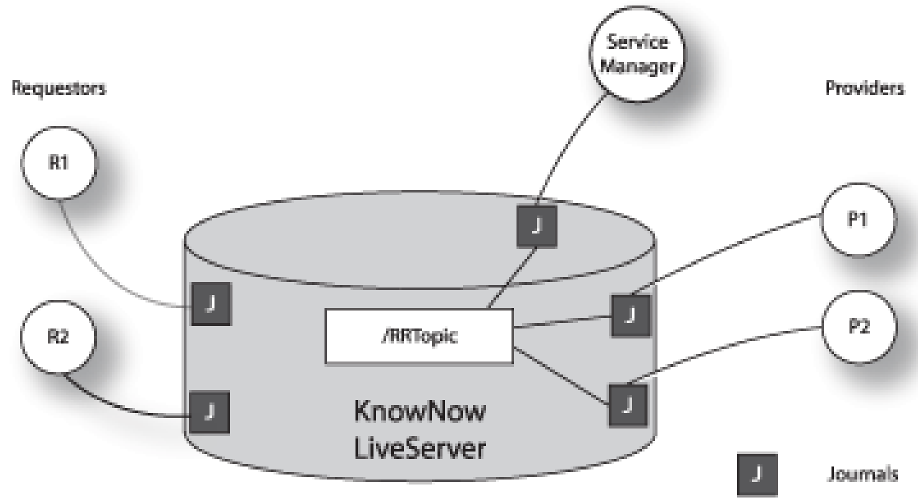
At this point, there is one more task to perform, and that is to create a journal for each requestor. Once that task is done, the system will be ready to receive requests.

To be able to send a request to a request/response topic, the outside requestor must create a journal for itself using the add_journal command, just as for the service manager and the provider. Once it sets up a journal for itself, it is ready to send requests and receive responses.

```
do_method=add_journal
kn_from=/who/R1/connector/789/kn_journal
```

At this point, the LiveServer will be set up as shown in Figure 5-1. The routes to providers will only pass along requests that have the correct provider ID (kn_provider_id) for that route.

Figure 5-1. LiveServer with providers and a service manager for a topic.



Using Request/response

After performing all the setup commands as described under "Setting Up Request/response" on page 150, you are now ready to run request/response. The tasks include

- "Sending Requests" on page 154
- "Updating Requests" on page 156
- "Sending Responses Back to the Requestor" on page 156
- "Deleting Service Managers and Providers" on page 156

Sending Requests

To send a request, the requestor uses an add_notify command.

```
do_method=add_notify
kn_to=/RRTopic
kn_id=RequestorID
kn_provider_name=/RRTopic
kn_response_uri=Requestor's Journal
kn_payload=requestpayload
```

kn_response_uri is the location to send responses to. In addition to these required headers, any other headers can also be sent as desired.

Note: The requestor must have proper credentials to make requests to a provider via the request/response topic. If the requestor is not authorized to make a request, the LiveServer will immediately return a status event with an error code.

After the request is received within the LiveServer, it is sent on to the service manager, which assigns a provider ID and forwards the request to the LiveServer, which then sends it on to the correct provider.

Note: Each request to a request/response topic must be assigned a provider ID. If it has no provider ID, it will not get sent and instead will just sit in the request queue. Normally, the service manager takes care of assigning the provider ID. However, if there is no service manager, the requestor is responsible for supplying the provider ID with the request.

Blocking versus Non-Blocking Requests

For the Windows Connectors, the API method for sending requests are AddRequest for non-blocking requests and, for blocking requests, a set of three methods, InitRequest-Blocking, AddRequestBlocking, and CleanupRequestBlocking. For the LiveJava Connector, the API methods for sending requests are addRequest and addRequest_BLOCKING.

The requestor uses AddRequest (or addRequest) to send a non-blocking request into the system, and uses the appropriate blocking methods to send a blocking request into the system. A blocking request is managed differently than a non-blocking request.

- In a non-blocking request, the requestor continues processing after submitting the request to one provider. The non-blocking request returns any immediate errors and then releases control back to the requestor. The application monitors the response URI to process the provider response. The application also monitors the request status handler to detect any errors.
- In a blocking request, the requestor makes a request to one provider and waits for the response before continuing to process. The blocking request returns the provider response. Any request errors or provider errors are also returned.

The kn_block header is used with blocking.

Updating Requests

After the LiveServer receives the request and identifies it as a request/response request, the LiveServer sends the request on to the service manager. The service manager then calls update_notify to assign a provider ID.

```
do_method=update_notify
kn_to=/RRTopic
kn_id=R1
kn_provider_id=P1
kn_provider_name=/RRTopic
kn_response_uri=URI_value_if_known
```

Once this is done, the request is sent off to the appropriate provider, and only to that provider. There is always only one provider per request, though a requestor could have different providers for different requests.

The Windows Connectors API method for this is UpdateRequest. The LiveJava Connector API method for this functionality is updateRequest.

Sending Responses Back to the Requestor

The final step in the request/response cycle is for the provider to send some sort of response back to the requestor, normally including the status of the event and information on the event to be passed on to the requestor. To send its response, the provider sends the update_notify command:

```
do_method=update_notify
kn_to=/RRTopic
kn_id=R1
kn_status=done
kn_payload=payloadcontent
```

In kn_status, the word **done** is a reserved string and must be provided in order for the response to be sent to the requestor.

The Windows Connectors API method for this functionality is AddResponse. The corresponding LiveJava Connector API method is addResponse. The provider uses this method to send the response for a specific request back to the requestor.

Note: In order to send responses, the provider must have the proper credentials on the request/response topic.

Deleting Service Managers and Providers

To delete either a service manager or a provider, you can use one of two ways:

1. You can use the delete_route command:

```
do_method=delete_route
kn_from=/RRTopic
kn_id=P1
```

   Where kn_id identifies the service manager or provider being deleted.

2. You can use the Windows Connectors RemoveProvider method or the LiveJava Connector deleteProvider method.

Request/response Authorization

Request/response topics use the same authentication and authorization schemes as all of the other topics on the LiveServer. This allows administrators to control and protect requestors and providers. This section describes the recommended authorization settings for

- Providers,
- Requestors, and
- Administrators

Providers

Providers usually need to have the ability to create, edit, and delete their own journals and routes and to add, edit, and delete events in the request/response topic (here called /RRTopic) and the /RRTopic/kn_routes. Suggested authorization settings for a request/response topic (/RRTopic) and a provider with a user name (provider_name) are shown in Table 5-1.

Table 5-1. Provider authorization settings (allow create/modify/delete topics).

Access	Inheritance	Command	Topic	User Name
allowuser	inherit	GET	/RRTopic	provider_name
allowuser	inherit	POST	/RRTopic	provider_name
allowuser	inherit	topic	/RRTopic	provider_name

Requestors

Requestors usually only need to have the ability to create, edit, and delete events in the /RRTopic/kn_requests topic. Requestors do not need to have direct access to any other request/response topics or other topics or to allow routes out of any topics. Table 5-2 shows the suggested authorization settings for a requestor (requestor_name) that makes requests to a request/response topic (/RRTopic).

Table 5-2. Requestor authorization settings.

Access	Inheritance	Command	Topic	User Name
allowuser	noinherit	GET	/RRTopic	requestor_name
allowuser	noinherit	POST	/RRTopic	requestor_name
allowuser	noinherit	notify	/RRTopic	requestor_name

Administrators

Administrators must be able to create, edit, or delete any topic or event. They must also be able to monitor the activity in request/response topics. Table 5-3 shows the suggested settings for an administrator that wants to control a request/response topic (/RRTopic).

Table 5-3. Administrator authorization settings.

Access	Inheritance	Command	Topic	User Name
allowuser	inherit	GET	/RRTopic	admin
allowuser	inherit	POST	/RRTopic	admin
allowuser	inherit	topic	/RRTopic	admin
allowuser	inherit	notify	/RRTopic	admin
allowuser	inherit	route	/RRTopic	admin

Monitoring

Administrators may sometimes want to monitor the requestors and the providers. They can easily do so by subscribing to the /RRTopic and the /RRTopic/kn_routes topics

Note: Administrators **must** set the kn_deletions flag to **true** so that subscriptions can receive notifications when requests or providers expire or are deleted.

Chapter 6 Using the JavaScript Connector

All Connectors manage communications between the LiveServer and your application. While you do not need to use a Connector, a Connector can make your life much easier by simplifying the calls and operations your application needs to perform. Therefore, when creating or customizing your JavaScript application to perform publish and subscribe operations with the LiveServer, you may want to have your application use the LiveJava Connector instead of dealing directly with the LiveServer. This chapter describes how the JavaScript Connector works and provides guidelines on using some of the most important calls in the API to create or customize your application so it can make best use of the JavaScript Connector. Be sure to also read Chapter 2 , "Events," for additional important information on the LiveServer's operations and architecture.

This chapter covers the following topics:

- "Using the JavaScript Connector" on page 162
- "Publishing Events" on page 166
- "Subscribing to Topics" on page 168
- "Unsubscribing" on page 171
- "Using Status Handlers" on page 172
- "JavaScript ActiveX Shim" on page 173
- "Simple Event Mapping Support" on page 177
- "Localization" on page 179
- "Writing Cross-Domain Web Applications" on page 181
- "Using Command-Line Parameters" on page 184
- "API Overview" on page 188

Using the JavaScript Connector

The JavaScript Connector supports JavaScript applications. It enables an application running in a Web browser to communicate with the LiveServer. The JavaScript Connector is installed when you install the LiveServer, as described in Chapter 2 . As with all Connectors, the JavaScript Connector mediates between your application and the LiveServer, managing publish, subscribe, and unsubscribe operations.

This section describes how to load the JavaScript Connector and provides an overview of how it operates under the following headings:

- "Including the JavaScript Connector" on page 162
- "Understanding the Connector's Frames" on page 163
- "Debugging Your Applications" on page 164

The rest of this chapter provides information on how the JavaScript Connector manages publish, subscribe, and unsubscribe operations.

You must also enable JavaScript support in the LiveServer as described in the chapter on configuring the LiveServer in the *KnowNow LiveServer Administration Guide*.

In addition to the information in this book, there is additional documentation available in the form of the *KnowNow LiveBrowser Developer's Guide* and the APIs in HTML format.

Including the JavaScript Connector

To use the JavaScript Connector with your JavaScript application, include a call to it in your application code, as shown in the following example. The Connector <script> element must be included at the very beginning of your application, inside the HTML <head> element and before any other <script> elements. When the page is loaded into a browser, the Connector is also loaded.

Example 6-1. inserting the JavaScript Connector.

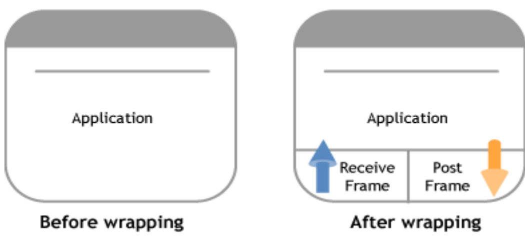
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <title>My JavaScript Application</title>
        <script type="text/javascript" src="../../js/kn_config.js"></script>
        <script type="text/javascript" src="../../js/kn_browser.js"></script>

        . . .
```

Understanding the Connector's Frames

Once the JavaScript Connector is loaded, the page stops its normal loading process and is wrapped in a frameset. The frameset contains three frames that enable communication with the LiveServer:

- A visible frame, which is where your application is loaded.
- A hidden post frame for sending requests to the LiveServer.
- A hidden tunnel (receive) frame that receives events from the LiveServer over a persistent HTTP connection.

Figure 6-1. JavaScript Connector frames.



The JavaScript Connector uses the top level of the hidden frameset, allowing the Connector to persist even when frames in your application are reloaded. When your application is loaded in the visible frame of the frameset, the Connector creates a kn object as a property of the browser window object. (For more on the kn object, see "The kn Object" on page 188.)

Since your applications will always be contained within a frameset, this may affect the way you construct your DOM (document object model) references to windows, frames, forms, and divs.

Note: If you are using Microsoft's Internet Explorer and have installed one of the Windows Connectors, see "JavaScript ActiveX Shim" on page 173.

JavaScript Access and Security Domains

If you have several HTML frames, and want to have JavaScript access across them, all frames must be within the same security domain. If you explicitly set the domain of a frame, you must also set **all** frames to that same domain. This is true even if the domain you are explicitly setting is identical to the domain that frame would have defaulted to.

For example, if two frames are being served from a LiveServer at abc.example.com, and you use JavaScript to set the domain of one frame explicitly to “abc.example.com,” the browser sees this as a completely different (and therefore hostile) domain, even though the domains are identical to we humans.

In other words, there is no implicit participation in domain security once the domain of one frame is set. **All** frames must be explicitly set in order to “play” with one another.

The recommended way to set the security domain in a frame is to use the JavaScript Connector or the do_method=whoami command. In a cross-domain setting, the URL must be fully qualified and include the full domain of the LiveServer:

```
<script type="text/javascript" src="http://kn-server.example.com/kn/
       ?do_method=whoami"></script>
```

Another way to set the domain is to use JavaScript directly, like so:

```
<script type="text/javascript">
       document.domain = "example.com";
</script>
```

Internet Explorer 2800 is much more strict in enforcing this model, while Internet Explorer 2600 is more forgiving. Regardless of approach, it is absolutely essential that all frames in a Web application be explicitly set to the same security domain with either whoami or JavaScript.

Debugging Your Applications

The JavaScript Connector suppresses errors and warnings, even those that your application generates. If you wish to see errors and warnings, set kn_debug to **on**. For more information on this and other options, see Table 6-4 on page 184.

If the LiveServer is down when you first attempt to fetch an application, then the Connector library will be missing. A test in your application can reveal this:

```
if (! self.kn) alert("We are experiencing technical difficulties. Please wait a
bit and reload the page, or send email to webmaster@example.com if the problem
persists."):
```

However, if the LiveServer is up when you first attempt to fetch the application, then the Connector will be present (and fully functional), but as soon as the LiveServer goes down, everything will stop working. In some cases you will get “permission denied” errors. The workaround is to send regular heartbeat events and start a watchdog timer from your onload handler:

```
// send a new pulse thirty seconds after we received the first one
watchdogInterval = 30 * 1000;
// allow up to five seconds for the pulse to make a round trip through the
// LiveServer; this is probably excessive, but slow network connections
```

```
// plus slow browsers have been known to delay this long.
maxHeartbeatLatency = 5 * 1000;
// watchdog subscription URI
watchdogName = "watchdog://" + self.name;
// watchdog timer handle
bone = null;
function watchdogCallback()
{
    clearTimeout(bone);
    bone = null;
    setTimeout(`watchdog()', watchdogInterval);
}

function bite()
{
    alert("We apologize, but we seem to be experiencing technical difficulties. \n" +
    "You may need to empty your browser's cache of stored Web pages. \n" +
    "Please wait a bit and then reload this Web page. \n\n" +
    "Please send email to webmaster@example.com if the problem persists.");
}

function watchdog()
{
    bone = setTimeout(`bite()', maxHeartbeatLatency);}
```

Publishing Events

Your application can use the kn.publish method to publish an event to a topic on the LiveServer. This method also gives you the option of providing status handler objects to be invoked after the event is published to handle the status event returned by the LiveServer. This method invokes the LiveServer's notify method, and allows all parameters allowed by notify. For more information on notify, see "notify" on page 52.

Your application must construct the event to be published as a JavaScript object and pass it to the kn.publish method. An event is simply a JavaScript object that has event header names as properties and corresponding event header values as property values. When you publish an event, the kn_id of the newly published event is returned.

The following example publishes an event with a user-defined kn_payload with the contents "Hello world!" and a custom property named specialProperty.

Example 6-2. Publishing an event with kn.publish.

```
// first create the object...
myEvent = new Object;

// then add some data to it...
myEvent.kn_payload = "Hello world!";
myEvent.specialProperty = "extra special value";

// finally, publish it.
kn.publish("/my/topic", myEvent);
```

After your application publishes the event, the LiveServer would receive an event with the headers shown below. The values set by your application are shown in boldface.

- kn_time_t: 987107695
- userid: test
- kn_id: 26451550
- kn_history_since_event_id: 987107695_12462_2224
- **specialProperty: extra special value**
- displayname: test
- kn_payload: **Hello world!**

Event Identifiers

Since the sample application does not specify the kn_id, a unique identifier is automatically generated, added to the event, and returned as a string for later use by your application.

Example 6-3. Saving the kn_id when publishing an event.

```
myEvent = new Object;
myEvent.kn_payload = "Hello world!";

eventId = kn.publish("/my/topic", myEvent);
```

Publishing Form Data

Your application can also use the kn.publishForm method to copy input elements from an HTML form into a new event object and then publish the event to a specified topic.

Subscribing to Topics

Your application can use the kn.subscribe method to subscribe to a topic and then receive events that are published to that topic. Any events published to the subscribed topic will be forwarded to your client and the appropriate destination function, specified by your application, will be invoked. This invokes the LiveServer's route method, and allows all parameters allowed by route.

The kn.subscribe method also gives you the option of defining the maximum number or age of events that should be returned to your application. You can also provide optional status handler objects to be invoked to handle status events returned by the LiveServer.

The following example defines an event handler that alerts the user of the kn_payload for each event received and then subscribes to the topic /my/topic.

Example 6-4. Defining an event handler and subscribing to a topic.

```
function onMessage(e){
    alert(e.kn_payload);
}

kn.subscribe("/my/topic", onMessage);
```

Understanding Destinations

The destination you specify for a subscription can be any of the following:

- Another topic.
- A JavaScript URI.
- A function or closure that accepts the received event as a parameter, as shown above.
- An object with an onMessage method that accepts the received event as a parameter.

Table 6-1. Specifying subscription destinations.

Destination	Specification
topic	kn.subscribe("/my/topic", **"/postit"**);
JavaScript URI	kn.subscribe("/my/topic", `"javascript:alert('OK')"`);

Table 6-1. Specifying subscription destinations. *(continued)*

Destination	Specification
function or closure	kn.subscribe("/my/topic", **onMessage**);
object with onMessage method	kn.subscribe("/my/topic", **{ onMessage: function (e) { alert(e.kn_id) } }**);

Subscription Options

The properties shown in Table 6-2 modify a subscription. For more information on headers for events and times, see "kn_history_*" on page 67.

Table 6-2. Subscription options.

Property	Description
do_max_n = *number*	Specifies the maximum number of previous events to be received on this subscription.
do_max_age = *seconds*	Fetches all events published to the topic up to the specified number of seconds before the subscription was opened. Specifying **infinity** will fetch all events in the topic regardless of their timestamp.
kn_deletions	Tells the LiveServer whether you want to receive notification of when events are deleted in a topic. **true** = receive notification. **false** = no notification. The default value is **false**.
kn_expires	Sets the expiration time on your subscription.

The following example shows how to specify options such that a maximum of ten events will be received by the subscription.

Example 6-5. Creating and using subscription options.

```
// first create the options object...
options = new Object;

// then set a do_max_n
options.do_max_n = 10;

// finally, subscribe to the topic and pass the options object as an argument
kn.subscribe("/my/topic", onMessage, options);
```

You may find it helpful to use the short form of object notation. Simply put the header names and values inside braces when you define the object (separating the header name from the value with a colon) as shown below:

```
kn.subscribe("/my/topic", onMessage, { do_max_n:10 })
```

Unsubscribing

When your application no longer wants to receive events from a topic, it must invoke the kn.unsubscribe method, specifying the route ID associated with the subscription. You may also provide optional status handler objects to be invoked to handle status events returned by the LiveServer.

Example 6-6. Unsubscribing from a topic.

```
function onMessage(e){
    alert(e.kn_payload);
}

aRouteID = kn.subscribe("/my/topic", onMessage);

. . .

kn.unsubscribe(aRouteID);
```

Using Status Handlers

Your application can use a status handler object to attach functions to status events received from the LiveServer. Your application must create an object with three methods and then pass it to the kn.publish, kn.publishForm, kn.subscribe, or kn.unsubscribe method when you are unsubscribing from a topic. These three methods are

- onSuccess, which is called every time a request completes successfully.
- onError, which is called every time a request fails.
- onStatus, which is called whenever any status message is received. If onStatus is defined, onError and onSuccess will not be called.

JavaScript ActiveX Shim

If your have installed the LiveActiveX Connector (one of the Windows Connectors) and your Web browser supports ActiveX, you can configure your Web browser and design your applications to use the JavaScript Connector's JavaScript ActiveX shim (to take advantage of ActiveX), instead of using the event transport mechanism described under "Understanding the Connector's Frames" on page 163.

Note: The JavaScript ActiveX Shim can only be used with Web browsers that support ActiveX and on machines where the LiveActiveX Connector is installed. The LiveActiveX Connector is described in Chapter 7 , "Using the Windows Connectors."

Configuring the Browser

Before using the shim, you will first need to change the security settings in Microsoft Internet Explorer. These settings can be changed using Internet Explorer's **Internet Options** dialog box.

Note: Access to Internet Explorer's security options may vary according to which version of Internet Explorer you are using.

To change the security settings in Internet Explorer,

1. Access the Internet Explorer **Tools** menu and choose **Internet Options**.
2. In the **Internet Options** dialog box, shown in Figure 6-2, choose **Custom Level**.

3. Locate the **Run ActiveX controls and plug-ins** item, shown in Figure 6-3, and choose **Enable**.

Warning: These settings may allow harmful programs to run on your computer, so you may wish to place the application's URL in the special **Trusted Sites** security zone, and only change the settings for that zone. Consult with your system administrator before making changes which affect your Web browser's security settings.

Figure 6-2. Explorer Internet Options dialog box.

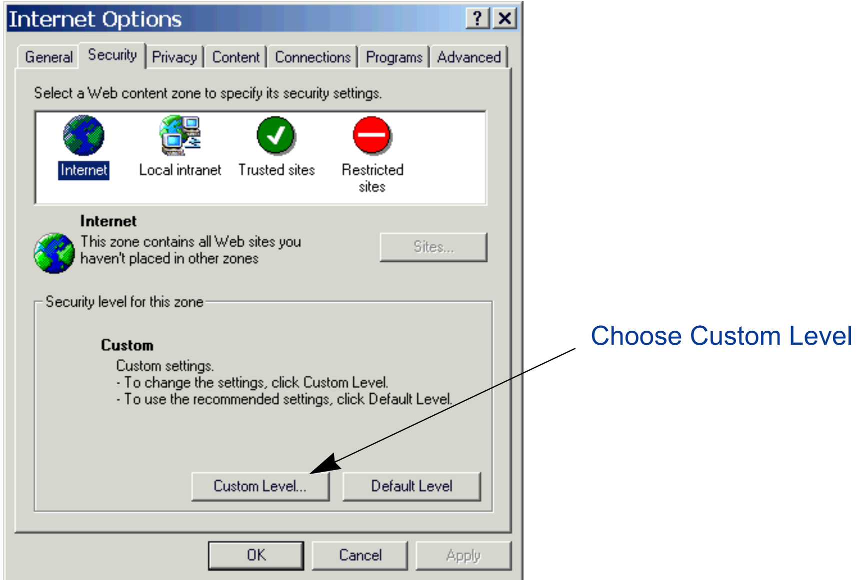
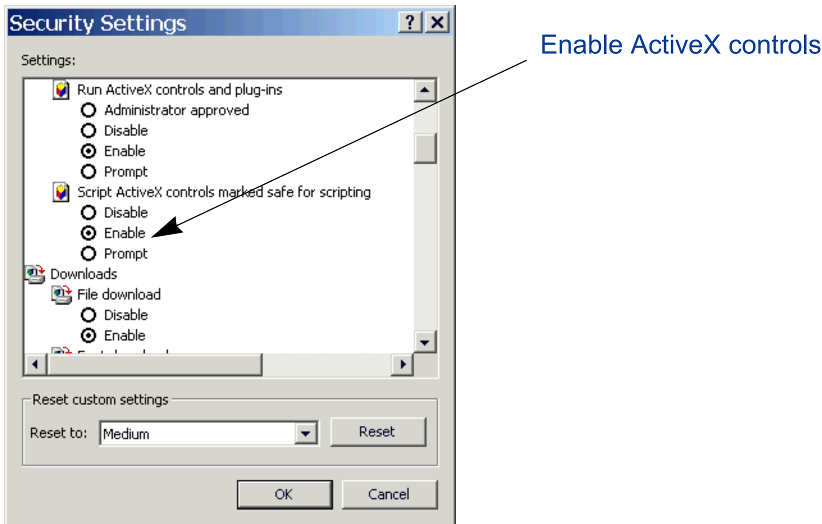


Figure 6-3. Enabling ActiveX controls and plug-ins.



Application Configuration

To use the JavaScript Connector's ActiveX shim with your application, simply include the script element shown below before including the JavaScript Connector.

Example 6-7. Including the ActiveX shim.

```
<!-- KnowNow JavaScript Connector ActiveX shim -->
<script type="text/javascript" src="/kn_apps/kn_lib/activex.js"></script>

<!-- KnowNow JavaScript Connector -->
<script type="text/javascript" src="/kn?do_method=lib"></script>
```

Shim API

The ActiveX shim disables the frameset wrapping normally performed by the JavaScript Connector. Most applications will not be affected by this change. Most of the high-level API calls of the JavaScript Connector are affected by the shim, with the following exceptions:

- Due to a bug in the Internet Explorer JavaScript-ActiveX bridge, NUL characters in received event header names and values are not preserved, and prematurely terminate strings. If the kn_options flag **single** is present, NUL characters have the same effect in outbound event header names and values.

- kn__submitRequest uses the LiveActiveX Connector and ignores the kn_options flag **noforward**. The kn_options flag **single** still has the usual effect.
- kn__restartTunnel has no effect.
- Unless the kn_options flag **quiet** is present, failure to open the tunnel connection will present an informational dialog briefly describing the shim configuration properties, and instructing the user to make any necessary changes and reload the page.

Shim Properties

The shim has several properties which you can set using either the command-line parameters or window properties. Window properties must be UTF-8 encoded and set before loading the KnowNow JavaScript Connector.

Table 6-3. Shim properties.

Property	Description
kn_activexPassword	The password with which to authenticate to the LiveServer. (This property is ignored by the LiveActiveX Connector.)
kn_activexProxy	URI of an HTTP proxy server to use for all accesses to the LiveServer. The format this field is looking for is: `ip_address:port`
kn_activexProxyPassword	The password with which to authenticate to the proxy server.
kn_activexProxyUsername	The user name with which to authenticate to the proxy server.
kn_activexServerURI	The URI of the KnowNow LiveServer to use, including user name and password. This defaults to the `kn_server` value. To access a password-protected LiveServer, you must encode the user name and password in the URI: `http://uname:pword@srvrname:8000/kn`
kn_activexUsername	The user name with which to authenticate to the LiveServer. (This property is ignored by the LiveActiveX Connector.)
kn_server	The URI or path name of the KnowNow LiveServer to use, such as: `http://servername:8000/kn`

Simple Event Mapping Support

This feature is for developers who wish to move an object from one platform to another. MIME types are used as the typing scheme, and it is recommended that users serialize their objects into an appropriate MIME type.

The simple event mapping support feature provides support for applications to map inbound events (coming in from subscriptions the applications have made) into objects/data types. It also allows applications to map objects, which they are attempting to publish, into events.

The mapping does not happen automatically. The application registers certain callbacks with the Connector, requesting that when an event with a particular header or value inside is received on a particular subscription, the Connector should call a particular callback to map the event data into an object and then process that object.

The JavaScript Connector supports plain MIME types ("text/plain," "application/xml", "image/gif", and the like) as well as the RFC-proposed "+xml" extensions for XML-derived content types.

Following is a list of examples (in an order of de-serialization preference):

- text/x-my-dtd+xml [for XML formats]
- text/x-my-dtd [for non-XML formats]
- text/*+xml [for generic XML text handling]
- text/* [for generic text handling]
- */*+xml [for generic XML handling]
- */* [the universal fallback]

The default MIME type of an event is "text/plain".

Mapping Outbound Objects to Events

On the publication side, you will need to implement JavaScript serialization (toString) methods on all objects, then use them as kn_payload values, and specify appropriate MIME types in the content-type header.

Example 6-8. Publisher side.

```
kn.publish( "/topic",
        {   "content-type": "application/x-js-date",
            kn_payload: new Date });
kn.publish("/topic",
        {   "content-type": "application/x-js-window",
            kn_payload: window });
```

```
kn.publish( "/topic", {}):
```

Mapping Inbound Events to Objects

On the subscription side, the programmer will need to register single-string-argument constructor functions corresponding to different MIME types or wildcards, and call kn.filter in event handlers to construct objects corresponding to events.

Example 6-9. Subscriber side.

```
kn.setFilter("application/x-js-date", Date);
kn.setFilter("application/*", String);
kn.subscribe("/topic",
           function (evt){
               var o = kn.filter(evt);
               alert("[" + (o ? o.constructor : typeof o) + "] : " + o);
           }
           );
```

Localization

The $$, $$_, and kn_createContext functions (discussed under "API Overview" on page 188) are all related to the localization system. The localization system is a way to make it easier to translate strings from one language to another. For example, someone could translate the strings in your application into one or more other languages. Those translations could then be loaded into your program. Once they are loaded in, you could have the Connector automatically determine which language the user has selected in their browser and use the correct translations for that user.

To support localization, the Connector makes it possible to have different contexts (localization domains). A context allows you to say in which context you want a translation, and how you want that translation to work. For example, in one context you could translate yes as appropriate for yes/no questions, and in another context you could use yes to mean something different—whatever purpose you want to put that string to.

Connector support works regardless of the size of your application. And contexts make it possible to have translations done flexibly. If you have different components or different parts of a program written by different people, each programmer can use the same string to mean different things.

In particular, there are some error messages that need to be translated differently inside some parts of the code than they are in others, even though they are the same string. To differentiate these messages, you place them inside different contexts. For example, all the strings provided by KnowNow, including the default Connector translations, are inside a context called kn. If you want to use the KnowNow translations, you can look inside the kn context. In most circumstances, you will probably want to use your own translations, which will either be in the default context (which is "", an empty string), or they will be in a context you are using for the application you are writing. (For more on creating contexts, see kn_createContext. There is also a sample application available on the KnowNow Web site that illustrates the use of contexts and translations.)

The default context, the empty string, is meant to be the primary context used in your application. If you have components you are pulling in from a third-party application, you can give each of them their own contexts. The default context is created by default; you don't need to create it explicitly. This means there is already a $ function and a $_ function that you can use in your application.

You may be wondering how you can define the translations themselves. You could call kn_localize individually for each string, but a tool, kn_localize.sh, is shipped with the LiveBrowser package to make the translation process easier. This tool automatically looks through a JavaScript file and extracts all the localized strings into a separate JavaScript file so you can translate them more easily. kn_localize.sh is located in the /*LiveBrowser_root*/js/tools/ directory. You will need a UNIX-style shell to run this tool.

When kn_localize.sh has extracted the strings, the resulting separate JavaScript file it creates will have a number of calls to kn_localizeDefault, each with a string that it is possible to translate. You can translate each of the lines in that separate JavaScript file as desired. (You don't need to translate all of them if you don't wish to.) If you want or need to, you can make multiple copies of that file, each with translations in a different language (and of course each with its own unique file name). Note that the original string must always be in double quotes. Also note that, when using kn_localize.sh, you can only have one $ call on any given line in your source code.

Once you have extracted and translated these strings, the usual way to put them back into your application is to explicitly include each of the different language translations on the applications page. To do so, use a <script> tag for each language (or, if using ASP or JSP, you can include them all on the server side when you build the page). Then the Connector will automatically choose the correct file based on browser settings and your program will use the correct translated file (assuming that the localization file (a) exists and (b) has been loaded in by the application). If you haven't defined localizations for a given language, the Connector will fall back on using the defaults or on using whatever strings you are passing in as arguments.

Writing Cross-Domain Web Applications

Cross-domain Web applications are applications where the HTML lives on a standard Web server (or is being generated by an application server), but the application wants to embed the KnowNow JavaScript Connector. For the sake of discussion, let us assume that the LiveServer is hosted at kn.example.com, and the pages that make up the application are being served from apps.example.com. The problem that you face is that the JavaScript security model in the browser does not allow the application, which comes from apps.example.com, to communicate with the KnowNow LiveServer at kn.example.com.

The JavaScript Connector library offers a built-in option to allow the HTML application and the Connector to sidestep the JavaScript security restrictions imposed by the browser, thus making your application a cross-domain application.

To create a cross-domain Web application, you must have the following:

- an intermediate level of knowledge of JavaScript
- a knowledge of the basic steps for writing a Web application
- a Web application that uses a single protocol, either HTTP or HTTPS

Note: To run your application in Netscape 4, both the Web server and the LiveServer must use the default port, either HTTP on port 80 or HTTPS on port 443. And although you must use the default port, you must **not** specify the port in the URL.

To make your application a cross-domain application,

1. Using the KnowNow System Administration console, change the **Configuration > JavaScript > Domain** and **Configuration > JavaScript > LiveServer URL prefix** parameters. For more information on these parameters, including how to set them, see the *KnowNow LiveServer Administration Guide*.
2. In the JavaScript Connector's configuration file, kn_config.js, which is installed with the LiveBrowser, modify the following two variables.
   - kn_browser_includePath. This variable sets the location of the KnowNow JavaScript components. Set this variable to the appropriate path to the KnowNow JavaScript components. You may need to set the path to a full URL.
   - kn_config_server. This variable must be set to the URL where the LiveServer accepts publish and subscribe requests; for example,

     ```
     "http://knserver.example.com/kn"
     ```

3. Modify your application code to use the URL of the LiveServer when requesting the JavaScript Connector. For more information on modifying the application code, refer to "Modifying the Application Code" on page 182.

Modifying the Application Code

In our discussion, we have assumed that the LiveServer is hosted at kn.example.com, and the pages that make up the application are being served from apps.example.com. The **src** attribute of the <script> tag should contain the URL of the LiveServer location, including fully-qualified domain name. This ensures that the Connector library is loaded at the correct location.

Note: Use the fully-qualified domain name corresponding to the IP address of the LiveServer to avoid JavaScript security issues. Also, the domain name must be entirely in lowercase.

```
<html>
<head>
<title>My KnowNow Application</title>
<script type="text/javascript"
src="http://kn.example.com/kn/?do_method=lib"></script>
</head>
```

Note: A cross-domain Web application that uses document.open() may produce errors due to the browser's security limitations.

HTML Pages Not Using the Connector

An application may contain HTML pages that do not use the Connector functionality but need scripting access to the other pages. For example, an application may use JavaScript across frames and framesets. These pages should contain a script tag that calls the do_method=whoami action of the KnowNow LiveServer. This way, the browser will allow any JavaScript within these pages to interact with the other pages of the application.

```
<script type="text/javascript"
src="http://kn.example.com/kn/?do_method=whoami"></script>
```

Alternatively, you can add a script tag to your application code that sets the document.domain property to the common suffix of the application and the LiveServer host domain names.

Using a Single IP Address

You can have the application server and the LiveServer on the same IP address, for example, with the application server using port 80, and the LiveServer using port 8000.

Note: You cannot run a cross-domain application on a single IP address if you want to support Netscape 4.x, because Netscape 4.x requires both the application server and the LiveServer to use HTTP on port 80 or HTTPS on port 443.

The **src** attribute of the <script> tag should contain the URL of the LiveServer, including fully-qualified domain name and port number. If you are not part of a domain, use the fully-qualified machine name instead.

```
<html>
<head>
<title>My KnowNow Application</title>
<script type="text/javascript"
src="http://kn.example.com:8000/kn/?do_method=lib"></script>
</head>
```

Using Command-Line Parameters

Named parameters can be passed to your application using parameters in the query string part of the URL or in kn_queryString. The query string is optional and appears following the path part of the URL, separated by a question mark. These parameters are listed in Table 6-4, with the following rules applying:

- Each parameter has a name and a value separated by a single equals sign (=). Both the parameter name and the parameter value are UTF-8 encoded and URL-escaped. The same parameter may be set several times, but only the last value has any effect.
- Multiple parameters can be given in either a semicolon-separated list (;) or an ampersand-separated list (&).
- Parameter values are available as string properties of the kn_argv object. For instance, the value for a parameter named daisy would be stored in kn_argv.daisy.
- Parameter names starting with kn_ are reserved.
- The name and equals sign are optional for the special parameter kn_topic.
- For all other kn_ parameters, the equals sign and value are optional. Supplying one of these parameters with the value omitted is the same as supplying the parameter with a Boolean value of true.

Table 6-4. Command-line parameters.

Parameter	Meaning
kn_autostart	Requests that your application start running immediately, rather than waiting for user input. You must implement support for this feature in your application if you want to support kn_autostart.
kn_debug	Comma-separated list of debugging options. Available options include: ■ all - Turn on all debugging flags. ■ events - Display an alert box every time an event is received. ■ posts - Display the contents of sent events in the posting frame. ■ receipts - Display the contents of received events in the tunnel frame. ■ routes - Display an alert box every time a subscription is canceled by the default message handler.
kn_displayname	Describes the user identified by kn.userid in human-readable format. Overrides the default value of kn.displayname.

Table 6-4. Command-line parameters. *(continued)*

Parameter	Meaning
kn_options	Comma-separated list of speed options. Available options include: ■ all. Turn on all options. ■ escape. Use JavaScript escape() instead of kn_escape(). ■ fastdoc. Use document.open() for KNDocument. ■ noforward. Do not forward status events to the tunnel frame. This disables status forwarding and hence 204 No Content and Connection:close for the JavaScript Connector. Note that this may lead to lots of browser history pollution. ■ nosocketshare. By default, the JavaScript Connector will use a leader-election algorithm that allows for sharing of one tunnel between multiple clients. However, in cases where you need to handle a high bandwidth, you can turn off leader election by setting this option. This in effect enables the creation of a tunnel per client. ■ noswap. Do not swap tunnel and post frames periodically. ■ single. Do not batch multiple requests. ■ unescape. Use JavaScript unescape() instead of kn_unescape().
kn_hashCache	Initial event hash cache value, loaded by the Connector at startup using kn.setHashCache().
kn_lang	A comma-separated list of language codes and optional quality values used when looking up localized messages. Overrides the default value of the kn_lang self property.
kn_lastTag_	Initial value for kn.lastTag_.
kn_response_flush	Controls the number of filler bytes (from 0 to 4,096) sent when the LiveServer detects a lull in the event stream for a particular tunnel (in other words, a persistent connection for event notification). If this value is not supplied, the number of bytes is determined automatically.
kn_target	Window name in which links to external resources should be followed, with _blank being the usual default. You must implement support for this feature in your application if you want to support kn_target.

Table 6-4. Command-line parameters. *(continued)*

Parameter	Meaning
kn_timestamp	Requests that event timestamps be displayed. You must implement support for this feature in your application if you want to support kn_timestamp.
kn_topic	Primary topic used by an application. You may choose to implement support for this feature in your application.
kn_tunnelID	Initial value for kn.tunnelID, the tunnel route kn_id.
kn_tunnelMaxAge	Initial value for kn.tunnelMaxAge, do_max_age value for the tunnel route. For more information on headers for events and times, see "kn_history_*" on page 67.
kn_tunnelURI	Initial value for kn.tunnelURI, the tunnel route source topic URI.
kn_userid	Identifies the user in the context of the Web application. Overrides the default value of kn.userid.

Command-Line Examples

Table 6-5 provides examples illustrating how to use the command-line options described in Table 6-4. All examples assume that your application's URL is .../MyApp/.

Table 6-5. Command-line examples.

Sample Command Line	Description
.../MyApp/?kn_debug	Basic debugging is enabled.
.../MyApp/?kn_debug=	All debugging is disabled.
.../MyApp/?kn_debug=all	All debugging flags are enabled.
.../MyApp/ or .../MyApp/?	No parameters specified.
.../MyApp/?color=red	kn_argv.color is set to the string red.
.../MyApp/?color=red;flavor=spicy or .../MyApp/?color=red&flavor=spicy	kn_argv.color is set to the string red and kn_argv.flavor is set to the string spicy.

Table 6-5. Command-line examples. *(continued)*

Sample Command Line	Description
.../MyApp/?/daisy/bar or .../MyApp/?kn_topic=/daisy/bar	kn_argv.kn_topic is set to the string /daisy/bar.
.../MyApp/?kn_options=escape,unescape	JavaScript escape() and unescape() will be used instead of kn_escape() and kn_unescape().

API Overview

The JavaScript Connector's API is documented in the LiveBrowser documentation available by accessing */LiveServer_root/*/livebrowser/docs/apidocs/index.html. In addition, the LiveBrowser has a number of calls and components. You can find more information about these calls and components in the same documentation.

This section provides a brief discussion of the most important objects in the JavaScript Connector's API..

Note: This section is provided to give you an idea of the capabilities of the API. For specific information on each call, information on what have changed since this document was written, and information on new calls, see the API documentation included in your LiveServer installation.

The self Object

self refers to a Window, FRAME, or IFRAME object in which the JavaScript Connector is active. self provides methods for managing frame attributes, localization, strings, and many other aspects of the program. For more information, see "The self Object" on page 189.

The kn Object

The kn object in the JavaScript API represents an instance of the JavaScript Connector and provides your application with several methods for publishing events to a topic, subscribing to a topic, and unsubscribing from a topic. The same kn object usually appears as a property of all frames in a window which access the same LiveServer, although programmer customization and complex frameset layout can cause multiple Connector instances to be created. For more information, see "The kn Object" on page 202.

The KN Object

The KN object is a holder for some constants used in JavaScript string and character processing. For a bit more information, see "The KN Object" on page 208.

The kn_argv Object

A hash table of name/value pairs (headers) as defined in the URL query string.

Status Handler Objects

The API provides three key callback methods that you will need to implement for creating status handler objects to process the status events returned by the LiveServer in response to requests. These methods are onSuccess, onError, and onStatus.

There are three status handler objects:

- onStatus
- onSuccess
- onError

These objects are used everywhere else as arguments. Just about every method in the JavaScript Connector API takes one of these status handlers as a parameter. Your program provides these to handle responses from requests sent to the LiveServer.

If onStatus exists, status events are passed to onStatus. Otherwise, status events are sent to onSuccess or onError, depending on the HTTP code that is being sent along with each event. If the HTTP code says the request was successful, the events are sent to onSuccess, if unsuccessful, they are sent to onError.

The normal usage is to either provide [onStatus or (onError and OnSuccess)], with the most common usage being to provide onError and onSuccess. All three are optional. There is no significant performance hit when using status handler objects, so we highly recommend that you provide them.

String and Character Support

The JavaScript Connector uses UTF-16 encoding for Unicode and UCS characters in JavaScript strings, but there is no byte order mark visible at the JavaScript level. This is backward-compatible with the UCS-2 character encoding employed by older Web browsers, thus providing cross-browser access to a rich character set.

As a result, binary data in events should be encoded using the MIME Base64 encoding or a similar technique.

The self Object

The self object provides APIs that refer to a window, a FRAME, or an IFRAME object in which the JavaScript Connector is active. It has properties and methods as documented under the following headings:

- “self Properties” on page 190
- “self Methods” on page 192

self Properties

Almost all of the self properties are query parameters that can be overwritten using a URL. For example, you could send the command ?kn_server=<*value*> in a URL command to overwrite the kn_server property. Some characters need to be encoded specially.

kn_appFrameAttributes

This property is only useful if you don't use the transport plug-ins from the LiveBrowser; i.e., if you use the original frameset-based application wrapping. This property is a string with additional HTML attributes written into the frameset tag. It is used typically to disable scroll bars so you can create a very small window, for example, such as banner-like applications.

kn_blank

Not useful to end programmers.

kn_defaultHacks

This property has been deprecated. Use kn_defaultOptions instead.

kn_defaultOptions

This property provides a list of Connector options to enable by default. It contains option flags to control the behavior of the Connector, as well as debug flags.

kn_displayName

You can use this property to set an initial value for kn.displayname.

kn_hashCache

Not useful to end programmers.

kn_lang

Used for internationalization. Tells you the currently selected language.

kn_lastTag

Not useful to end programmers.

kn_lbversion

A property that is generated when you use the kn_lbversion method. Contains your LiveBrowser version number as a string. See also the kn_lbversion method.

kn_maxHits

Used for performance tuning. Not normally useful to end programmers.

kn_queryString

If this property is set, it overrides the browser query string for the purposes of argument parsing. This property is useful and important from a security point of view. If you are worried that people might pass in harmful arguments, set this to the set of options you want to use, and those settings will take precedence over anything passed in.

kn_response_flush

The value in this property is passed directly to the LiveServer. It tells the LiveServer how many bytes to send during each lull in the event stream to force buffering proxy servers and buffering browsers to flush and to execute the JavaScript content.

As of LiveBrowser version 2.0.1 and LiveServer version 2.0.4, this property is determined automatically at LiveServer startup, so in most cases you will not need to use it. There are some cases when you may wish to use it, such as if you want to avoid the 12-second delay when going over an Apache 2.0 proxy server.

kn_server

This property is only useful if you are not using the new LiveBrowser framework and are instead just using do_method=lib. Use this parameter to tell do_method=lib which LiveServer to contact. The value is just the URL of the target LiveServer.

This property is still useful with new applications if you want to take an application that was written for or installed for one LiveServer and you want to run it against a different LiveServer. If your application and your LiveServer are both set up for cross-domain use, you can use this parameter to tell the application to use a different LiveServer than it normally would. For more information, see "Writing Cross-Domain Web Applications" on page 181.

A similar parameter is a setting called kn_configServer, which sets the value of kn_server. The difference between these two parameters is that kn_server is a self property that connects to the core library and can be overridden from the URL.

kn_strings

A built-in table of localized strings used for localization. It cannot be set from the URL. This table is built by calling the kn_localize function.

kn_tunnelID

Not useful to end programmers.

kn_tunnelMaxAge

Not useful to end programmers.

kn_tunnelURI

Not useful to end programmers.

kn_userid

You can use this property to set an initial value for kn.userid.

self Methods

The self methods relate to localization and to publishing, subscribing to, and otherwise managing events.

$$

Takes as input a string and returns a localized version of that string, if it has been defined, in a context. Otherwise, it returns the string. For example,

```
aLocalizedString=$$(aContext, aString)
```

See also $$_ and kn_formatString.

There is a family of related functions that are created at programmer request. Two are created when the Connector is loading: $ and $_.

$$_

Used for retrieving localized versions of strings. This method takes multiple arguments. The first argument is a format string. All remaining arguments are passed in as parameters. This method calls kn_formatString and passes to that method the localized version of its own first parameter.

This method is like kn_formatString, but it first takes the formatting template and passes that template on to $$, then takes the result of $$, a context, and a formatting template and uses that as the formatting template for its call to kn_formatString. Instead of passing an object as you would with kn_formatString, you pass $$_ a list of parameters, which gets turned into an array. When you are referring to those parameters in your formatting template in $$_, use numbers as parameter names (for positional indicators); for example, %{0} refers to the first parameter after a formatting template, %{1} refers to the second parameter, and so on.

An example of this function:

```
aFormattedLocalizedString=$$_("aContext", "aFormattingTemplate", parameter, ...)
```

See also $$ and kn_formatString.

kn__hacks

Obsolete. This method has been replaced by kn__options.

kn__debug

This method checks the debugging flags that have been set by kn_debug. To use this method, pass it a debugging flag name. This method returns true or false to indicate whether that flag is enabled (true) or not (false).

kn__options

This method checks for the presence of a specified word (feature) in the comma-separated list of options that are set by kn_options.

To use this method, pass it a feature name. This method returns true or false to indicate whether that feature occurs in the options string (true if it does, false otherwise). The special flag kn_options=all causes this method to return true for all features.

If you don't specify a feature name, this method checks for the presence of the kn_options parameter.

kn_charCodesFromString

One of the localization functions. Correctly identifies and decodes Unicode characters, including those that are outside the first 16-bit range of Unicode. This method takes a string and turns it into an array, in which one element corresponding to each character in the string. Each element is a Unicode number.

This is method a replacement for a built-in JavaScript function that is buggy in every browser we've tested.

kn_createContext

Takes a context as a parameter and generates two new self methods:

1. aContext$(...), which is equivalent to $$(aContext,...)
2. aContext$_(...), which is equivalent to $$_(aContext,...)

Whatever parameters you pass to aContext$ get passed internally to $$ after the context. Likewise, whatever parameters you pass to aContext$_ get passed internally to $$_ after the context.

Call this method once for each context you use.

This method does not return anything.

See also $$ and $$_.

kn_debug

The Connector has a debugging flag system that allows you to turn on debugging mode and enable more verbose debugging of different parts of the Connector and the various plug-ins. This system is controlled by the kn_debug flags.

It takes string as a parameter; the string is a comma-separated list of debug flags. There are two special debugging option flags: none and all. None disables all debugging options; all enables all debugging options. You can use a URL parameter with kn_debug to turn on general Connector debugging.

When you enable debugging, it turns off the Connector's normal exception-handling behavior, which is to suppress exceptions (that is, the normal behavior is to not display error messages and to allow the rest of page to run). Also, when debugging is on, it shows normally hidden frames or IFrames where communication with the LiveServer takes place.

kn_decodeRequest

Decodes requests for the LiveServer. Essentially, this method decodes parameters with values from the application/x-www form URL encoding used by the LiveServer and in URL parameters. If you are sending requests to the LiveServer, sometimes those requests need to be decoded for use on the client side. Use this function to do so. This method is useful for batch commands, debugging, and client-side use.

kn_defaultOnError

A default method that is called when a status event is unsuccessful if you do not provide your own onError method. The default behavior of this method is to ignore the error unless you are in debug mode, in which case you will see a pop-up message.

kn_defaultOnMesssage

You can use this method to override the default behavior of the JavaScript Connector. The default behavior of the JavaScript Connector is to try to delete any route that it gets events from for which it doesn't have any registered handler for. If you provide a kn_defaultOnMessage handler, you can override that behavior by doing something different in your kn_defaultOnMessage. This method only works when it is defined in a leader window.

kn_defaultOnStatus

A default method that is called if you do not provide your own onStatus method. Depending on the HTTP code that is sent with the event, this default onStatus method calls either the default onError or the default onSuccess method. The status handler object is passed as the (optional) second argument, or as this.

kn_defaultOnSuccess

A default method that is called when a status event is successful if you do not provide your own onSuccess method. The default behavior of this method is to ignore the success message.

kn_doStatus

You will probably not need to use this method.

kn_encodeRequest

Encodes requests for the LiveServer. Essentially, this method encodes parameters with values (with separators between) into the application/x-www form URL encoding used by the LiveServer and in URL parameters. If you are sending requests to the LiveServer, those requests need to be encoded first. Use this function to do so. This method is useful for batch commands, debugging, and client-side use.

kn_escape

Converts a string from UTF16 to URL-encoded UTF8. Replaces the JavaScript escape method, which is buggy in most browsers.

kn_formatString

Used to take bits of an object and interpolate those bits into formatted messages (usually that are sent to the user), where some parts of that message are variable. It takes a formatting template and an object and returns a string that has been formatted using the formatting template.

```
aFormattedString=kn_formatString("aFormattingTemplate", anObject)
```

The formatting template is just a text string containing special notation so that some parts of the message can be variables. Every place the special notation appears in the formatting template, it is replaced with something from the object. The special notation is:

- %% becomes a single % in the output.
- %{propertyname} becomes anObject.propertyName.

kn_hacks

Obsolete. This method has been replaced by kn_options.

kn_htmlEscape

Takes an input string and changes special characters, such as < and >, into the proper HTML codes, such as < and >. Also, it turns Unicode characters outside the basic ASCII range into numeric character references (i.e, performs UTF16 encoding). The resulting HTML can be written into a page; what the user will see will be the original string. Essentially, makes any string safe for doc.write() as an HTML attribute or text area.

kn_inspectAsHTML

Not useful for the end programmer.

kn_inspectAsText

Not useful for the end programmer.

kn_inspectInWindow

Not useful for the end programmer.

kn_isReady

Pass a window, frame, or IFrame handle as a parameter to this method. This method will say whether that window, frame, or IFrame is an accessible object.

kn_lbversion

Provides the version number of the LiveBrowser. If you include this method in your code, you get a self property called self.kn_lbversion, which is your LiveBrowser version number contained in a string. You can use this version number to track compatibility with different versions of the LiveBrowser.

In addition to the entire LiveBrowser having a version number, every component in the API also has an interface version number. These version numbers can be used to track and manage compatibility at the component level. The calls that manage component versions are kn_version_define, kn_version_checkk, and kn_version_toFloat, which see.

We encourage you to check the LiveBrowser version number and possibly that of individual components at the start of your code. To check the LiveBrowser version, include the following at the end of the <head> section of your LiveBrowser application:

```
<script>
        kn_include("kn_lbversion");
```

```
</script>
<script>
        alert(self.kn_lbversion);
</script>
```

kn_localize

One of the localization functions. Use this to add a translation to the string tables (contained in the self property kn_strings). Essentially the same as kn_localizeDefault, except that this method overrides any strings that have already been defined.

This call takes several arguments: a language code (with an optional country suffix), a context, an input string, and a result string (or null if the result string is the same as the input string for that language and that context).

These two methods, kn_localize and kn_localizeDefault, make it possible to have project-wide translations for a large Web site or large Web application, and then have specific local overrides for different parts that need their own, slightly different translations of the same localized string without breaking out into separate contexts.

kn_localizeDefault

One of the localization functions. Essentially the same as kn_localize, except that it only adds the localization if one hasn't already been defined for that string. Note that the original string passed to kn_localizeDefault must always be in double quotes.

These two methods, kn_localize and kn_localizeDefault, make it possible to have project-wide translations for a large Web site or large Web application, and then have specific local overrides for different parts that need their own, slightly different translations of the same localized string without breaking out into separate contexts.

kn_onTunnelStatus

This method is called each time a tunnel starts. This method only works when it is defined in a leader window.

kn_onTunnelStop

This method is supposed to be called when a tunnel stops. However, most of the time, the notification for tunnel stops doesn't work. This method only works when it is defined in a leader window.

kn_opts

Obsolete. This method has been replaced by kn_options.

kn_options

This method controls a separate system of options. The kn_options method provides control over these options; use this method to set the options. If you want to check how the options are set, use kn__options.

This method takes a string as a parameter; the string is a comma-separated list of option flags. There are two special option flags: none and all. None disables all options; all enables all options.

The options system controls optional features of the Connector, such as performance enhancements, that sometimes have undesirable side-effects. For example, you can make things go faster at the expense of occasionally corrupting data. Because of these side-effects, these options are off by default.

kn_presence_register

An interface to presence. Presence provides a programmatic interface to accessing the subscriber presence for a topic, so you can find out when subscribers start and stop listening to a topic. This is the preferred method to use (in preference over kn_presence_obj).

For this method, specify a topic, a callback (listener) object, and some status handlers. This method notices changes and invokes a separate handler object. The listener has OnConnect and OnDisconnect, which you must implement.

The Live Buddy (and anything that uses the Live Buddy) uses presence to indicate whether a user is online or offline. For more information on the Live Buddy and presence, see the *KnowNow LiveBrowser Developer's Guide*.

kn_presence_obj

An interface to presence. Presence provides a programmatic interface to accessing the subscriber presence for a topic, so you can find out when subscribers start and stop listening to a topic. This method is a constructor for objects that you then use as prototypes in JavaScript's inheritance system to extend your own methods to handle presence changes. Use of this method is not recommended; use kn_presence_register instead. You must override this method. You must also implement this methods onRouteStatus function.

To use this method, create a new kn_presence_obj and pass in a topic to listen to. Set the kn_presence_objs onRouteStatus method to be your own handler for route status changes. That callback gets called with route events and transition streams. The two transition streams are deleted and new, which say what has happened with that route.

kn_publish

Publishes events to a topic. Identical to kn.publish. Takes three arguments and calls kn.NOTIFY:

- a destination URI (the topic)
- an event object
- a status handler object

The return value is the kn_id of the event object, either an existing one or one that is added during the publish command. For information on kn_id and other LiveServer headers, see "kn_ Header Reference" on page 62.

kn_publishForm

Similar to kn_publish and identical to kn.publishForm. This method takes a form, turns it into an event, and then publishes it as a form. You can use this method to take an application that was built around form submission and turn it into a publish/subscribe application.

This method takes three arguments and calls kn.NOTIFY:

- a destination URI
- an HTML form object
- a status handler object

The return value is the kn_id of the form object, either an existing one or one that is added during the publish command.

kn_redrawCallback

Not useful for the end programmer.

kn_resolvePath

A utility function that resolves relative path names. Given a base path and a relative path, it returns the relative path qualified using the base path, therefore creating an absolute path out of the relative path. If the second argument is already an absolute path, it returns the second argument. This method understands the .. directory navigation notation, but does not understand the . notation.

The absolute path must end with a slash.

kn_sendCallBack

Not useful for the end programmer.

kn_stringFromCharcode

One of the localization functions. The reverse of kn_charCodesFromString. Correctly encodes Unicode characters, including those that are outside the first 16-bit range of Unicode. This method takes a list of one or more character codes as parameters and returns the corresponding string.

This is method a replacement for a built-in JavaScript function that is buggy in every browser we've tested.

kn_subscribe

Identical to kn.subscribe. Subscribes to a topic so that it can receive events published to that topic. Takes four parameters and uses kn.ROUTE:

- a source URI
- a destination URI
- an options object (a set of parameters)
- a status handler object

This method returns the route ID for that subscription, which is a dynamically generated URL. You can set that route ID in options by setting the kn_statusFrom header.

kn_tunnelLoadCallBack

Not useful for the end programmer.

kn_unescape

Handles the URL encoding. Replaces the JavaScript unescape method, which is buggy in most browsers.

kn_unsubscribe

Identical to kn.unsubscribe. Deletes a subscription from the LiveServer and client sides. Takes as parameters

- a subscription ID
- a status handler object

It deletes the subscription identified by the subscription ID.

kn_utf8decode

Converts UTF8 characters into UTF16. Web browsers use UTF16 inside their JavaScript patches.

kn_version_check

Checks for a version number of a compatible subsystem interface. After defining component version numbers with kn_version_define, use kn_version_check to check components for a particular feature with a particular flavor. If the feature is not present, kn_version_check returns false. To use this method, pass it a component name, the version number you are checking for (or null if you don't care and are just interested in, for example, feature sets), and a feature set object (a list of features and the particular flavors you require).

We encourage you to check the LiveBrowser version number and possibly that of individual components at the start of your code.

In order to have access to the component version number and the version methods, you need to include the kn_lbcheck component.

See also kn_version_define.

kn_version_define

Defines the version number of an interface. It keeps a registry for all defined components, which keeps track of all the different versions for each component that has been so defined. You can define multiple versions for the same component by making multiple calls to this method. This method takes the name of a subsystem (component) that a version is being defined for, the version that is being defined, the oldest compatible version, and an optional object with options.

The optional object with options consists of a string of feature strings (properties). Each property of the object consists of a name, which is a feature, and a value that is the style of that feature that is supported. This fourth argument allows you to define for a module which feature sets the module supports. It also allows someone using the module to check to make sure the feature set they are using is present. For example, you would use this when you have to switch your component from using one style to another, incompatible style of the same functionality. You might, for example, change the implementation of one method so that the parameter list is different. To manage versions, use this method to define which features with which flavors you implement. Then you would use kn_version_check to check for a particular feature with a particular flavor. There are three possible flavors: the original flavor, old flavor, and new flavor. You can create any number of flavors as you wish.

Once you have defined a version number, you will need to manually code in the version number of a component into your function calls. Those version numbers can then be checked with kn_version_check.

Later, you should run kn_version_define on a component each time you make a change to that component in order to increment the max version. If you have also made a change that makes it incompatible with older versions, you should also change the min version to equal the current new max version. You could then also warn a user that they need an updated version of the LiveBrowser, if you made changes that made your application incompatible with an older version of the LiveBrowser.

In order to have access to the component version number and the version methods, you need to include the kn_lbcheck component.

See also kn_version_check.

kn_version_toFloat

A helper function for the other two version methods (kn_version_define and kn_version_check). This method takes a version string as input and provides a version number as output.

In order to have access to the component version number and the version methods, you need to include the kn_lbcheck component.

kn_watchdog_register

Registers a watchdog listener object. The listener's optional onTimeout() method is invoked when no events from the LiveServer's internal data source have been received for 40 consecutive seconds. The listener's optional onUpdate() method is invoked each time there is an update from the LiveServer's internal data source.

The kn Object

This object is used for publishing events, establishing subscriptions to topics, and removing established subscriptions. This object has both properties and methods as documented under the following headings:

- "kn object Properties" on page 202
- "kn object Methods" on page 204

kn object Properties

The kn object's properties control windows and user names as well as other things.

kn.displayname

Contains the human-readable version of the name of the person using the application, or Guest User if there is no name. The name Guest User is localized, so it could be translated if and as desired. This value is set by the LiveServer, so the client-side defaults don't affect this.

If you are using the LiveServer, the default is Anonymous User, which is not localized. This overrides the LiveBrowser defaults.

kn.documents

Used internally.

kn.lastError

Normal error-trapping behavior stores the most recent exception here.

kn.lastError.date

An example of a use of kn.lastError.

kn.leaderWindow

A window handles that tells you which window controls the running tunnel. This is important because some methods (kn_defaultOnMesssage, kn_onTunnelStatus, and kn_onTunnelStop) don't work correctly unless they are in the leader window.

kn.ownerWindow

A window handle that tells you which window owns your instance of the Connector object. A single instance of the Connector object (the actual kn object) is shared throughout a frame set. If you instantiate kn at one level in a frame set, all the children of that level also use the same one recursively.

kn.tunnelURI

Not useful for the end programmer.

kn.tunnelID

Not useful for the end programmer.

kn.userid

A machine-readable version of the display name. The default is guest. It is not localized. If you are using the LiveServer, the default is anonymous, which is also not localized. This overrides the LiveBrowser defaults.

kn object Methods

A number of these methods are essentially one method that is parameterized by the name of the LiveServers do_methods. For detailed information on each of the do_methods, see "do_method Command Reference" on page 49. You may need to have the proper permissions set in the LiveServer (such as administrative access) in order to use some of these methods.

kn.ADD_JOURNAL

A direct call to do_method=add_journal. Takes the following parameters:

- an options object (similar to kn.subscribe)
- a status handler object

Nothing is returned. Take care using this call because it can lock up the Connector if it is trying to start two journals simultaneously.

kn.ADD_NOTIFY

A direct call to do_method=add_notify. Takes the following parameters:

- an options object (similar to kn.subscribe)
- a status handler object

Nothing is returned.

kn.ADD_ROUTE

A direct call to do_method=add_route. Takes the following parameters:

- an options object (similar to kn.subscribe)
- a status handler object

Nothing is returned.

kn.ADD_TOPIC

A direct call to do_method=add_topic. Takes the following parameters:

- an options object (similar to kn.subscribe)
- a status handler object

Nothing is returned.

kn.BATCH

A direct call to do_method=batch. Takes the following parameters:

- an options object (similar to kn.subscribe)
- a status handler object

Nothing is returned.

kn.CLEAR_TOPIC

A direct call to do_method=clear_topic. Takes the following parameters:

- an options object (similar to kn.subscribe)
- a status handler object

Nothing is returned.

kn.DELETE_NOTIFY

A direct call to do_method=delete_notify. Takes the following parameters:

- an options object (similar to kn.subscribe)
- a status handler object

Nothing is returned.

kn.DELETE_ROUTE

A direct call to do_method=delete_route. Takes the following parameters:

- an options object (similar to kn.subscribe)
- a status handler object

Nothing is returned.

kn.DELETE_TOPIC

A direct call to do_method=delete_topic. Takes the following parameters:

- an options object (similar to kn.subscribe)
- a status handler object

Nothing is returned.

kn.NOTIFY

A direct call to do_method=notify. Takes the following parameters:

- an options object (similar to kn.subscribe)

- a status handler object

Nothing is returned.

kn.ROUTE

A direct call to do_method=route. Takes the following parameters:

- an options object (similar to kn.subscribe)
- a status handler object

Nothing is returned.

kn.SAVE_CONFIG

A direct call to do_method=save_config. Takes the following parameters:

- an options object (similar to kn.subscribe)
- a status handler object

Nothing is returned.

kn.SET_TOPIC_PROPERTY

A direct call to do_method=set_topic_property. Takes the following parameters:

- an options object (similar to kn.subscribe)
- a status handler object

Nothing is returned.

kn.UPDATE_NOTIFY

A direct call to do_method=update_notify. Takes the following parameters:

- an options object (similar to kn.subscribe)
- a status handler object

Nothing is returned.

kn.UPDATE_ROUTE

A direct call to do_method=update_route. Takes the following parameters:

- an options object (similar to kn.subscribe)
- a status handler object

Nothing is returned.

kn.clearHandler

Used in advanced applications. Use to internally delete your own handler from the table.

See also kn.setHandler.

kn.iw

Used for debugging.

kn.publish

Publishes events to a topic. Identical to kn_publish. Takes three arguments and calls kn.NOTIFY:

- a destination URI (the topic)
- an event object
- a status handler object

The return value is the kn_id of the event, either an existing one or one that is added during the publish command.

kn.publishForm

Similar to kn.publish and identical to kn_publishForm. This method takes a form, turns it into an event, and then publishes it as a form. You can use this method to take an application that was built around form submission and turn it into a publish/subscribe application.

This method takes three arguments and calls kn.NOTIFY:

- a destination URI
- an HTML form object
- a status handler object

The return value is the kn_id of the form object, either an existing one or one that is added during the publish command.

kn.sendQueue

Not useful for the end programmer. Sends batch requests; however, kn.BATCH is the preferred method for sending batch requests.

kn.setHandler

Used in advanced applications. Use to internally add your own handler when an event with a new route label comes off the wire. You can have at most only one handler for any given string, which is the same string you get back when you call a subscribe function. You can make your own strings and register them, and if you get events with a corresponding kn.ROUTE location, they get sent to the handler you registered using this method.

See also kn.clearHandler.

kn.subscribe

Identical to kn_subscribe. Subscribes to a topic so that it can receive events published to that topic. Takes four parameters and uses kn.ROUTE:

- a source URI
- a destination URI
- an options object (a set of parameters)
- a status handler object

This method returns the route ID for that subscription, which is a dynamically generated URL. You can set that route ID in options by setting the kn_statusFrom header.

kn.unsubscribe

Identical to kn_unsubscribe. Deletes a subscription from the LiveServer and client sides. Takes as parameters

- a subscription ID
- a status handler object

It deletes the subscription identified by the subscription ID.

The KN Object

These are constants used for string and character processing. They are not useful for the end programmer.

- KN.hexDigits
- KN.ucsMaxChar
- Kn.ucs2max
- Kn.ucsNoChar
- KN.utf16firstHighHalf

- KN.utf16firstLowHalf
- KN.utf16max
- KN.utf16offset
- KN.utf16mask
- KN.utf16shift
- KN.utf8mask
- KN.utf8shift

Chapter 7 Using the Windows Connectors

All Connectors manage communications between the LiveServer and your application. While you do not need to use a Connector, a Connector can make your life much easier by simplifying the calls and operations your application needs to perform. Therefore, when creating or customizing your Windows application to perform publish and subscribe operations with the LiveServer, you may want to have your application use one of the Windows Connectors instead of dealing directly with the LiveServer.

This chapter provides information that is common to all Connectors in the suite of Windows Connectors, as well as some information that is specific to the LiveActiveX Connector. It also provides guidelines on using some of the most important calls in the APIs to create or customize your application so it can make best use of the Connector you wish to use. Be sure to also read Chapter 2 , "Events," for additional important information on the LiveServer's operations and architecture, as well as Chapter 5 , "Common Connector Capabilities," for information on capabilities that are common to all KnowNow Connectors. For information that is specific to the Live.NET Connector and LivePDA Connector, see Chapter 8 , "Using the Live.NET and LivePDA Connectors."

- "Introducing the Windows Connector Suite" on page 213
- "Pre-installation Tasks and Connector Dependencies" on page 216
- "Installing the Windows Connectors" on page 219
- "Uninstalling the Windows Connectors" on page 225
- "Using the Connector APIs" on page 226
- "The Connector Class" on page 232
- "Request Status Events" on page 236
- "Publish Operations" on page 239
- "Subscribe Operations" on page 242

- "Unsubscribe" on page 246
- "Cursors" on page 247
- "Heartbeat" on page 248
- "Logging" on page 249
- "Presence" on page 253
- "Request/response" on page 254
- "Disconnecting from the LiveServer" on page 258
- "The LiveActiveX Connector: Use Fully Qualified Names" on page 259
- "The LiveC++ Connector: Exceptions and Error Codes" on page 260

Introducing the Windows Connector Suite

Connectors manage communications between the application and the LiveServer. All Connectors perform the following primary functions:

- publish commands
- subscribe and unsubscribe requests
- connection and event status handling
- offline queuing

In order for your Windows applications to best and most easily use the capabilities of the LiveServer, you have a choice of using any of the following Connectors:

- LiveC++ Connector for Windows
- LiveActiveX Connector
- Live.NET Connector
- LivePDA Connector

Which Connector you use depends on what language your application is written in. Each of these Connectors performs the same kinds of functions. As described in the next section, the application programming interfaces (APIs) for these Connectors support a wide range of tools and Windows platforms.

Note: The Live.NET Connector is .NET compliant and has been tested with Visual Basic .NET and C#.

This chapter contains information that is common to all these Connectors, as well as a few pieces of information that are specific to the LiveActiveX Connector and LiveC++ Connector. Chapter 8 , "Using the Live.NET and LivePDA Connectors," contains information that is specific to those two Connectors.

This section contains information under the following headings:

- "Connector APIs Overview" on page 214
- "Supported Tools and Platforms" on page 214
- "SSL and the Windows Connectors" on page 215

Connector APIs Overview

The Connector APIs (C++, COM/ActiveX, and .NET) offer language-specific functions for modifying or creating your applications to take advantage of the capabilities of the desired Connector. By using the APIs, your application can make the proper calls to the Connector and properly receive information from the Connector.

The LiveC++ Connector API is written in C++; the Live.NET Connector API is written in .NET. The remaining APIs (Com/ActiveX) are wrappers around the LiveC++ Connector core. Each API shares the same names for objects, functions, and calls, making it easy to move from using one API to another, and are threadsafe.

You can find out more about these APIs starting on "Using the Connector APIs" on page 226.

Supported Tools and Platforms

Table 7-1 and Table 7-2 list the supported development tools and platforms for each of these APIs. For additional platform information, please see the Connector release notes.

In Table 7-1, the Connector APIs are indicated by the following numbers:

1. C++ static API.
2. COM/ActiveX API.
3. .NET API. Any language that supports .NET (such as C# and Visual Basic .NET) should be able to use the KnowNow .NET API. For information on dependencies for the Live.NET Connector, see "Live.NET Connector Requirements" on page 216.

Table 7-1. Supported development tools and platforms.

Development Tool	Windows Operating System							
	98	ME	NT	2000	XP	Server 2003	.NET	Pocket PC 2003
Visual Studio 6 and Visual Basic 6	1, 2	1, 2	1, 2	1, 2	1, 2	1, 2		
Visual Studio .NET	1, 2	1, 2	1, 2	1, 2, 3	1, 2, 3	1, 2, 3	3	3
Visual Studio .NET 2003	1, 2	1, 2	1, 2	1, 2, 3	1, 2, 3	1, 2, 3	3	3

Table 7-2. Live.NET and LivePDA Connectors: Product support.

Connector	Visual Studio .NET 2003	Visual Studio .NET 2005	.NET 1.1	.NET 2.0	.NET 1.0 CF	.NET 2.0 CF
Live.NET Connector	Yes	Yes	Yes	Yes	N/A	N/A
LivePDA Connector	Yes	Yes	N/A	N/A	Yes	Yes

SSL and the Windows Connectors

In order to use HTTPS connections with a developer certificate or a certificate signed by a corporate CA Root, you must install the root certificate of the issuing certification authority on your computer. This can be done in Internet Explorer by simply downloading the root certificate from the CA. Follow the prompts to install this root certificate into Windows.

Note: The LiveServer name passed to the Connector must exactly match the LiveServer name in the certificate; otherwise, the HTTPS connection will fail.

Pre-installation Tasks and Connector Dependencies

Table 7-1 on page 214 lists the supported development tools and platforms for each of the Connector APIs. In addition to that information, before installing the Connectors, make sure the environment is ready for them as described under the following headings:

- "LiveActiveX Installation Note" on page 216
- "Live.NET Connector Requirements" on page 216
- "LivePDA Connector Requirements" on page 217

When you are ready, install the desired Connector(s) as described under "Installing the Windows Connectors" on page 219.

LiveActiveX Installation Note

When installing the LiveActiveX Connector, you will be given a choice of which version to use: One for use with Visual Studio 6, one for Visual Studio .NET, or one for Visual Studio .NET 2003. Although you choose one, you are actually installing all three versions, but are only registering one of them.

Note: In Windows, you can switch between ActiveX controls by registering or unregistering a DLL using the Microsoft Windows regsvr32.exe tool. You can use this tool to change to another version of the LiveActiveX Connector.

Live.NET Connector Requirements

Although it is not a requirement for non-custom use of the Live.NET Connector, before installing the Live.NET Connector, we recommend installing Microsoft Visual Studio .NET 2000.

If you are building custom applications, the Live.NET Connector requires a clean installation of the .NET Framework SDK or Microsoft Studio .NET.

In all cases, the Live.NET Connector depends on the .NET Framework, which in turn has dependencies that are listed in Table 7-3. The platform and software requirements for redistributing the .NET framework are available at Microsoft's Web site. We bundle the .NET Framework library with the Live.NET Connector; however, you will need to make sure your system supplies the other dependencies.

Table 7-3. Dependencies matrix for the Live.NET Connector.

Software	Version	Configuration
Internet Explorer	5.01 or later	Client and server
Core WMI for Windows Instrumentation	1.5	Client and server
Windows Installer	2.0	Client and server
MDAC	2.6 or later	Client
MDAC	2.7 or later	Server
Internet Information Services (IIS)	5	Server (with ASP .NET applications)

If you are building custom applications, the Live.NET Connector requires a clean installation of the .NET Framework SDK or Microsoft Studio .NET. For more information, see "Bundling the Live.NET Connector with Custom Applications" on page 278.

LivePDA Connector Requirements

For development purposes, you can install the LivePDA Connector on your desktop computer without having to have a Pocket PC connected. You also do not need to have ActiveSync installed. You can build and run the applications with LivePDA Connector by using the emulator that is shipped with Microsoft Visual Studio 2003.

If you plan to develop applications using the LivePDA Connector API, make sure you have the following software installed and configured before you install the LivePDA Connector:

- Pocket PC 2003
- Pocket PC 2003 SDK

Note: There are additional requirements that must be met when installing the Pocket PC SDKs; see the Microsoft Web site for details.

When you are ready to run the LivePDA Connector on an actual Pocket PC, you will need to connect a Pocket PC to your desktop computer, and you will need to have Microsoft ActiveSync installed on your desktop computer to synchronize the application files between computer and Pocket PC. If you are installing the LivePDA Connector just to use it, there are no further pre-installation requirements.

When your system is ready, you can install the Connector as described under "Installing the Windows Connectors" on page 219.

Installing the Windows Connectors

The installation programs for the Windows suite of Connectors are available at the following Web site.

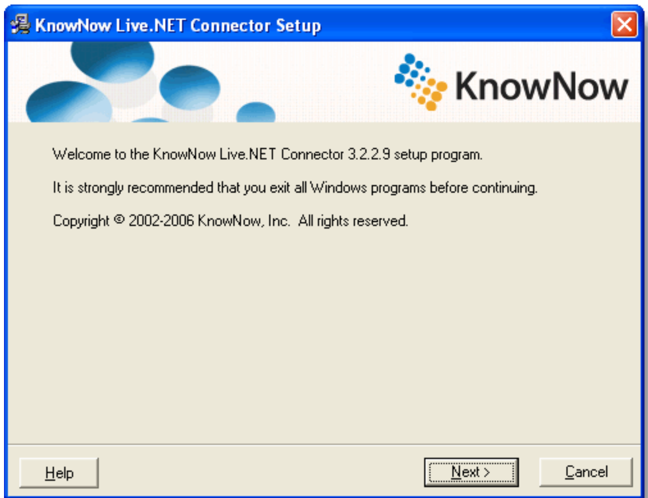
http://www.knownow.com/support/

The following instructions cover the installation process for all the KnowNow Connectors for Windows, using primarily the Live.NET Connector as an example.

To install a Windows Connector,

1. Run the chosen Connector installation program. The Welcome window appears.
2. The next window is a welcome message. Choose **Next**.

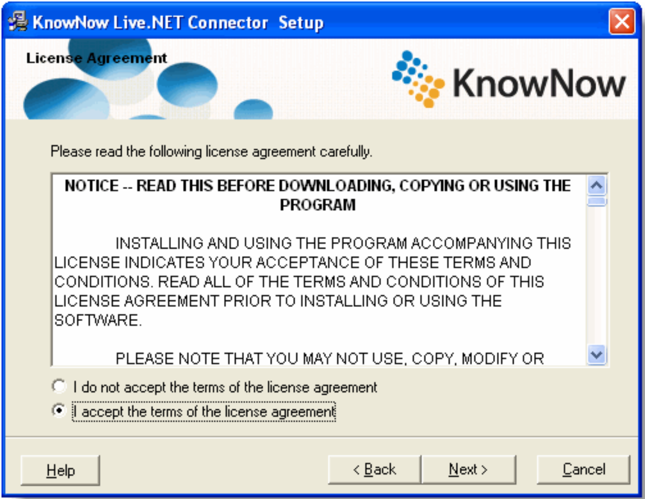
Figure 7-1. Welcome dialog box.



3. If you have a prior installation of the Connector, either one that is older than the one you are installing, or newer, another window will appear.
   - If you have an older installation, you will be notified and given the choice of continuing (and thereby upgrading your existing Connector) or canceling the installation process.
   - If you have a newer installation than the one you are trying to install, the Existing Installation window appears. It is recommended that you use the newer version. You can choose to install the older version, though in that case you will need to exit the installation program and uninstall the newer version before you can run the installation program again to install the older version.

4. The License Agreement window is displayed next. Choose **I accept the terms of the license agreement**, then choose **Next.** (**Next** is disabled until you accept the terms.)

Figure 7-2. License agreement.



5. Next, the Setup Type window appears, asking whether you want a **Typical** or **Custom** installation.

   - If you wish to change the default installation directory or make other changes to the default installation, select **Custom**, then choose **Next** and proceed with step 7 on page 222.

- If you select **Typical**, choose **Next**, then skip to step 9 on page 223.

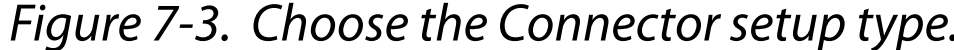
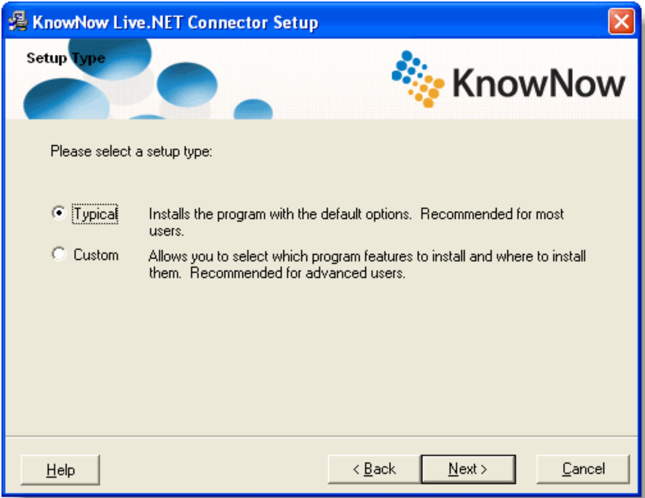


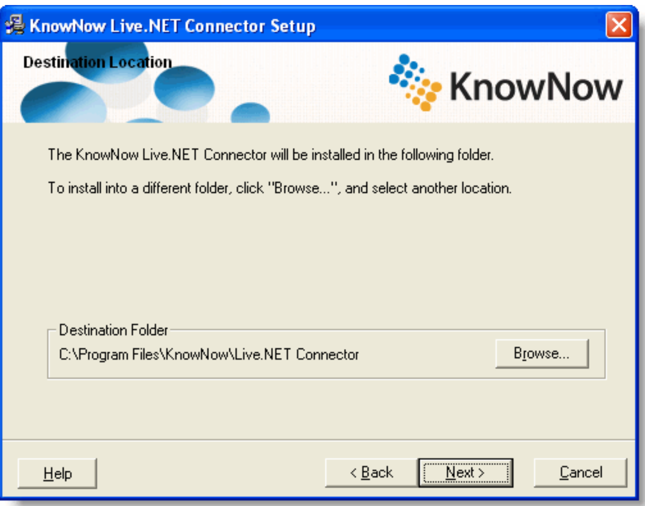
Figure 7-3. Choose the Connector setup type.



6. If you are installing the LiveActiveX Connector, you will see a screen prompting you to specify a which version to use: One for use with Visual Studio 6, one for Visual Studio .NET, or one for Visual Studio .NET 2003. Although you choose one, you are actually installing all three versions, but are only registering one of them. In Windows, you can switch between ActiveX controls by registering or unregistering the desired DLL using the Microsoft Windows regsvr32.exe tool. You can use this tool to change to another version of the LiveActiveX Connector.

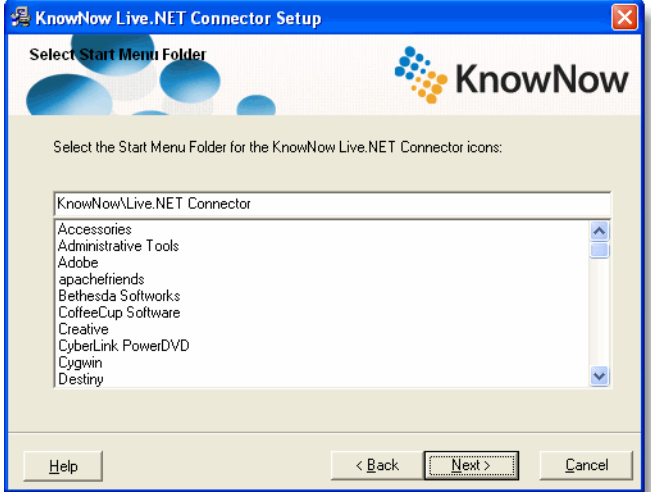
7. Choose the destination folder for the Connector, or accept the default. The default location is C:\Program Files\KnowNow*ConnectorName*. Choose **Browse** to choose a location other than the default.

Figure 7-4. Choose a destination.



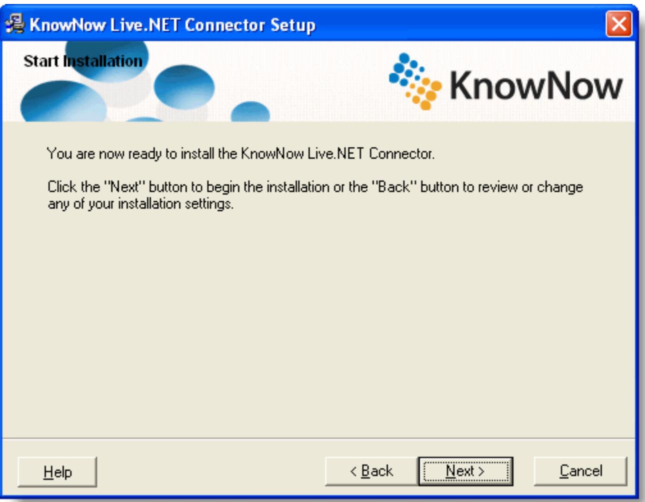
8. After specifying a destination. choose **Next**. Now select a Start Menu folder for the Connector.

Figure 7-5. Select a Start Menu folder.



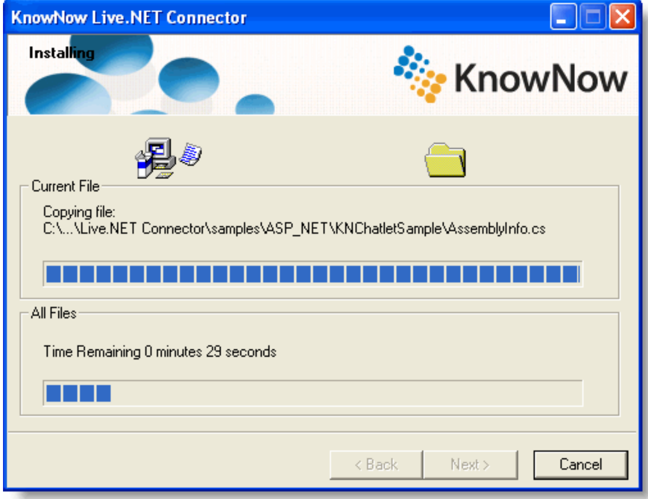
9. Choose **Next**. The installation program is now ready to start.

Figure 7-6. Ready to roll.



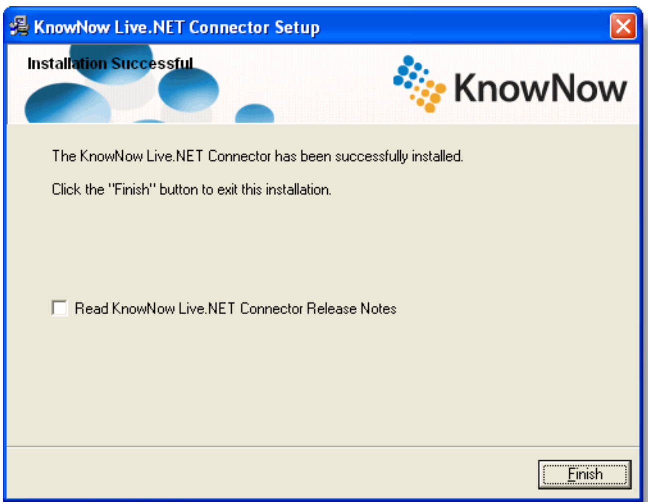
10. Choose **Next**. The progress window reports the installation's progress. If you are installing the LivePDA Connector, the installation program will display some additional dialog boxes.

Figure 7-7. Installing.



11. When the installation is complete, the success window appears. You can choose to view the release notes if you wish. Choose **Finish**.

Figure 7-8. Success!



Uninstalling the Windows Connectors

To uninstall any of the Windows Connectors, access the **Start > Programs > KnowNow *ConnectorName*** menu, then select the uninstall option.

Using the Connector APIs

In order to access a LiveServer, your application needs to be able to talk with the appropriate Connector using calls from the Connector's API. Through the Connector, your application can then exchange information with a LiveServer. As mentioned previously, the basic tasks that your application needs to perform are to generate publish, subscribe, and unsubscribe requests, and to receive data sent from the LiveServer (including from other applications that are sending the data through the LiveServer), as well as to handle status events and optionally manage offline queuing. Of course, your application will need to use the Connector APIs to manage these operations.

Because the Windows Connectors and their APIs are consistent across the board, once you understand how to use one of the language-specific APIs, you should be able to use any of the APIs with little change beyond what is required by the individual languages. The basic use of the APIs is discussed under the following headings and in the remainder of this chapter:

- "Sample Code and Additional Documentation" on page 226
- "Connector Libraries" on page 227
- "Essential Tasks" on page 228
- "Providing Access to the APIs" on page 230

Sample Code and Additional Documentation

Each Connector ships with a set of samples illustrating many of the calls in the APIs. The sample code is heavily commented to make clear what each step of the code is accomplishing. You may find that the samples make excellent base files for your own applications; use them as leverage to reduce your development time.

The sample code is located in the *Connector_root*\samples\ directory. The default location of the sample files is

C:\Program Files\KnowNow*ConnectorName*\samples\

Each set of samples is described in an HTML file in the *Connector_root*\samples\ directory, each HTML file is named for the Connector whose samples it describes; thus,

- LiveActiveX_samples.html
- LiveCPP_samples.html
- LiveNET_samples.html
- LivePDA_samples.html

In addition to the sample code, each Connector ships with a set of HTML files that describe the API calls in more detail. The API documentation can be accessed from any Web browser. The location of the API files is relative to the Connector's installation location. For example, if you installed the LiveC++ Connector for Windows in its default location, the HTML documentation for its API files is located here:

C:\Program Files\KnowNow\LiveC++ Connector\docs\doxygen\index.html

Additional information on the Live.NET Connector's API documentation is provided under "Accessing the Live.NET Connector's API Documentation" on page 268.

Connector Libraries

Each Windows Connector has a DLL, the names of which are listed in Table 7-4.

Table 7-4. Connector libraries.

Connector	Library or Libraries
LiveActiveX Connector	LibKNCom.dll
Live.NET Connector	LibKNDotNet.dll
LivePDA Connector	LibKNPda.dll.

The LiveC++ Connector uses a number of libraries, which are listed in Table 7-5.

Table 7-5. LiveC++ Connector libraries.

Library	Description
LibKN10X86D.lib x86	Debug library built using Visual C++ 6.0
LibKN10X86ND.lib x86	Non-Debug library built using Visual C++ 6.0
LibKN10X86UD.lib x86	Unicode Debug library built using Visual C++ 6.0
LibKN10X86UND.lib x86	Unicode Non-Debug library built using Visual C++ 6.0
LibKN10_71_X86D.lib x86	Debug library built using Visual C++ 7.1
LibKN10_71_X86ND.lib x86	Non-Debug library built using Visual C++ 7.1
LibKN10_71_X86UD.lib x86	Unicode Debug library built using Visual C++ 7.1
LibKN10_71_X86UND.lib x86	Unicode Non-Debug library built using Visual C++ 7.1
LibKN10_7X86D.lib x86	Debug library built using Visual C++ 7.0

Table 7-5. LiveC++ Connector libraries. *(continued)*

Library	Description
LibKN10_7X86ND.lib x86	Non-Debug library built using Visual C++ 7.0
LibKN10_7X86UD.lib x86	Unicode Debug library built using Visual C++ 7.0
LibKN10_7X86UND.lib x86	Unicode Non-Debug library built using Visual C++ 7.0

Essential Tasks

At a very high level, you use a Connector to establish a connection with a LiveServer. Once the connection is established, you can use various methods in the Connector class to perform a variety of tasks, such as publish, subscribe, and so on. You choose which methods you want depending on how you are using the connection. At a minimum, your application needs to perform the following essential tasks:

1. Access the appropriate APIs as described under "Providing Access to the APIs" on page 230.
2. Create an instance of the Connector and set up the required and desired parameters as described under "Creating a Connector" on page 232. The LiveServer to connect to is required; you can optionally supply a user name and password, proxy information, and so on.
3. Perform publish, subscribe, and unsubscribe operations, and control offline queuing. Publish and subscribe examples are provided under "Publish Operations" on page 239 and "Subscribe Operations" on page 242. Offline queuing is described under "Enabling and Using Offline Queuing" on page 239.
4. Catch and manage connection status events (for example, connection up, connection down, and connection retry; connection retry is automatic). You will also need to manage event status events. Status events are discussed under "Request Status Events" on page 236.
5. Disconnect from the LiveServer as described under "Disconnecting from the LiveServer" on page 258.

These operations are discussed in greater detail from a general (non-API-specific) point of view in Chapter 2 , "Events," including explanations of what publish, subscribe and unsubscribe operations are, what events are, and what the LiveServer does with events. The following sections in this chapter provide guidelines for performing these operations specifically using the Windows Connector APIs.

API Classes for the Windows Connectors

All the Windows Connector APIs provide classes to perform the operations listed under "Essential Tasks" on page 228. The following classes are in the LiveC++ Connector's API; the same or similar classes are in the other Windows Connector APIs.

- The Connector class has methods to open a connection, publish, subscribe, unsubscribe, and control offline queuing.
- The ITransport::Parameters method passes parameters to Connector.Open for opening a connection to a LiveServer.
- The Message class is used to build messages for Connector.Publish. Note that message (event) property names are not meant to be very large; keep them under a few thousand characters in size at the most. Smaller names are better for performance. Also, a message with a zero-length property name will cause the Connector to throw an exception.
- IConnectionStatusHandler is used to handle connection status events from Connector.Open and LiveServer connection activity.
- IRequestStatusHandler is used to handle status events for Connector.Publish, Connector.Subscribe, and Connector.Unsubscribe.
- IListener provides the callback mechanism by which you will be receiving events for Connector.Subscribe.

Some applications may only publish data; others may subscribe and unsubscribe, and still others may perform all three operations. In order to understand what is happening with publish, subscribe, and unsubscribe operations, you should also understand what topics, routes, and events are and how they are managed by the LiveServer. For that information, see Chapter 2 , "Events."

In order to use the API calls, you must supply access to the APIs as described under "Providing Access to the APIs" on page 230. For detailed information about the APIs, their syntax, parameters, and such, see the separate API documentation (as mentioned under "Sample Code and Additional Documentation" on page 226).

If you are using the Live.NET Connector and wish to take advantage of the KnowNow WinForms or KnowNow ASP.NET controls, see Chapter 8 , "Using the Live.NET and LivePDA Connectors," for more information on those components.

Providing Access to the APIs

Depending on the language you are using, you will need to create some kind of command in your source code that provides access to the KnowNow APIs. Guidelines for doing so are provided under the following headings:

- "C++ Static Libraries" on page 230
- "COM/ActiveX" on page 231
- ".NET" on page 231

C++ Static Libraries

If you are planning to use the LiveC++ Connector for Windows and are going to write a C++ application to do so, you will need to have access to the KnowNow C++ static library.

Typically, most other C++ static libraries require you to both include the header files and also to modify the link options in order to link in the proper library. The problem with this is that, with so many different development tools, it is often difficult to know which library is the proper one to use. One of the useful aspects of the KnowNow C++ static library is that the header files pull in the proper libraries for you, regardless of what tool you are using.

The static library also has advantages over DLLs in that you don't need to ship an additional DLL with your application. This helps keep your application as small as possible. The static library only pulls in the classes you need, instead of containing every possible call whether your application uses them or not. Of course, DLLs have some nice features, but the C++ static library provides a great deal of flexibility and helps you to keep your application compact.

To use this functionality, if you are using the C++ static libraries for your application, you must reference the library at the top of your source code with a line like this:

```
#include <libkn\Connector.h>
```

This includes the fundamental header file, which in turn includes header files for any other classes needed. There are other header files that you may also wish to include depending on what activities you wish to perform; these header files are listed in the API documentation, which is provided separately as HTML files with the Connector installation. The default location of the documentation is

C:\Program Files\KnowNow\LiveC++ Connector\docs\docygen\index.html

The default location of the header files is

C:\Program Files\KnowNow\LiveC++ Connector\include\libkn\

COM/ActiveX

If you are planning to use the LiveActiveX Connector and are going to write a COM/ActiveX application, you will need to have access to the KnowNow COM/ActiveX APIs.

Because Visual Basic and Visual C++ are very common tools for COM/ActiveX applications, this section discusses how to provide access to the APIs in those two tools. Essentially, whatever language you are using, you will need to provide a reference to the LibKNCom.dll.

- To access LibKNCom.dll in Visual Basic, set a project reference and point it to LibKNCom.dll. Once you've got your project pointing to this DLL, the DLL pulls in references to the proper classes and so forth, making them available to you.
- With Visual C++, use

  ```
  #import "LibKNCom.dll"
  ```

If you don't know how to do this in your language, or if you need a graphical user interface (GUI), you can use a free tool from Microsoft called OLEView to open LibKNCom.dll so you can see the program IDs and GUIDs we use so you can create those objects by hand.

OLEView is bundled with Visual Studio and is also contained within the Microsoft Platform SDK, which you can download for free from the Microsoft Web site.

.NET

If you are planning to use the Live.NET Connector and are going to write a .NET application, you will need to have access to the KnowNow .NET APIs. To access those APIs, you just need to add a reference to LibKNDotNet.dll. You can add a reference for each class in the Solution Explorer by right-clicking on **References** for that class. Or you can add a reference by choosing the **Project** menu and choosing **Add Reference**.

The Connector Class

The Connector class provides the core functionality for the Connector's publish, subscribe, and unsubscribe operations. Other classes that support the operations of the KnowNow Connector are IConnectionStatusHandler, IListener, Message, and RequestStatusHandler. See the API documentation for descriptions of those classes (and the contents of their header files) and for important information on implementing supporting methods for the Connector's operations within your application. This section provides information on creating a Connector under the following headings:

- "Creating a Connector" on page 232
- "Specifying a LiveServer" on page 233
- "Connection Status Handlers" on page 234

Creating a Connector

For an overview of Connector operations and some rules governing their use, see "Introducing Connectors" on page 15.

To create a Connector,

1. To use the Connector class, create an instance of the Connector object.
2. Next, after you have created the instance of the Connector object, use the Open call to specify the LiveServer to connect to. Related to that, you will generally use a connection status handler to respond to connection status messages.
3. At this point, you can publish, subscribe, or perform other tasks as desired. For publish and subscribe operations, you will use some parameters to specify aspects of the publication or subscription, including specifying status handlers.
4. At the end of the session, use the Close command.

Number of Connector Instances

We recommend that you reuse a Connector instance to a LiveServer. If your application requires multiple Connector instances to a LiveServer, then you must ensure that the registry key is set properly as described in this section.

Warning: Editing the Windows registry is fraught with danger. Before you start, back up the registry and be sure of what you are doing.

To enable any WinInet-based application to have more than default number of connections to a LiveServer (usually three connections), you will need to edit a registry setting. Add/Edit a "DWORD value" key "MaxConnectionsPerServer" to

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Internet Settings
```

If you are interested, contact KnowNow Support for more information on this topic.

Examples

Here's an example of creating an instance of a Connector in C#.

```
Connector pubConn = new Connector();
```

And here is Visual Basic example:

```
Dim pubConn As New KnowNow.LiveDotNet.Connector()
```

Specifying a LiveServer

After you create the Connector, you call Open, a method in the Connector class. With Open, you specify the LiveServer to connect to, as well as any other parameters needed. Open initializes the Connector instance with information required for Publish and Subscribe calls later on. Unlike Subscribe, Open does not create a persistent connection.

Note: Although Open returns a value of **true** or **false** for the success or failure of the connection, because of how Windows works, you cannot rely on the return value to determine whether the connection is open or whether a LiveServer exists.

With Open implemented, on application startup, the Connector automatically attempts to connect with the specified LiveServer. If the initial connection fails, the Connector will not try to reconnect. However, if the initial connection succeeds, and then the connection later fails, the Connector will automatically try to reconnect. As this is all done automatically, your application does not need to take any action except for those mentioned under "Enabling and Using Offline Queuing" on page 239 and "Subscribe Operations" on page 242.

Connection Parameters

Open uses ITransport::Parameters to identify the LiveServer you want to test the connection to. The LiveServer identification is stored in the **m_serverURL** parameter, which is required by Open. The URL must be complete, not relative, and must be in this format:

```
http://yourLiveServer:8000/kn
```

Caution: The /kn portion of this URL is vital.

Open also uses ITransport::Parameters to set other, optional parameters as desired, such as user name, password, and so on. You can examine ITransport::Parameters in the separate API documentation (described under "Sample Code and Additional Documentation" on page 226) to see what kinds of parameters are passed and how they are passed. Some parameters include user names, passwords, proxy information, and parameters relating to dialog boxes and other aspects of the connection.

The utility function GetParameters retrieves the parameters stored in ITransport::Parameters.

Connection Status Handlers

If you wish, you can create a connection status handler to receive status updates regarding the connection to the LiveServer—whether the connection is up or down, the fact that the connection was down, and when the connection was re-established. This section describes the connection status handler; for information on creating status handlers for Publish, Subscribe, and Unsubscribe operations, see "Request Status Events" on page 236.

Connection status handlers give the following information on the Connector's status:

- Connected
- Disconnected
- Queue files
- Queue flush
- Retry connection

You do not need to create a connection status handler; however, if you wish to do so, derive your connection status handler, OnConnectionStatus, from the IConnectionStatusHandler base class.

OnConnectionStatus checks the status of the network. When it is implemented, your application will receive status callbacks whenever the status of the network changes.

You must implement this status handler in each instance of the Connector class so that it performs operations as you desire. You can, for example, have your implementation of OnConnectionStatus provide information on the connection, or you can have it take an action depending on the status. For example, when the connection goes down, you could use this method to start an offline queue.

After you open a connection, you must make a call to Connector::AddConnectionStatusHandler to register your OnConnectionStatus status handler. The connection status handler will then be called whenever the status of the connection changes. You can remove the status handler with Connector::RemoveConnectionStatusHandler.

Checking the Connection Status

For checking the connection status, you can use IsConnected. IsConnected checks to see if the Connector is connected to a LiveServer.

Caution: IsConnected only returns a valid value after you have used Subscribe. Until this, you cannot rely on IsConnected.

Request Status Events

The LiveServer uses request status events to tell your application what happened to the publish, subscribe, or unsubscribe request you just made; 200 OK, 401 Authentication Required, and the like. Request status events provide information specific to the Publish, Subscribe, and Unsubscribe calls. (From now on, we will refer to request status events more simply as status events.)

- For Publish, the status events provide information on the success or failure of the Publish command.
- For Subscribe, the status events provide information on the status of the events being subscribed to.
- For Unsubscribe, the status events provide information on the success or error of the Unsubscribe call.

You must provide some way to manage those status events, using IRequestStatusHandler as a base class, implementing OnError, OnStatus, or OnSuccess from that base class.

The following sections discuss the need for using status handlers and provide publish and subscribe examples in several languages:

- "The IRequestHandler Class" on page 236
- "Publish Operations" on page 239
- "Subscribe Operations" on page 242
- "Unsubscribe" on page 246

The IRequestHandler Class

The IRequestStatusHandler class provides three possible status handling methods relating to the Connector::Publish, Connector::Subscribe, and Connector::Unsubscribe commands. These status handling methods are OnError, OnSuccess, and OnStatus. You must derive your own IRequestHandler class and implement (by overriding) your choice of these methods.

Although you can use all three methods, OnSuccess and OnError are normally used together. OnStatus can be used to replace both OnSuccess and OnError. Which status handler(s) you use is up to your programming needs and preferences. If you use all three, you will get duplicate calls. We recommend using either (OnError and OnSuccess) OR (OnStatus), but not all three.

For information on creating a connection status handler, see "Connection Status Handlers" on page 234.

Note: Success is defined as a 200 Okay message. Anything else is regarded as an error.

OnError

In OnError, you define what actions to take if a Connector::Publish/Subscribe/Unsubscribe command is not successful. With OnError implemented, you will only receive the error callbacks.

If you are using OnError, you may also wish to implement OnSuccess to receive the success callbacks, though you do not need to.

Normally, you would not also want to implement OnStatus if you have implemented OnError. If you do choose to implement both OnError and OnStatus, you will receive duplicate calls.

Alternatively, you can implement OnStatus only to take the place of OnError and OnSuccess.

OnStatus

In OnStatus, you define what actions to take if a Connector::Publish/Subscribe/Unsubscribe command is successful or if the command is not successful. OnStatus is triggered any time one of these commands is called. With OnStatus implemented, you will receive all callbacks, both for errors and success.

If you wish to use OnStatus, we recommend that you use it in the place of OnError and OnSuccess, rather than in addition to those calls. Otherwise, if you implement OnStatus and OnError and/or OnSuccess, you will receive duplicate calls.

OnSuccess

In OnSuccess, you define what actions to carry out if a Connector::Publish/Subscribe/Unsubscribe command is successful. With OnSuccess implemented, you will only receive the success callbacks.

If you are using OnSuccess, you may also wish to implement OnError to receive the error callbacks, though you do not need to.

Normally, you would not also want to implement OnStatus if you have implemented OnSuccess. If you do choose to implement both OnSuccess and OnStatus, you will receive duplicate calls.

Alternatively, you can implement OnStatus only to take the place of OnError and OnSuccess.

Publish Operations

The Connector class contains the core commands for publishing, subscribing, and unsubscribing. The command for publishing is Publish. A Publish command publishes an event to a topic on the specified LiveServer, which then in turn manages those events, storing them in the appropriate topics and sending those events to subscribers to those topics. If the topic does not already exist, the LiveServer will create the topic when it receives the Publish command. For more general information on publish and subscribe operations, see Chapter 2 , "Events."

Each Connector can have multiple publications, and each application can have multiple Connectors, one for each LiveServer of interest, though the needs of your application will determine your exact implementation. For example, one application may only publish, while another may only subscribe.

The parameters that Publish takes are a Message and an IRequestStatusHandler.

- The Message is the actual message (the event) that is being published.
- An IRequestStatusHandler. When you call Publish, you must also pass in a status handler. The LiveServer returns a status (success or error) when it receives a Publish command. It is up to you to decide what actions to take upon success or error. When the Connector receives a status from the LiveServer, it checks the returned status code and calls your status handler. For more on status handlers, see "The IRequestHandler Class" on page 236.

Enabling and Using Offline Queuing

Offline queuing is a related and very useful operation for publish operations. Normally, if your application is disconnected from the LiveServer and your application sends a publish event, the publish operation will fail. But if offline queuing is enabled in the Connector, the Connector detects when the connection is down and saves Publish events in memory (or, optionally, a file on disk), creating a queue. Later, when the connection is re-established (which is done automatically), the Connector automatically sends the events in memory to the LiveServer. Status events are also sent to indicate when the queue is being flushed upon reconnection.

With offline queuing, the event flow is something like this:

```
Connected
Disconnected
       publish request #1 ==> sent to queue
       publish request #2 ==> sent to queue
Reconnected
Status message: Queue flushing has begun
       flush publish event #1 to LiveServer
```

```
        flush publish event #2 to LiveServer
Status message: Queue flushing has ended
Back to normal status
```

Offline queuing is an optional capability. You do not have to implement or use offline queuing.

Offline Queuing API Calls

The primary method for offline queuing is SetQueueing, a method in the Connector class. SetQueueing activates or deactivates offline queuing individually for each publisher. SetQueueing is off by default.

To enable offline queuing, set SetQueueing to **on**. You don't need to do anything further if you don't wish, though there are optional offline queuing capabilities available, such as writing the queue file to disk and later reading that file from disk, as well as managing some of the otherwise automatic functions. And for persistence, you may wish to use SaveQueue and LoadQueue.

With the additional offline queuing methods,

- You can turn queuing on and off dynamically.
- You can check to see if the queue has items in it.
- You can call SaveQueue to save the queue to a text file before the application exits.
- At restart, you can call LoadQueue to load the saved queue.

For example, before the application shuts down, you may wish to have it check to see if the queue has items in it. If so, you can have the application send a SaveQueue command, then shut down. On restart, it can call LoadQueue so the saved messages can be loaded and flushed to the LiveServer upon connection.

In alphabetical order, the other offline queuing methods (in the Connector class in the C++ API) are

- Clear, which deletes the queued events from memory without sending them to the LiveServer.
- Flush, which sends the queued events in memory to the LiveServer.
- GetQueueing, which retrieves information on whether queuing is enabled or disabled.
- HasItems, which checks to see if the queue in memory has any events in it.
- LoadQueue, which loads the events from the file created by SaveQueue. Normally, you would use LoadQueue followed by a Flush.

- SaveQueue, which saves the queued events to a file. When called, SaveQueue performs a Clear call as part of its actions.

For detailed information about the APIs, their syntax, parameters, and such, see the separate API documentation, whose location is provided under "Sample Code and Additional Documentation" on page 226.

Offline Queuing Status Codes

The Connectors use the following status codes for offline queue messages.

- 15000 - Flush Start
- 15001 - Flush End
- 15002 - Error in Flush
- 15003 - Sent for every message flushed

The standard format of a status event is as follows:

- status: The value is a string that is a response from the LiveServer; for example, "200 Watching topic."
- status_code: The code associated to this string; for example, "200."
- ErrorString: This header is filled if the status code is not in the 200-300 range

Offline Queuing Examples

The following example shows how to enable offline queuing in C#.

```
pubConn.SetQueueing(true);
```

And here is how it is done in Visual Basic:

```
pubConn.SetQueueing(True)
```

Subscribe Operations

The Connector class contains the core commands for publishing, subscribing, and unsubscribing. Depending on which capabilities you want to take advantage of, there are different commands for subscribing as described in this section.

Essentially, a subscribe command sends a request to the LiveServer asking the LiveServer to send to the application all events related to the topic specified in the Subscribe call. When the LiveServer receives this command, it sets up a route and sends events from that topic to the application over that route. If the topic does not already exist, the LiveServer will create the topic when it receives the Subscribe command.

Note: A call to Subscribe returns a unique route ID. You must use that same route ID when unsubscribing to that topic.

For more information on publish and subscribe operations and on topics, routes, and events, see Chapter 2 , "Events."

Each Connector can have multiple subscriptions, and each application can have multiple Connectors, one for each LiveServer of interest, though the needs of your application will determine your exact implementation. For example, one application may only publish, while another may only subscribe.

Subscribe

Subscribe takes four parameters:

- The topic to subscribe to; for example, /what/chat.
- A listener. This is the callback mechanism by which you will be receiving events. It "listens" to the topic being subscribed to in this Subscribe call. Whenever the Connector gets a message for the topic defined in this Subscribe call, the Connector automatically makes a call to your implementation of OnUpdate and forwards the message contained in OnUpdate to the subscriber. You **must** have a listener in order to receive events. For more information, see "The IListener Class" on page 244.
- A Message, which is used to pass in options for this subscription. A Message is required. It can be empty, but it cannot be null. The following subscription options can be used. You can find more information about these subscription options in Chapter 2 , "Events."
    - kn_history_since_n, which obtains the most recent *n* events.

    - kn_history_since_age, which obtains events that are newer than a specified age. Specifying infinity will fetch all events in the topic regardless of their timestamp.
    - kn_hold_new_events, which is used with kn_history_since_age and kn_history_since_n. This header guarantees that any events posted to the kn_from topic in the route while the initial route publication (IRP) is being processed will not be delivered to the kn_to topic until after the IRP is complete.
    - kn_deletions, which tells the LiveServer whether you want to receive notification of when events are deleted in a topic. **True** = Receive notification. **False** = No notification. The default value is **False**.
    - kn_expires, which sets the expiration time on your subscription.
- An IRequestStatusHandler. This status handler is derived from the IRequestStatusHandler class, which is described under "The IRequestHandler Class" on page 236. Just as for a Publish or Unsubscribe command, you need to implement status handlers for your Subscribe command. These status handlers relate to the events being subscribed to. It is up to you to decide what is an error and what is success, and what actions to take upon each.

SubscribeASLC

This command adds some capabilities to the Subscribe command. When you set up a subscription using this command, you could specify a filter that will enable you to replay all messages sitting in a topic in one of two ways:

- based on the time that the LiveServer receives the event (using kn_time_t)
- based on the event ID (using kn_event_id)

This makes it possible to, for example, receive all events that had been posted since the last connection.

For example, suppose you are subscribing to a topic and are receiving events from that topic, and there is a network failure between your application and the LiveServer. If another application is still connected and is still publishing to that topic, of course your application will not see those events. If you are using SubscribeASLC, you can retrieve all unexpired events from the point of failure in one of two ways listed above.

SubscribeASLC takes seven parameters:

- The topic name to subscribe to.
- An enumerator type to identify which filter you want to use. The type is either 0 or 1, depending on whether you want to use a time (0) or an event ID (1).

- The information in string format which contains the value of the filter (time or event).
- A vector that is a list of all the events being sent from the LiveServer that you want the Connector to filter out (i.e., that you want the Connector to ignore). For example, if the LiveServer sends four events with the same timestamp, each with a unique event ID, then you can supply the event ID to replay events from. The Connector will ignore events preceding that event ID, even though events before that ID may have the same timestamp. Alternatively, you could supply the time to replay events from.
- A listener, which is the same as the listener used in Subscribe.
- A Message, which can contain the same types of options as Subscribe.
- An IRequestStatusHandler, which is the same as the IRequestStatusHandler used in Subscribe.

The IListener Class

In addition to connection and status events, there is a special kind of handler called a listener. Subscribing applications must provide a listener event handler to get the new events that are published. In order to provide this listener, you must derive your own IListener class from the IListener base class. In creating your own IListener class, you must override the OnUpdate method. The Connector calls OnUpdate whenever someone publishes an event to the topic you are listening to.

OnUpdate is essential to the Subscribe command of the Connector class. Note that this callback is called from a separate thread than the Connector's. It is the application's responsibility to ensure that anything done in OnUpdate is thread-safe.

Every time an event is published to the topic you are subscribing to, that update is sent through OnUpdate, which the Connector automatically calls. That update contains a message concerning the subscription. This message comes from the LiveServer through the Connector to the application's OnUpdate. When the application receives that message, it should take some action (as defined by you in your implementation of OnUpdate). This message contains potentially quite a lot of information in the form of kn_ headers, again as defined by OnUpdate.

The most important kn_ headers are

- kn_from and kn_to
- kn_id
- kn_route_id
- kn_time_t

- kn_uri

For more information on these headers, as well as a list of the other kn_ headers, what they can contain, and how they are used, see Chapter 2 , "Events."

Other Subscribe Operations

Internally, the Subscribe method calls GetRouteId and SubscribeRouteId. Therefore, you do not need to call these methods explicitly. However, there may be times when you want to separate the functionality of the Subscribe call. In such cases, you can use the pair of methods GetRouteId and SubscribeRouteId. If you are not using Subscribe's automatic GetRouteId, you must call GetRouteId to create a route to a topic you are subscribing to. The route ID that is returned from this call must then be passed on to SubscribeRouteId to set up the subscription for that route.

Unsubscribe

The Connector class contains the core commands for publishing, subscribing, and unsubscribing. The command for unsubscribing is Unsubscribe. Unsubscribe tells the LiveServer that the application is no longer interested in receiving events that are published to the specified topic. The Unsubscribe call also closes the route (the tunnel) from the application to that topic. When the LiveServer receives the Unsubscribe command, it stops sending events on that topic to the application, though it does not destroy the topic itself. (After all, other applications may still be subscribing to that topic.)

Unsubscribe takes two parameters:

- A unique route ID that identifies the route that is being unsubscribed from. This route ID is returned by a call to Subscribe. You must use that same route ID when unsubscribing to that topic.
- An IRequestStatusHandler. This status handler is implemented from the IRequestStatusHandler class, which is described under "The IRequestHandler Class" on page 236. Just as for a Publish or Subscribe command, you need to implement status handlers for your Unsubscribe command. These status handlers are similar to those needed for Publish and Subscribe, though in this case the status relates to the success or error of the Unsubscribe call. When the Connector sends the Unsubscribe call to the LiveServer, the LiveServer acknowledges with an "okay" message. If no "okay" is received from the LiveServer, then it is up to you to decide what to do, such as retrying the Unsubscribe call.

Cursors

Cursors support the ability to page through events in a topic a page at a time. You can also move through those events in other ways when using cursors. A discussion of how cursors work is provided under "Cursors" on page 146 as well as in the separate API documentation. The API calls that support cursors are:

- CursorParameters
- ICursor
- ICursorListener
- IPage

To use the Cursor class,

1. Create an instance of the Connector class.
2. After you have created the instance of the Connector class, use the Open call to initialize the Connector.
3. Call the Connector class's InitializeCursor method to obtain an instance of the Cursor class.
4. Next, call the GetNextPage method to get the first page of events. After that, you can call any of the methods GetNextPage, GetPreviousPage, and so on. You can reset the starting position and manipulate other aspects of the cursor as well.
5. At the end of the session, we recommend that you use the Connector class's CleanupCursor API to deinitialize the Cursor rather than destroying the object in your application.
6. Finally, use the Close command to close the connection with the LiveServer.

Heartbeat

The LiveServer's heartbeat capability makes it possible to determine whether a Connector is present and for a Connector to confirm the presence of a LiveServer. The Windows Connectors support the heartbeat with the callbacks and interfaces discussed in this section.

Connectors start the heartbeat by subscribing to the statistics topic /kn_system/heartbeat, and stop the heartbeat by unsubscribing from the same topic. The LiveServer notifies Connectors about its presence in two ways:

- by sending events every 10 seconds when the Connector subscribes to /kn_system/heartbeat
- by sending white spaces through the tunnel after a particular interval; the Connector sends a heartbeat event whenever it receives those white spaces

The LiveServer sends the white space heartbeats only when there is no activity down the tunnel otherwise. In other words, if there are no pending events to be sent to the Connector from the LiveServer, to keep the tunnel alive, the LiveServer sends a whitespace heartbeat after 30 seconds, and every 30 seconds thereafter. Connectors can notice both types of events and notify the application about the LiveServer's presence. When the Connector does not receive events in either of these ways, then it sends a timeout message to the subscriber application.

API Support for the Heartbeat Feature

To take advantage of this feature, subscriber applications need to implement IHeartbeatListener. IHeartbeatListener is a callback listener type with two methods named OnUpdate and OnTimeout. The Connector calls OnUpdate whenever there is a heartbeat event from LiveServer. The Connector calls OnTimeout when no events have been received from LiveServer for 40 seconds.

Additional APIs that support the heartbeat feature are

- TimeoutStatus, which is an enumeration to notify the subscriber application about the reason for the timeout. It has two values: one to indicate that the LiveServer is down, and the other to indicate that the LiveServer is in an unstable state.
- The listener interface IHeartbeatListener, which you can use to catch the heartbeat event and the heart beat timeout.

Logging

For debugging purposes, the Windows Connectors support logging. Using the logging capabilities, and depending on the Connector, you can enable or disable logging, set a target log file name, set a log error level, declare whether rolling log files are used, and set the maximum size of the rolling log files before the oldest files are overwritten. The default settings are shown in Table 7-6.

Table 7-6. Default settings for logging.

Connector	Target Log File	Log Level	Logging	Rolling	Maximum Size of the Rolling File
LiveActiveX Connector	LiveCpp.log	KN_ERROR	Enabled	N/A	N/A
LiveC++ Connector	LiveCpp.log	KN_ERROR	Enabled	N/A	N/A
Live.NET Connector	LiveDotNet.log	KN_ERROR	Enabled	On	1000KB
LivePDA Connector	LivePDA.txt	KN_ERROR	Enabled	On	1000KB

The remainder of this section provides information on logging under the following headings:

- "Live.NET Connector and LivePDA Connector Logging" on page 249
- "Configuring Logging" on page 250
- "Logging Messages and Log Level" on page 251

Live.NET Connector and LivePDA Connector Logging

The Live.NET Connector uses a third-party open-source software package named log4net. Different versions of log4net are available for .Net Framework and .Net CF. log4net makes logging highly configurable and easy. The Live.NET Connector uses log4net 1.2.0.8.

Configuring Logging

There are three ways you can change the Connector's logging configuration: through a configuration file, through the API, and through the default settings. If configuration settings have been made using both the API and a configuration file, the order of precedence for configuration settings in which the Connector executes or uses the settings is as follows:

- Any configuration settings made using the configuration file override configuration settings made using the API, and also override any default settings. Using a configuration file is discussed under "Using Configuration Files" on page 250.
- Any settings made using the API override the default settings. Using the API is discussed under "Using the API" on page 251.
- If there are no settings in the configuration file or in the API for a logging capability, the Connector uses the default configuration settings listed in Table 7-6 on page 249.

Using Configuration Files

You can make configuration settings in a configuration file. You can change some or all of the default configuration settings at the configuration file level; any settings you do not set in this file will be set through either the API (if you use the proper call) or through the default settings.

To enable logging based on a configuration file,

- for the LiveActiveX Connector, LiveC++ Connector, and Live.NET Connector, place the configuration file in your application's directory.
- For the LivePDA Connector, place the configuration file in the device's root.

At boot time, the Connector first looks for the configuration file in the same directory in which the application is running. (For the LivePDA Connector, it looks in the device's root.) If the Connector finds such a file, it loads the configuration settings from that file and ignores any configuration settings that were made using the API, and also ignores any default settings that the configuration file sets differently.

Configuration settings passed through the configuration file are the same for all instances of an application.

The names of the configuration file vary according to the Connector:

- For the LiveActiveX Connector and the LiveC++ Connector, the Connector looks for a KnowNow INI file named KNLogConfig.ini.

- For the Live.NET Connector and the LivePDA Connector, the Connector looks for an XML-formatted log4Net configuration file named KnLogConfig.xml.

A sample log configuration file is provided with each Connector; you can review the information in that file to see how logging can be configured, and you can also use that file as a basis for creating your own configuration file. The sample configuration files are located in the *LiveServer_root**Connector_root*\conf\ directory.

Using the API

Some of the logging options can be set using the appropriate class:

- For the LiveActiveX Connector, the ILogger class
- For the LiveC++ Connector, the Logger class
- For the Live.NET Connector and LivePDA Connector, the KNLogger class

For details on these classes, see the API documentation.

Using the methods in these classes, you can set the target log file and log level, and you can enable or disable logging. For the Live.NET Connector, you can also set whether to use rolling log files, and you can set the maximum size for the log files.

These APIs can be called statically.

Logging Messages and Log Level

Logged messages have the following structure:

```
[DateTime][ProcessId.ThreadId][Class name] Log Severity Level: Text message
```

The *text message* contains *Method Name: Message*. For the Live.NET Connector and LivePDA Connector, the value of *ProcessID* and *ThreadID* is 0 for all logged messages.

The log level is the level at which you wish to receive messages. Any messages below that level are not reported. You can set the log message level as described earlier. Table 7-7 lists the logging error levels.

Table 7-7. Logging error levels.

Sr. Number	Log4Net Level	Connector Level	Remarks
1.	DEBUG	KN_ASSERTION	Informational messages.
2.	INFO	KN_NOTICE	An internal verification failed.
3.	WARN	KN_WARNING	Something needs to be addressed.

Table 7-7. Logging error levels. *(continued)*

Sr. Number	Log4Net Level	Connector Level	Remarks
4.	ERROR	KN_ERROR	A serious problem that did not stop operations.
5.	FATAL	KN_FATAL	Causes the Connector to stop operation.A critical error has taken place.

Presence

Presence allows applications to monitor topics so that information about subscribers can be known: whether they are online or offline, subscribing, unsubscribing, and so on. Presence is supported by the following methods in the Connector class:

- "PresenceSubscribe" on page 253
- "PresenceUnsubscribe" on page 253

PresenceSubscribe

To start listening to a topic for the purposes of presence, use PresenceSubscribe. This function takes the following parameters:

- The name of the topic to monitor.
- The presence listener. When you set up presence, you pass in this presence listener (which is essentially a handler). This handler has two callback mechanisms that your application must implement:
  - OnConnect. This is called whenever an application subscribes to a particular topic for which you are setting up presence. When an application connects to that topic, the LiveServer sends an event containing information within standard headers. Your application can handle that information as needed.
  - OnDisconnect. As with OnConnect, except that it is called whenever an application unsubscribes to the topic being listened to.
- A request status handler.

PresenceUnsubscribe

If you want to stop paying attention to a topic, use PresenceUnsubscribe to specify the topic to stop listening to.

Request/response

The Windows suite of Connectors supports the LiveServer's heavyweight request/response system, which is described under "Request/response" on page 148. That discussion covers all the capabilities of request/response. The discussion in this section is based on that material, providing information on how the Windows Connectors support request/response. It is assumed that you will have already read that detailed discussion of request/response before reading this section.

Note: The Windows Connectors support both blocking and non-blocking requests. The differences between blocking and non-blocking requests are described under "Blocking versus Non-Blocking Requests" on page 155.

The following functions in the Windows Connector APIs support the LiveServer's heavyweight request/response capabilities:

- "AddRequest" on page 254
- "InitRequestBlocking" on page 255
- "AddRequestBlocking" on page 255
- "CleanupRequestBlocking" on page 256
- "UpdateRequest" on page 256
- "AddResponse" on page 256
- "AddProvider" on page 257
- "RemoveProvider" on page 257
- "AddRequestManager" on page 257

AddRequest

This function is for non-blocking requests. This function takes the following parameters:

- A topic name to be shared by the requestor, provider, and service manager (if there is one).
- A unique request ID that is used by the requestor. If you do not provide this ID, then the Connector will create one.
- The request that the requestor is sending to the provider.
- The name of the response topic that the requestor will listen to for the response.

- The ID of the provider (which can be null). If this is null, then the service manager will decide which provider to use.
- A status handler.

InitRequestBlocking

For blocking requests using the Windows Connectors, instead of one method as with the LiveJava Connector, there are three methods, each of which represents a step in the blocking request process. By having three methods, it is possible to send multiple blocking requests using the same initialized object.

The first step for creating blocking requests is to initialize the data for a blocking request. You would do this once using InitRequestBlocking, which takes the following parameters:

- A topic name to be shared by the requestor, provider, and service manager (if there is one).
- The name of the response topic that the requestor will listen to for the response.
- A request status handler.

AddRequestBlocking

The next step in creating blocking requests is to make one or more blocking requests. Each blocking request is made by using a call to AddRequestBlocking for each request. This method takes the following parameters:

- The ID that was returned by InitRequestBlocking.
- A unique request ID that is used by the requestor. If you do not provide this ID, then the Connector will create one.
- The request that the requestor is sending to the provider (the data of the request).
- A timeout specified in milliseconds. If no response has been received within the time specified in this timeout, then the program unblocks and provides an error message.
- The ID of the provider (which can be null). If this is null, then the service manager will decide which provider to use.
- A status handler.
- A reference to a response object. The Connector will populate this message object.

CleanupRequestBlocking

The last step in creating blocking requests is to perform cleanup by unsubscribing to the shared topic and cleaning up the initialization. This cleanup is done using CleanupRequestBlocking. You would call this method once after making all your requests with AddRequestBlocking. The CleanupRequestBlocking method takes the following parameters:

- A handle from InitRequestBlocking.
- A request status handler.

UpdateRequest

At times you may want to update or modify a request that was previously made using AddRequest or AddRequestBlocking. To do so, use UpdateRequest. This method returns **true** or **false**, indicating whether the call was successful (true) or not (false). This method takes the following parameters:

- A topic name to be shared by the requestor, provider, and service manager (if there is one).
- A unique request ID that is used by the requestor. If you do not provide this ID, then the Connector will create one.
- The request that the requestor is sending to the provider (the data of the request).
- The name of the response topic that the requestor will listen to for the response.
- The ID of the provider (which can be null). If this is null, then the service manager will decide which provider to use.
- A status handler.

AddResponse

When a provider processes a request, it needs to post a response by calling AddResponse. This method returns true or false, where true indicates that the action was successful. This method takes the following parameters:

- A topic name to be shared by the requestor, provider, and service manager (if there is one).
- The original incoming request.
- The outgoing response.
- A status handler.

AddProvider

A provider must register itself with the LiveServer or a service manager. To do so, it must use AddProvider. This function takes the following parameters:

- A topic name to be shared by the requestor, provider, and service manager (if there is one).
- Its provider ID.
- The provider topic.
- Options for setting up the route between the shared topic and the journal for the provider. These options are the standard route options, which are set by using kn_ headers as described under "An Overview of kn_ and Other Headers" on page 56.
- Status handlers.

RemoveProvider

To remove a provider from the system, use RemoveProvider, which returns **true** (removal was successful) or **false** (removal was not successful). This function takes the following parameters:

- A topic name to be shared by the requestor, provider, and service manager (if there is one).
- The name of the provider to remove.
- A status handler.

AddRequestManager

To add a service manager to the system, use AddRequestManager, which returns **true** (action was successful) or **false** (action was not successful). This function takes the following parameters:

- A topic name to be shared by requestors, providers, and the service manager (if there is one).
- The service manager's topic.
- Options for setting up the route between the shared topic and the service manager's topic.
- A status handler.

Disconnecting from the LiveServer

Disconnection from the LiveServer is automatic when your application shuts down. However, you can use the Connector.Close method to close the connection to the LiveServer. Although the connection is closed, the Connector itself is not destroyed, nor are any topics that have been created on the LiveServer. You can reuse the Connector after it is closed by using a Publish, Subscribe, Unsubscribe, or EnsureConnected command; all these commands are in the Connector class.

When using the Close() call, be aware that it should never be called from the connection status handler.

The LiveActiveX Connector: Use Fully Qualified Names

In order to avoid possible conflicts with known Microsoft DLL problems, when using the LiveActiveX Connector's Parameters object, use fully qualified namespaces.

Also, as described under "Cursors" on page 146, all the Windows Connectors support cursors. For the LiveActiveX Connector, when coding in Visual Basic using the Windows Connector APO SetPos, to avoid the use of Visual Basic's reserved words STATIC and END, we recommend that you use a fully qualified name while declaring a cursor of type Static. For example, declare the cursor like this:

```
LIBKNCOMLib.STATIC
```

rather than simply like this:

```
STATIC
```

Likewise, when setting the position to END, use a fully qualified name, such as LIBKN-COMLib.END, rather than simply END.

The LiveC++ Connector: Exceptions and Error Codes

The KNException class supports native exception handling in C++. KNException makes it possible for you to pass errors around and to roll back your stack (very much like a try/catch block) in a very clean way. Exceptions are thrown by a number of methods in the API. See the API documentation for details.

The LiveC++ Connector internally uses WinInet to communicate with a LiveServer using the HTTP protocol. Therefore, all the HTTP protocol status codes and WinInet's internal errors are forwarded without any mapping to internal codes. There are no KnowNow internal errors.

Table 7-8 and Table 7-9 on page 262 list the error codes that WinInet supports; the tables are sorted by error code number.

Table 7-8. HTTP status codes.

Text Code	Code	Description
HTTP_STATUS_OK	200	Request completed
HTTP_STATUS_CREATED	201	Object created, reason = new URI
HTTP_STATUS_ACCEPTED	202	Async completion (TBS)
HTTP_STATUS_PARTIAL	203	Partial completion
HTTP_STATUS_NO_CONTENT	204	No information to return
HTTP_STATUS_RESET_CONTENT	205	Request completed, but clear form
HTTP_STATUS_PARTIAL_CONTENT	206	Partial GET fulfilled
HTTP_STATUS_AMBIGUOUS	300	Server couldn't decide what to return
HTTP_STATUS_MOVED	301	Object permanently moved
HTTP_STATUS_REDIRECT	302	Object temporarily moved
HTTP_STATUS_REDIRECT_METHOD	303	Redirection with new access method
HTTP_STATUS_NOT_MODIFIED	304	If-modified-since was not modified
HTTP_STATUS_USE_PROXY	305	Redirection to proxy, location header specifies proxy to use
HTTP_STATUS_REDIRECT_KEEP_VERB	307	HTTP/1.1: keep same verb
HTTP_STATUS_BAD_REQUEST	400	Invalid syntax

Table 7-8. HTTP status codes. *(continued)*

Text Code	**Code**	**Description**
HTTP_STATUS_DENIED	401	Access denied
HTTP_STATUS_PAYMENT_REQ	402	Payment required
HTTP_STATUS_FORBIDDEN	403	Request forbidden
HTTP_STATUS_NOT_FOUND	404	Object not found
HTTP_STATUS_BAD_METHOD	405	Method is not allowed
HTTP_STATUS_NONE_ACCEPTABLE	406	No response acceptable to client found
HTTP_STATUS_PROXY_AUTH_REQ	407	Proxy authentication required
HTTP_STATUS_REQUEST_TIMEOUT	408	Server timed out waiting for request
HTTP_STATUS_CONFLICT	409	User should resubmit with more information
HTTP_STATUS_GONE	410	The resource is no longer available
HTTP_STATUS_LENGTH_REQUIRED	411	The server refused to accept request without a length
HTTP_STATUS_PRECOND_FAILED	412	Precondition given in request failed
HTTP_STATUS_REQUEST_TOO_LARGE	413	Request entity was too large
HTTP_STATUS_URI_TOO_LONG	414	Request URI too long
HTTP_STATUS_UNSUPPORTED_MEDIA	415	Unsupported media type
HTTP_STATUS_RETRY_WITH	449	Retry after doing the appropriate action.
HTTP_STATUS_SERVER_ERROR	500	Internal server error
HTTP_STATUS_NOT_SUPPORTED	501	Required not supported
HTTP_STATUS_BAD_GATEWAY	502	Error response received from gateway
HTTP_STATUS_SERVICE_UNAVAIL	503	Temporarily overloaded
HTTP_STATUS_GATEWAY_TIMEOUT	504	Timed out waiting for gateway
HTTP_STATUS_VERSION_NOT_SUP	505	HTTP version not supported

Table 7-9 lists the WinInet error codes.

Table 7-9. WinInet error codes.

Text Code	Code
ERROR_INTERNET_OUT_OF_HANDLES	12001
ERROR_INTERNET_TIMEOUT	12002
ERROR_INTERNET_EXTENDED_ERROR	12003
ERROR_INTERNET_INTERNAL_ERROR	12004
ERROR_INTERNET_INVALID_URL	12005
ERROR_INTERNET_UNRECOGNIZED_SCHEME	12006
ERROR_INTERNET_NAME_NOT_RESOLVED	12007
ERROR_INTERNET_PROTOCOL_NOT_FOUND	12008
ERROR_INTERNET_INVALID_OPTION	12009
ERROR_INTERNET_BAD_OPTION_LENGTH	12010
ERROR_INTERNET_OPTION_NOT_SETTABLE	12011
ERROR_INTERNET_SHUTDOWN	12012
ERROR_INTERNET_INCORRECT_USER_NAME	12013
ERROR_INTERNET_INCORRECT_PASSWORD	12014
ERROR_INTERNET_LOGIN_FAILURE	12015
ERROR_INTERNET_INVALID_OPERATION	12016
ERROR_INTERNET_OPERATION_CANCELLED	12017
ERROR_INTERNET_INCORRECT_HANDLE_TYPE	12018
ERROR_INTERNET_INCORRECT_HANDLE_STATE	12019
ERROR_INTERNET_NOT_PROXY_REQUEST	12020
ERROR_INTERNET_REGISTRY_VALUE_NOT_FOUND	12021
ERROR_INTERNET_BAD_REGISTRY_PARAMETER	12022
ERROR_INTERNET_NO_DIRECT_ACCESS	12023

Table 7-9. WinInet error codes. *(continued)*

Text Code	Code
ERROR_INTERNET_NO_CONTEXT	12024
ERROR_INTERNET_NO_CALLBACK	12025
ERROR_INTERNET_REQUEST_PENDING	12026
ERROR_INTERNET_INCORRECT_FORMAT	12027
ERROR_INTERNET_ITEM_NOT_FOUND	12028
ERROR_INTERNET_CANNOT_CONNECT	12029
ERROR_INTERNET_CONNECTION_ABORTED	12030
ERROR_INTERNET_CONNECTION_RESET	12031
ERROR_INTERNET_FORCE_RETRY	12032
ERROR_INTERNET_INVALID_PROXY_REQUEST	12033
ERROR_INTERNET_NEED_UI	12034
ERROR_INTERNET_HANDLE_EXISTS	12036
ERROR_INTERNET_SEC_CERT_DATE_INVALID	12037
ERROR_INTERNET_SEC_CERT_CN_INVALID	12038
ERROR_INTERNET_HTTP_TO_HTTPS_ON_REDIR	12039
ERROR_INTERNET_HTTPS_TO_HTTP_ON_REDIR	12040
ERROR_INTERNET_MIXED_SECURITY	12041
ERROR_INTERNET_CHG_POST_IS_NON_SECURE	12042
ERROR_INTERNET_POST_IS_NON_SECURE	12043
ERROR_INTERNET_CLIENT_AUTH_CERT_NEEDED	12044
ERROR_INTERNET_INVALID_CA	12045
ERROR_INTERNET_CLIENT_AUTH_NOT_SETUP	12046
ERROR_INTERNET_ASYNC_THREAD_FAILED	12047
ERROR_INTERNET_REDIRECT_SCHEME_CHANGE	12048
ERROR_INTERNET_DIALOG_PENDING	12049

Table 7-9. WinInet error codes. *(continued)*

Text Code	Code
ERROR_INTERNET_RETRY_DIALOG	12050
ERROR_INTERNET_HTTPS_HTTP_SUBMIT_REDIR	12052
ERROR_INTERNET_INSERT_CDROM	12053
ERROR_INTERNET_FORTEZZA_LOGIN_NEEDED	12054
ERROR_INTERNET_SEC_CERT_ERRORS	12055
ERROR_INTERNET_SEC_CERT_NO_REV	12056
ERROR_INTERNET_SEC_CERT_REV_FAILED	12057
ERROR_HTTP_HEADER_NOT_FOUND	12150
ERROR_HTTP_DOWNLEVEL_SERVER	12151
ERROR_HTTP_INVALID_SERVER_RESPONSE	12152
ERROR_HTTP_INVALID_HEADER	12153
ERROR_HTTP_INVALID_QUERY_REQUEST	12154
ERROR_HTTP_HEADER_ALREADY_EXISTS	12155
ERROR_HTTP_REDIRECT_FAILED	12156
ERROR_HTTP_NOT_REDIRECTED	12160
ERROR_HTTP_COOKIE_NEEDS_CONFIRMATION	12161
ERROR_HTTP_COOKIE_DECLINED	12162
ERROR_HTTP_REDIRECT_NEEDS_CONFIRMATION	12168
ERROR_INTERNET_SECURITY_CHANNEL_ERROR	12157
ERROR_INTERNET_UNABLE_TO_CACHE_FILE	12158
ERROR_INTERNET_TCPIP_NOT_INSTALLED	12159
ERROR_INTERNET_DISCONNECTED	12163
ERROR_INTERNET_SERVER_UNREACHABLE	12164
ERROR_INTERNET_PROXY_SERVER_UNREACHABLE	12165
ERROR_INTERNET_BAD_AUTO_PROXY_SCRIPT	12166

Table 7-9. WinInet error codes. *(continued)*

Text Code	Code
ERROR_INTERNET_UNABLE_TO_DOWNLOAD_SCRIPT	12167
ERROR_INTERNET_SEC_INVALID_CERT	12169
ERROR_INTERNET_SEC_CERT_REVOKED	12170

Chapter 8 Using the Live.NET and LivePDA Connectors

All Connectors manage communications between the LiveServer and your application. While you do not need to use a Connector, a Connector can make your life much easier by simplifying the calls and operations your application needs to perform. Therefore, when creating or customizing your Windows application to perform publish and subscribe operations with the LiveServer, you may want to have your application use one of the Windows Connectors instead of dealing directly with the LiveServer.

The Live.NET Connector and the LivePDA Connector are part of the Windows suite of Connectors. For an overview of the capabilities of all Windows Connectors, including system requirements and installation instructions, see Chapter 7 , "Using the Windows Connectors." Be sure to also read Chapter 2 , "Events," for additional important information on the LiveServer's operations and architecture, as well as Chapter 5 , "Common Connector Capabilities," for information on capabilities that are common to all KnowNow Connectors.

- "About the Live.NET and LivePDA Connectors" on page 268
- "About the WinForm Components" on page 269
- "About the KnowNow ASP.NET Controls" on page 272
- "Programming Notes" on page 278

About the Live.NET and LivePDA Connectors

The Live.NET and LivePDA Connectors work on the Windows and PDA platforms. For system requirements, supported operating systems, supported development tools, and installation instructions, see Chapter 7 , "Using the Windows Connectors." That chapter also contains an overview of the API capabilities of all Windows Connectors, including basic and essential tasks that all Connectors can or must perform.

This chapter addresses what is unique to the Live.NET and LivePDA Connectors, which we refer to as the Live.NET Connector for short.

Accessing the Live.NET Connector's API Documentation

This chapter provides a descriptive overview of the Live.NET Connector's components and controls. This chapter is not an API reference, however. For detailed programming information on using these components, please refer to the API documentation included with the Live.NET Connector.

- To access the API documentation in Windows help format, open the following file:

  LiveServer_root\Live.NET Connector\docs\index.chm

  Where *LiveServer_root*\ is where the KnowNow directory where you installed your Live.NET Connector. For example, if you chose to install in the default location, the path would be

  C:\Program Files\KnowNow\Live.NET Connector\docs\index.chm

- You can also access the same content in HTML format by accessing

  LiveServer_root\Live.NET Connector\docs\doxygen\index.html

Also, the numerous samples that ship with the Live.NET Connector are heavily commented to illustrate the use of the various forms and controls. You can make use of any of those samples as a basis for creating your own applications. The samples are included in

LiveServer_root\Live.NET Connector\samples\

Also note that the default HttpWebRequest.Timeout value is 100 seconds. It will handle no more than 8,000 events in a single topic. Any topic that has more than 8,000 events will see a timeout exception.

About the WinForm Components

Microsoft Windows Forms (WinForms) is a set of classes in the Microsoft .NET Framework that provide extensible libraries for creating controls for user interfaces. In addition to offering all the API capabilities described in Chapter 7 , "Using the Windows Connectors," the Live.NET Connector has a rich set of KnowNow WinForm components. The KnowNow WinForm components are KnowNow classes that can be added to the .NET IDE toolbox and dragged and dropped onto a form for further UI customization and to add a number of excellent capabilities to the UI. These classes offer a number of customization options, and can also be used programmatically as well.

The KnowNow WinForm components are described under the following headings:

- "An Overview of the KnowNow WinForm Components" on page 269
- "Windows Form and Control Methods" on page 271

An Overview of the KnowNow WinForm Components

The KnowNow WinForm components are:

- Live Data Source
- KNChatlet
- KNEventViewer
- KNImage
- KNMarqueeBand
- KNTree

Live Data Source

The Live Data Source component can be used to manage and prepare a bindable collection of events present at the LiveServer or to manage XML data using KnowNow's XML format. The Live Data Source components are KNCommand, KNConnection, KNDataAdapter, and KNDataSet.

KNDataSet is the central Live Data Source object. It is a memory-resident representation of events present in a topic at the LiveServer. KNDataSet represents a collection of KNDataTables, where each table represents data in a topic at the LiveServer.

In addition to KNDataSet and KNDataTable, you can use the KNCommand, KNDataAdapter and KNConnection objects to query the LiveServer to receive events and save events back to the LiveServer.

KNChatlet

The KNChatlet component can be placed in any Form application to create a chatlet. When creating each chatlet, you provide a chat topic name. Chatlets are public chat areas that any number of users can access (as long as they have the correct server access permission). The users of all the KNChatlet controls that use the same topic name can chat with each other using that chatlet.

KNEventViewer

The KNEventViewer component can be used to display events in table format. Using KNEventViewer, one can view an event's contents and properties. The associated event form makes it possible to add an event to the view, delete an event from the view, or publish an acknowledgement of an event.

The KNEventViewer can be run in two modes: A publisher mode, where the user has access to publishing data and changing the properties of the event viewer, and a subscriber mode, where the user can only view data. The publisher can provide necessary access rights to the subscriber.

KNImage

The KNImage component can be used to display dynamic images in a desktop application. The control can be placed in any Form application; when doing so, you give a topic name. The topic name is where events with image information are published.

When publishing events holding image information, providing its URL can specify the location of the image. The URL can be a Web URL (starting with http:// or https://) or a file URL (starting with file://).

KNMarqueeBand

The KNMarqueeBand component can be used to display a dynamic scrolling marquee of events. The data is added, updated, or deleted as soon it is added, updated, or deleted on the LiveServer. The marquee renders one item per event in the data topic. The entries in the marquee can optionally contain hyperlinks and can optionally display a tool tip. Marquees can scroll horizontally and vertically, up, down, left, and right. You can configure the speed at which the marquee scrolls.

KNTree

KNTree is a simple topic tree that makes it possible for you to drop in a tree component that allows the dynamic browsing of a topic space via recursive discovery.

KNTree also facilitates adding and deleting a topic at the LiveServer. A context menu with add, update, and delete options is provided for that purpose.

Windows Form and Control Methods

By design, Windows Form and Control methods cannot be called on a thread other than the one that created the form or control. If you attempt to do this, an exception is thrown.

This is applicable for developers building applications using the KnowNow Connectors. The Connectors create a separate thread, a tunnel thread, for handling events coming down the tunnel. The Connectors then deliver the events to an application via a call-back mechanism on this thread. So, if an application were to create a Windows form or control on a main thread, and then tried to update it using the callbacks on the tunnel thread, then the application would throw an exception. For relevant information, see the following Web page:

http://support.microsoft.com/default.aspx?scid=kb;en-us;318604

About the KnowNow ASP.NET Controls

ASP.NET as a framework does not provide the ability to make live updates in real time and without submitting pages. Normally, the only way to get those abilities would be for you to write JavaScript code. To solve this problem, the Live.NET Connector includes a set of KnowNow ASP.NET controls. By taking advantage of the LiveBrowser's components, the KnowNow ASP.NET controls make it possible for you to have the live, dynamic capabilities of a number of the LiveBrowser controls without having to know JavaScript. The KnowNow ASP.NET controls generate the JavaScript code required to create and initialize a particular LiveBrowser control in a Web page. This code runs on the client side and works just as any LiveBrowser control works.

This provides following benefits to ASP.NET control developers,

1. You do not need to have a thorough knowledge of JavaScript. Instead, you can program in your choice of language (C#, VB.NET, *et cetera*).
2. You can use live controls at design time. This helps you to get an initial idea at design time of what the page will look like and how it will behave. There is no need for you to run the page to verify the look and feel of the page every time you make a change.

You can use these components in conjunction with the rest of the Live.NET Connector API. For example, you can create the Message object in any .NET language, fill it with desired filter values, and pass it to an ASP.NET control.

In addition to other capabilities, you can create Web Parts using Web controls and Sharepoint services.

This section provides the following information on these controls:

- "An Overview of the KnowNow ASP.NET Controls" on page 272
- "Cascading Style Sheet Support" on page 275
- "System Configuration for the KnowNow ASP.NET Controls" on page 275
- "KnowNow ASP.NET Deployment" on page 276

An Overview of the KnowNow ASP.NET Controls

The KnowNow ASP.NET controls are:

- KNChatlet
- KNEventViewer
- KNImage
- KNList

- KNMarqueeBand
- KNTable
- KNTree

This section provides an overview of each of these components. For more information, see the following documentation:

- For detailed descriptions of the capabilities of each component, including supported CSS, see the *KnowNow LiveBrowser Developer's Guide*, which ships with the LiveBrowser and LiveServer.
- For programming information and details on the KnowNow ASP.NET controls, see the Live.NET Connector's API documentation. Access to the API documentation is described under "Accessing the Live.NET Connector's API Documentation" on page 268.

KNChatlet

Similar to the WinForm version, the KnowNow ASP.NET version of the KNChatlet component can be placed in any ASP.NET application to create a chatlet. When creating each chatlet, you provide a chat topic name. Chatlets are public chat areas that any number of users can access (as long as they have the correct server access permission). The users of all the KNChatlet controls that use the same topic name can chat with each other using that chatlet.

KNEventViewer

Similar to the WinForm version, the KnowNow ASP.NET version of the KNEventViewer component can be used to display events in table format. Using KNEventViewer, one can view an event's contents and properties. The associated event form makes it possible to add an event to the view, delete an event from the view, or publish an acknowledgement of an event.

The KNEventViewer can be run in two modes: A publisher mode, where the user has access to publishing data and changing the properties of the event viewer, and a subscriber mode, where the user can only view data. The publisher can provide necessary access rights to the subscriber.

KNImage

Similar to the WinForm version, the KnowNow ASP.NET version of the KNImage component can be used to display dynamic images in an ASP.NET Web application. The control can be placed in any ASP.NET form; when doing so, you give a topic name. The topic name is where events with image information are published.

KNList

Any ASP.NET Web application can use the KnowNow KNList component to display a dynamic list of data in a Web page.

Either a combo or a simple list box is visible with list headers, depending on the type of list. All the valid headers of one particular chosen event are visible in the list. Lists are dynamic and are updated as soon as the event gets updated on the LiveServer. Lists support cursors and paging.

KNMarqueeBand

Similar to the WinForm version, the KnowNow ASP.NET version of the KNMarqueeBand component can be used to display a dynamic scrolling marquee of events. The data is added, updated, or deleted as soon it is added, updated, or deleted on the LiveServer. The marquee renders one item per event in the data topic. The entries in the marquee can optionally contain hyperlinks and can optionally display a tool tip. Marquees can scroll horizontally and vertically, up, down, left, and right. You can configure the speed at which the marquee scrolls.

KNTable

The KNTable component can be used to display events in a topic. These events can be simple flat events or KnowNow XML events. There are various ways in which the events can be displayed.

- KNTable can display a static snapshot of the events in the topic at the time the KNTable initializes. In this mode, any updates made to events in the topic are not reflected in the table.
- KNTable can also be dynamic. In this mode, just as with the static tables, the table is initialized with the events in the topic. In addition, you get notifications when events are added, updated, and deleted. The KNTable automatically updates itself with the changes.

For both modes, events can be displayed either all on one page, or in pages of a given number of events. The ability to display the table in pages is particularly useful and efficient when the number of events in the topic is large.

The KNTable component has a number of capabilities, including

- Sortable columns
- Columns that can be reordered
- Cursor and Page capabilities
- Dynamic resizability with automatic slider and scroll bars

- Selectable KnowNow headers for populating column labels and column values
- Event-enabled headers for columns and rows
- Custom properties API
- Event-configurable custom properties
- Publish-from-table capabilities

Data can be added to the table from the top or the bottom as a configurable API option without needing to change the subscription or the table reverting back then next time data is published to it.

KNTree

KNTree is a simple topic tree. Using it, developers can drop in a tree component that allow the dynamic browsing of a topic space via recursive discovery. KNTree supports the LiveServer's cursor capabilities.

Cascading Style Sheet Support

Many of the KnowNow ASP.NET controls support cascading style sheet (CSS) classes. Each such control exposes various properties, each of which is mapped to one CSS class. These properties can be set at design time.

Each property is of type KNStyle. KNStyle is a KnowNow stylesheet class that is inherited from .NET framework's Style class. KNStyle provides various "look and feel" properties, like color, font, border style, and so on.

For information on KNStyle, and on whether a control supports CSS, and on what elements can be controlled using CSS for each control, read the information in the API documentation for that individual control.

System Configuration for the KnowNow ASP.NET Controls

Before using the KnowNow ASP.NET controls, configure your system for cross-domain use as described under "Writing Cross-Domain Web Applications" on page 181.

After creating the cross-domain settings as described starting on page 181, you must also specify the complete URL for an installed LiveBrowser directory in the Web.Config file of your ASP.NET application. When specifying the URL, use the following format:

```
<appSettings>
       <add key="KN_LIVEBROWSER_PATH" value="URL"/>
</appSettings>
```

This information should be added in the <configuration> node of Web.Config. For example, the URL could be

```
http://knserver.example.com/livebrowser
```

For some examples of KnowNow ASP.NET Web application deployments, see "KnowNow ASP.NET Deployment" on page 276.

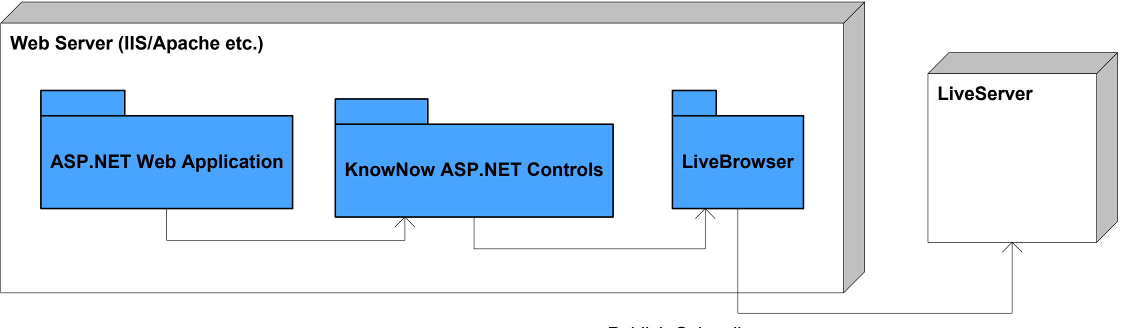
KnowNow ASP.NET Deployment

This section describes a few deployment scenarios in which the KnowNow ASP.NET controls can be used. For information on configuring your system for cross-domain Web applications, see "System Configuration for the KnowNow ASP.NET Controls" on page 275.

Case 1

In this case, the LiveBrowser is deployed as virtual directory in the Web server where the ASP.NET Web application is running. The LiveBrowser interacts with the LiveServer, which can be installed anywhere over the network in same domain. The URL to the LiveBrowser folder in this case is the Web server's URL (http://*WebServer_URL*).

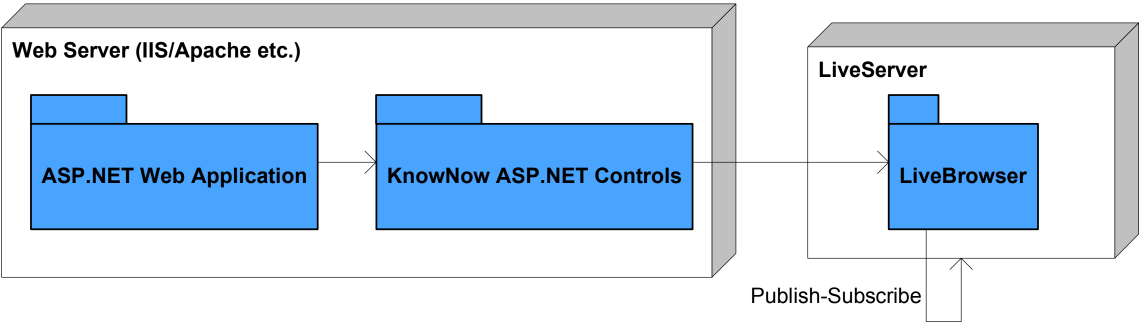
Figure 8-1. Case 1 illustration.



Case 2

In this case, the LiveBrowser is in its default installation with the LiveServer. The ASP.NET Web application is running in a separate Web server. The URL to the Live-Browser folder in this case is simply the LiveServer's URL, which by default is http://*machine_name*:8000.

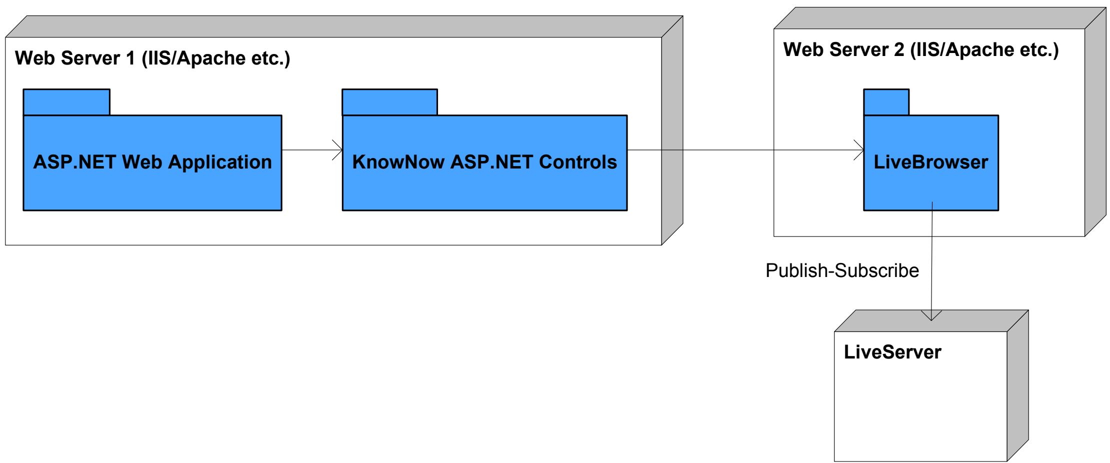
Figure 8-2. Case 2 illustration.



Case 3

In this case, three servers are involved: Web Server 1, where an ASP.NET Web application is running, Web Server 2, where the LiveBrowser is deployed as a virtual directory, and the LiveServer. The URL to the LiveBrowser in this case is the URL of Web Server 2 (http://*WebServer_2_URL*).

Figure 8-3. Case 3 illustration.



Programming Notes

This section provides information specific to creating applications with the Live.NET Connector under the following headings:

- "Bundling the Live.NET Connector with Custom Applications" on page 278
- "Providing Access to the APIs" on page 278
- "Accessing Header Values" on page 279
- "Live.NET Connector Exception Handling" on page 279
- "Installing Web Parts that Use the Live.NET Connector" on page 279

Bundling the Live.NET Connector with Custom Applications

If you are building custom applications that use the Live.NET Connector, the following information may be useful, especially if you wish to bundle the Connector with your application and you don't want to install the Connector separately. You will also need the information provided under "Live.NET Connector Requirements" on page 216.

If you are building custom applications, the Live.NET Connector requires a clean installation of the .NET Framework SDK or Microsoft Studio .NET.

The name of the Live.NET Connector's DLL is LibKNDotNet.dll. After installation, the default location of this DLL is the /vs71/ folder (for .NET Framework 1.1) or the /vs80/ directory (for .NET Framewrok 2.0) in the *KnowNow_Install_root*\Live.NET Connector\bin\ directory.

The Live.NET Connector depends on the MSCOREE.dll (the Microsoft .NET Framework library) and on the logging DLL (Log4Net.dll). If these DLLs are not present, the Connector will not work properly. If your application needs these DLLs in a single directory (such as a \bin\ directory), you can simply copy them into that directory and deliver your custom application with these DLLs in that location. With such an arrangement, the Live.NET Connector will only be available for that application, but it will work just as well as ever.

Providing Access to the APIs

If you are planning to use the Live.NET Connector and are going to write a .NET application, you will need to have access to the KnowNow .NET APIs. To access those APIs, you just need to add a reference to LibKNDotNet.dll. You can add a reference for each class in the Solution Explorer by right-clicking on **References** for that class. Or you can add a reference by choosing the **Project** menu and choosing **Add Reference**.

If you wish to browse the Live.NET Connector API documentation for information needed while creating your application, see the information under "Accessing the Live.NET Connector's API Documentation" on page 268.

Accessing Header Values

To enumerate the various headers in a Message class, you will need to use the IEnumerator interface defined in the .NET framework. Classes derived from IEnumerator must support the three methods Current, Reset, and MoveNext. (For further information regarding the IEnumerator interface, see the .NET documentation.) Each element in the collection is a structure of type MessageEntry that has two data fields Field and Value. Here is a code snippet that extracts field and value pairs from a Message object and prints them.

Example 8-1. Extracting field/value pairs from a Message.

```
private void OnUpdate(KnowNow.LiveDotNet.Message msg)
{
        System.Collections.IEnumerator ienum = msg.GetEnumerator();
        while(ienum.MoveNext())
        {
            KnowNow.LiveDotNet.MessageEntry me =
(KnowNow.LiveDotNet.MessageEntry)ienum.Current;
            string field = me.Field;
            string val = me.Value;
            Console.WriteLine("Field: " + field + ", Value: " + val + "\n");
        }
        }
```

One of the included Live.NET Connector samples uses a MessageEntry structure; see that sample for more ideas and information.

Live.NET Connector Exception Handling

The Live.NET Connector throws one exception: "InvalidOperationException." This exception could get thrown when a call to the get_Current method of the IEnumerator object is invoked.

Installing Web Parts that Use the Live.NET Connector

ASP.NET Web Parts are controls that make it possible for users to personalize the content of a Web page, including the ability to add new Web Parts to the page.

You can install a Web Part that uses the Live.NET Connector as described here.

To install a Web Part,

1. Use the STSADM.exe command-line tool to install the packaged Web Part (e.g., YourWebPartPkg.CAB). The command would look something like this:

   ```
   STSADM.exe  -o  addwppack  -filename YourWebPartPkg.CAB -url
   http://YourVirtualServerName
   ```

   The command must succeed in order to install your Web Part.

2. Create a new security policy file or modify an existing policy file to add a few customization statements. The simplest way might be to make a copy of the existing file wss_minimaltrust.config. The first set of statements to add are given in Example 8-2 on page 281. Add these statements in the beginning, right after the tag `<CodeGroup class=”FirstMatchCodeGroup” version=”1” PermissionSetName=”Nothing”>`. After making this change, the section in the configuration file would look similar to Example 8-3 on page 281.

3. In the custom security file, search for the following statements :

   ```
   <SecurityClass Name="ZoneMembershipCondition"
   Description="System.Security.Policy.ZoneMembershipCondition, mscorlib,
   Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
   ```

4. Add the following two statements after the above statement:

   ```
   <SecurityClass Name="SharePointPermission"
   Description="Microsoft.SharePoint.Security.SharePointPermission,
   Microsoft.SharePoint.Security, Version=11.0.0.0, Culture=neutral,
   PublicKeyToken=71e9bce111e9429c"/>

   <SecurityClass Name="FileIOPermission"
   Description="System.Security.Permissions.FileIOPermission, mscorlib,
   Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
   ```

5. In the web.config file, define a new policy definition (a new entry for WSS_MinimalTrust_*YourWebPartName*.config) as shown here:

   ```
   <securityPolicy>
       <trustLevel name="WSS_Medium" policyFile="D:\Program Files\Common
   Files\Microsoft Shared\Web Server Extensions\60\config\wss_mediumtrust.config"
   />
       <trustLevel name="WSS_Minimal" policyFile="D:\Program Files\Common
   Files\Microsoft Shared\Web Server
   Extensions\60\config\wss_minimaltrust.config" />
       <trustLevel name="WSS_Minimal_YourWebPartName" policyFile="D:\Program
   Files\Common Files\Microsoft Shared\Web Server
   Extensions\60\config\WSS_MinimalTrust_YourWebPartName.config" />
   ```

```
</securityPolicy>
```

6. Activate the new security policy by setting the trust level to the new policy using one of the two options as follows.

   For a site running under ASP.NET 1.1:

```
<trust level=" WSS_Minimal_YourWebPartName " originUrl="" />
```

   For a site running under ASP.NET 2.0:

```
<trust level="WSS_Minimal_YourWebPartName" originUrl=""
processRequestInApplicationTrust="false" />
```

7. Restart the WSS service using the **IISReset** command.

Example 8-2. Sample code 1 for the security policy file.

```
<CodeGroup
       class="UnionCodeGroup"
       version="1"
       PermissionSetName="FullTrust">
       <IMembershipCondition
           class="StrongNameMembershipCondition"
       version="#.#.#.#"     <The version of your WebPart Assembly>
           PublicKeyBlob="XXXXXXX" <--Your WebPart Assembly's PublicKeyBlob-'
       Name="YourWebPartName"
       />
</CodeGroup>
<CodeGroup
       class="UnionCodeGroup"
       version="1"
       PermissionSetName="FullTrust">
       <IMembershipCondition
           class="StrongNameMembershipCondition"
           version="3.2.2.13"
PublicKeyBlob="0024000004800000940000000602000000240000525341310004000001000100635B7410EF8380967FB0F452BF78AF35B5D6FFB738EB4D069F2A51738D75681E9B83B09241D25F48513
B4DBB44E1BFC27C3B8E4B0DF05BB7B5C21097099A1801FF33693B2EE90CB87A535EC5749A407E18BC
C132C38D1D0DD44FD2A5EB4510F6FC989F65CB07B6C9CEC86E27AC375EC118BAF427FBD03A23305DA
662B91F26B1"
           Name="LibKNDotNet"
        />
</CodeGroup>
```

Example 8-3. Security policy file after adding code from Example 8-2.

```
...    ... ... ... ... ...
...    ... ... ... ... ...
```

```
...     ... ... ... ... ...

        </PermissionSet>
</NamedPermissionSets>

<CodeGroup
class="FirstMatchCodeGroup"
        version="1"
        PermissionSetName="Nothing">
        <IMembershipCondition
            class="AllMembershipCondition"
                       version="1"
        />
        <CodeGroup
            class="UnionCodeGroup"
            version="1"
            PermissionSetName="FullTrust">
            <IMembershipCondition
                class="StrongNameMembershipCondition"
                version="<-------Your WebPart's Version---------'"
                PublicKeyBlob="<-----Your WebPart Assembly's PublicKeyBlob-----'"
                Name="YourWebPartName"
            />
        </CodeGroup>
        <CodeGroup
            class="UnionCodeGroup"
            version="1"
            PermissionSetName="FullTrust">
            <IMembershipCondition
                class="StrongNameMembershipCondition"
                version="3.2.2.13"
PublicKeyBlob="0024000004800000940000000602000000240000525341310004000001000100063
5B7410EF8380967FB0F452BF78AF35B5D6FFB738EB4D069F2A51738D75681E9B83B09241D25F48513
B4DBB44E1BFC27C3B8E4B0DF05BB7B5C21097099A1801FF33693B2EE90CB87A535EC5749A407E18BC
C132C38D1D0DD44FD2A5EB4510F6FC989F65CB07B6C9CEC86E27AC375EC118BAF427FBD03A23305DA
662B91F26B1"
                Name="LibKNDotNet"
            />
        </CodeGroup>
        <CodeGroup
            class="UnionCodeGroup"
            version="1"
            PermissionSetName="ASP.Net">
            <IMembershipCondition
                class="UrlMembershipCondition"
                version="1"
                Url="$AppDirUrl$/*"
            />
        </CodeGroup>
```

...	...	...	...	...	...
...	...	...	...	...	...
...	...	...	...	...	...

Chapter 9 Using the LiveJava Connector

All Connectors manage communications between the LiveServer and your application. While you do not need to use a Connector, a Connector can make your life much easier by simplifying the calls and operations your application needs to perform. Therefore, when creating or customizing your Java application to perform publish and subscribe operations with the LiveServer, you may want to have your application use the LiveJava Connector instead of dealing directly with the LiveServer. This chapter describes how the LiveJava Connector works and provides guidelines on using some of the most important calls in the API to create or customize your application so it can make best use of the LiveJava Connector. Be sure to also read Chapter 2 , "Events," for additional important information on the LiveServer's operations and architecture, as well as Chapter 5 , "Common Connector Capabilities," for information on Connector capabilities. This chapter describes the LiveJava Connector under the following headings:

- "System Requirements" on page 287
- "Installing the LiveJava Connector" on page 288
- "Using the LiveJava Connector" on page 296
- "Publishing Events" on page 301
- "Handling Status Events" on page 302
- "Subscribing" on page 305
- "Unsubscribing" on page 309
- "Heartbeat" on page 310
- "Logging" on page 312
- "Heartbeat" on page 310
- "Request/response" on page 315

- "Journal Connection Callback" on page 316
- "Journal Connection Callback" on page 316
- "Simple Event Mapping Support" on page 318
- "Using the Java Messaging Service" on page 323
- "A Guide to the Java API" on page 329

System Requirements

The LiveJava Connector is supported on all platforms that support Java.

To use the LiveJava Connector, you must have the following software installed:

- JDK 1.3.0 or later

If you wish to use Java modules with the LiveServer, you must also have the LiveServer's Java module (knjava) loaded into the LiveServer as described in the chapter on configuring the LiveServer in the *KnowNow LiveServer Administration Guide*.

Installing the LiveJava Connector

The LiveJava Connector can be installed on Windows, UNIX, Linux, and Mac OSX platforms. It is available for download from

http://support.knownow.com

The installation file contains release notes, documentation files, examples, jar files, and other files needed for the LiveJava Connector.

- The release notes and the documentation files are under the /*Connector_root*/docs/ directory and include JavaDoc-generated HTML files that document the LiveJava Connector API classes and methods. To access the API documentation, open /*Connector_root*/docs/javadoc/index.html.
- The example files, which illustrate common ways of using the Connector API, are in various subdirectories under the /*Connector_root*/examples/ directory.
- The jar files are under /*Connector_root*/lib/. After installing the LiveJava Connector, you will need to set the CLASSPATH environment variable for the needed files as described in the installation instructions ("Installing on Windows" on page 290 and "Installing on UNIX" on page 295). The jar files are discussed under jar Files.

jar Files

The LiveJava Connector is shipped with several jar files. Some are essential and you must set your CLASSPATH to include them always. Others are needed to support various APIs or other functions; set your CLASSPATH to include them if your situation calls for it./

The core LiveJava Connector jar files are microserver.jar and knjms.jar (which depends on jms.jar). These files implement two LiveJava Connectors.

- The microserver.jar file is the core and most commonly used LiveJava Connector; it contains the KnowNow LiveJava Connector API.
- The knjms.jar file is an alternative LiveJava Connector; it contains KnowNow's API implementation of the Java Messaging Service (JMS) for performing publication and subscription operations. KnowNow's implementation of JMS provides some functionality that the Sun JMS specification does not support, such as request/response.
- The log4j-1.2.8.jar supports the LiveJava Connector.

In order to use one of the LiveJava Connectors in UNIX, you will need to set the CLASSPATH environment variable to include either the microserver.jar file or the knjms.jar and jms.jar files. (Under Windows, the CLASSPATH is automatically set correctly.) You can use both LiveJava Connectors if you like, but you must use at least one of the LiveJava Connectors to support Java applications. You also need to set the CLASSPATH to include other jar files as needed according to Table 9-1.

Table 9-1. LiveJava Connector jar files.

jar File Name	Function	Set CLASSPATH to Include this File
jcert.jar	Supports SSL.	When using SSL.
jms.jar	Supports JMS API.	When using the JMS API.
jnet.jar	Supports the LiveJava Connector; used by KNJServer.	Always.
jsse.jar	Supports the LiveJava Connector; used by KNJServer.	Always.
knjms.jar	Implements one version of the LiveJava Connector. Supports Java Messaging Service (JMS) API, but provides some functionality that the Sun JMS specification does not support, such as request/response. Also, depends on jms.jar.	When using the JMS API.
log4j-1.2.8.jar	log4j logging library. Supports the LiveJava Connector's logging capabilities.	Always.
microserver.jar	Implements another version of the LiveJava Connector. Supports the LiveJava Connector API.	When using the LiveJava Connector API.

Installing on Windows

The LiveJava Connector has an executable installation program. The following pages provide step-by-step instructions for this program. However, if you wish, you can run this program (livejava-*version*.exe) and follow the prompts. In any case, be sure to set the CLASSPATH environment variable as described in step 10 on page 295.

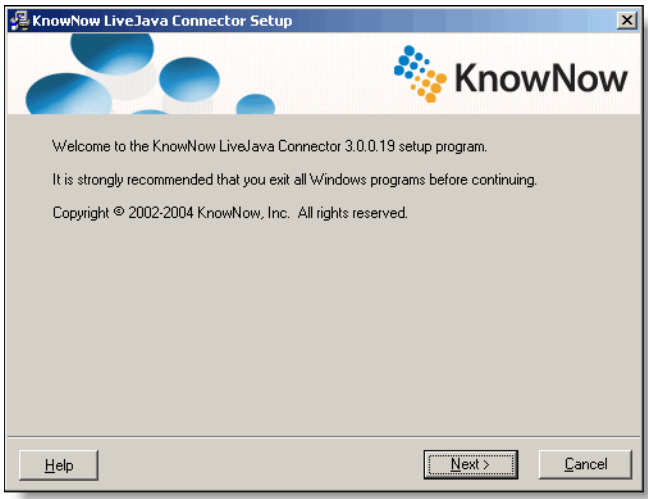
To install the LiveJava Connector on Windows,

1. Run the installation program. The installation program checks to see if you have a previous installation of the Connector. If it finds one, it also compares the versions of the two installations. If you do not have a previous or newer installation, the Welcome window appears as shown in step 2. Otherwise, one of two windows will appear, depending on whether you have a Connector that is older than the one you are installing, or newer.

- If you have an older installation, the Prior Installation Detected window appears. You can upgrade the older installation by choosing **Next**.
- If you have a newer installation, the Existing Installation window appears. If this happens, it is recommended that you use the newer version. You can choose to install the older version, though in that case you will need to exit the installation program and uninstall the newer version before you can run the installation program again to install the older version. Whatever your preference, if you have arrived at this window, choose **Finish**. This ends the installation program so you can perform your next desired step.

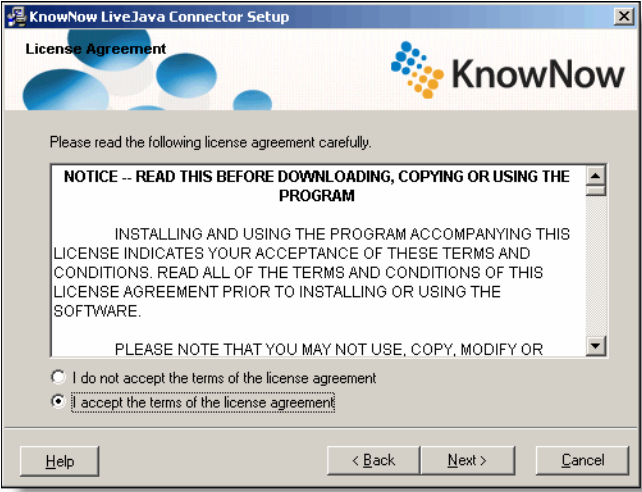
2. The next window is a welcome message. Choose **Next**.

Figure 9-1. Welcome dialog box.



3. The License Agreement window is displayed next. Choose **I accept the terms of the license agreement**, then choose **Next.** (**Next** is disabled until you accept the terms.)

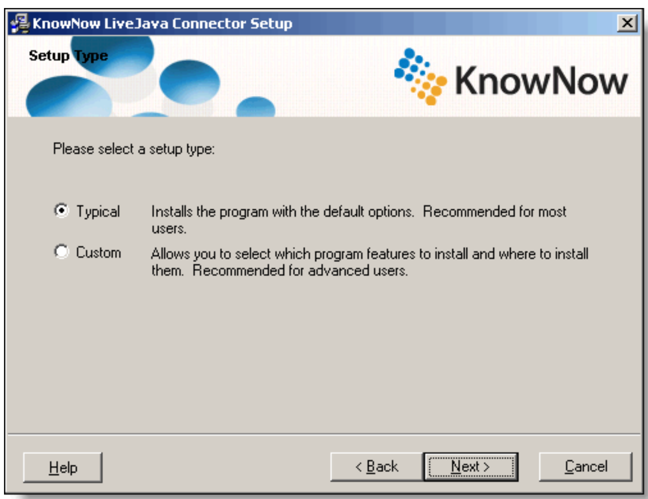
Figure 9-2. License agreement.



4. Next, the Setup Type window appears, asking whether you want a **Typical** or **Custom** installation.

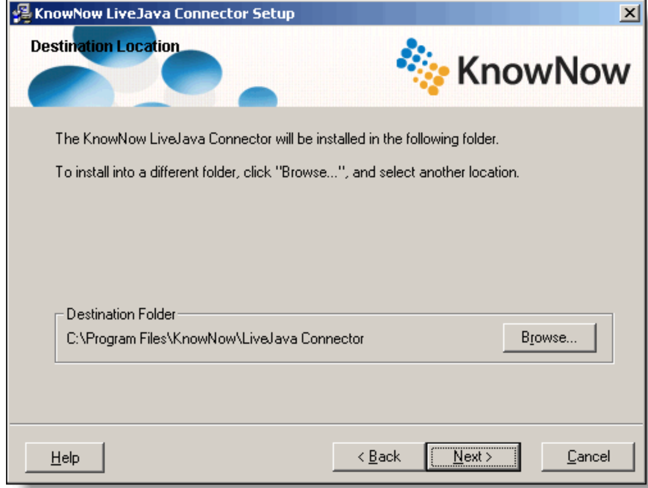
- If you wish to change the default installation directory or make other changes to the default installation, select **Custom**, then choose **Next** and proceed with step 5 on page 292.
- If you select **Typical**, choose **Next**, then skip to step 7 on page 293.

Figure 9-3. Choose the Connector setup type.



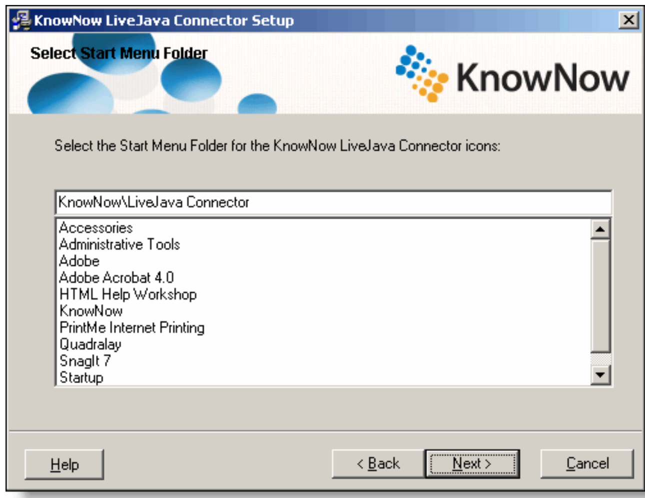
5. Choose the destination folder for the Connector, or accept the default. The default location is C:\Program Files\KnowNow\LiveJava. Choose **Browse** to choose a location other than the default.

Figure 9-4. Choose a destination.



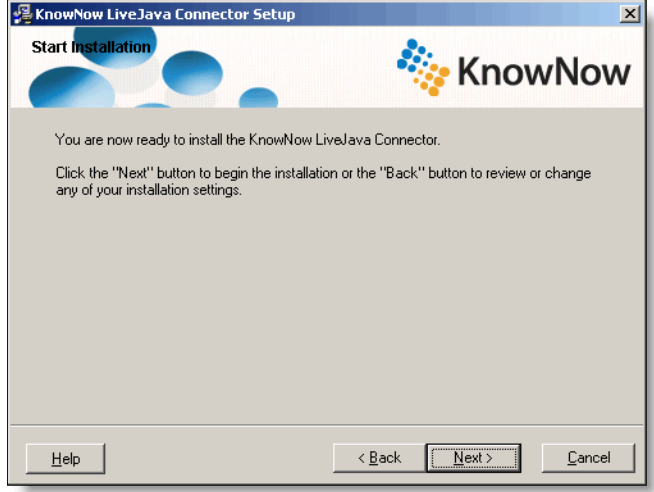
6. After specifying a destination. choose **Next**, then select a Start Menu folder for the Connector.

Figure 9-5. Select a Start Menu folder.



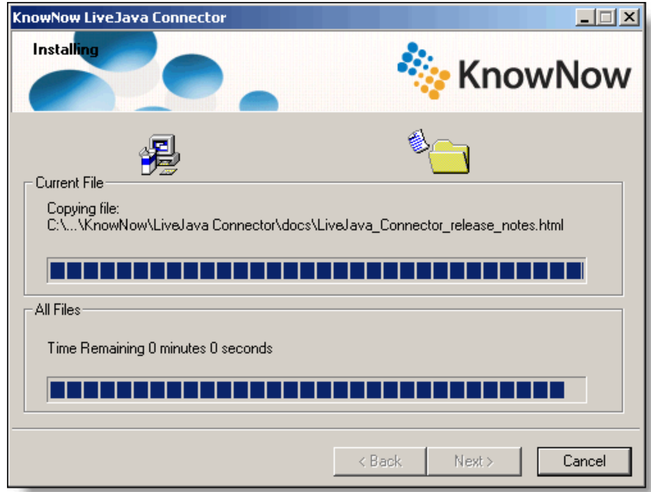
7. Choose **Next**. The installation program is now ready to start.

Figure 9-6. Ready to roll.



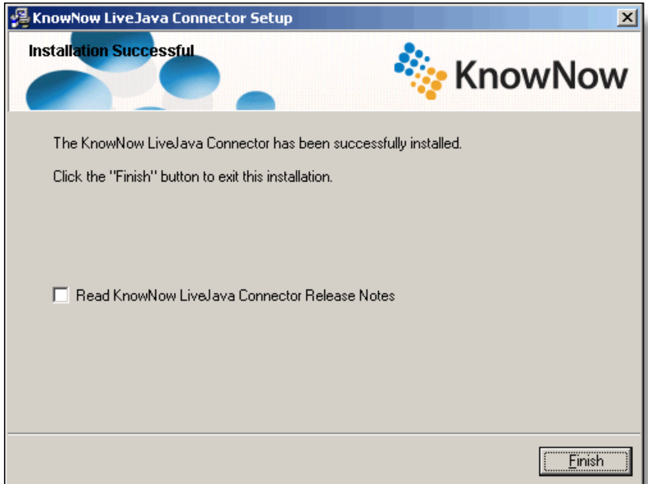
8. Choose **Next**. The progress window reports the installation's progress. If there was a prior installation, you may briefly see a pop-up window stating that the prior installation is being removed before the installation process continues.

Figure 9-7. Installing.



9. When the installation is complete, the success window appears. You can choose to view the release notes if you wish by enabling the **Read KnowNow LiveJava Connector Release Notes** check box. Choose **Finish**.

Figure 9-8. Success!



10. The installer automatically sets the CLASSPATH environment variable to the appropriate values to include all the jar files. For a list of the LiveJava Connector's jar files, see "jar Files" on page 288

Installing on UNIX

This section describes how to install the UNIX LiveJava Connector and set the CLASSPATH environment variable for the needed jar files.

To install the LiveJava Connector on UNIX,

1. Retaining the directory structure, untar the livejava-*version*.tgz file to a desired location, such as /usr/local/kn. A /livejava-*version/* directory will be created with the LiveJava Connector files underneath.

   Here is a sample set of installation commands.

   ```
   cd /usr/local/kn
   gzip -dc livejava-version.tgz | tar xvf -
   ```

2. Set the CLASSPATH environment variable to the appropriate values to include the needed jar files. For a list of the LiveJava Connector's jar files, see "jar Files" on page 288. The following example sets the CLASSPATH environment variable to include all the jar files. In this example, KNJAVA_HOME is not required; it is just used to simplify the CLASSPATH setting.

   ```
   export KNJAVA_HOME=/usr/local/kn/livejava-version
   export CLASSPATH=$KNJAVA_HOME/lib/microserver.jar: \
       $KNJAVA_HOME/lib/knjms.jar: \
       $KNJAVA_HOME/lib/jms.jar: \
       $KNJAVA_HOME/lib/jnet.jar: \
       $KNJAVA_HOME/lib/jsse.jar: \
       $KNJAVA_HOME/lib/log4j-1.2.8.jar: \
       $KNJAVA_HOME/lib/jcert.jar:$CLASSPATH
   ```

Using the LiveJava Connector

The LiveJava Connector performs publish, subscribe, and unsubscribe operations and manages other communications between your Java application and the LiveServer. Most of the functionality of this Connector—publish, subscribe, add requests, create journals, and so on—is contained in the KNJServer class.

This section describes how to use the basic features of the LiveJava Connector API to communicate with the LiveJava Connector under the following headings. For information on publishing, subscribing, and other functions, see the remainder of this chapter.

- "Communicating with the Connector" on page 296
- "System Properties" on page 297
- "Connecting to a LiveServer through a Proxy" on page 298
- "Connecting to a LiveServer Using SSL" on page 299

Communicating with the Connector

Your Java application needs to communicate with the LiveJava Connector. Example 9-1 shows how a Java client program could create a connection to the LiveJava Connector by creating an instance of the KNJServer class, specifying the URL of the desired LiveServer.

Example 9-1. Creating a KNJServer instance.

```
import com.knownow.common.*;       /* KnowNow common stuff */
import com.knownow.microserver.*;  /* KnowNow pub/sub API */

/** serverURL specifies the path to the KnowNow LiveServer */
String  serverURL = "http://yourliveserver.example.com:8000/kn";

KNJServer aServer = new KNJServer(serverURL);
```

Your Java application can then use the KNJServer instance to publish events to topics and subscribe to events from topics hosted on the associated LiveServer.

When the LiveJava Connector is configured to require that Java client programs be authenticated, your Java client program would use the KNJServer constructor that accepts a user name, password, and realm, as shown in Example 9-2.

Example 9-2. Creating a KNJServer instance with authentication parameters.

```
import com.knownow.common.*;       /* KnowNow common stuff */
import com.knownow.microserver.*;  /* KnowNow pub/sub API */
```

```
/** serverURL specifies the path to the KnowNow LiveServer */
String serverURL = "http://yourliveserver.example.com:8000/kn";
String username = new String("test");
String passwd = new String("test");
/* authorization realm */
String realm = new String("LiveServer.example.com-auth");

KNJServer aServer = new KNJServer(serverURL, username, passwd, realm);
```

The authorization realm refers to the HTTP basic authorization. If you do not know the authorization realm for your LiveServer, you can use the utility com.knownow.util.GetAuthInfo as follows:

```
java com.knownow.util.GetAuthInfo http://LiveServer.example.com:8000/kn
```

System Properties

The LiveJava Connector has the following system properties. These can be set in the system.properties file, or they can be passed as parameters to start a specific application.

- KN.log.level, which enables and disables logging
- KN.log.file, which specifies the log file
- KN.forceHTTP_1.0, which controls which version of HTTP the LiveJava Connector is to use
- KN.disableKeepAlives, which disables HTTP keepalives

KN.log.level

This property controls logging. Set to **true** to enable logging. For example,

```
java -DKN.log.level=true classname
```

KN.log.file

This property specifies the name of the log file. By default, if you enable logging (using KN.log.level) but don't use KN.log.file, a log file with the name of microserver.log will be created and used. If you set this property, but KN.log.level is not set to **true**, this property will be ignored. You can specify a path as part of the file name.

Here is an example of setting this property on the command line (this example sets logging to **true** at the same time).

```
java -DKN.log.file=true -DKN.log.file=myserverlog.log classname
```

KN.forceHTTP_1.0

By default, the LiveJava Connector uses HTTP 1.1. There are times when you may wish to use HTTP 1.0 only; for example, if you are working with a proxy that doesn't accept the HTTP 1.1 protocol. To turn off the connection keepalive and cause the LiveJava Connector to use the HTTP 1.0 features only, set this property to **true**. For example,

```
java -DKN.forceHTTP_1.0=true classname
```

KN.disableKeepAlives

This property disables HTTP 1.0 keepalives. It takes as values **true** and **false**.

Setting Multiple System Properties

You can set multiple system properties with a single command, like so:

```
java -DKNJServer.log=true -DKNJServer.log.file=mylog.log classname
```

Connecting to a LiveServer through a Proxy

To create an instance of the LiveJava Connector that enables a Java application to publish events to topics and subscribe to events from topics hosted on a LiveServer through a proxy server, your client application should use the setProxyServer method after creating a KNJServer instance. If the proxy requires authorization, your client application will need to use the setProxyAuthorization to specify the user name and password, as shown in Example 9-3.

Example 9-3. Specifying KNJServer proxy server parameters.

```
import com.knownow.common.*;      /* KnowNow common stuff */
import com.knownow.microserver.*; /* KnowNow pub/sub API */

/** serverURL specifies the path to the LiveServer */
String serverURL = "http://LiveServer.example.com:8000/kn";

KNJServer aServer = new KNJServer(serverURL);
/** specify you are connecting to the LiveServer through a proxy */
aServer.setProxyServer("proxy.example.com", 8080);

/** if your proxy requires authorization, specify that */
aServer.setProxyAuthorization("proxy.example.com-auth",
           "proxyUser", "proxyPasswd");
```

Connecting to a LiveServer Using SSL

The Connector uses JSSE1.0.2 to provide secure sockets layer (HTTPS) support. If you are using a self-signed certificate, you will need to import the certificate to the appropriate keystore using the Java keytool utility found in the $JAVA_HOME/jre/lib/security/ directory.

You will also need to have the CLASSPATH set to use the jcert.jar file. For more information on the jar files, see "jar Files" on page 288. For instructions on setting the CLASSPATH environment variable, see either step 10 on page 295 (for Windows) or step 2 on page 295 (for UNIX).

The following command imports the certificate from the file mycert.cer and stores it into the keystore jssecacerts:

```
keytool -import -file mycert.cer -keystore jssecacerts
```

Once you have imported the necessary certificates, you can connect to the LiveServer using the same procedure described in Example 9-1 on page 296.

Note: The name specified in the serverURL statement **must** match the LiveServer name specified in the **CN** field of the certificate, or the HTTPS connection will fail.

When connecting to a secure LiveServer, your Java client application must define the property java.protocol.handler.pkgs. You can do this within your Java client application, as follows:

```
System.getProperties().put("java.protocol.handler.pkgs",
"com.sun.net.ssl.internal.www.protocol");
```

You can also define this property on the command line when you start your Java client application:

```
java -Djava.protocol.handler.pkgs=com.sun.net.ssl.internal.www.protocol \
    YourJavaClient
```

Testing SSL

Developers will usually want to test out their Java applications with the LiveServer that uses a test certificate. This enables the LiveServer to implement HTTPS. A test certificate can be obtained from authorities like http://www.thawte.com.

A browser like Internet Explorer presents a user with a dialog box informing the user that the certificate is not trusted, allowing them an option to proceed. Java does not present such a dialog box but instead exits with an error. The following sample solution assumes that the test certificate was obtained from Thawte and the user is running Java version 1.4.2.

Follow the instructions under "Connecting to a LiveServer Using SSL" on page 299 before proceeding.

Java's keytool is used to load the LiveServer's certificate as a trusted certificate:

```
keytool -import -alias thawtecert -file "C:\Program Files\KnowNow
    \Server2023\conf\cert.pem.txt" -keystore jssecacerts
```

In addition, since the test certificate's certificate authority is Thawte, Thawte's test certificate root needs to be added to the keystore to ensure that the LiveServer's test certificate can be verified. The root certificate can be retrieved from http://www.thawte.com/ssl-digital-certificates/free-trial/. (Note that Web sites can change, so if this link is no longer valid, access http://www.thawte.com and browse from there.)

Issue the following command once you have saved the root certificate in thawtetestcert.txt:

```
keytool -import -trustcacerts -alias thawtetestcertCA -file thawtetestcert.txt
    -keystore jssecacerts
```

Run your Java program while ensuring that the trust store used for trusted certificates is the jssecacerts store in the local directory:

```
java -Djavax.net.debug=ssl  -Djavax.net.ssl.trustStore=jssecacerts
    RMDGateway https://rajeev-laptop.knownow.com/kn
```

The **-Djavax.net.debug=ssl** parameter is shown here so that you may see the SSL handshake between the client program and the LiveServer.

Publishing Events

Your client application can use the KNJServer.publish method to publish an event to a topic on a LiveServer. The publish function takes two arguments, the topic and the event, and has no return value. The LiveJava Connector keeps one session alive as it publishes multiple events.

The Java code fragment in Example 9-4 creates an event and uses the KNEvent.set method to populate the event with kn_payload, displayname, and user_id headers. It then publishes the event to the topic /what/chat. (For a list of the kn_ headers, see "kn_ Header Reference" on page 62.)

Example 9-4. Creating and publishing an event.

```
import com.knownow.common.*;       /* KnowNow common stuff */
import com.knownow.microserver.*;  /* KnowNow pub/sub API */

/** topic to publish the event */
String pubTopic = "/what/chat";

/** create the event */
KNEvent anEvent = new KNEvent();

/** add some event data to it */
anEvent.set("kn_payload", "This is the event payload");
anEvent.set("displayname", "cartman");
anEvent.set("user_id", "cartman");

/** finally publish it */
aServer.publish(pubTopic, anEvent, null);
```

The LiveServer receives an event resembling the following:

```
kn_time_t: 987107696
user_id: cartman
kn_id: 26541550
displayname: cartman
kn_payload: This is the event payload
```

Note: If the kn_id is not specified as part of the event data, the LiveServer automatically generates a unique kn_id and adds it to the event.

Handling Status Events

When the LiveServer processes a request made by your client application, it often generates status events that describe the outcome of the request. These status events are normally written in response to an inbound HTTP request to the LiveServer.

Your application can implement a KNStatusHandler to handle status events or redirect the events to any URI, including other topics on the LiveServer. When deriving a class from this interface, you provide an implementation of the onSuccess and onError methods, which process the status event in the manner you choose.

The KNJournalStatusHandler interface is used to receive status events about journal connections.

Example 9-5 publishes an event to a topic and uses a KNStatusHandler to handle status events generated by the LiveServer.

Example 9-5. Handling status events.

```
import com.knownow.common.*;
import com.knownow.microserver.*;

public class StatusHandlerSample implements KNStatusHandler {

public void onStatusEvent(KNEvent anEvent) {

        String status = anEvent.get("status");
        int statusCode = anEvent.getHttpStatus();

        if (statusCode >= 200 && statusCode < 300) {
            System.out.println("PASSED\n" + status);
            }
        else {
            System.out.println("FAILED\n" + status);
            }
        }
}
```

Sample Status Event

Example 9-6 illustrates a status event generated by the LiveServer.

Example 9-6. Sample status event.

```
kn_id: e557ac76-a31b-11d5-080020fee0cf
kn_time_t: 998017657
```

```
status: 200 Notified
```

Exceptions

The following classes supports native exception handling. These classes make it possible for you to pass errors around and to roll back your stack (very much like a try/catch block) in a very clean way. See the API documentation for each of these classes for their error codes and other details.

- KNException
- CursorException
- PresenceException
- ReqRespException

Subscribing

The KNJServer class provides a number of subscribe methods. With one of them, you can subscribe a topic to a callback object for handling events received on that topic, as described under "Subscribing a Topic to a Callback" on page 305. With another, you can subscribe to a topic and also support all since last connect (ASLC) capabilities, as described under "All Since Last Connect (ASLC)" on page 306. With the third method, you can subscribe a subtopic to a source topic, effectively routing all events from the source to the subtopic, as described under "Subscribing One Topic to Another Topic" on page 307.

All subscribe methods allow you to specify options for controlling the number and age of events to be received. For an example of setting subscription options, see "Subscription Options" on page 307.

Subscribing a Topic to a Callback

Your Java client application can use the KNJServer.subscribe method to assign a KNEventListener to be invoked to handle events received from a specific topic.

This KNJServer.subscribe method takes three arguments: a topic, an instance of a KNEventListener that will function as a callback, and options for the subscription. Options can be used to specify the maximum number of events that can be received or the maximum event age allowed. The options can be null if no options are to be specified. For complete details on the options allowed, see "Subscription Options" on page 307.

This KNJServer.subscribe method returns the route created by the subscription as a KNRoute. Your application can store the route for future reference, since it is required for your application to unsubscribe from the topic.

The Java code fragment in Example 9-7 implements a KNEventListener class, creates an instance of the listener, and then subscribes the event received on the topic /what/chat to the listener. The maximum event age allowed is five minutes earlier than the time the subscription was opened.

Example 9-7. Subscribing a topic to a callback.

```
import com.knownow.common.*;          /*KnowNow common stuff */
import com.knownow.microserver.*;    /*KnowNow pub/sub API */

public class MyTest implements KNEventListener
{
 public void onEvent(KNEvent anEvent)
 {
   System.out.println(anEvent);
 }
```

```
public MyTest() {};

        public static void main(String[] args)
        {
            String serverURL = "http://LiveServer.knownow.com:8000/kn";
            String subtopic = "/what/chat";
            KNJServer aServer = new KNJServer(serverURL);
            KNOptions userData = new KNOptions();

            userData.set("displayname",  "cartman");
            userData.set("user_id", "cartman");
            userData.set("do_max_age", "300");

            MyTest aTest =  new MyTest();
            KNRoute route2Callback =
                aServer.subscribe(subtopic, aTest, userData, null);
            if (route2Callback.getHttpStatus() >= 300)
            {
                System.err.print(
                "Failed to create route to callback, HTTP status ");
                System.err.println(route2Callback.getHttpStatus());
            }
        }
}
```

All Since Last Connect (ASLC)

The subscribeASLC command adds some capabilities to the subscribe command. When you set up a subscription using this command, you could specify a filter that will enable you to replay all messages sitting in a topic in one of two ways:

- based on the time that the LiveServer receives the event (using kn_time_t)
- based on the event ID (using kn_event_id)

This makes it possible to, for example, receive all events that had been posted since the last connection.

For example, suppose you are subscribing to a topic and are receiving events from that topic, and there is a network failure between your application and the LiveServer. If another application is still connected and is still publishing to that topic, of course your application will not see those events. If you are using subscribeASLC, you can retrieve all unexpired events from the point of failure in one of two ways listed above.

Subscribing One Topic to Another Topic

Your Java client application can use the KNJServer.subscribe method to subscribe one topic to another topic, creating a route from one topic to another. You don't need to define a listener with this method, since events posted to the source topic are simply routed to the subtopic.

This KNJServer.subscribe method, which takes a subtopic, a source topic, and options for the subscription, returns the route created by the subscription as a KNRoute. Your application should store the route for future reference, since it is required if your application needs to unsubscribe from the topic.

The Java code fragment in Example 9-8 subscribes the source topic /what/quotes to the destination topic /what/chat with no event options specified.

Example 9-8. Subscribing a subtopic to a topic.

```
import com.knownow.common.*;              /*KnowNow common stuff */
import com.knownow.microserver.*;         /*KnowNow pub/sub API */

/** create an instance of the LiveJava Connector */
KNJServer aServer = new KNJServer(
       "http://LiveServer.example.com:8000/kn");

/** topic to subscribe */
String  subtopic = "/what/quotes";

/** topic that subscribes to the above topic */
String  destinationTopic = "/what/chat";

KNRoute  route2Topic = aServer.subscribe( subtopic, destinationTopic, null, null);

if (route2Topic.getHttpStatus() >= 300)
{
       System.err.print(
           "Failed to create route to topic, HTTP status ");
       System.err.println(route2Topic.getHttpStatus());
}
```

Subscription Options

The subscribe methods provide the ability to pass optional data to the LiveServer using the KNOptions class. In the code fragment in Example 9-9, a maximum of five events will be received by the subscription.

Example 9-9. Setting subscription options.

```
/** create KNOptions */
KNOptions userData = new KNOptions();

userData.set("display_name", "cartman");

userData.set("user_id", "cartman");

userData.setDoMaxN(5);

KNRoute route2Callback = aServer.subscribe(subTopic, myTest, userData, null);
```

Unsubscribing

Your client application uses the KNJServer.unsubscribe function is used to unsubscribe a route that was previously subscribed to a topic and halt further delivery of events published to that topic.

This method requires that you specify the route to be unsubscribed. The route is the same one returned by one of the two KNJServer.subscribe methods that was used to established the subscription.

The unsubscribe function does not return any values.

The Java code fragment in Example 9-10 unsubscribes from a topic, which stops further delivery of events that are published to that topic.

Example 9-10. Unsubscribing from a topic.

```
import com.knownow.common.*;                /*KnowNow common stuff */
import com.knownow.microserver.*;           /*KnowNow pub/sub API */

/** create an instance of the LiveJava Connector */
KNJServer aServer = new KNJServer(
           "http://LiveServer.example.com:8000/kn");
KNRoute aRoute;

/** subscribe to a topic */
aRoute = aServer.subscribe(...);
...
...
/** unsubscribe from topic that we previously subscribed to*/
aServer.unsubscribe(aRoute, null);
```

Heartbeat

The heartbeat capability makes it possible to determine whether a Connector is present and for a Connector to confirm the presence of a LiveServer.

Connectors start the heartbeat by subscribing to the statistics topic /kn_system/heartbeat, and stop the heartbeat by unsubscribing from the same topic. The LiveServer notifies Connectors about its presence in two ways:

- by sending events every 10 seconds when the Connector subscribes to /kn_system/heartbeat
- by sending white spaces through the tunnel after a particular interval; the Connector sends a heartbeat event whenever it receives those white spaces

The LiveServer sends the white space heartbeats only when there is no activity down the tunnel otherwise. In other words, if there are no pending events to be sent to the Connector from the LiveServer, to keep the tunnel alive, the LiveServer sends a whitespace heartbeat after 30 seconds, and every 30 seconds thereafter. Connectors can notice both types of events and notify the application about the LiveServer's presence. When the Connector does not receive events in either of these ways, then it sends a timeout message to the subscriber application.

API Support for the Heartbeat Feature

To take advantage of this feature, subscriber applications need to implement a HeartBeatListener class with two functions, onUpdate and onTimeout. onUpdate is called whenever there is a heartbeat event from LiveServer. onTimeout is called when no events have been received from LiveServer for 40 seconds.

Additional APIs that support the heartbeat feature are

- HeartBeatHandler. This class handles the events coming from the LiveServer and in turn calls the subscriber's callback. The HeartBeatHandler class has a startHeartbeat method and a stopHeartbeat method for starting and stopping the heartbeat. HeartBeatHandler is an implementation of the class KNEventListener.
  - startHeartbeat subscribes to the /kn_system/heartbeat topic; basically, it listens in on that topic for the heartbeat and starts a 40-second timer. You can use the KNJServer startHeartBeat method to register the HeartBeatListener with the LiveServer.
  - Whenever a heartbeat event comes from LiveServer, the onHeartbeatEvent method resets the 40-second timer. When the timer runs out, or if it is a whitespace event coming from the LiveServer, then OnHeartbeatEvent calls the subscriber's OnTimeout.

  - stopHeartbeat unsubscribes from the /kn_system/heartbeat topic and deletes the timer.
  - HeartBeatHandler's onEvent implementation routes the Heartbeat events to your listener instance.
- TimeoutStatus, which is an enumeration to notify the subscriber application about the reason for the timeout. It has two values: one to indicate that the LiveServer is down, and the other to indicate that the LiveServer is in an unstable state.
- You can use the listener interface HeartBeatListener to catch the heartbeat event and the heartbeat timeout.

Logging

For debugging purposes, the LiveJava Connector supports logging using Jakarta's third-party open-source software package named log4j. Using the logging capabilities, you can set a target log file name, a log error level, whether rolling log files are used, and the maximum size of the rolling log files before the oldest files are overwritten. The default settings are

- Target log file: ./livejava.log
- Log level: ERROR
- Rolling: On
- Max rolling file size: 100KB

There are three ways you can change the Connector's logging configuration. They are executed in order of precedence, as follows:

- **./livejava_log.conf**: The Connector looks for an XML-formatted log4j configuration file at boot time. It looks for such a file in the same directory in which the application is running. If it finds such a file, it loads the configuration settings from that file and ignores any configuration settings that were made using the API. The configuration passed through the configuration file is the same for all instances of an application.
- **Environment variables**: Some of the log4j options are available as LiveJava Connector environment variables. These variables can be passed to the application using a Java command line switch. Different values can be passed to different application instances. Any configuration passed through the configuration file overrides configuration settings made through environment variables. The environment variables supported are as follows.
  - **KN.log.file:** The complete path of the target file to which the Connector's logging messages are to be redirected.
  - **KN.log.level:** The lowest level of log entries to be included. It takes values from the set {Assertion, Notice, Warn, Error, Fatal}. Setting the log level enables all log entries of that level and higher.
- **API**: Some of the log4j options can be set using methods in the KNLog class. These methods allow you to set the target log file and log level and to set rolling logs on and off. These APIs can be called statically. Any configuration passed via the configuration file overrides configurations set through API. Any settings made using the API override the default settings.
- **Default configuration**: The Connector uses the default configuration settings listed above.

- **Redirection**: You can also redirect the Connector's log to any OutputStream via an API method. This method can be used to redirect log output to system log daemon.

Logged messages have the following structure:

```
[Date time][0.0][Threadname] log Severity Level : Text message
```

The *Text message* contains *Method Name: Message*.

Table 9-2 provides a guideline for logging messages.

Table 9-2. Logging error levels.

Sr. Number	Log4Net Level	Connector Level	Remarks
1.	DEBUG	KN_ASSERTION	Informational messages.
2.	INFO	KN_NOTICE	An internal verification failed.
3.	WARN	KN_WARNING	Something needs to be addressed.
4.	ERROR	KN_ERROR	A serious problem that did not stop operations.
5.	FATAL	KN_FATAL	Causes the Connector to stop operation.A critical error has taken place.

Presence

Presence allows applications to monitor topics so that information about subscribers can be known: whether they are online or offline, subscribing, unsubscribing, and so on. Presence is supported by the following:

- "presenceSubscribe" on page 314
- "PresenceUnsubscribe" on page 314

presenceSubscribe

To start listening to a topic for the purposes of presence, use KNJServer.presenceSubscribe. This function takes the following parameters:

- The name of the topic to monitor.
- The presenceHandler. This handler has two callback mechanisms that your application must implement:
  - OnConnect. This is called whenever an application subscribes to a particular topic for which you are setting up presence. When an application connects to that topic, the LiveServer sends an event containing information within standard headers. Your application can handle that information as needed.
  - OnDisconnect. As with OnConnect, except that it is called whenever an application unsubscribes to the topic being listened to.
- A request status handler.

PresenceUnsubscribe

If you want to stop paying attention to a topic, use presenceUnsubscribe to specify the topic to stop listening to.

Request/response

The LiveJava Connector supports the LiveServer's request/response system, which is described under "Request/response" on page 148. That discussion covers all the capabilities of request/response. All request/response functionality is provided in the LiveJava Connector's KNJServer class. For more information on that class, see the API documentation for the LiveJava Connector. The LiveJava Connector API documentation is installed in the LiveJava Connector's installation directory (which is accessed by browsing to /*LiveJava_Installation*/docs/javadocs/index.html).

The methods that support request/response are:

- addRequest_BLOCKING
- addProvider
- addRequest
- addRequestManager
- addResponse
- deleteProvider
- updateRequest

Journal Connection Callback

The LiveJava Connector allows a callback to be specified to handle journal status. This callback, if set, is invoked every time a tunnel route connection is started, stopped, or closes unexpectedly. Example 9-11 demonstrates the use of a journal connection callback.

Example 9-11. Journal connection callback code sample.

```
import com.knownow.microserver.*;
import com.knownow.common.*;

public class MyTest implements KNJournalStatusHandler, KNEventListener
{
        public void onEvent(KNEvent anEvent)
        {
            System.out.println(anEvent.toString());
        }
        public void onJournalStatus(KNEvent jsEvent)
        {
            System.out.println(jsEvent.toString());
        }

        public static void main(String[] args)
        {
            MyTest myTest = new MyTest();
            KNJServer aServer = new
KNJServer("http://LiveServer.example.com:8000/kn");

            aServer.setJournalStatusListener(myTest);

            KNRoute aRoute = aServer.subscribe("/what/chat",myTest, null, null);
        }
}
```

When a tunnel route connection is started, the Connector generates an event which contains the following headers:

```
status="200"
kn_payload="Watching Topic"
```

When a connection is closed normally, the Connector generates an event which contains the following headers:

```
status="0"
error="0"
kn_payload="Journal connection closed normally"
```

When a journal connection dies unexpectedly and explicitly, the Connector generates an event which contains the following headers:

```
status="0"
error="1"
kn_payload="Journal connection closed unexpectedly"
```

If the journal status callback is specified, the Connector invokes the callback with this journal status event.

Simple Event Mapping Support

The LiveJava Connector supports simple event mapping. Simple event mapping makes it possible for applications to map inbound events (coming in from subscriptions the applications have made) into objects/data types. It also allows applications to map objects, which they are attempting to publish, into events.

The mapping does not happen automatically. The application registers certain callbacks with the Connector, requesting that when an event with a particular header or value inside is received on a particular subscription (or on any of the application's subscriptions), the Connector should call a particular callback to map the event data into an object and then process that object.

Similarly, the application registers a callback (or implements an interface on the type) asking that when an object of type "daisy" is published, the Connector should first call a particular function which will translate it into an event to be published.

Mapping Outbound Objects to Events

In order to map outbound objects/data types into events, the application registers with the Connector that when publishing objects of "type x," a particular callback should be used to create the event to be transmitted.

The application object implements the KnowNow interface KNConvertible. The method objectToEvent() of the KNConvertible interface is called by the Connector when it publishes the object. If the object does not implement this interface, the Connector throws a NoSuchMethodException.

Example 9-12 and Example 9-13 demonstrate mapping outbound objects to events.

Example 9-12. Implementing the KNConvertible interface.

```
import com.knownow.microserver.*;
import com.knownow.common.*;

public class KNObject implements KNConvertible
{
	private int    _knID = 0;
	private int    _knExpires = 0;
	private String _knPayload = "This is a test payload: PUBOBJ";
	public KNObject() {};
	public KNObject(int kn_id, int kn_expires)
	{
	    _knID = kn_id;
	    _knExpires = kn_expires;
	}
```

```
        public void setKNId(int id)
        {
            _knID = id;
        }

        public void setKNExpires (int expires)
        {
            _knExpires = expires;
        }
        public KNEvent objectToEvent()
        {
                KNEvent anEvent = new KNEvent();

            anEvent.set("kn_id", new Integer(_knID));
            anEvent.set("mykn_expires", new Integer(_knExpires));
            anEvent.set("kn_payload", _knPayload);

            return anEvent;
        }
}
```

Example 9-13 demonstrates publishing an object that implements the KNConvertible interface.

Example 9-13. Mapping outbound events to objects.

```
import java.lang.reflect.*;
import com.knownow.microserver.*;
import com.knownow.common.*;

/** simple test that publishes an object */

public class TestPubObj {

        private static KNJServer aServer;

        public static void main(String args[]) throws NoSuchMethodException,
            IllegalAccessException, InvocationTargetException
        {
            String aURL = "http://10.10.13.25:8001/kn";

            aServer = new KNJServer(aURL);
            long startTime = 0;
            //System.out.println("Version = " + aServer.getVersion());

            if (args.length < 1) {
            System.out.println("Usage: TestPubObj [iterations]");
            System.exit (1);
```

```
        }

        int TIMES = Integer.parseInt (args[0]);

        boolean bval;

        KNObject anObject = new KNObject();
        anObject.setKNExpires(100);

        for (int i = 1; i <= TIMES; i++)
        {
            anObject.setKNId(i);
            bval = aServer.publish("/what/mgktest", anObject, null);
                if (!bval)
            {
                System.err.println("Published " + i + " events");
                    System.exit(0);
            }
        }
        System.exit(0);
    }
}
```

Mapping Inbound Events to Objects

In order to implement the functionality of mapping inbound events into objects in the application, the subscribe() call of the Connector has been extended to include a set of maps of the type-callback mappings. The basic functionality available through these maps is as follows:

- Any event with a header *x* and value *y* calls a particular callback. This allows mapping of all events with, for example, the header StockSymbol and the value IBM to the IBMCallback(), while events with the header StockSymbol and the value AAPL map to the AAPLCallback() instead.
- A special value * (an asterisk) is supported in place of either the header or the value settings. This allows users to map events with, for example, the header StockSymbol, no matter what the value, to a particular callback.
- Note the special case of a map containing header * and value * going to a particular callback: Every event, regardless of data within, is sent to the particular callback specified for the subscription it came from. The shorthand of kn.subscribe(topic,callback) is just shorthand for the map of header * and value * to a callback.

Example 9-14 demonstrates mapping inbound events to objects.

Example 9-14. Mapping inbound events to objects.

```
import java.io.*;
import java.util.*;
import java.sql.*;

import com.knownow.microserver.*;
import com.knownow.common.*;

public class TestEventMap {

        private static KNJServer aServer;

        public static void main(String args[]) {

            String aURL = "http://10.10.13.25:8001/kn";

            aServer = new KNJServer(aURL);

            IBMCallback aCB1 = new IBMCallback();
            AAPLCallback aCB2 = new AAPLCallback();
            QuotesCallback aCB3 = new QuotesCallback();

            KNEventMap aMap = new KNEventMap();

            KNHVPair aPair1 = new KNHVPair("StockSymbol", "IBM");
            KNHVPair aPair2 = new KNHVPair("StockSymbol", "AAPL");
            KNHVPair aPair3 = new KNHVPair("Quotes", "*");

            aMap.addMap(aPair1, aCB1);
            aMap.addMap(aPair2, aCB2);
            aMap.addMap(aPair3, aCB3);

            KNRoute aRoute = aServer.subscribe("/what/mgktest", aMap, null, null);

        }

        static class IBMCallback implements KNEventListener
        {
            public void onEvent(KNEvent anEvent)
            {
                System.err.println("Invoked IBM Callback");
            System.err.println(anEvent.toString());
        }
}

static class AAPLCallback implements KNEventListener
{
        public void onEvent(KNEvent anEvent)
```

```
        {
            System.err.println("Invoked AAPL Callback");
            System.err.println(anEvent.toString());
        }
}

static class QuotesCallback implements KNEventListener
{
        public void onEvent(KNEvent anEvent)
        {
            System.err.println("Invoked Quotes Callback");
            System.err.println(anEvent.toString());
        }
        }
}
```

Using the Java Messaging Service

This section describes the KnowNow Java Messaging Service under the following topics:

- Java Messaging System API
- JMS Overview

Java Messaging System API

The Java Messaging System (JMS) is a Java API that provides your Java applications with a common programming model that is portable across messaging systems. KnowNow's implementation of the JMS API supports the publish-and-subscribe model and works with BEA WebLogic 6.1 and IBM WebSphere 4.0 application servers.

In the simplest sense, publish-and-subscribe is intended for a one-to-many broadcast of messages, while point-to-point is intended for one-to-one delivery of messages. In the publish-and-subscribe model, a producer can send (or publish) a message to many consumers through a topic. Consumers who wish to receive messages simply subscribe to a topic. Any message addressed to a topic is delivered to all of the topic's consumers.

The information provided in the following sections complements the JMS tutorial available from Sun Microsystems; it has been modified to use KnowNow as the JMS provider interface.

For more information on basic JMS API concepts and JMS architecture, refer to Sun Microsystems documentation and JMS specifications, which are available at

http://java.sun.com/products/jms

Note: The KnowNow JMS API supports a large subset of the Sun interfaces and methods. For a list of unsupported interfaces and methods, see "Unsupported JMS Interfaces" on page 327 and "Unsupported JMS Methods" on page 327.

JMS Overview

A JMS application is made up of

- "Administered Objects" on page 324
- "Connection Factories" on page 324
- "Destinations" on page 325
- "Connections" on page 325
- "Sessions" on page 325

- "Message Producers" on page 325
- "Message Consumers and Message Listeners" on page 326
- "Messages" on page 326

Administered Objects

Two parts of a JMS application, destination and connection factories, are maintained administratively rather than programmatically. JMS clients access these objects through portable interfaces, so client applications run with very little or no change. JMS applications look up administered objects through the Java Naming and Directory Interface (JNDI).

The KnowNow JMS API provides a tool called JmsAdmin to create JMS Destination and TopicConnectionFactory administered objects on the BEA WebLogic JNDI namespace.

Connection Factories

A connection factory is the object the client uses to create a connection with a LiveServer. A connection factory encapsulates a set of configuration parameters that can be defined using the jmsadmin.conf properties file. Each connection factory is an instance of the TopicConnectionFactory interface.

You can create new connection factories on the WebLogic JNDI namespace using the following command. Before you do so, edit the knjmsadmin.properties file with appropriate entries and add knjms.jar to the CLASSPATH.

```
java com.knownow.jms.admin.JmsAdmin TopicConnectionFactory jndi_name
```

A JMS client program usually performs a JNDI lookup of the connection factory. The following code fragment illustrates obtaining an InitialContext object and using it to look up the TopicConnectionFactory by name.

Example 9-15. Connection factories.

```
Hashtable env = new Hashtable(10);
env.put(Context.INITIAL_CONTEXT_FACTORY,
       "weblogic.jndi.WLInitialContextFactory");
env.put(Context.PROVIDER_URL, "t3://localhost:7001");

Context jndiContext =  new InitialContext(env);

(TopicConnectionFactory) jndiContext.lookup("TopicConnectionFactory");
```

Destinations

A destination is a object your client uses to specify the target of messages it produces and the source of messages it consumes. In the publish/subscribe messaging domain, destinations are called topics.

You can create new topics on the WebLogic JNDI namespace using the following command. Before you invoke this command, add knjms.jar to the Java CLASSPATH.

```
java com.knownow.jms.admin.JmsAdmin Topic topic_name
```

The following code fragment performs a JNDI lookup of a previously created and administered topic.

```
Topic myTopic = (Topic) jndiContext.lookup("MyTopic");
```

Connections

A connection represents an open TCP/IP socket connection with the LiveServer. You can use the connection to create one or more sessions. Once you have a TopicConnectionFactory object, you can create a connection as follows:

```
TopicConnection topicConnection = topicConnectionFactory.createTopicConnection();
```

When your application completes, it should close the connection that was created. Closing a connection closes its message producers and message consumers.

```
topicConnection.close();
```

Sessions

A session represents a context for producing and consuming messages. Use session to create message producers, message consumers, and messages. If you created a TopicConnection object, you would use it to create a TopicSession as follows:

```
TopicSession topicSession = topicConnection.createTopicSession(false,
Session.AUTO_ACKNOWLEDGE);
```

Message Producers

A message producer is an object created by a session and is used for sending messages to a destination. The publish/subscribe form implements the TopicPublisher interface.

```
TopicPublisher topicPublisher = topicSession.createPublisher(myTopic);
```

Once you have created a message producer, you can use it to send messages. You must first create the messages.

```
topicPublisher.publish(message);
```

Message Consumers and Message Listeners

A message consumer is an object created by a session and is used for receiving messages from a destination. The publish/subscribe form implements the TopicSubscriber interface.

```
TopicSubscriber topicSubscriber = topicSession.createSubscriber(myTopic);
```

Note: The KnowNow JMS API does not support durable subscriptions.

Once you have created a message consumer, you can use it to receive messages asynchronously with a message listener.

A message listener is an object that acts as an asynchronous event handler for messages. It implements the MessageListener interface, which contains one method, onMessage. In the onMessage method, you define the actions to be taken when a message arrives. You register the message listener with a specific TopicSubscriber by using the setMessageListener method.

For example, if you define a class named TopicListener that implements the MessageListener interface, you could register the message listener as follows:

```
TopicListener topicListener = new TopicListener();
topicSubscriber.setMessageListener(topicListener);
```

After you register the message listener, you call the start method on the TopicConnection to start message delivery. Once message delivery begins, the message consumer automatically calls the message listener's onMessage method whenever a message is delivered. The onMessage method takes one argument of type Message, which the method can cast to any of the other message types.

Messages

JMS applications produce and consume messages that can be used by other applications. For complete documentation of message headers, properties, and bodies, see the documentation of the Message interface in the JMS API JavaDoc.-generated HTML.

The JMS API defines five different message body formats, also known as message types, with which you can send and receive data in many forms. The JMS API provides methods for creating messages of each type and for filling in their contents.

For example, to create and send a TextMessage to a topic, you might use the following code segment:

```
TextMessage message =  topicSession.createTextMessage();
message.setText("This is the actual message");
topicPublisher.publish(message);
```

For instructions on how to write a simple JMS publish/subscribe application using the BEA WebLogic 6.1 application server, refer to the following example provided in the LiveJava Connector's installation directory:

```
/Connector_root/examples/jms/client/weblogic/ (for Weblogic)
```

For instructions on how to write a simple JMS pub/sub application using the IBM WebSphere 4.0 application server, refer to the following example provided as part of the LiveJava Connector API:

```
/Connector_root/examples/jms/client/websphere/ (for WebSphere)
```

Unsupported JMS Interfaces

None of the point-to-point JMS API calls are supported and therefore cannot be invoked:

- javax.jms.Queue
- javax.jms.QueueBrowser
- javax.jms.QueueConnection
- javax.jms.QueueConnectionFactory
- javax.jms.QueueReceiver
- javax.jms.QueueSender
- javax.jms.QueueSession
- javax.jms.TemporaryQueue

Unsupported JMS Methods

The following JMS methods are not supported and therefore cannot be invoked:

- javax.jms.Session.commit
- javax.jms.Session.rollback
- javax.jms.Session.recover
- javax.jms.Session.close
- javax.jms.Session.getMessageListener
- javax.jms.Session.setMessageListener

- javax.jms.Session.acknowledgeMessage
- javax.jms.TopicSession.createDurableSubscriber
- javax.jms.TopicSession.createTopic
- javax.jms.TopicSession.createTemporaryTopic
- javax.jms.MessageConsumer.receive
- javax.jms.MessageConsumer.receiveNoWait

A Guide to the Java API

The LiveJava Connector has an associated API with classes and methods to support the functionary discussed in this chapter. Many of those classes and methods have already been mentioned previously in this chapter. This section provides a quick overview of some of the most important classes and methods in the LiveJava Connector's API that have not already been mentioned. For detailed information about each class and method, see the API's separate documentation, which can be found under the Connector's installation directory as described under "Installing the LiveJava Connector" on page 288.

- "Creating and Managing Events" on page 329
- "Publishing, Subscribing, and Unsubscribing" on page 330
- "Managing Topics, Events, and Routes" on page 330
- "Setting Queue Options" on page 330

Creating and Managing Events

The KNEvent class is used to represent an event that can be published or which an application can receive through a subscription. Events can contain multiple headers. This class provides a variety of methods for creating events and manipulating their contents.

The KNEventListener interface is used to process KNEvent objects that your application receives. You provide an implementation of the onEvent method.

Cursors

Cursors support the ability to move through events in a topic a page at a time. You can also move through those events in other ways when using cursors. A discussion of how cursors work is provided under "Cursors" on page 146. The API calls that support cursors are:

- Cursor
- CursorException
- CursorListener
- CursorParameters
- Page

Publishing, Subscribing, and Unsubscribing

The KNJServerInterface interface has a complete set of methods for managing publish and subscribe operations for your application to implement.

KNJserver is a subclass of KNJServerInterface. It also provides a complete set of methods for managing publish and subscribe operations, as well as add requests, create journal commands, and so on.

The KNOptions class is used to pass optional data to the LiveServer when subscribing to a topic. Options are represented as headers. The following options have special meanings:

- do_max_age
- do_max_n
- kn_expires

Managing Topics, Events, and Routes

The KNRoute class is used to represent the route for a subscription. An object of this type is returned by the KNJServer.subscribe methods and is a required parameter for the KNJServer.unsubscribe method.

KNJServer provides several methods for managing topics, events, and routes on a LiveServer.

- KNJServer.addTopic
- KNJServer.clearTopic
- KNJServer.deleteEvent
- KNJServer.deleteRoute
- KNJServer.deleteTopic
- KNJServer.setEventProperty
- KNJServer.setRouteProperty
- KNJServer.setTopicProperty

Setting Queue Options

The following methods are provided for managing the Connector's event queue:

- KNJServer.clearOutboundQueue
- KNJServer.flushOutboundQueue
- KNJServer.isAutoQueueing

- KNJServer.isOutBoundQueueing
- KNJServer.loadOutboundQueue
- KNJServer.saveOutboundQueue
- KNJServer.setAutoQueueing
- KNJServer.setOutboundQueueing

Index

Symbols

- (minus sign) as time specifier 60
* (asterisk), inbound event mapping in Java and 320
+ (plus sign) as time specifier 60
@ (at sign) as time specifier 60

A

about:blank 51
ActiveX
    Connector. *See* LiveActiveX Connector.
    controls, registering 216
    enabling 173
    JavaScript shim. *See* JavaScript Connector.
    Windows Connectors and 18, 213
AddConnectionStatusHandler 235
add_journal 49
    providers and 153
    requestors and 153
    service managers and 151
add_notify 49
    examples 34
    request/response and 154
addProvider (LiveJava Connector method) 153, 315
AddProvider (Windows Connectors method) 153, 257
addRequest (LiveJava Connector method) 155, 315
AddRequest (Windows Connectors method) 155, 254
addRequest_BLOCKING (LiveJava Connector method) 155, 315
AddRequestBlocking (Windows Connectors method) 155, 255
addRequestManager (LiveJava Connector method) 152, 315
AddRequestManager (Windows Connectors method) 152, 257
addResponse (LiveJava Connector method) 156, 315
AddResponse (Windows Connectors method) 156, 256
addresses, IP. *See* IP addresses.
add_route 49
    offhost routing example 135
    providers and 153
    service managers and 152
add_topic 50
    topic filter modules and 116, 118
addTopic (LiveJava Connector method) 330
administered objects 324
admission filters 111
all since last connect (ASLC)
    Connector support for 142
    LiveJava Connector 306
    topic properties and 82
    Windows Connectors 243
APIs (application programming interfaces). *See* LiveServers; modules; *and individual Connector names.*

applications
See also individual Connector names.
communications with Connectors 16
customizing
overview 7
reasons for 15
number of Connectors and 138
Web. See cross-domain Web applications; JavaScript Connector.
architecture
LiveServer, overview 2
ASP.NET controls. See Live.NET Connector.
ASP.NET Web Parts, Live.NET Connector and 272, 279
asterisk (*), inbound event mapping in Java and 320
at sign (@) as time specifier 60
audience for book xix
authenticate (module function) 133
authentication/authorization modules. See security.
Authentication policy (LiveServer parameter) 5, 139
authorize (module function) 133
average (Expr parameter) 105

B

bandwidth
high, leader election and 185
Basic. See Visual Basic.
batch class of commands 45
batch command 50, 139
capabilities of 45
status events and 60
batching requests, JavaScript Connector 185
batchPost 50, 139
BEA WebLogic, Java Messaging Service and 327
blank command 51
JavaScript events and 42
boldface type, meaning of in book xxi
book
contents xix
conventions xxi
browsers, using to send commands to the LiveServer 36
bytes, filler, JavaScript 185

C

C# code examples
enabling offline queuing 241
instantiating a Connector 233
C++
See also Visual C++.
Connector. See LiveC++ Connector for Windows.
static libraries 230
DLLs versus 230
CA (certificate authority). See certificate authority (CA) files.
cache, JavaScript hash 185
callbacks
Java, subscribing topics to 305, 307
JavaScript 42, 51
journal status 316
cascading style sheets (CSS), ASP.NET controls and 275
certificate authority (CA) files
using
with the LiveJava Connector 299
with the Windows Connectors 215
certificates. See certificate authority (CA) files.
characters
allowable in events 26
delimiters in Expr module 103
encoding 40
escaped 40
JavaScript Connector 189
LF 26
literal space 26
NUL 26, 175
SP 26
CLASSPATH
setting in UNIX 295
setting in Windows 295
CleanupCursor (Windows Connectors method) 247
CleanupRequestBlocking (Windows

Connectors method) 155, 256
Clear (Windows Connectors method) 240
clearOutboundQueue (LiveJava Connector method) 330
clear_topic 51
kn_topic_nodelete and 51, 81
clearTopic (LiveJava Connector method) 330
Close (Windows Connectors method) 247, 258
clusters
kn_event_id 64
kn_route_cluster_state 75
shutting down and restarting 70
support of 4
code
ellipsis in xxi
examples
C#. *See* C# code examples.
Visual Basic. *See* Visual Basic.
samples, location of 226
typeface, meaning of xxi
COM (common object model) programming. *See* LiveActiveX Connector.
com.knownow.util.GetAuthInfo 297
command-line parameters
JavaScript. *See* JavaScript Connector.
LiveJava Connector system properties. *See* LiveJava Connector.
commands
batching 62
status events and 60
LiveServer, URL root and 23
sending to LiveServer 36
communication flow 7
compilers, supported. *See* platforms and tools.
configuration files, jmsadmin.conf 324
connection factories 324
creating 324
connection parameters, Windows Connectors 233
connections, changing default number of in Windows registry 232
connections (Java Messaging Service) 325
connection status
handlers, registering and removing 235
connection status (*continued*)
managing, in Windows 234
Windows Connectors, testing 235
Connector (class) 229
instantiating, code examples 233
number of instances 232
overview 232
Connectors
APIs
location of documentation 227
overview 17–19
uses for 15
applications and, number of 138
capabilities 16
common capabilities 137–159
communications with applications 16
disconnections and (reconnect/retry) 145
features 17
filter modules and 140
functions of 16
JavaScript. *See* JavaScript Connector.
LiveActiveX. *See* LiveActiveX Connector.
LiveBrowser. *See* JavaScript Connector.
LiveC++. *See* LiveC++ Connector for Windows.
LiveJava. *See* LiveJava Connector.
LiveServers and 138
overview 15
overview of operations 16
Pocket PC. *See* LivePDA Connector.
publishing and subscribing, overview 139
publishing and subscribing and 7
publishing in batch mode 50, 139
rules for use 138
session management 139
status events 140
tunnel route formats 12
tunnel routes for 39
unsubscribing, overview 139
Windows. *See* Windows Connectors.
constants
JavaScript Connector 188
security modules 132
contacting KnowNow xxii

contents of book xix
content-type (kn_payload parameter value) 72
control characters, escaping 40
core modules 96
count (Expr parameter) 105
cross-domain Web applications 181–183
  See also JavaScript Connector.
  application code notes 182
  document.open and 182
  HTML pages not using Connector 182
  instructions on creating 181
  IP addresses 183
  JavaScript Connector library and 181
  Netscape 4 and 181
CSS (cascading style sheets), ASP.NET controls and 275
Cursor (class) 146, 329
CursorException 329
CursorListener 329
CursorParameters 247, 329
cursors
  client side 146
  Connector support for 143, 146
  dynamic 146
  LiveActiveX Connector and 259
  LiveJava Connector and 329
  server side 146
  static 146
  Windows Connectors and 247
custom events. *See* events.

D

database recovery, kn_time_t and 81
data headers. *See* headers.
debugging
  logging support for
    LiveJava Connector 312
    Windows Connectors and 249
  options, JavaScript Connector 164, 184
deleted (kn_payload parameter value) 72
deleteEvent (LiveJava Connector method) 330
delete_notify 51
deleteProvider (LiveJava Connector method) 157, 315
delete_route 51
  request/response and 156
deleteRoute (LiveJava Connector method) 330
delete_topic 51
  kn_topic_nodelete and 51, 81
deleteTopic (LiveJava Connector method) 330
deletions, guidelines 83
delimiter characters 103
Destination (JMS object), creating 324
destinations, JavaScript Connector and 168
destinations (Java Messaging Service) 325
  creating 325
  topics and 325
detachFilter (LiveJava Connector method) 315
developer Web pages xxii
directories, documentation 130
disconnections, Connectors and 145
displayname (JavaScript Connector method), overriding default value 184
display names, JavaScript Connector 184
DLLs (dynamic linked libraries)
  C++ static libraries versus 230
  for the Live.NET Connector 278
  for the Windows Connectors 227
  LibKNCom.dll 231
  LibKNDotNet.dll 231, 278
  Log4Net.dll 278
  modules and 96
  MSCOREE.dll 278
/docs/ directory 130
documentation
  conventions xxi
  KnowNow xx
    APIs xx, 130
    LiveBrowser xxi
document.domain 55
document object model (DOM), JavaScript and 163
document.open
  cross-domain Web applications and 182
  using instead of KNDocument 185
DOM (document object model), JavaScript and 163
Domain (parameter), overriding whoami

with 55
domains
security, JavaScript access and 163
do_max_age
JavaScript Connector and 186
LiveJava Connector and 330
replacement header for 67
do_max_n
LiveJava Connector and 330
replacement header for 68
do_method commands 44–82
See also individual command names.
batching 50, 62, 139
status events and 60
categories 44
classes 45
permissions and 45
example 34
filter modules and 116, 118
format 44
kn_ headers and 44, 45
reference 49–55
reserved name status 33
set_topic_property, kn_max_queue_size and 71
status events and 60
duplicate squashing 35
Filterduplicates and 111
durable subscriptions, KnowNow JMS support of 326

E

email
SMTP, sending in response to events 115
encoded event. *See* events.
EnsureConnected (Windows Connectors method), reusing Connectors and 258
environment variables, LiveJava Connector 295
escape, using instead of kn_escape 185
escaped characters 40
ESS (Enterprise Syndication Solution), won't start 141
events 21–84
See also individual Connector names; headers; topics.
allowable characters in 26
authorization and 134
blocking 128
communication flow 7
components of 9
contents, retrieving and setting 131
definition of term 2
deleting from topics 51, 51
deletion events
headers for 169
kn_deleted 63
kn_deletions 63
duplicate squashing 35
Filterduplicates and 111
encoded 40
expiration time 64
JavaScript 169
filter modules and 127
formats 25
js tunnel route format and 41
forwarding
specifying maximum age 67, 169
unexpired events 68, 169
history 67
JavaScript, callbacks 42
mapping, simple event. *See* simple event mapping.
maximum number of in a topic, setting 71
notify, adding or replacing 49
offhost routing of 135
order of headers within 25
padding, js tunnel route and 39
periodic 79
possible filter module actions upon 128
queue size, setting maximum 71
request, forwarding 46
route IDs 76
route URIs 82
kn_routed_from 75
kn_route_location 76

events (*continued*)
- sending 52
  - to other topics or URIs 128
- size of 27–32
- special headers for 59
- status. *See* status events.
- struct for 132
- subscribe, callback mechanism for 229
- syntax 26
- timer 79
- timestamps 80
  - JavaScript 186
- topics and 9, 9
- topics for 81
- tunnel route formats and 39
- updating 54, 70

exceptions
- C++ 260
- classes for 143
- Connector support for 143
- Java, NoSuchMethodException 318
- LiveJava Connector 304
- Live.NET Connector 279

Expr filter module 102
- delimiter characters 103
- example 110
- parameters 104
- using 102

F

feedback, providing xxii

files
- configuration. *See* configuration files.
- formats, for modules 96
- jar, LiveJava Connector 288
- jmsadmin.conf 324
- knjmsadmin.properties 324
- LiveJava Connector log 297
- LiveJava Connector system.properties 297
- microserver.log 297
- Web.Config 275

filler bytes, JavaScript 185

filter (JavaScript Connector method) 178

Filterduplicates, hop-by-hop headers and 111

Filterduplicates filter module 111
- parameters 112

filter modules 99–120
- *See also* modules.
- attaching 65, 99, 124
  - threadsafe implementations 125
- capabilities of 99
- changing 99
- creating 99
- detaching 66, 126
  - in Java 315
  - threadsafe implementations 126
- do_methods and 116
- events and 127
  - event management 124
- Expr 102
  - parameters 104
  - using 102
- file formats 100
- Filterduplicates 111
  - parameters 112
- for sending email 115
- functions of 99
- KnModuleStart and 123
- KnModuleStop and 124
- loading into LiveServer 122
- naming 97
  - headers for 65
- number of 100
- overview 13, 96
- ownership options 66
- parameters
  - headers for 66, 66
  - setting 100
- possible functions of 128
- removing 99
- routes and 100
- start-up and shutdown processing 122
- Tcl email filter 115
- threadsafe implementations 127
- topics and 100
  - do_methods 118
- using 99
- XPath 116, 116

filter modules (*continued*)
XSLT 118
XSLTtoHeaders 120
/filters 65, 97
filters, admission 111
filters. *See* filter modules.
Flat Files permissions module 5
Flush (Windows Connectors method) 240
flushOutboundQueue (LiveJava Connector method) 330
form data, JavaScript and 167
forward slashes (/). *See* slashes (/)
frames
callbacks and 43
JavaScript access and 163
JavaScript Connector 188
swapping, JavaScript Connector options 185
framesets
JavaScript Connector 163
ActiveX shim and 175

G

GetNextPage (Windows Connectors method) 247
GetParameters (Windows Connectors method) 234
GetPreviousPage (Windows Connectors method) 247
GetQueueing (Windows Connectors method) 240
GetRouteId (Windows Connectors method) 245
GET. *See* HTTP and HTTPS.
GoogleSpelling.cpp, route creation snippets 89
Greek language example 26

H

hash cache, JavaScript 185
hash table of headers, JavaScript 188
HasItems (Windows Connectors method) 240
header (Expr parameter) 105
headers 33–82
See also individual header names.
batching 50, 139
custom 34
data, defined 35
definition of term 23
do_method commands. *See* do_method commands.
duplicate squashing and 35
examples 25
extracting values using the Live.NET Connector 279
hop-by-hop
defined 35
Filterduplicates and 111
JavaScript Connector 188
kn_. *See* kn_ headers.
names of 25, 33
reserved 33, 38
route, define 35
rules for using 37
special 59
status event. *See* status events.
tunnel routes, LiveJava Connector 316
using 34
UTF-8 and 37, 41
heartbeat
Connector support for 143
LiveJava Connector and 310
Windows Connectors and 248
HeartBeatHandler (LiveJava Connector method) 310
HeartBeatListener (LiveJava Connector method) 310, 311
help command 52
helper modules 96
hop-by-hop headers
defined 35
Filterduplicates and 111
hop-by-hop headers. *See* headers.
HTML, JavaScript access and 163
HTML pages
cross-domain applications and 182
rendering dynamically 42

HTTP 1.0 keepalive, Connector support for 144
HTTP and HTTPS
See also secure sockets layer (SSL).
ESS and 141
requests, encapsulating 133
using 36–38
version, specifying for LiveJava Connector 298
HTTPClient.disableCertificateVerification 141
HttpWebRequest.Timeout 268

I

IBM WebSphere, Java Messaging Service and 327
IConnectionStatusHandler
connection status events and 229
Open and 234
ICursor 146, 247
ICursorListener 247
identification numbers (IDs)
for events. *See* events.
kn_id and 70
user, JavaScript Connector and 186
IHeartbeatListener 248, 248
IListener (class) 229, 244
ILogger 251
inbound objects
mapping to events in Java 320
mapping to events in JavaScript 178
InitializeCursor (Windows Connectors method) 247
initial route publication (IRP), headers 53, 69
InitRequestBlocking (Windows Connectors method) 155, 255
installation programs, Windows Connectors 219
Internet Explorer. *See* Microsoft Internet Explorer.
IP addresses
cross-domain applications and 183
security modules 132
IPage 247
IRequestStatusHandler (class) 236, 236
IRP (initial route publication), headers 53, 69
isAutoQueueing (LiveJava Connector method) 330
IsConnected (Windows Connectors method) 235
isOutBoundQueueing (LiveJava Connector method) 331
Italic text, meaning of in book xxi
ITransport::Parameters 229, 233

J

Japanese language example 26
jar files, LiveJava Connector 288
Java. *See* LiveJava Connector.
Java Messaging Service (JMS) 323–328
administered objects 324
APIs 323
durable subscriptions and 326
unsupported interfaces 327
unsupported methods 326, 327
BEA WebLogic and 327
connecting to LiveServer 324
connection factories 324
creating 324
connections 325
destinations 325
creating 325
topics and 325
IBM WebSphere and 327
KnowNow, request/response support 288
message consumers 326
message listeners 326
message producers 325
messages 326
overview 323
sessions 325
Java Naming and Directory Interface (JNDI), administered objects and 324
java.protocol.handler.pkgs 299
JavaScript
See also JavaScript Connector.
about:blank and 51
callbacks 42, 51

JavaScript (*continued*)
- journal creation and 49
- route commands and 42
- same domain security policy 51
- security domains 163
- setting string properties 55
- UTF encoding 41
- whoami and 52

JavaScript Connector 161–187
- ActiveX shim 173–176
  - API notes 175
  - Internet Explorer security options and 173
  - LiveActiveX Connector and 173
  - properties 176
  - using 175
- APIs 188–209
  - constants 188
  - documentation for 188
  - frames 188
  - headers 188
  - localization 188
  - status handlers 189
  - strings 188
- command-line parameters 184–187
  - debugging options 164, 184
  - event timestamps 186
  - examples 186
  - filler bytes 185
  - hash cache value 185
  - kn_argv and 184
  - kn_queryString and 184
  - kn_topic and 184
  - language codes 185
  - rules for 184
  - speed options 185
  - start application immediately 184
  - table of 184
  - target window names 185
  - topics 186
  - tunnel route age 186
  - tunnel route IDs 186

JavaScript Connector (*continued*)
- command-line parameters (*continued*)
  - tunnel route URI 186
  - user display names 184
  - user IDs 186
- events
  - identifiers 167
  - kn object and 188
  - publishing 166
- framesets 163
  - ActiveX shim and 175
- library 52
  - cross-domain Web applications and 181
- LiveServer status and 164
- loading 162
- localization 179–180
- MIME, Base64 and 189
- operations of 163
- overview 18
- publish operations, form data 167
- simple event mapping. *See* simple event mapping.
- status handlers
  - methods for 172
  - publishing 166
  - subscribing 168
  - unsubscribing 171
- subscribing to topics 168
  - destinations 168
  - options 169
- supported MIME types 177
- tunnel route for 39
- tunnel route format 12
- unsubscribe operations 171
- writing cross-domain Web applications. *See* cross-domain Web applications.

javascript_domain 55

jcert.jar 289

JMS (Java Messaging Service). *See* Java Messaging Service (JMS).

JmsAdmin (JMS tool) 324

jmsadmin.conf, connection factories and 324

jms.jar 289
JNDI (Java Naming and Directory Interface), administered objects and 324
jnet.jar 289
journals
  as sessions 11
  callbacks for 316
  Connectors and 138
  creating 49
  duplicate squashing and 35
  LiveJava Connector API 302
  names of 138
  overview 12
  providers and 153
  random numbers and 12
  reconnecting to 53
  request/response, creating 153
  topics, Maximum Session Reconnect Time and 54
  tunnels and 138
  using 11
jsse.jar 289
js tunnel route format 40–43
  See also tunnel routes.
  about 39
  event format and 41
  flushing 74
  JavaScript callbacks 42
  JavaScript Connector and 12
  selecting 74

K

keytool utility 299
key/value pairs. *See* headers.
/kn 22
KN (JavaScript Connector object) 188, 208
kn (JavaScript Connector object) 188, 202–208
  displayname, overriding default value 184
  filter 178
  methods 204–208
  properties 202–203
  publish 166
  publishform 167
kn (JavaScript Connector object) (*continued*)
  setHashCache 185
  subscribe 168
  tunnelID, initial value 186
  tunnelMaxAge, initial value 186
  tunnelURI, initial value 186
  unsubscribe 171
  userid
    displaying value of 184
    overriding default value 186
kn_activexPassword 176
kn_activexProxy 176
kn_activexProxyPassword 176
kn_activexProxyUsername 176
kn_activexServerURI 176
kn_activexUsername 176
KnApiEvent (class) 127, 131
kn_argv (JavaScript Connector object) 188
  command-line parameters and 184
kn_atomic 62
KnAuthenticateSet (module function) 123
KnAuthObject_Command 133, 134
KnAuthObject_e 133
KnAuthObject_Event 133, 134
KnAuthObject_Route 133, 134
KnAuthObject_Topic 133, 134
KnAuthObject_Web 133, 134
kn_authorize, offhost routing and 136
KnAuthorizeSet (module function) 123
kn_autostart (JavaScript Connector command-line option) 184
kn_batch 50, 62
kn_block 62
  request/response and 155
KNChatlet
  KnowNow ASP.NET 273
  WinForm 270
kn_connection 62
KNConvertible (class) 318
  examples 319
kn_cookie, offhost routing example 136
kn_debug (JavaScript Connector command-line option) 164, 184
kn_default_kn_expires 63

set_topic_property and 54
kn_deleted 63
kn_deletions 63
add_route and 49
JavaScript Connector subscription options 169
request/response and 159
route command and 53
sensing deletions and 83
update_route and 55
Windows Connectors Subscribe and 243
KN.disableCertificateVerification 141
KN.disableKeepAlives 298
kn_displayname (JavaScript Connector command-line option) 184
KNDocument (class) 42
using document.open instead 185
KnErrorForbidden 136
KnErrorSuccess 136
KnErrorUnauthorized 136
kn_escape, using escape instead 185
KNEvent (class) 329
set 301
kn_event_id 64
notify and 52
KNEventListener (class) 305, 329
examples 305
KnEventPost (module function) 128
KNEventViewer
KnowNow ASP.NET 273
WinForm 270
KNException 143
LiveC++ Connector 260
LiveJava Connector 304
kn_expires 64
closing tunnel routes with 39
JavaScript Connector subscription options 169
LiveJava Connector and 330
sensing deletions and 83
tunnel routes and 12
Windows Connectors Subscribe and 243
kn_filtername 65
set_topic_property and 54
kn_filteroptions 66
kn_filterparams 66, 100
set_topic_property and 54
KnFilterRouteAttach (module function) 125
KnFilterRouteDetach (module function) 126
KnFilterRouteEvent (module function) 127
KnFilterTopicAttach (module function) 125
KnFilterTopicDetach (module function) 126
KnFilterTopicEvent (module function) 127
KN.forceHTTP_1.0 298
kn_from 67
add_journal and 49
add_route and 50
delete_route and 51
offhost route creation and 87
offhost routing example 135
providers and 153
route class of commands and 46
route command and 53
update-route and 55
Windows Connectors Subscribe and 244
kn_hashCache (JavaScript Connector command-line option) 185
kn_ headers 26, 56–82
See also individual kn_ header names.
do_method commands and 44, 45
globally supported 59
locally supported 56
reference 62–82
summary table 46, 56
Windows Connectors Subscribe and 244
kn_history_* headers 67
kn_history_since_age 67
cursors and 147
JavaScript Connector and, subscription options 169
kn_hold_new_events and 69
Windows Connectors Subscribe and 243
kn_history_since_event_id 67
kn_route_checkpoint and 75
kn_history_since_n 68
cursors and 147
JavaScript Connector and, subscription options 169
kn_hold_new_events and 69

kn_history_since_n (*continued*)
  LiveServer shutdown and 70
  Windows Connectors Subscribe and 242
kn_history_since_time 68
kn_history_until_age 68
kn_history_until_event_id 68
kn_history_until_n 68
kn_history_until_time 68
kn_hold_new_events 53, 69
  Windows Connectors Subscribe and 243
kn_id 70
  delete_notify and 51
  delete_route and 51
  examples 25, 34
  for shutting down and restarting the LiveServer 70
  JavaScript Connector and 167
  kn.publish and 166
  Process Manager and 70
  providers and 153
  sensing deletions and 83
  shutting down LiveServer and 70
  update_notify and 54
  update_route and 55
  Windows Connectors Subscribe and 244
KNImage
  KnowNow ASP.NET 273
  WinForm 270
KNJAVA_HOME environment variable 295
knjava module 96, 287
knjmsadmin.properties, connection factories and 324
knjms.jar 289
  connection factories and 324
/kn_journal 12
  routes and 53
  temporary topics and 77
KNJournalStatusHandler (class) 302
KNJserver, LiveJava Connector request/response and 315
KNJServer (class) 296, 330
  addTopic 330
  clearOutboundQueue 330
  clearTopic 330
  deleteEvent 330
KNJServer (class) (*continued*)
  deleteRoute 330
  deleteTopic 330
  detachFilter 315
  flushOutboundQueue 330
  isAutoQueueing 330
  isOutBoundQueueing 331
  loadOutboundQueue 331
  proxy servers and 298
  publish 301
  saveOutboundQueue 331
  setAutoQueueing 331
  setEventProperty 330
  setOutboundQueueing 331
  setRouteProperty 330
  setTopicProperty 330
  subscribe
    KNRoute and 330
    use of 305, 307
  subscribeASLC, use of 306
  subscription methods 305
  unsubscribe 309
    KNRoute and 330
KNJServerInterface (class) 330
kn_lang (JavaScript Connector command-line option) 185
kn_lastTag_ (JavaScript Connector command-line option) 185
KNList 274
KNLog 312
KNLogConfig.ini 249, 251
KNLogConfig.xml 249
KnLogConfig.xml 251
KN.log.file 297
KNLogger 251
KN.log.level 297
kn_mailbcc 115
kn_mailcc 115
kn_mail_filter 115
kn_mailfrom 115
kn_mailsubject 115
kn_mailto 115
KNMarqueeBand
  KnowNow ASP.NET 274

KNMarqueeBand (*continued*)
WinForm 270
kn_max_queue_size 71
set_topic_property and 54, 71
kn_module 71
providers and 153
KnModuleStart 131
KnModuleStart (module function) 97, 123
KnModuleStop (module function) 97, 124
KNOptions 307
KNOptions (class) 330
kn_options (JavaScript Connector command-line option) 185
JavaScript ActiveX shim and 176
kn__submitRequest and 176
NUL characters and 175
kn_owner 52, 71
KnowNow
contacting xxii
documentation set xx
Web pages xxii
KnowNow LiveBrowser Developer's Guide xxi
KnowNow LiveServer Administration Guide xx
KnowNow LiveServer Developer's Guide xx
kn_password 88
kn_payload 72
as a status event 72
deletions and 83
examples 25, 34
kn.publish and 166
kn.subscribe and 168
message bodies and 25
simple tunnel route format and 40
kn_provider_id 72
providers and 153
Service Managers and 73
service managers and 151
kn_provider_name 73
kn_queryString (JavaScript Connector method) 184
kn_redrawCallback 42
KnRequest (struct) 132, 132, 133
security modules and 132
kn_request_format 73
offhost route creation and 88
kn_request_id 73
kn_request_manager 73
kn_request_response 73
kn_response_flush 73
kn_response_flush (header)
add_journal command and 49
add_route command and 50
route command and 53
kn_response_flush (JavaScript Connector command-line option) 185
kn_response_flush_interval 74
add_journal command and 49
add_route command and 50
route command and 53
kn_response_format 74
add_journal and 49
add_notify and 49
add_topic and 50
batch command and 50
delete_notify and 51
delete_route and 52
notify and 52
route command and 53
set_topic_property and 54
status events and 60
tunnel routes and 39
update_notify and 55
update_route and 55
kn_response_uri 74
kn_restartTunnel (JavaScript Connector method), JavaScript ActiveX shim and 176
kn_retry 75
Web services and 91
KNRoute (class) 305, 307, 330
kn_route_checkpoint, replacement header for 75
kn_route_cluster_state 75
LiveServer shutdown and 70
kn_routed_from 75
duplicate squashing and 35
sensing deletions and 83

kn_route_id 76
    Windows Connectors Subscribe and 244
kn_route_location 76
    duplicate squashing and 35
    kn_status_to and 76
    kn_uri and 82
/kn_routes 11
    kn_owner and 72
    security and 134
kn_sendCallback 42
kn_server 176
kn_since_checkpoint 76
    deprecation of 69
/knSOAP 88
kn_status 76
kn_status_from 76
KNStatusHandler (class) 302
    use of 302
kn_status_to 76
    batch command and 50
    notify and 52
    status events and 60
kn__submitRequest (JavaScript Connector method)
    JavaScript ActiveX shim and 176
    kn_options and 176
/kn_subtopics 11
    security and 134
/kn_system/heartbeat 248, 310
KNTable 274
kn_target (JavaScript Connector command-line option) 185
kn_temporary 77
kn_temporary_expires 79
kn_timer_fired 79
kn_timer_interval 79
kn_timer_nextupdate 79
kn_timer_updated 79
kn_timestamp (JavaScript Connector command-line option) 186
kn_time_t 80
    database recovery and 81
    sensing deletions and 83
    Windows Connectors Subscribe and 244
kn_to 81
    add_notify and 49
    add_route and 50
    add_topic and 50
    clear_topic and 51
    delete_notify and 51
    examples 34
    notify and 52
    notify and topic command classes and 46
    offhost route creation and 87
    offhost routing example 135
    providers and 153
    route command and 49, 53
    set_topic_property and 54
    update_notify and 55
    update_route and 55
    Windows Connectors Subscribe and 244
kn_topic (JavaScript Connector command-line option) 184, 186
kn_topic_allow_update 82
kn_topic_nodelete 81
    clear_topic and 51
    delete_topic and 51
    set_topic_property and 54
kn_topic_nopersist 81
    set_topic_property and 54
kn_topic_nopost 81
    set_topic_property and 54
kn_topic_order 82
/kn_topics 11
KNTree
    KnowNow ASP.NET 275
    WinForm 270
kn_tunnelID (JavaScript Connector command-line option) 186
kn_tunnelLoadCallback 42
kn_tunnelMaxAge (JavaScript Connector command-line option) 186
kn_tunnelURI (JavaScript Connector command-line option) 186
kn_unescape, using unescape instead 185
kn_uri 82
    kn_route_location and 76
    Windows Connectors Subscribe and 245
kn_user 88

kn_userid (JavaScript Connector command-line option) 186
kosme (header example) 25

L

language examples, Greek and Japanese 26
LDAP module 121
LDAP permissions module 5
leader election 185
leaks, routes and topics 78
LF character 26
lib command 52
  security and 134
LibKNCom.dll 227, 231
LibKNDotNet.dll 227, 231, 278, 278
  required DLLs for 278
LibKNPda.dll 227
libraries, Windows Connectors 227
licensing, Connector support for 144
literal space character 26
LiveActiveX Connector
  See also Windows Connectors.
  APIs, accessing 231
  cursors and 259
  installing different versions of 216
  JavaScript ActiveX shim and 173
  JavaScript shim. *See* JavaScript Connector.
  library 227
  namespace recommendations 259
  Visual Basic and Visual C++ and 231
LiveBrowser, documentation xxi
LiveBrowser Web service (LWWS) 148
LiveC++ Connector for Windows
  See also Windows Connectors.
  APIs
    accessing 230
    documentation location 230
  C++ static libraries and 230
  error codes 260
  exceptions 260
  header files, location 230
Live Data Source 269
LiveJava Connector 285–331
  APIs 329–331
    addProvider 153
LiveJava Connector (*continued*)
  APIs (*continued*)
    addRequest 155
    addRequest_BLOCKING 155
    addRequestManager 152, 152
    addResponse 156
    cursors 329
    deleteProvider 157
    event management methods 330
    events 329
    journals 302
    queueing methods 330
    request/response methods 315
    route management methods 330
    status events 302
    topic management methods 330
    updateRequest 156
  callbacks, for journal status 316
  connecting to 296
  do_max_age and 330
  do_max_n and 330
  events
    code examples 301
    managing 329
    methods for managing 330
    publishing 301
  examples 288, 327
  exceptions 304
    NoSuchMethodException 318
  filter modules and, detaching 315
  heartbeat capabilities 310
  HTTP version and 298
  installing 288
    on UNIX 295
    on Windows 290
  jar files 288
  Java Messaging Service. *See* Java Messaging Service.
  journals 302
  kn_expires and 330
  logging 289, 312
    enabling 297
  overview 19
  presence 314

LiveJava Connector (*continued*)
proxy servers and, using 298
queues, methods for setting options 330
request/response 148–159
jar files for 288
methods for 315
routes 330
methods for managing 330
secure servers and 299
security information 296
obtaining 297
simple event mapping. *See* simple event mapping.
SSL and 289
using 299
status events 302
API call 302
examples 302
subscription operations
methods for 305
options 307
supported platforms 19, 288
system properties 297
setting multiple 298
system requirements 287
topics
methods for managing 330
options 330
setting properties 145
subscribing to a callback 305, 307
unsubscribing from 309
tunnel routes, callbacks for operations 316
user names 298
using 296–322
./livejava_log.conf 312
Live.NET Connector 267–283
See also Windows Connectors.
accessing header values 279
APIs
accessing 231, 278
documentation 268
ASP.NET controls 272–277
CSS support and 275
Live.NET Connector (*continued*)
ASP.NET controls (*continued*)
deployment 276
KNChatlet 273
KNEventViewer 273
KNImage 273
KNList 274
KNMarqueeBand 274
KNTable 274
KNTree 275
Sharepoint and 272
system configuration and 275
Web.Config and 275
bundling with applications 278
dependencies 216
exceptions 279
installation requirements 216
libraries 227
Live Data Source 269
name of DLL 278
number of events per topic 268
programming notes 278
required DLLs 278
Web Parts and 272, 279
Windows form and control methods and 271
WinForm components 269
KNChatlet 270
KNEventViewer 270
KNImage 270
KNMarqueeBand 270
KNTree 270
LivePDA Connector 267–283
See also Windows Connectors.
APIs, documentation 268
dependencies 217
installation requirements 217
library 227
LiveServer_release_notes.html xxi
LiveServers 1–20
APIs, documentation of xx
closing connection to 258
Connectors and 138

LiveServers (*continued*)
documentation xx
functions of 4
JavaScript ActiveX shim and 176
number of 4
overview 3
shutting down 70
status, determining with JavaScript Connector 164
testing 54
won't start due to bad filter name 97
load balancing, request/response and 151
loadOutboundQueue (LiveJava Connector method) 331
LoadQueue (Windows Connectors method) 240
persistence and 240
usage 240
localization
JavaScript Connector 179–180, 188
JavaScript language codes 185
log4j 312
log4j-1.2.8.jar 289
log4net 249
log4net 1.2.0.8 249
Log4Net.dll 278
Logger 251
logging
Connector support for 144
enabling, LiveJava Connector 297
JavaScript Connector debugging options and 184
Live.NET Connector 312
support files for 289
Windows Connectors 249
APIs for 251
login (Connector method) 139
logout (Connector method) 139
lowercase (Expr parameter) 105
LWWS (LiveBrowser Web service) 148

M

max (Expr parameter) 105
Maximum Session Reconnect Time, topic properties and 54
Max Queue Size (KSA field) 71
Message (class) 239, 279
Publish method and 229
message consumers (Java Messaging Service) 326
MessageListener (class) 326
message listeners (Java Messaging Service) 326
message producers (Java Messaging Service) 325
messages
Java Messaging Service. *See* Java Messaging Service.
using kn_payload for 25
messaging, Java. *See* Java Messaging Service.
meta-topics. *See* topics.
microserver.jar 289
microserver.log 297
Microsoft Internet Explorer
JavaScript security domains and 164
security options 173
Microsoft OLEView 231
Microsoft Visual Studio .NET 2003
Live.NET Connector and 216
MIME
Base64
JavaScript and 189
types, JavaScript Connector support 177
min (Expr parameter) 105
minus sign (-) as time specifier 60
miscellaneous class of commands 45
blank 51
help 52
lib 52
test 54
whoami 55
modules 20, 95–136
APIs 122–136
documentation of 130
overview 130
required functions 130
authentication/authorization. *See* security modules.

modules (*continued*)
definition of term 96
event management 124
Expr 102
parameters 104
using 102
failure of, LiveServer actions regarding 123
file formats 96
Filterduplicates 111
parameters 112
filter. *See* filter modules.
initializing 97
Java support 96, 287
knjava 96, 287
KnModuleStart and 123
KnModuleStop and 124
loading into LiveServer 97, 97, 122
naming 97
nsperm and LDAP 121
overview 13
security. *See* security modules.
shutdown processes 97, 122, 124
start-up processes 97, 122, 123
types of 96
XPath 116
XSLT 118
XSLTtoHeaders 120
MSCOREE.dll 278
m_serverURL 233
multiple requests, not batching in JavaScript 185

N

names
filter modules 97
headers for 65
headers 25
reserved 33, 38
user display names, JavaScript Connector 184
name/value pairs. *See* headers.
Netscape
cross-domain Web applications and, IP addresses and 183
version 4
cross-domain applications and 181
nosocketshare 185
NoSuchMethodException 318
notify class of commands 45
add_notify 49
notify and 49
default topic for 38
delete_notify 51
kn_to and 46, 81
notify 52
update_notify 54
notify command 52
nsperm module 121
ns_sendmail 115
n-tier deployments 140
NUL character 26
JavaScript ActiveX shim and 175
null user name 72

O

objects
administered 324
inbound
mapping to events in Java 320
mapping to events in JavaScript 178
outbound
mapping to events with Java 318
mapping to events with JavaScript 177
objectToEvent (LiveJava Connector method) 318
offhost routes, creating for Web services requests 87
offhost routing 135
offline queuing
LiveJava Connector, methods 330
persistence and 240
queue size, setting 71
Windows Connectors 239–241
capabilities 240

offline queuing (*continued*)
  Windows Connectors (*continued*)
    methods 240
    status codes 241
OLEView 231
OnConnectionStatus (Windows Connectors method) 234
onError (JavaScript Connector method) 172
onError (JavaScript Connector object) 189
OnError (Windows Connectors method) 237
  base class 236
  using with other status handlers 236
onEvent (LiveJava Connector method) 329
onMessage (LiveJava Connector method) 326
onStatus (JavaScript Connector method) 172
onStatus (JavaScript Connector object) 189
OnStatus (Windows Connectors method) 237
  base class 236
  using with other status handlers 236
onSuccess (JavaScript Connector method) 172
onSuccess (JavaScript Connector object) 189
OnSuccess (Windows Connectors method) 237
  base class 236
  using with other status handlers 236
onTimeout (LiveJava Connector method) 310
OnTimeout (Windows Connector method) 248
onUpdate (LiveJava Connector method) 310
OnUpdate (Windows Connector method) 248
OnUpdate (Windows Connectors method), Subscribe and 244
op (Expr parameter) 106
Open (Windows Connectors method) 229, 233, 247
  connection status events and 229
  IConnectionStatusHandler and 234
  parameters and 233
/operations topic 70
outbound objects
  mapping to events
    Java 318
    JavaScript 177
ownership options for filters 66
owners of routes 52, 71

P

Page 329
parameters
  command-line, JavaScript. *See* JavaScript Connector.
  filter modules 100
Parameters (Windows Connectors method)
  Connector connections and 229
  Connectors and 233
  LiveActiveX Connector and 259
passwords
  JavaScript ActiveX shim 176
  kn_password and 88
  requiring 121
payloads, overview 9
periodic events 79
permissions, do_method command classes and 45
persistence
  offline queuing and 240
  topics, headers for 81
  topics and 62
platforms and tools 17
  Windows suite of Connectors 214
plus sign (+) as time specifier 60
Pocket PCs. *See* LivePDA Connector.
ports, cross-domain installations and 183
POST. *See* HTTP and HTTPS.
presence
  Connector support for 144
  detecting presence of a person 83
  LiveJava Connector 314
  Windows Connectors 253
presenceSubscribe (LiveJava Connector method) 144, 314
PresenceSubscribe (Windows Connectors method) 144, 253
presenceUnsubscribe (LiveJava Connector method) 314
PresenceUnsubscribe (Windows Connectors method) 144, 253
processes, controlling with kn_id 70
properties
  event. *See* events.

files. *See* files.
JavaScript ActiveX shim 176
route. *See* routes.
topics. *See* topics.
property:value pairs. *See* headers.
providers. *See* request/response.
proxies, Connector support for 144
proxy servers
JavaScript ActiveX shim and 176
LiveJava Connector and 298, 298
pubConn.SetQueueing 241
publications
See also individual Connector names.
communication flow 7
overview 2
publish (JavaScript Connector method) 166
publish (LiveJava Connector method) 301
Publish (Windows Connectors method) 239–241
messages and 229
parameters 239
reusing Connectors and 258
status event handlers 239
status events and 229
status handlers 236
publishform (JavaScript Connector method) 167
publishing. *See* publications.

Q

queuing. *See* offline queuing.

R

random numbers, journals and 12
reconnect/retry, Connector support for 145
regex (Expr parameter) 108
regsvr32.exe 216
regular expressions
debugging 106
delimiter characters 103
filter module for 102
parameters 104
tags 106
regular expressions (*continued*)
syntax 103
using 102
release notes, location of xxi
RemoveConnectionStatusHandler 235
RemoveProvider (Windows Connectors method) 157, 257
replace (Expr parameter) 108
replaceHeader (Expr parameter) 108
requestors. *See* request/response.
request/response 148–159
add_notify and 154
administrators, security settings 158
blocking 62, 155
Connector support for 145
headers for 73, 73
journals, creating 153
kn_deletions and 159
LiveJava Connector support for 288
load balancing 151
providers
add_journal and 153
add_route and 153
defined 149
delete_route and 156
deleting 156
monitoring 159
registering 152
security settings 157
tasks 152
requestors
add_journal and 153
defined 149
monitoring 159
security settings 158
requests
blocking versus non-blocking 155
sending 153, 154
updating 156
responses, sending 156
security settings 157
service managers
add_journal and 151

request/response (*continued*)
    service managers (*continued*)
        add_route and 152
        creating and registering 151–152
        defined 149
        deleting 156
        provider IDs and 151
        tasks 151
    support for 7
    tasks 149
        setup phase 149
    update_notify and 156, 156
    using 154
requests
    batching 62
        status events and 60
    forwarding 46
    JavaScript, not batching 185
    sending 36
request status events. *See* status events.
response_format, add_route and 50
REST
    overview 86
    requests 91
retries for Web services requests 75
return values, security 136
route class of commands 45
    add_journal 49
    add_route 49
    default topic for 38
    delete_route 51
    kn_connection and 62
    kn_from and 46
    route 52
        add_journal and 49
        add_route and 49
        with no kn_to 42
    update_route 55
route command 52
    add_journal and 49
    add_route and 49
    creating offhost routes with 87
    kn_connection and 62
    using 100
route command (*continued*)
    with no kn_to 42
    XSLT filter and 118
route headers. *See* headers.
route properties. *See* routes.
routes
    authorization and 134
    creating 49, 52, 81
    definition of term 2
    deleting 51, 52
    event forwarding 67
        JavaScript 169
    expiration time 64
        JavaScript 169
    filters for. *See* filter modules.
    flushing 49, 50, 53
    forwarding requests 46
    headers. *See* headers.
    IDs and events 76
        Subscribe and Unsubscribe and 242
    /kn_journal and 53
    leaks, possible reasons for 78
    LiveJava Connector and 330
    not reaped, reasons for 78
    offhost 135
        Web services requests and 87
    owners 52, 71
    ownership options 66
    replacing 49
    request/response and
        providers 152
        service managers 152
    starting points, specifying 67
    tunnel. *See* tunnel routes.
    updating 55, 100
    URIs 75, 76, 82
/RRTopic 150

S

sales contact information xxii
sample code. *See* code.
saveOutboundQueue (LiveJava Connector
        method) 331

SaveQueue (Windows Connectors method) 241
- persistence and 240
- usage 240

secure sockets layer (SSL)
- *See also* HTTP and HTTPS.
- LiveJava Connector and 289, 299
- Windows support 215

security
- *See also* modules.
- authentication, definition of term 121
- authorization, definition of term 121
- LiveJava Connector and 296
- *n*-tier deployments and 140
- offhost route permissions and 88
- request/response settings 157
- SSL. *See* secure sockets layer (SSL).

security modules 121
- *See also* modules.
- APIs, overview 132–136
- constants 132
- functions of 121
- KnAuthenticateSet and 123
- KnAuthorizeSet and 123
- KnModuleStart and 123
- KnModuleStop and 124
- loading into LiveServer 122
- nsperm and LDAP 121
- return values 136
- start-up and shutdown processing 122

self (JavaScript Connector object) 188, 189–202
- methods 192–202
- properties 190–192

service managers. *See* request/response.

session management 139
- overview 5

sessions (Java Messaging Service) 325

set (LiveJava Connector method) 301

setAutoQueueing (LiveJava Connector method) 331

setEventProperty (LiveJava Connector method) 330

setHashCache (JavaScript Connector method), kn_hashCache and 185

setMessageListener (LiveJava Connector method) 326

setOutboundQueueing (LiveJava Connector method) 331

setPos (LiveJava Connector method) 147

SetPos (Windows Connectors method) 147

setProxyAuthorization (LiveJava Connector method) 298

setProxyServer (LiveJava Connector method) 298

SetQueueing (Windows Connectors method) 240

setRouteProperty (LiveJava Connector method) 330

set_topic_property 54
- kn_max_queue_size and 71
- kn_topic_nodelete and 81
- kn_topic_nopersist and 81
- kn_topic_nopost and 81
- topic filter modules and 116, 118
- update_notify and 54

setTopicProperty (LiveJava Connector method) 145, 330

SetTopicProperty (Windows Connectors method) 145

Sharepoint 272, 279

shim, ActiveX. *See* JavaScript Connector.

shutdown (do_method), kn_id and 70

shutting down the LiveServer 70

simple event mapping
- JavaScript Connector and 177
  - inbound objects 178
  - outbound objects 177
  - supported MIME types 177
- LiveJava Connector and 318–322
  - asterisk (*) and 320
  - inbound objects 320
  - outbound objects 318

simple tunnel route format 39–40
- *See also* tunnel routes.
- about 39
- Connectors and 12
- flushing 74
- selecting 74

size of events 27–32
slashes (/), /*name*/ and /*name* conventions xxi
SMTP email, sending 115
SMTP Server (Service Network parameter) 115
SOAP and WSDL 93–94
 /knSOAP and 88
 LiveServer generation of 90
 obtaining a LiveServer's WSDL 88
 overview 86
 requests 90
SOAPRequestor.cpp
 REST request snippet 91, 92
 SOAP request snippet 90
SOs (shared objects), modules and 96
space character, literal 26
SP character 26
speed options, JavaScript Connector 185
SSL (secure sockets layer). *See* secure sockets layer (SSL).
startHeartbeat (LiveJava Connector method) 310
start-up and shutdown processing, modules 122
status (status event header) 60
status codes
 offline queuing, Windows Connectors 241
status events 36
 See also events.
 Connectors and 140
 definition of success 237
 do_method commands and 60
 headers 60
  kn_payload 72
  kn_route_location 76
  kn_status_from 76
  kn_status_to 60, 76
  status 60
 JavaScript Connector 189
  forwarding options 185
 LiveJava Connector 302, 302
  examples 302
 supporting commands 45
 topics for 76
 Windows Connectors. *See* Windows Connectors.
std (Expr parameter) 109
stock exchange topic example 9
stopHeartbeat (LiveJava Connector method) 310
stopping the LiveServer 70
strings
 JavaScript Connector 188
 window, properties 55
structs, KnRequest 132, 132, 133
subscribe (JavaScript Connector method) 168
subscribe (LiveJava Connector method) 305, 320, 330
 use of 307
Subscribe (Windows Connectors method) 242
 events, callbacks 229
 kn_deletions and 243
 kn_expires and 243
 kn_ headers and 243, 244
 kn_history_since_age and 243
 kn_history_since_n and 242
 kn_hold_new_events and 243
 OnUpdate and 244
 parameters 242
 reusing Connectors and 258
 status events and 229
 status handlers 236
 Unsubscribe and 242
subscribeASLC (LiveJava Connector method) 306
SubscribeASLC (Windows Connectors method) 243
SubscribeRouteId (Windows Connectors method) 245
subscribing. *See* subscriptions.
subscriptions
 See also individual Connector names.
 communication flow 7
 durable, support for 326
 overview 2
subtopics. *See* topics.
sum (Expr parameter) 109
supported platforms and compilers. *See* platforms and tools.

syntax, events 26
system configuration, ASP.NET controls and 275
system.properties file 297

T

target (HTML command), JavaScript Connector and 185
Tcl, email filter 115
TCP ports, security modules 132
technical support xxii
Temporary Topic Expires 78, 79
Temporary Topic Recognize 78
  kn_temporary and 77
test command 54
thawtetestcert.txt 300
threadsafe implementations
  filter modules and 127
    attaching 125
    detaching 126
time, specifying 60
TimeoutStatus (LiveJava Connector method) 311
TimeoutStatus (Windows Connector method) 248
timer events 79
timestamps
  event 80
    JavaScript 186
tools, supported. *See* platforms and tools.
topic (Expr parameter) 109
topic class of commands 45
  add_topic 50
  clear_topic 51
  delete_topic 51
  kn_to and 46
  set_topic_property 54
TopicConnectionFactory (JMS object), creating 324
topics
  See also events; *and individual topic names*.
  authorization and 134
  clearing events from 51

topics (*continued*)
  commands, kn_to and 81
  creating 13, 50
  default 38
  definition of term 9
  deleting 51
    preventing 81
  deletion events and 63
    JavaScript 169
  events and 9
    event routing 9
    filter modules and 127
  examples, stock exchange 9
  expiration 63, 77
  /filters 97
  filters. *See* filter modules.
  Java Messaging Service destinations and 325
  JavaScript Connector command-line option 186
  journal 11
  journals. *See* journals.
  /kn_system/heartbeat 248, 310
  leaks, possible reasons for 78
  managing 13
  meta-topics, definition of term 11
  modules and 13
  naming 13
  not reaped, reasons for 78
  /operations 70
  overview 9
  persistence 81
  persistent 62
  posting to, preventing 81
  properties 13, 82
    ASLC and 82
    Connector support for 145
    headers for 63
    setting or modifying 54
  referencing 13
  request/response. *See* request/response.
  requests, forwarding 46
  routes. *See* routes.
  security and 134

topics (*continued*)
  setting maximum number of events in 71
  temporary 50, 54
    expiration time 79
    header for 77
    routes and 53
    uses for 78
  URIs for 75, 76
TopicSubscriber (class) 326
toString, outbound object mapping and 177
tunnelID (JavaScript Connector method), initial value 186
tunnelMaxAge (JavaScript Connector method) 186
tunnel routes
  See also routes.
  age of, JavaScript 186
  closing 12, 39
  Connector creation of 138
  definition of term 12
  flushing 73
  formats 39
    js. *See* js tunnel route format
    kn_connection and 62
    selecting 74
    simple. *See* simple tunnel route format
  IDs, JavaScript 186
  journals and 138
  kn_expires and 65
  operations, Java callbacks for 316
  owners 52
  specifying 39
  URI for, JavaScript 186
tunnels, sharing 185
tunnelURI (JavaScript Connector method) 186
tyatto (header example) 25
type (Expr parameter) 109

U

unescape, using instead of kn_unescape 185
UNIX, specifying time 60
unsubscribe (JavaScript Connector method) 171
unsubscribe (LiveJava Connector method) 309, 330
Unsubscribe (Windows Connectors method) 246
  parameters 246
  reusing Connectors and 258
  status events and 229
  status handlers 236
  Subscribe and 242
unsubscribe operations. *See individual Connector names.*
update_notify 54
  request/response and 156, 156
  set_topic_property and 54
updateRequest (LiveJava Connector method) 156, 315
UpdateRequest (Windows Connectors method) 156, 256
update_route 55
update_route command, using 100
uppercase (Expr parameter) 109
URIs (uniform resource identifiers)
  for routes 76, 82
  for topics 75
  JavaScript ActiveX shim and 176
  JavaScript Connector and 186
URL root 22
  LiveServer commands and 23
URLs (uniform resource locators), of Web pages for downloads and information. *See* Web pages.
userid (JavaScript Connector method)
  displaying value of 184
  overriding default value 186
user IDs, JavaScript Connector 186
user names
  JavaScript ActiveX shim 176
  kn_user and 88
  null 72
  requiring 121
  specifying, LiveJava Connector 298
U suffix, meaning of 41
UTF-16
  JavaScript and 41

UTF-8
headers and 37, 41
JavaScript and 41
utilities
com.knownow.util.GetAuthInfo 297
keytool 299

V

value (Expr parameter) 110
version numbers, header for 62
Visual Basic
accessing LibKNCom.dll 231
code examples
enabling offline queuing 241
instantiating a Connector 233
cursors and 259
Visual C++
See also C++.
accessing LibKNCom.dll 231

W

Web applications, cross-domain. *See* cross-domain Web applications.
Web.Config file 275
WebLogic, BEA, Java Messaging Service and 327
Web pages
See also URLs (uniform resource locators).
KnowNow xxii
making interactive 52
Web Parts, Live.NET Connector and 272, 279
Web services 85–94
See also REST *and* SOAP and WSDL.
kn_request_format 73
requests 87–92
creating offhost routes for 87
kn_retry and 75
managing responses to 91
REST 91
retries 75
retrying 91
SOAP 90
WebSphere, IBM, Java Messaging Service and 327
whoami command 55
JavaScript and 52
overriding 55
security and 134
/who/*anystring*/s/ 77
/who/java 77
/who/wms 77
window (parameter) 43
window names, JavaScript target 185
Windows
form and control methods 271
registry, editing for number of connections 232
Windows Connectors 211–258
APIs
accessing 230
AddProvider 153
AddRequest 155
AddRequestBlocking 155
AddResponse 156
C++ static libraries 230
classes 229
CleanupRequestBlocking 155
InitRequestBlocking 155
overview 18, 214, 228
RemoveProvider 157
UpdateRequest 156
using 226–258
ASLC 243
creating 232
cursors 247
dependencies 216
DLLs for 227
events, sending and receiving 239, 242
heartbeat capabilities 248
heartbeats 248
installing 219–224
instantiating a Connector, examples 233
libraries for 227
list of 18, 213
LiveActiveX Connector. *See* LiveActiveX Connector.

LiveC++ Connector for Windows. *See* LiveC++ Connector for Windows.
Live.NET Connector. *See* Live.NET Connector.
LivePDA Connector. *See* LivePDA Connector.
LiveServers and
closing the connection to 258
specifying 233
logging 249
offline queuing and 239–241
methods 240
status codes 241
overview 18
presence 144, 253
publish operations 239–241
offline queuing and 239–241
status events and 239
request/response 148–159, 254–257
sample code, location of 226
SSL support 215
status events 236
definition of success 237
determining which handler to use 236
OnError 237
OnStatus 237
OnSuccess 237
subscribe operations 242–245
IListener 244
supported tools and platforms 214
topics, setting properties 145
uninstalling 225
unsubscribe operations 246
Windows Form (WinForm) controls. *See* Live.NET Connector.
Windows XP, Live.NET Connector and 216
WinForm (Windows Form) controls. *See* Live.NET Connector.
ws_rest 73
ws_soap 73

X

XML
evaluating documents with XPath 116
evaluating documents with XSLT 118
evaluating documents with XSLTtoHeaders 120
JavaScript Connector support of 177
XPath filter module 116
XSLT filter module 118
XSLTtoHeaders filter module 120